Plastic Self Organising Maps and Classification of Radar Data

Ben Sparks

Somerville College,
University of Oxford

September 4, 2008

Abstract

This report presents a new implementation and extension to the Plastic Self Organising Map (PSOM) algorithm proposed by Lang [8]. The main aims of the project are to implement the algorithm in Matlab with adequate commentary to enable further development, to reduce the 'black-box' status of the algorithm and parameters and to use the algorithm to classify radar data supplied by THALES Aerospace.

We successfully provide a Matlab implementation of the general PSOM algorithm and suggest several modifications to improve results. In particular we give an adaptation of the PSOM that we apply to the THALES radar data, and suggest extra features to help classify this specific data set.

We test the PSOM's continually learning behaviour on several different data sets, including benchmark sets (demonstrating non-stationary data) and data sets from THALES. The benchmark data sets displaying required features are created as part of the project, and can be generated with random distributions while retaining the important characteristics.

We document systematic experiments done on these data sets - to assess necessary parameters, and where possible to specify optimal values. The tests are also assessed for quality of classification and demonstration of the benefits of the modifications.

The new algorithm shows drastically improved results when the modifications are included, and gives a good classification on the THALES data set.

Finally we discuss where more work can be done in this area. Several different lines of enquiry are presented, including further adjustments to facilitate classification of the THALES data set, more modifications for the PSOM algorithm, and ways to implement the PSOM differently in order to set the work on a more rigorous foundation.

Contents

Acknowledgments

This project is written for submission as the 4-module dissertation on the MSc in Mathematical Modelling and Scientific Computing, at Oxford University in 2008. The project was proposed by THALES Aerospace, and supervised by Mason Porter and Philip Bond from Oxford University, and Diven Topiwala, Richard Lavery and Ian Ellis from THALES. The project builds largely on previous work on the Plastic Self Organising Map (PSOM) by Rob Lang [8].

Figures in the project are produced by myself unless otherwise stated, using Matlab and The Geometer's Sketchpad.

I'd like to thank Mason and Philip, my supervisors at the university, for their guidance, support and encouragement - and technical contributions. Mason has kindly offered to host the relevant computer files that go with this project, and they can be found at `http://www.maths.ox.ac.uk/~porterm/research/PSOM`

Many thanks go also to THALES Aerospace for sponsoring the project, and to Diven, Richard, and the others at THALES for offering their perspectives on the work, and creating a useful working relationship with the company.

1 Introduction

1.1 Motivation

In many different situations there is a need to classify data into classes or patterns, based on common features of some sort. Much of neural computing development has been motivated by this need. Our human senses and brains prove excellent at recognising patterns in situations where the data is readily accessible, but there are clearly situations where the sheer amount of data or dimensions make human classification not only difficult, but impractical. This motivates the vast and swiftly growing area of pattern recognition of which this project will merely focus on a very small part [1], [2].

Narrowing the field of view considerably we consider sets of data that exhibit *non-stationary* behaviour. By *non-stationary* we mean where classes that we wish to recognise actually change over time, and possibly appear or disappear. This can pose fundamental problems for systems not designed for such an environment [8]. In a *stationary* environment the most efficient method remains for a system to be 'trained' in some way to recognise certain classes, and then the system will hopefully recognise any repeated instances of similar classes. Primary examples of these are neural networks, which are electronic structures designed to model in some way part of the function of the human brain [1]. They typically consist of 'neurons' (nodes) containing some ability to process data and 'weights' (links) containing some information on relationships between neurons. Haykin offers this general definition [4],

> *A Neural Network is a massively parallel distributed processor made up of simple processing units which has a natural propensity for storing experiential knowledge and making it available for use.*

Defining our terms more precisely, we say a *stationary* data set is one where the class means and variances do not change over time [8], and a *non-stationary* data set is one where either the means or variances do change, and the number of classes may change. In this project we focus on the latter aspect, that is, we will not be considering data where the class means drift over time, rather that new classes may appear, and old classes may need to be forgotten. In fact, the algorithm proposed would certainly be able to take account of drift, and does do this as a matter of course, but we will not concentrate on that property in this study.

Static neural networks are designed for use on *stationary* data, and they do not work well in *non-stationary* environments, [8]. A key problem is that information is needed *a priori*. That is, a training period is needed to set the topology of the network and give example patterns to recognise. If new classes appear after the training period they will not be recognised in the same way as others that have been trained for, and if classes disappear from the data they will still be stored in the network, taking up valuable space. For an example of the problems encountered when using inappropriate clustering techniques see Appendix B, where we compare the results from a k-means clustering approach.

The self organising map (SOM), first developed by Kohonen [6], is an example of a so-called *static* Neural Network, which exhibits the above problems when used on *non-stationary* data. We discuss the working in more detail in the next section, but it is important to note here why an extension is needed. In our particular example of radar data (from THALES) we have a *non-stationary* situation. If an algorithm is classifying radar data in real time, then you ideally want to be able to handle new and unseen types of signals as they appear, without having to retrain your system. A *dynamic* Neural Network aims to do exactly this. Lang offers this definition to differentiate a new type of Neural Network, [8],

> *A dynamic Neural Network (DNN) is a Neural Network that can alter its own topology to accept novelty within a non-stationary signal space.*

To summarise, in many situations (including the radar data from THALES) it is an essential feature of a classification system to be able to adapt and learn new classes (and forget old ones) as it runs, without retraining. The *plastic* self organising map (PSOM) presented in this project is one possible solution to this problem.

1.2 The Self Organising Map

In order to explain the working of the Plastic Self Organising Map a brief description of the simple Self Organising Map (SOM), often called a Kohonen Map [6], will be useful.

Kohonen's 1990 article [7] contains a useful review of these concepts - here we present an overview using descriptions that are more in keeping with what follows in the rest of the project.

We initialise the feature map (the network that will organise itself into a format representing the data structure), by assigning each node in the map a random 'weight vector' of equal dimension to the input data we are using. Each node is connected to its neighbours by a link, and the aim is for neighbouring nodes in the feature map to be identified with vectors that are 'nearby' in the input data. If the feature map is one or two dimensional then we will produce a reduced dimensionality map that incorporates much of the clustering and structure information of the original data.

The algorithm works as follows:

1. Initialise - by assigning all nodes a random weight vector (this could be done in several different, and potentially significant ways - see [1] p.115-116 for a discussion).

2. Present an input $\mathbf{u}$ - from the data set, picked at random or in a pre-assigned order.

3. Calculate distances - by computing the Euclidean distance from the input vector to each weight vector at the nodes.

4. Select minimum distance - i.e. find the 'closest' node in the network (the focus - $\mathbf{x}_f$)

5. Update weight vectors - The focus node and nodes in the neighbourhood (the $\mathbf{x}_j$s) are changed by the following equation,

$$\mathbf{x}_j \rightarrow \mathbf{x}_j + \eta(t)\gamma(t)(\mathbf{u} - \mathbf{x}_j) \tag{1.1}$$

   where $\eta(t)$ is the 'learning rate', and $\gamma(t)$ is the 'neighbourhood function'.

6. Go back to step 2 unless all data has been used, in which case stop.

The key step is step 5, where the network is adjusted. The equation has the simple effect of making the node in question ($\mathbf{x}_j$) more like the input. How much more like the input is decided by $\eta(t)$ and $\gamma(t)$. The learning rate ($0 < \eta(t) < 1$) is a function that decreases in time and determines how drastically to adjust the node, while the neighbourhood function $\gamma(t)$ determines how big the 'neighbourhood' is, and effects a different learning rate at different points in the neighbourhood. We will use it to decrease the size of the neighbourhood over time.

Examples of possible $\eta(t)$ and $\gamma(t)$ are (from [3] and [4])

$$\eta(t) = \eta_0 \exp\left(-\frac{t}{\tau}\right), \tag{1.2}$$

$$\gamma(t) = \exp\left(-\frac{\|\mathbf{x}_f - \mathbf{x}_j\|^2}{2\sigma^2(t)}\right), \tag{1.3}$$

with constants set as

$$\eta_0 = 0.1, \tag{1.4}$$

$$\tau = 1000, \tag{1.5}$$

where $\sigma(t)$ is a monotonically decreasing function of time.

An important feature is that a 'training period' is built in to these functions. The learning rate reduces as time goes on, so that early on drastic changes are produced in the weight vectors, but later, as the map is hopefully becoming more representative, the learning rate decreases and smaller adjustments are made, until the map does not change. At this point the 'training period' is over. In parallel, the size of the neighbourhood decreases, so that early changes affect a large portion of the map, while later only small changes are made, close to the focus. These effects create a rough ordering of the map at an early stage, and then provide a fine tuning at the end.

The SOM provides a simple way to represent a data structure, but it will not change its behaviour after the training period is over. If we require classification in real time, and the ability to accept hitherto unseen patterns we have to retrain the whole map. It is not enough to simply increase the learning rate again, as this would almost certainly corrupt any structure already present. All the required patterns need to be present during the training phase for this to be a successful classification algorithm.

2 The Plastic Self Organising Map

The PSOM is an attempt to extend the basic ideas of the SOM to a situation where new patterns are able to be incorporated during the algorithm's running. In the process, the working of the PSOM has changed significantly from its ancestor (the SOM), and the algorithm is perhaps best thought of as a new process rather than a simple extension of the SOM.

We first summarise Lang's description of the PSOM, and explain how we implemented the algorithm in Matlab for this project. Following this we explain the changes made during this project and the reasoning behind them. Frequent references are made to Lang's thesis [8] for ease of comparison by those who wish to do so.

The PSOM consists of a network of neurons (nodes) and links, which will change over the course of the algorithm. Each neuron, $\mathbf{x}_j$, is identified with a vector (the weight vector), of the same dimension as the input space. Each link, c_{ij}, is a scalar representing the 'size' of the link between $\mathbf{x}_i$ and $\mathbf{x}_j$. If no link exists between the two then the value of c_{ij} is zero. The aim of the algorithm, as it works through the data, is to represent clusters of similar input data by a cluster of neurons with similar vectors, and with neurons connected by links. Eventually we will aim to create a graph where each data cluster is represented by a disconnected subgraph in our network.

The links actually represent both the distance between two neurons, and how recently these neurons have been updated (the 'age' of the link). This is one of the most significant differences when compared with other network clustering devices (like the SOM).

2.1 Lang's Algorithm

Figure 1 is a flowchart of the PSOM algorithm, slightly updated from Lang's [8] to include the details of assigning a class to each data input , hence providing the classification - the whole point of the algorithm.

It is stressed that what follows is Lang's work, explained here to make this project self-contained, and to give insight on the changes we make to the algorithm.

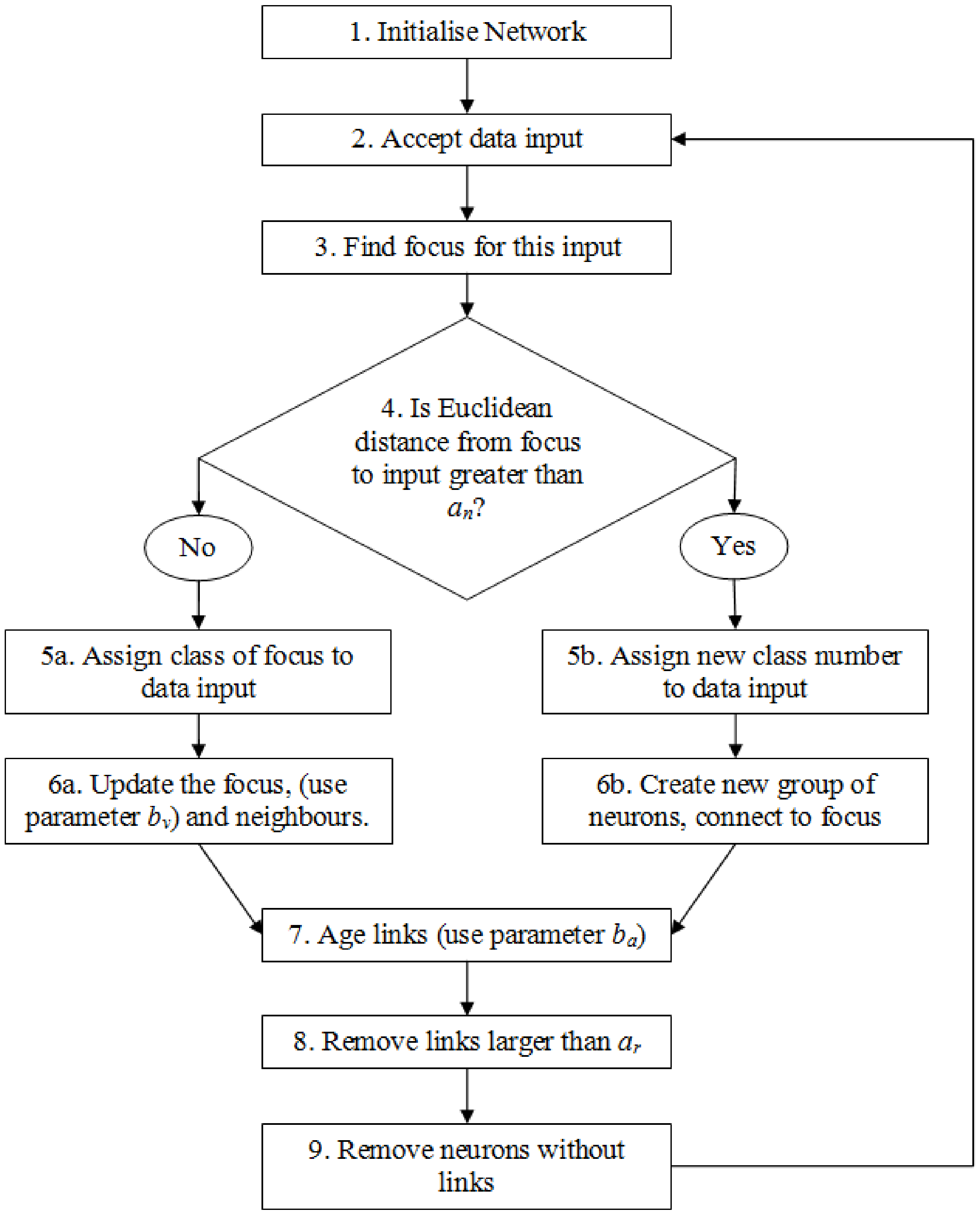

1. Initialise Network
2. Accept data input
3. Find focus for this input
4. Is Euclidean distance from focus to input greater than a_n?
No
Yes
5a. Assign class of focus to data input
5b. Assign new class number to data input
6a. Update the focus, (use parameter b_v) and neighbours.
6b. Create new group of neurons, connect to focus
7. Age links (use parameter b_a)
8. Remove links larger than a_r
9. Remove neurons without links

Figure 1: Flowchart of PSOM Algorithm

2.1.1 Parameters

We proceed by explaining each step and discussing relevant details of the implementation. It is worth noting that what follows will be partially specific to this Matlab implementation of the PSOM. We will need to use several parameters that Lang introduced, and we reuse his conventions so that threshold parameters are labelled with as (and subscripts) while scaling and incremental parameters use bs [8]. While these parameters will be the subject of much discussion, and this nomenclature may no longer be most appropriate, we will use the same convention for ease of comparison. Table 1 shows the details of the parameters as used by Lang[1]. We have clarified some of the descriptions, but left the names unchanged. See Table 2 in Section 2.3.1 for our revised set of parameters.

Parameter	**Name**	**Purpose**	**Range**
a_n	Node Building Parameter	The focus-input distance threshold after which a new group of neurons will be created.	$[0, 1]$
a_{cl}	Cluster Threshold	Threshold for neighbouring neurons to be considered 'near'.	$[0, 1]$
a_r	Maximum Link Length	Ageing threshold, links longer than this will be removed	Lang uses $[1, 100]$
b_c	Neuron Update Parameter	Either positive or negative depending on a_{cl}. Set algorithmically.	± 0.01
b_a	Link Ageing Parameter	How much links are aged each iteration (additive)	$[0, 1]$
b_v	Focus Update Parameter	How much to make the focus like the input	$[0, 1]$

Table 1: Parameters used in Lang's PSOM [8]

[1]Lang does not include the parameter b_v in his list of important parameters, but mentions it in passing while describing the algorithm. We include it here since it proves to have drastic effects on the algorithm's performance

2.1.2 Step-by-Step

Refer to Figure 1 to see the whole process with each step numbered as in the following description.

0) Set-up

The data we wish to classify must be presented as an $q \times dim$ matrix, where q is the number of signals to classify and dim is the dimension of the data. That is - the data signals are read as rows of a matrix one by one. The data must be normalised so all values lie in $(0, 1)$ ([8] p. 41). This is in order that each dimension of the data set is treated equally, otherwise when calculating Euclidean distances any dimension with larger numerical ranges would be treated with greater importance. Normalising solves this problem, but requires that maximum and minimum values are known for each dimension of the data. At this stage it is worth noting that a different normalisation, or weighting dimensions differently, will significantly alter the operation of the algorithm and may in fact be a useful way to add emphasis to certain dimensions. We revisit this discussion in section 5.3.

1) Initialise

We define our network in this implementation by keeping track of a list of neurons, and the links between them. The link matrix L, where

$$L_{ij} = \begin{cases} c_{ij} & \text{if a link exists between nodes } i \text{ and } j \\ 0 & \text{if no link exists,} \end{cases} \tag{2.1}$$

holds all information about the link locations and lengths. L will be a symmetric matrix, and we choose to implement the full matrix rather than just one half for ease of access. This means that whenever a link length changes we restate the change on the opposite side of the matrix.

Neurons are recorded as a row of the $n \times dim$ matrix X, where n is the number of neurons and dim is the dimension of the input space, as before. In this way each neuron is labelled as $\mathbf{x}_j$ for some j, and we will also assign a class number to every neuron, which labels it as part of a particular class.

The network is initialised with a group of nodes with random vectors, each given the same initial class number. In this implementation we choose a random point in the input space, create a node there, and then create two more nodes at random small deviations from this point. We then connect the two deviations to the initial random node with links equal to their Euclidean distance. Lang discusses several options for initialising the network, and suggests this version as the most useful. In practice it does not appear to matter to any great extent, as this group will tend to disappear over time anyway. See Section 5.1 for other suggestions.

2) Accept Input

This step involves simply taking a row of the input data set and presenting it to the algorithm. We will refer to this input vector as $\mathbf{u}$.

3) Find Focus

We calculate which of the neurons in the network is most similar to $\mathbf{u}$ by looking for the shortest Euclidean distance between it and each neuron - that is, the 2-norm of the difference in the vectors (and we will use $\|\cdot\|$ to denote the 2-norm unless otherwise stated). The closest neuron is called the focus and is labelled $\mathbf{x}_f$ (Lang uses $\mathbf{z}$). The distance from focus to input we denote d. We find the index of the focus neuron, f, as follows, ,

$$f = \arg\min_j \{\|\mathbf{u} - \mathbf{x}_j\|\}, \tag{2.2}$$

$$\text{so } d = \|\mathbf{u} - \mathbf{x}_f\| = \min_j \{\|\mathbf{u} - \mathbf{x}_j\|\}, \tag{2.3}$$

where $\arg\min_j$ is an instruction to return the value of j that gives the minimum result.

4) Check Size of d

The crucial step in the algorithm is to check the size of d. If $d < a_n$ (the node building parameter or threshold) then we conclude that the input is like enough to the focus and we proceed to update the focus accordingly. If $d \geq a_n$ we decide that no neuron in the network is sufficiently close to the input and we create a new group of neurons.

5a) & 6a) Focus and Neighbourhood Update: $d < a_n$

In this case we first adjust the focus to be more like the input using

$$\mathbf{x}_f \rightarrow \mathbf{x}_f + b_v(\mathbf{u} - \mathbf{x}_f). \tag{2.4}$$

Here Lang arbitrarily sets b_v to 0.9 and leaves this constant throughout the rest of the algorithm. We then update the 'neighbourhood' of the focus - by 'neighbourhood' we mean those neurons that are directly connected to the focus. Lang's algorithm uses two steps: first updating the links, then using the link lengths to update the neurons. The new link lengths are simply recalculated as the Euclidean distance from each neighbour to the new focus. Lang also uses the multiplier a_r (and sets this equal to 100), which seems to be only useful to make the link lengths easily rounded to integers between 0 and a_r in order that they may be written on network diagrams more easily. The new link length is given by

$$c_{fj} = a_r \|\mathbf{x}_f - \mathbf{x}_j\|, \tag{2.5}$$

remembering to adjust the symmetric other side of our L matrix when doing this.

Having updated the link lengths we now use them to update the neuron vectors themselves. Lang's method does this by using the a_{cl} parameter to decide whether neurons should be pulled towards the focus or pushed away. He does this using

$$\begin{gathered}\mathbf{x}_j \rightarrow \mathbf{x}_j + b_c c_{fj}(\mathbf{u} - \mathbf{x}_j) \\ \text{where } b_c = \begin{cases} 0.01 & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| < a_{cl} \\ -0.01 & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| > a_{cl}. \end{cases}\end{gathered} \tag{2.6}$$

The b_c value obscures the simplicity of this process. Simply, if the neighbour is at a Euclidean distance of less than a_{cl} the update will make the neighbour *more* like the input, by an amount proportional to that distance. If the distance is greater than a_{cl} then the update will make the neighbour *less* like the input, by a factor proportional to that distance. A consequence of this is that distant neighbours will be pushed away a large amount, due to their distance being large (see Section 2.2.3 for discussion of this point). The b_c parameter seems to only play the role of undoing the a_r multiplication from (2.5) (it is set equal to $\pm\frac{1}{a_r}$ [8] p. 39), while also changing sign to facilitate the pushing or pulling of the neuron.

Note that the value of c_{fj} is not necessarily the same as the Euclidean distance $\|\mathbf{x}_f - \mathbf{x}_j\|$ - it also takes account of the ageing process, detailed in step 7.

In summary, the aim of this part of the algorithm is to make the focus and its closest neighbours more like the input - so that the network learns to be more like the incoming data - while separating them from any neighbours that are too far away. The parameter a_{cl} determines which neighbours are considered to be close.

Figure 2 shows the network finding a focus close to an example input, and performing the neighbourhood update. These diagrams should be treated as an aid to understanding

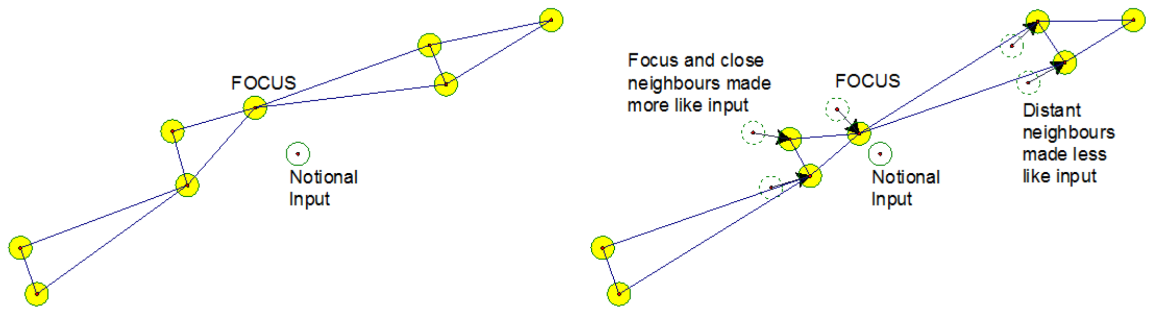

FOCUS
Notional Input
Focus and close neighbours made more like input
FOCUS
Notional Input
Distant neighbours made less like input

Figure 2: The left diagram shows the initial network with a notional input. The right shows the update process (dashed circles are the original positions of the neurons).

and can necessarily only represent a two-dimensional data set, but they are useful to show the behaviour of the various neurons.

The input data is classified by simply returning the class number of the focus. Lang mentions at a different stage that any neurons that end up in the close neighbourhood (less than a_{cl}) will have their class number reassigned to be the same as the focus. This means that groups of neurons with different classes can merge into one, thereby giving the network the opportunity to remove unnecessary classes by combining them with a useful one.

5b) & 6b) Create new group: $d \geq a_n$

Alternatively (this is the step that allows the algorithm to remain dynamic) if $d \geq a_n$, then we conclude that no neuron in the network represents the input data sufficiently well, and we create a new group of neurons. Lang is somewhat vague about the details, saying only that 'a group of neurons is created with vectors similar to the input, and attached to the focus.' [8] p. 38. In this implementation we choose to create a new neuron with identical vector to the input, and then create two other neurons with a small random deviation from this. Each of the new neurons is connected to the other new neurons, and also to the focus (possibly very distant). There are many possible ways of implementing this that stay within Lang's description. For example, only one of the new neurons could be connected to the focus, or they could be connected to points other than the focus. This highlights the need for further experimentation - see Section 5.1 for some discussion of this. Figure 3 shows the creation of a new group of neurons in the chosen manner.

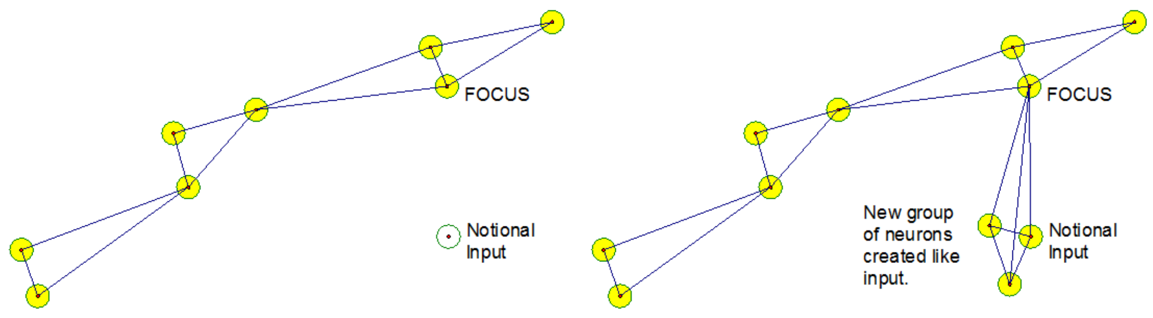


Figure 3: Again the left diagram is the starting network, this time with a distant input, the right shows the new group of neurons connected to the focus.

The new neurons are given a new unused class number, and the input data is classified with this new number.

7) Age Links

After processing the input following one of the two paths above we age all links in the network. Lang achieves this by adding a small amount - parameter b_a - to each link:

$$c_{ij} \rightarrow c_{ij} + b_a. \tag{2.7}$$

Links that have not been updated in the focus-neighbourhood update for some time get progressively older. In this way we can detect which areas of the graph have not seen an input for some time.

8) Remove Old Links

After the ageing process we check for any links that are longer than parameter a_r, and remove them. This represents the removal of links that connect very distant neurons, and links connecting neurons that have not seen any similar input for a long time. This is the process by which disconnected subgraphs are created.

9) Remove Isolated Neurons

Finally, we check if we have isolated any neurons by removing all their links. If so, we remove the neuron from the network. This will occur when a cluster's links become very 'old', and are removed one by one. When neurons are left unconnected we will remove them

Now return to step (2) and repeat with the next row of the input data matrix.

2.1.3 Discussion

Lang introduced six parameters that completely control the working of the algorithm. The question remains of how to choose them effectively. Here we summarise Lang's suggested heuristics and include some discussion, bearing in mind his qualifying remark that his parameters are not necessarily optimal - [8] p. 41.

- a_n - after normalisation of the data (if one already has the data and class information in advance) one can calculate the maximum radius of the classes. This is the distance from the notional centre of the class to the furthermost point of the class. Then we must simply ensure that a_n is set just larger than the largest class radius. ([8] p. 41)

- a_{cl} - Lang states that the clustering threshold should then be set smaller than a_n - he suggests that an appropriate value, which he obtains empirically, is 80% of a_n ([8] p. 41).

- a_r - the value Lang states in his description is 100, ([8] p. 39), but then he uses a value of 90 in his tests ([8] p. 56). 100 was chosen in order to give (rounded) integer link lengths for display purposes ([8] p. 39).

- b_c - this is a simple compensator for the a_r parameter ([8] p. 39), and appears to be generally equal to $\frac{1}{a_r}$.

- b_v - Lang states that he chose the focus update parameter as $b_v = 0.9$ ([8] p. 38). No further comment is given.

- b_a - the ageing parameter is given as typically 0.01 at first, but he uses 0.3 later in the tests. He later discusses a heuristic to choose this. Discussion of this heuristic follows.

Lang explains that the ageing parameter b_a can be chosen if we know some information about the input data - namely the longest gap between presentations of the same class. We obviously do not want the network to forget patterns between presentations, so if we know the longest likely time we have to wait we can set the ageing parameter accordingly. For a given neuron $\mathbf{x}_j$ we can calculate how long it will remain in the network if it is not updated, Lang calls this the 'temporal persistence' - t_j. We need to know the length, c_{sj}, of the shortest link connected to it. Then, (as in [8])

$$t_j = \frac{(a_r - c_{sj})}{b_a}. \tag{2.8}$$

In words this is simply finding the difference between the shortest link and the maximum link length, and dividing by the ageing amount to see how many iterations will pass before this link disappears. Rearranging gives

$$b_a = \frac{(a_r - c_{sj})}{t_j}, \tag{2.9}$$

and if we substitute in our longest time between presentations for the temporal persistence then we have a measure of the maximum value of b_a. Any higher and the network will forget classes between presentations ([8] p. 170).

In the next section we discuss some modifications to the algorithm and parameter choices before presenting a new algorithm with these modifications included.

2.2 Changes to Algorithm and Parameters

Here we discuss and lay out in detail the modifications we made to Lang's algorithm during this project. In the next section we present our complete algorithm.

2.2.1 Parameter Choice/Reduction

The parameters are an obvious target for improvement. Lang himself says that it should be possible to remove or combine some of them. We here suggest some changes - for evidence as to what constitute effective values, see the discussion and numerical testing in Section 3.

Changing a_r and b_c

In our opinion a confusing feature in the algorithm is the use of $a_r = 100$. The links at some stage all start as measuring Euclidean distance in the normalised $[0,1]^{dim}$ space (recalling that *dim* is the dimension of the input data), and Lang states that he uses $a_r = 100$, and scales up all the links lengths by this amount in order to be able to display link lengths as integers. If there is no need to display the links in this way then we may as well not bother scaling up the links by any amount. This would alter the link update equation (2.5) by removing the need for the a_r, and the link update simply becomes a restating of the Euclidean distance,

$$c_{fj} = \|\mathbf{x}_f - \mathbf{x}_j\|. \tag{2.10}$$

The neuron update equation (2.6) is also simplified, and there is no need for the b_c parameter to cancel out the a_r from (2.5). The equation becomes

$$\mathbf{x}_j \rightarrow \begin{cases} \mathbf{x}_j + c_{fj}(\mathbf{u} - \mathbf{x}_j) & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| < a_{cl}, \\ \mathbf{x}_j - c_{fj}(\mathbf{u} - \mathbf{x}_j) & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| > a_{cl}. \end{cases} \tag{2.11}$$

This leaves the a_r purely as a measure of maximum link length, and no longer tied up with display purposes. Lang's value of 100 becomes equivalent to 1, which still seems a sensible value to choose for now - any links longer than 1 in $[0,1]^{dim}$ space are likely to be overly long for any clustering purpose - at least for small dim, where the maximum link length is initially physically limited to $\sqrt{dim}$.

The role of b_c is revealed as nothing more than a scaling device, and is not (as implied by the description in Table 1) a parameter determining how drastically neurons are updated. From (2.11) we can see that the device controlling the extent to which neighbouring neurons are made closer to the input is purely the link length c_{fj}, so that close neurons are changed less than further ones (still within our clustering threshold a_{cl}). As a consequence there is a space for a new parameter - one that acts in the same way as b_v acts for the focus, but for the neighbourhood neurons. See Section 5.2 on ideas for further work and a more detailed discussion of this idea.

Choosing a_{cl}

It is not altogether clear what the exact role of the a_{cl} parameter is. It seems to function in a similar way to a_n in choosing which neurons are within a 'clustering zone' of some sort, and Lang's comment of choosing a_{cl} to be about 80% of a_n emphasises this. Our numerical tests in Section 3 attempt to justify a choice of this parameter, but it is worth asking the question here: Does this need to be a different parameter from a_n? Can we reduce the number of parameters by combining a_{cl} with a_n?

Choosing b_a

See Section 2.2.2 on changing the ageing method for a discussion of this parameter.

Choosing b_v

In early runs of our implementation of Lang's algorithm we observed our network behaving erratically and rapidly moving neurons around the space as inputs arrived. One reason for this was the high value of b_v at 0.9. This means that when a close focus has been identified (with a_n) it will immediately be moved 90% of the way towards the input vector. Setting b_v to a lower value produced a more gradual shifting of neurons and seemed to help reduce apparently erratic behaviour of the neurons during the algorithm. In practice, we used a value of around 0.2 from here onwards. See the numerical tests in Section 3 for evidence of this behaviour.

2.2.2 Ageing Method

Lang ages the links in the algorithm by adding a small amount, b_a, at every iteration. Thus, if they are never reset to their Euclidean distances they will eventually disappear when they reach the maximum link length a_r and are removed by the algorithm. This is done by addition but an alternative is to use a multiplicative approach. Using a multiplier larger than 1 (e.g. 1.01) we will have links that age exponentially. This will accelerate the removal of old links and perhaps improve performance when lots of links are present (e.g. in the presence of noise).

If this method is used then the link ageing process is

$$c_{ij} \to c_{ij} b_a. \tag{2.12}$$

The equations for temporal persistence used to set the b_a parameter will also change. A given neuron $\mathbf{x}_j$ with shortest link of length c_{sj} will disappear when the link's length becomes equal (or greater than) to a_r - that is it will happen at the earliest when

$$c_{sj} b_a^{t_j} = a_r, \tag{2.13}$$

recalling that t_j is the temporal persistence (i.e. the number of iterations until the neuron is removed). This gives

$$b_a = \left(\frac{a_r}{c_{sj}} \right)^{\frac{1}{t_j}}, \tag{2.14}$$

so that

$$t_j = \frac{\log \frac{a_r}{c_{sj}}}{\log b_a}. \tag{2.15}$$

2.2.3 Neuron Pushing

In step 6a of the algorithm (the neighbourhood update) we can see that the aim is to pull neurons within the clustering threshold (the 'near' ones) closer to the input, and to separate from neurons that are still connected but outside the clustering threshold (the 'far' ones). Lang's method achieves this by actually updating the vectors of both near and far neighbours, in proportion with the length of the link to the focus, but crucially changing the direction of the far ones (see equations (2.6) and (2.11)). This particular method has a clear disadvantage: the near neurons are updated a small amount (because their links are short), but the far ones are changed by large amounts (and moved away from the focus). Potentially very dissimilar neurons are therefore affected massively, and moved large distances across the space (see section 2.2.4 on neurons going 'out of bounds'). This has the effect of seriously distorting neurons that are connected to the current focus but not within the cluster threshold. The whole purpose of these neurons is to adjust themselves to become more like input vectors. If they are pushed around in this way then we lose any usefulness of their previous adjustments. All that is actually needed is an emphasis of the separation between the near and far neurons in order to aid the clustering process.

A simple remedy is to not change the far neighbours at all, and rely on the near neighbours' movement to create the separation. This has the advantage of only causing a change in one cluster of neurons (the one with the focus), and leaving all other neurons unchanged.

This would change the neuron update equation from (2.11) to

$$\mathbf{x}_j \rightarrow \begin{cases} \mathbf{x}_j + c_{fj}(\mathbf{u} - \mathbf{x}_j) & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| < a_{cl}, \\ \mathbf{x}_j & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| > a_{cl}. \end{cases} \tag{2.16}$$

However, we can still aid the clustering by pushing far neurons away in a different manner. After all, link length is a measure of similarity in some sense, despite the fact that it combines a measure of age and distance in the same number. We can simply extend the link length from the focus to these far neurons, thereby accomplishing a similar result without distorting their weight vectors themselves. Currently we reset the link lengths of all neighbours to be their Euclidean distance. In the new scheme we do this only for the near neighbours, while we perform an extra ageing iteration on the links to the far neighbours to emphasise their 'distance'. We can do this by changing the link update equation (2.10) to be

$$c_{fj} \to \begin{cases} \|\mathbf{x}_f - \mathbf{x}_j\| & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| < a_{cl} \\ c_{fj} b_a & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| > a_{cl} \end{cases} \tag{2.17}$$

Clearly there are many possible algorithmic variations, and there are many that would be interesting to investigate in the future. See Section 5 for more discussion of such ideas.

2.2.4 Neurons 'Out Of Bounds'

As a result of the phenomenon of 'neuron pushing' discussed in section 2.2.3 it is entirely possible that neurons are adjusted to have values outside of the $[0, 1]$ range for each dimension. This is clearly undesirable since they cannot represent any of the incoming normalised data. A preliminary fix to this involves testing for any neuron values outside the range $[0, 1]$ and removing them (and their links). The result of this is that if classes migrate out of bounds then they are removed and a new class is created next time an input arrives. We found it necessary to have this change in place before any useful results could be produced from the original algorithm, but after the changes we implement in Section 2.2.3 this situation should never occur. However we leave the code in as a safeguard.

2.2.5 Using Old Classes

When an input arrives that justifies the creation of a new group of neurons we assign a new and unused class number to it (and the new neurons). If however the input is an instance of a class that the network has 'forgotten' (either it has aged enough to be deleted, or been 'pushed' out of bounds) then it would be better to restart the old class, rather than create a new one. Otherwise the data from one input class will be spread over (at least) two algorithm classes. If we can achieve this effect then even a poorly functioning PSOM (which keeps forgetting classes) can still classify correctly. To do this we need to somehow keep track of all the classes the algorithm is using, and check if any old ones can be reused, before creating a brand new one.

There are a number of ways to implement this: we could store the average weight vector of all neurons in the class, thereby representing the current 'position' of the class, or we could store the mean position of all input vectors that have been assigned to the class, which would result in an overall general position of the vectors associated with each class.

We chose to use the second of these options, in order to incorporate the most information about the class. This involves creating several new class variables in order to keep track of the classes over the course of the algorithm: the overall mean, standard deviation, number of entries, and the current state of existence of the class; present, forgotten or combined with another class. After each classification of a signal each one of these variables is updated. In order to calculate the standard deviation on the fly we record the total sum-squared of each class as well. For the existence state use a value of 0 if the class still exists in the network, -1 if the class has been forgotten (see step 9 of the algorithm in Section 2.1.2), and if the last remaining neuron in a class has been combined with a different class (step 6a - the neighbourhood update) we record that class number.

These class variables (implemented as `classmean, classstd, classno, classexit,` and `classsq`) provide a useful way to keep track of the clusters recognised in the running of the algorithm and also give an easy way to test the final results against known classes in the input data.

We implement these re-occurring classes by checking through the `classmean`s of every 'old' (i.e. `classexit` $\neq 0$) class and comparing them with the current input vector. We find the closest old class and check if the distance to it is less than the maximum standard deviation in it. If so, then we assume that we are seeing a re-occurrence and use the old class number. If no old class matches the input closely enough we will go ahead and create a new class.

Again there are decisions to be made as to exactly how tolerant to be with these re-occurrences.

2.3 New Algorithm

Here we summarise our new implementation of the algorithm. See the previous section for the motivation for each of the changes we make. Figure 4 shows the algorithm with updated steps to reuse old classes (see section 2.2.5). Also refer to Appendix C for the pseudo-code for this algorithm. The algorithm as presented represents the best of current modifications - but there remain further options to test. These are detailed in Section 5. Any steps that are identical to the original algorithm are discussed in Section 2.1. The numbering of the steps is also identical to the original version for comparison purposes.

2.3.1 Parameters

Table 2 gives a suggested set of parameter values, which we have reduced in both number and complexity. Note than a_{cl} is now identical to a_n, b_c has been removed, and the other parameters have clear heuristics to choose them (see Section 2.2.1 on choosing parameters and Section 3 on numerical testing). We hope that a reduced number of parameters will make the PSOM algorithm more transparent and more easily adjustable.

Parameter	**Name**	**Purpose**	**Values**
a_n	Cluster Threshold	The focus-input distance threshold after which a new group of neurons will be created.	$[0, 1]$ - we use 0.14
a_r	Maximum Link Length	Ageing threshold, links longer than this will be removed	≈ 1
b_a	Link Ageing Parameter	How much links are aged each iteration (multiplicative)	≈ 1 but > 1
b_v	Focus Update Parameter	How much to make the focus like the input	$[0, 1]$ - we use 0.2

Table 2: Parameters used in the modified PSOM

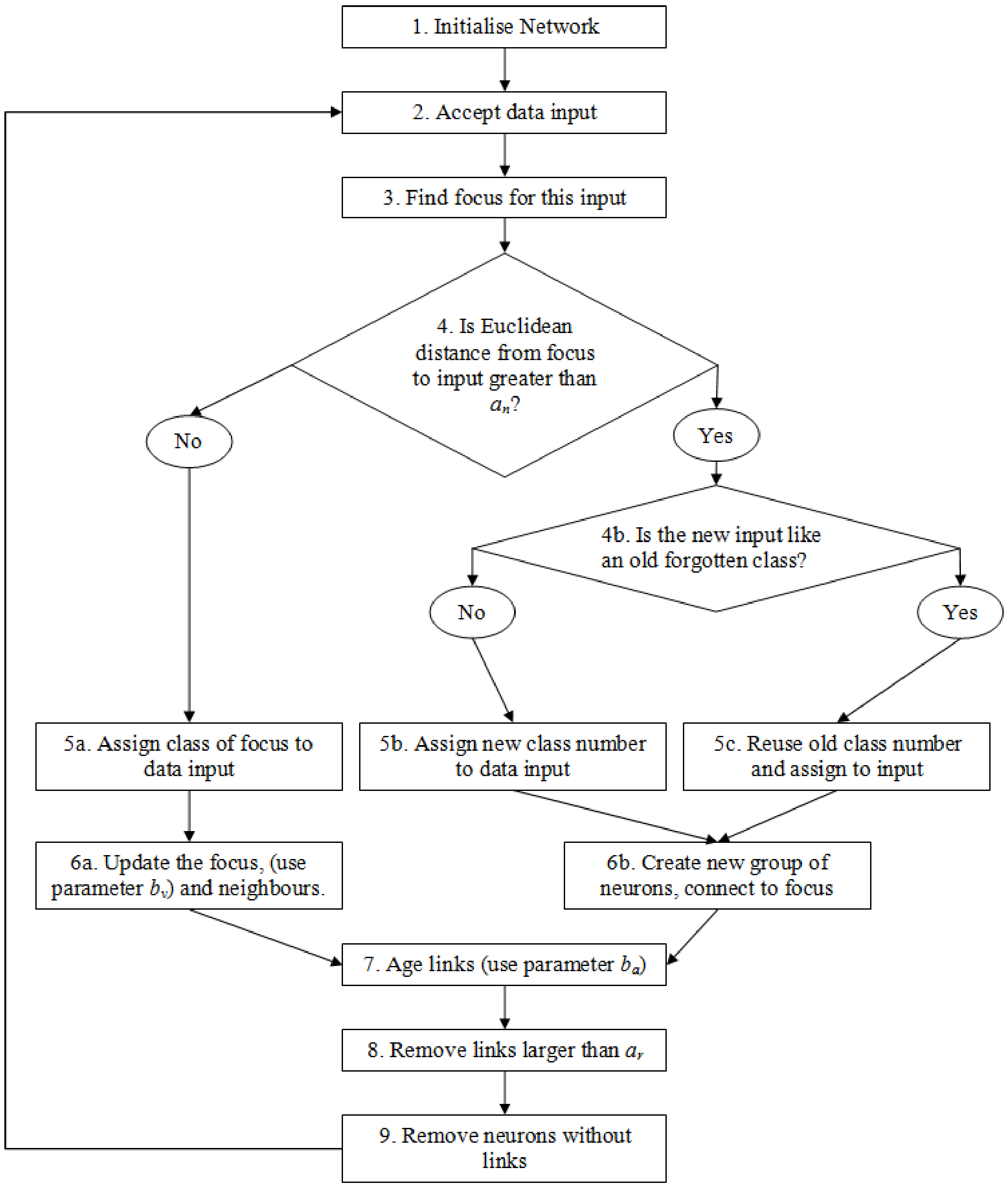

1. Initialise Network
2. Accept data input
3. Find focus for this input
4. Is Euclidean distance from focus to input greater than a_n?
No
Yes
4b. Is the new input like an old forgotten class?
No
Yes
5a. Assign class of focus to data input
5b. Assign new class number to data input
5c. Reuse old class number and assign to input
6a. Update the focus, (use parameter b_v) and neighbours.
6b. Create new group of neurons, connect to focus
7. Age links (use parameter b_a)
8. Remove links larger than a_r
9. Remove neurons without links

Figure 4: Flowchart of modified PSOM Algorithm

2.3.2 Step-by-Step - Modifications

5a) & 6a) Focus and Neighbourhood Update - $d < a_n$

When $d < a_n$ the focus is updated in the same way as before,

$$\mathbf{x}_f \rightarrow \mathbf{x}_f + b_v(\mathbf{u} - \mathbf{x}_f). \tag{2.18}$$

but the b_v parameter is set to about 0.2, a much lower value than before (see 3.3.1). The neighbourhood is updated by first calculating which neighbours are 'near' and 'far' using the Euclidean distance and the a_n threshold (no a_{cl} this time). We update the links of the near neighbours, setting them equal to their Euclidean distance from the focus, while the far neighbours' links are left as they were but aged by one more iteration. From (2.17), we get

$$c_{fj} \rightarrow \begin{cases} \|\mathbf{x}_f - \mathbf{x}_j\| & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| < a_n, \\ c_{fj} b_a & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| > a_n. \end{cases} \tag{2.19}$$

Having updated the link lengths we now use them to update the neuron vectors. Near neighbouring neurons are drawn closer, in proportion to their link length, while far neighbours are left alone. From (2.16) we get

$$\mathbf{x}_j \rightarrow \begin{cases} \mathbf{x}_j + c_{fj}(\mathbf{u} - \mathbf{x}_j) & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| < a_{cl}, \\ \mathbf{x}_j & \text{if } \|\mathbf{x}_f - \mathbf{x}_j\| > a_{cl}. \end{cases} \tag{2.20}$$

If a neuron with a different class number is identified as a near neighbour its class number will be reassigned to that of the focus.

Figure 5 shows the network finding a focus close to an example input, and performing the neighbourhood update. Compare with Figure 2 to see the differences.

Classification of the input data is given as before by returning the class number of the focus.

4b), 5b), 5c) and 6b) - New group of neurons - $d \geq a_n$

Having identified an input a long way from any neuron, we first compare the input to a list of old classes that have been forgotten by the algorithm. If one of these classes is sufficiently close then we will reuse that class number, rather than generating a new

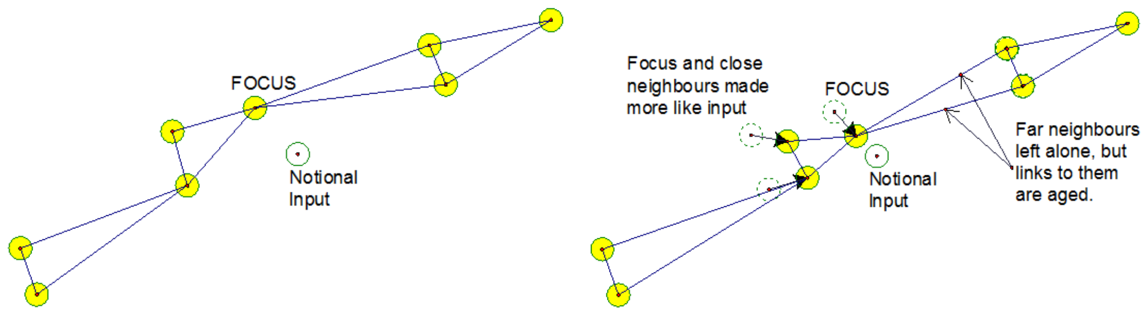

FOCUS
Notional Input
Focus and close neighbours made more like input
FOCUS
Notional Input
Far neighbours left alone, but links to them are aged.

Figure 5: The left diagram shows the initial network with a notional input. The right shows the update process (dashed circles are the original positions of the neurons).

one. The current algorithm decides this by looking at the Euclidean distance between the class means of the old classes and the input vector. If the distance to the closest class is less than the maximum standard deviation (each dimension's standard deviation is recorded) within the dimensions of that class then we decide that the input signal is a re-occurrence of the old class, and assign that class number to the input. We then create a new group of neurons as before, but allocate them this reused class number.

If the shortest distance to the old classes is not within the standard deviation then we assume that we are seeing a new class, and create a new group of neurons as before, incrementing the class number to create a new class.

7) Age Links

This is now done multiplicatively, so all links are made older using

$$c_{ij} \rightarrow c_{ij} b_a \quad \forall i, j < n \tag{2.21}$$

3 Numerical Experiments

In this section we present results of our systematic numerical experiments on the algorithms, using a variety of data.

3.1 Test Data Sets

We first explain the data we use to test the algorithm. It is important to demonstrate correct functioning of the algorithm on several types of data. For example, one of the key features of the PSOM that give it an advantage over the SOM and other static neural networks is its ability to deal with new classes of data, and to forget obsolete ones. Accordingly we describe five classes of data we would be like to be able to classify, similar to Lang's tests in [8].

1) Regular, frequently occurring data: obviously the algorithm must be able to handle such 'ordinary' data.

2) Stopping data - similar to class 1, but which stops occurring at some point. We want the network to be able to eventually forget about this class.

3) Infrequent but still regular data - less frequent then class 1.

4) Irregular data, which occurs at random intervals.

5) Late starting data - again similar to class 1, but this time the class appears at some point mid way through the algorithm.

In addition to these five we will use a sixth class for adding noise to the data set to test if the algorithm still functions in noisy environments. Table 3 shows how many inputs we will use from each class.

For visualisation purposes we generate the test data in three dimensions, but any other number of dimensions will still work. For each class we pick a random point in $[0, 1]^3$, and generate random vectors of the correct dimension using the `randn` command. This creates normally distributed random data with mean zero and standard deviation one. We scale the points by multiplying by a factor (we use 0.03 here) and then add

Class Number	**Name**	**Occurrences**
1	Regular	500
2	Stopping	200
3	Infrequent	100
4	Irregular	100
5	Late Starting	100
6	Noise	0
	TOTAL	**1000**

Table 3: Amount of signals in each class for test data set.

the centre point for the class. This creates a normally distributed cluster of points at a random location with standard deviation 0.03.

Having done this for each class we then allocate the order of the data. To acheive this we create a $500 \times 15)$ matrix, and each set of 3 columns will contain entries for one of the 5 classes. We allocate the regular class (1) to every row in the first three columns of the matrix while the stopping class will occupy the first 200 rows of the second three columns. The infrequent data we allocate to every 5th row in the third set of three columns, while the irregular data is assigned to random rows throughout the fourth set of three columns. Finally the late starting class data is written in the last 100 rows of the last 3 columns. If the matrix is read from left to right in collections of threes across each row, and zeros are ignored, we have the required order of the data. By reshaping the matrix and removing all zero rows we achieve a data set with the classes occurring as desired. Figure 6 shows the incoming order of the data.

Each time the code is run the classes will be located in a different position in $[0, 1]^3$ space, and in order to standardise part of the tests we chose a particular set of data as a benchmark data set. Figure 7 shows a 3-D visualisation of the benchmark data, and from this we can judge approximately the maximum radius of the clusters to be about 0.14 and so the parameter a_n is set at this value by default when running on these data sets. It can also be seen that this particular data set has clearly-defined clusters that are well separated in the input space. Naturally there is no guarantee that randomly generated data sets will always have this property.

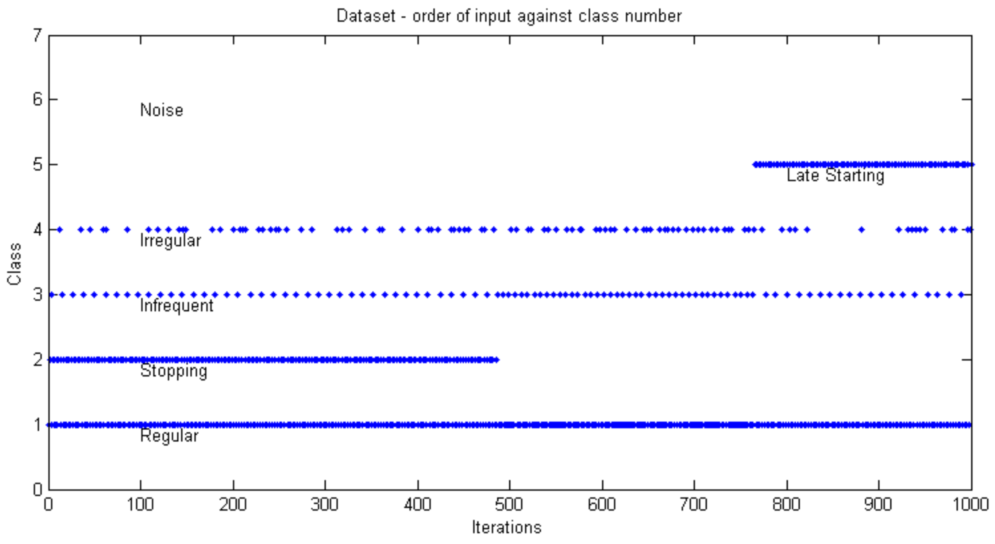


Figure 6: The order data arrives from the classes, in this case there is no noise added

3.2 Measuring Performance

There are several ways we can measure the performance of the algorithm and we detail them here.

At every iteration of the algorithm we calculate what we will call the *recognition error*. This is the Euclidean distance between the input vector and the focus - denoted as d in Section 2.1.2. It provides a useful measure of how successfully the current neurons represent the incoming data. A large recognition error suggests that the input signal is unlike anything in the current network.

To gain insight into the efficiency of the process we record the number of links and the number of nodes in the network. If these remain constant over a long period of time it suggests that the network is correctly representing the data (i.e. neither creating new groups nor forgetting old ones.

We can also record the number of classes the network is using to classify the data. A higher than expected number of classes is an indication that the network is subdividing incoming clusters and mis-labelling them as separate patterns.

When using the test data set we can also directly check the success of the classification process, because we know the correct class of every input signal. A simple scoring

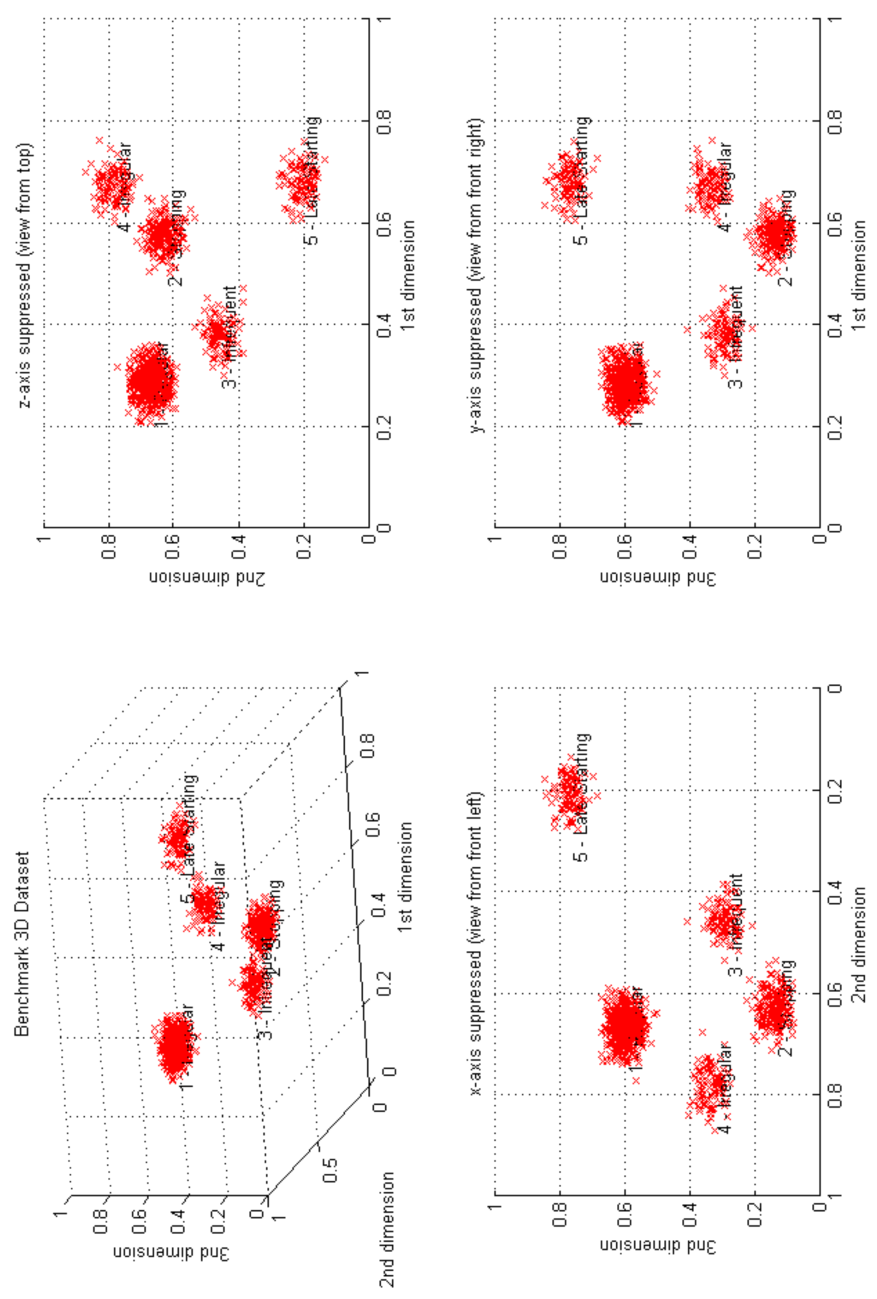


Figure 7: Benchmark data set

mechanism is to count how many inputs from each class are mis-classified. In this situation we are counting the false negatives, i.e. how many signals are assigned to the wrong class. False positives are when classes are assigned extra data that does not belong there. In this situation (with no noise present) the most helpful statistic is the false-negatives, which we will refer to as ‘missed’ data.

Notice that we do not necessarily need to label the algorithm classes with the same numbers as the input classes (as these are essentially arbitrary), so we choose the algorithm class that has the most data points from an input class to represent it. We then record how many data points from this class have *not* been allocated the same label. This provides a simple count of mis-classifications. If one of the algorithm classes is found to represent more than one of the input classes then the algorithm is clearly not working properly, as it has classified the majority of two separate input classes in the same algorithm class. We score this situation by looking for the next largest number of data points in an algorithm class and using that class to judge how many are missed. This will drastically increase the count of missed data points.

In the main section of the results on the test data we will mainly use the data from the mis-classifications, but graphs of the other measurements can be found in Appendix A.

A table detailing the location of the data from each input class, and where it has been classified, is a useful way to assess the performance of the algorithm. Table 4 shows shows an ideal result for a classification scheme. Each column represents an input class, while the rows give the algorithm classes. In the ideal example, all data in each input class has been allocated to a unique algorithm class, indicating a successful recognition of a distinct pattern. False negatives will show up when a column has more than one non-zero entry, while false positives occur when a particular row has more than one non-zero entry.

The first row is empty because it represents the initial class from the random initialisation of the network, which ages away over the course of the algorithm unless by chance data occurs nearby, when it will adjust to represent one of the clusters. The last column is empty because there were no noise elements added in this example.

	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**	**Noise Class**
1	0	0	0	0	0	0
2	500	0	0	0	0	0
3	0	200	0	0	0	0
4	0	0	100	0	0	0
5	0	0	0	100	0	0
6	0	0	0	0	100	0

Table 4: Comparison of classifications

3.3 Tests

In this section we present the results of varying the parameters on the two different versions of the algorithm. We use our implementation of Lang's algorithm (discussed in Section 2.1) and the new algorithm described in Section 2.3.

Unless otherwise stated we generate, for each test, a random data set (see Section 3.1) and then run the algorithm on this data repeatedly while varying the chosen parameter in small steps. We repeat this process 50 times (using a different randomly generated data set each time). In the figures Lang's algorithm appears first, the new algorithm is second, and the axes are set similarly for comparison purposes. The mean results over the 50 trials are plotted, with broken lines indicating $\pm$ one and two standard deviations.

Table 5 lists the values of all parameters used by default in these tests (where necessary), while changing the parameter in question. Parameter b_a takes the appropriate value depending on the ageing system used.

Parameter	**Value**
a_n	0.14
a_{cl}	80% of a_n, i.e. 0.112
a_r	1
b_c	1
b_a	0.01 or 1.01
b_v	0.2

Table 5: Parameters in tests

3.3.1 Parameter b_v

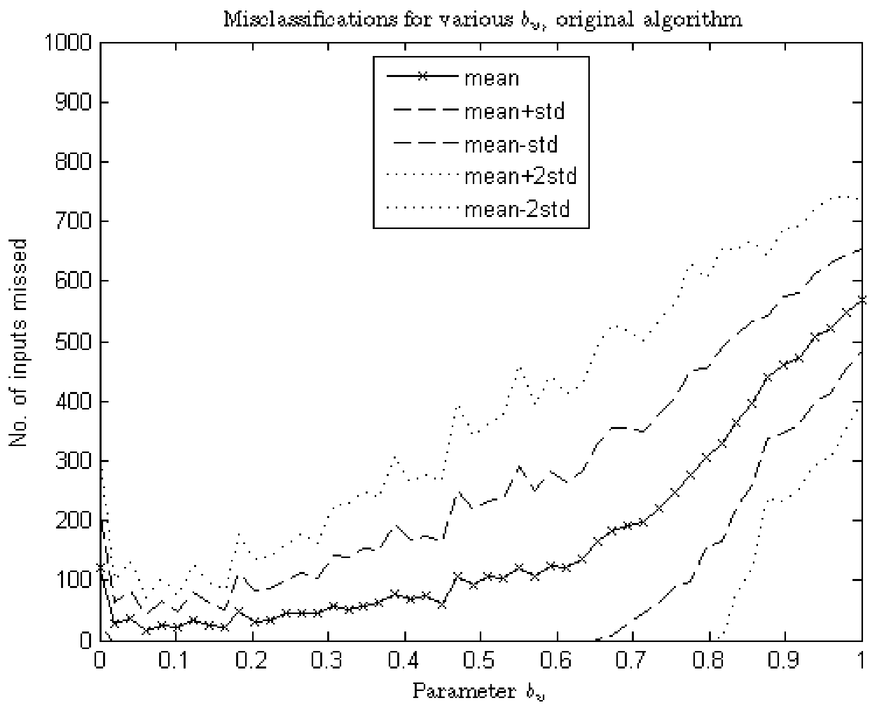
Parameter b_v controls how much the focus is adjusted to become like the input (if it is within the cluster threshold). Lang chooses a value of 0.9 but we demonstrate here that this is an ineffective choice. Figure 8 shows how the number of data points missed (false negatives) changes with b_v.

It is clear that Lang's algorithm works better with a much lower value of b_v, but there is still a significant amount of data missed, and as b_v increases the spread of data shows that the algorithm performs somewhat unpredictably. The new algorithm has a consistently lower number of mis-classifications, and does not seem to be very sensitive to the choice of b_v, although lower values do appear to perform better.

It makes intuitive sense that a lower value of b_v gives better results by adjusting the network more gradually, and allowing it to accommodate data over time, rather than drastically altering at every input. Beyond this observation, the main conclusion is that the new version of the algorithm performs significantly better.

It is useful to compare the classification tables from Lang's algorithm at different b_v values. Tables 6 and 7 show these when b_v is 0.2 and 0.9 respectively. At a value of 0.2, the algorithm recognises most classes completely accurately, only missing one data point from class 2. At 0.9 however classes 3 and 4 are spread over many different algorithm classes, while class 1 is split in two. The poor performance in classes 3 and 4 - the infrequent and irregular classes - demonstrate that the errors are arising in data which is less 'regular'. This seems to be because the clustering neurons have more time to be 'disturbed' by the working of the algorithm before they see a new signal. The problem appears to be solved by the new algorithm, which successfully reproduces the ideal result in Table 4.

Notice that all the errors shown here are false negatives and that no false positives occur. It seems that the only time false positives occur are when classes in the data set are actually overlapping, and so the algorithm labels both two classes with the same number. This is a problem with the data, and not the algorithm. For this reason we use the false negatives as the main statistic for judging success.



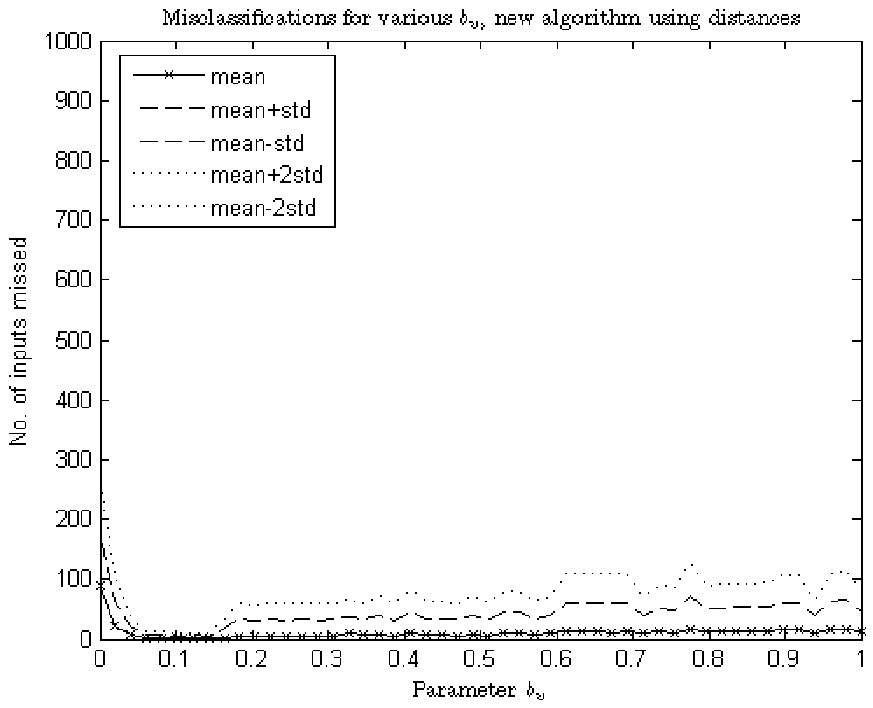


Figure 8: Performance of the algorithms as b_v is changed. The top graph is Lang's algorithm and the bottom graph is the new algorithm

	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**	**Noise**
1	0	0	0	0	0	0
2	500	0	0	0	0	0
3	0	1	0	0	0	0
4	0	0	100	0	0	0
5	0	199	0	0	0	0
6	0	0	0	100	0	0
7	0	0	0	0	100	0

Table 6: Lang's algorithm with $b_v = 0.2$

	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**	**Noise**
1	0	0	0	0	0	0
2	116	0	0	0	0	0
3	0	200	0	0	0	0
4	0	0	2	0	0	0
5	0	0	3	0	0	0
6	0	0	0	5	0	0
7	0	0	2	0	0	0
8	0	0	0	1	0	0
9	0	0	0	1	0	0
10	0	0	1	0	0	0
11	0	0	0	93	0	0
12	0	0	1	0	0	0
13	0	0	1	0	0	0
14	0	0	2	0	0	0
15	0	0	1	0	0	0
16	0	0	69	0	0	0
17	384	0	0	0	0	0
18	0	0	0	0	100	0
19	0	0	18	0	0	0

Table 7: Lang's algorithm with $b_v = 0.9$

3.3.2 Parameter a_n

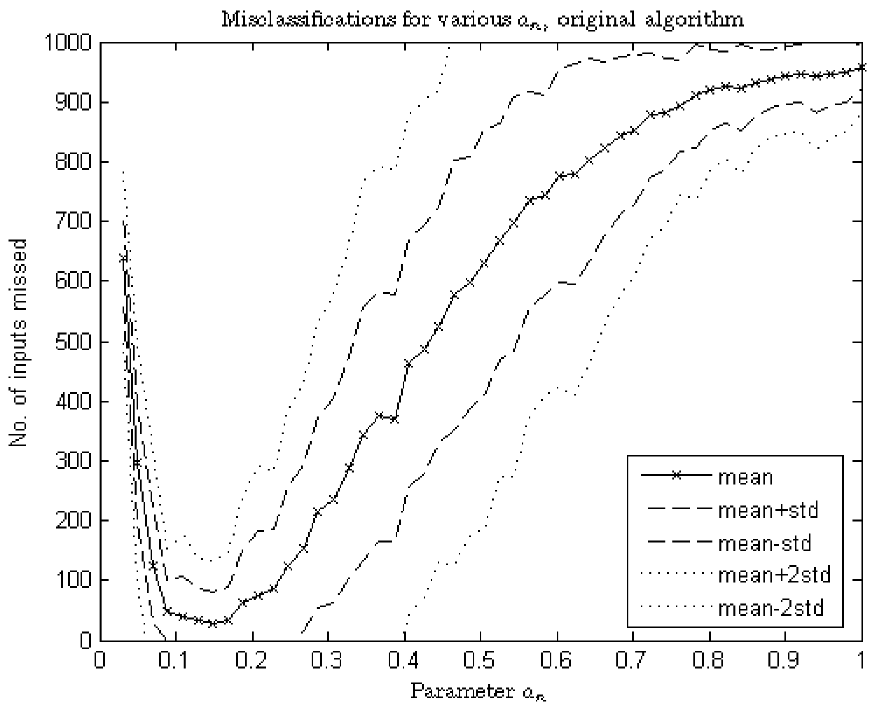
The parameter a_n is crucial to define how big a 'cluster' can be, and has a big effect on the results, as can be seen in Figure 9. When a_n is around the value suggested by the data (about 0.14) it is obvious that both versions of the algorithm do fairly well, although again there is a noticeable difference in the consistency of the results around this point. Lang's algorithm again has a larger standard deviation, compared to the new version. Both algorithms classify the data badly as a_n moves away from the indicated value; if it is too small then clusters are too small and single input classes are classified as several different algorithm classes. If a_n is too large then algorithm classes encompass wider areas and it becomes increasingly likely that algorithm classes will overlap two or more input classes (and hence mis-classify the data).

It is clear from these results that choosing an effective value of a_n is a very important factor in any useful operation of this algorithm. If the characteristics of the data are not known beforehand this poses a problem. A potentially very useful direction for further work would be to investigate how to allow the algorithm to adjust a_n dynamically as the algorithm progresses. If a test data set is available then an approximate a_n can be calculated (see Section 2.1.3), in this case we calculate the maximum radius of the classes at around 0.13, so a suitable value of a_n is just larger than this at 0.14.

See Appendix A for graphs showing the other statistics for this parameter. Computational time (Figure 28) is much the same for both versions and does not alter across the range of a_n. The recognition error is noticeably worse in Lang's algorithm, and tends to get higher as a_n increases. The other graphs (Figures 20 and 21) showcase the effect of a very small a_n when the algorithm over-classifies data and creates far too many classes.

3.3.3 Parameter a_{cl}

As discussed earlier in Section 2.2.1 it is not altogether clear why a_{cl} is a necessary extra parameter. The results in Figure 10 demonstrate that for each version of the algorithm, once $a_{cl} > a_n$ it has a relatively small effect on the performance of the algorithm. In each case the best performance is when $a_{cl} \approx a_n$. This supports the earlier suggestion that



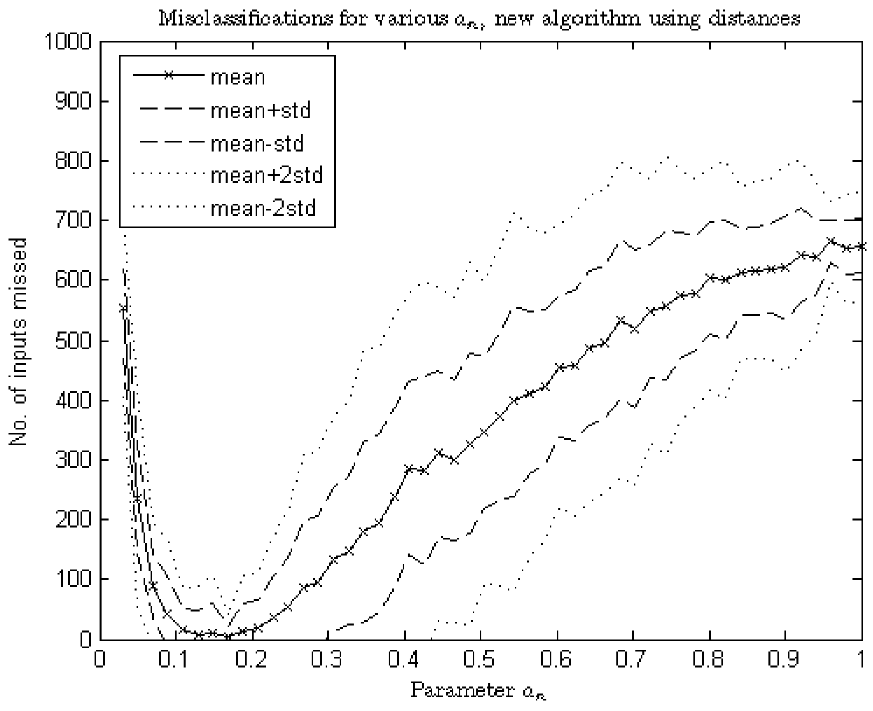
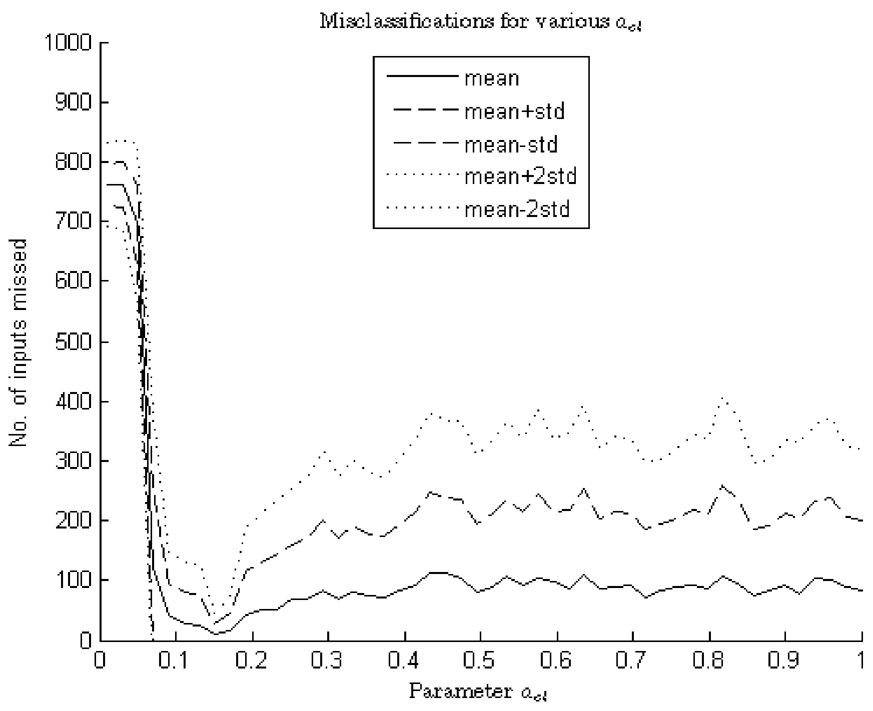


Figure 9: Performance of the algorithms as a_n is changed. The top graph is Lang's algorithm and the bottom graph is the new algorithm.



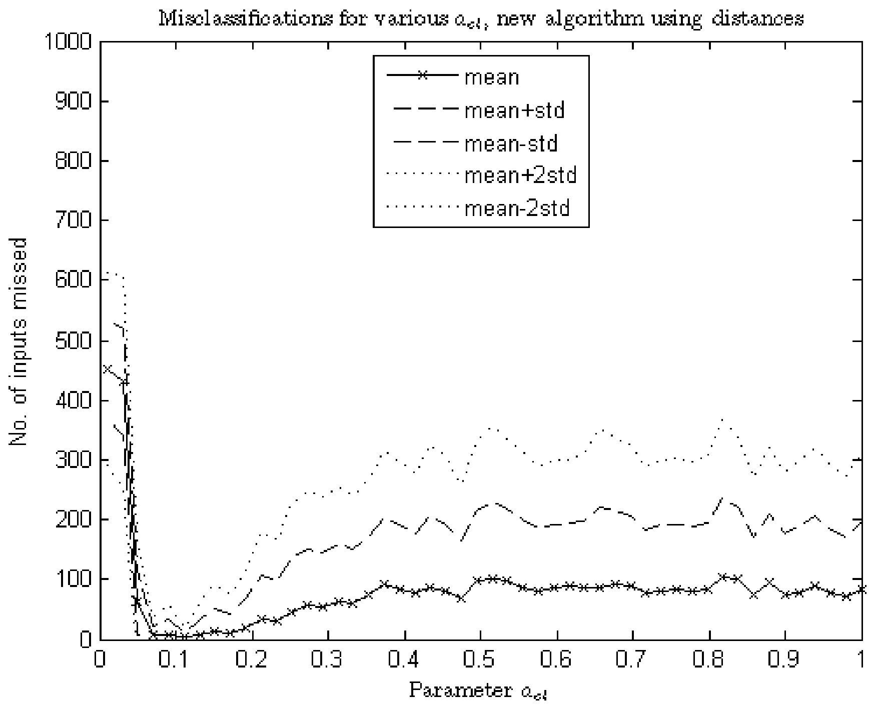


Figure 10: Performance of the algorithms as a_{cl} is changed. The top graph is Lang's algorithm and the bottom graph is the new algorithm

the two parameters could be combined. The other graphs in Appendix A also support this, and show that this parameter has very little effect on recognition error.

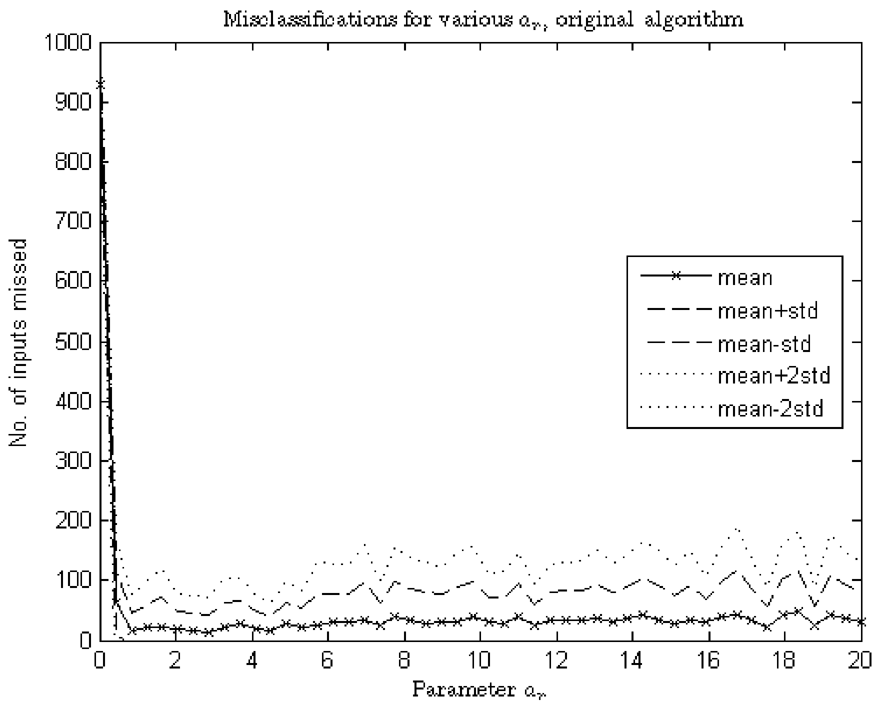
3.3.4 Parameter a_r

Parameter a_r represents the maximum length of a link before it is deleted. As all link lengths in Lang's algorithm were scaled by this amount, and we set $b_c = \frac{1}{a_r}$, this has little effect on the algorithm, other than the fact that it clearly must be above zero. If the parameter gets too large then it will simply mean that ageing links has less effect, and classes are forgotten less easily. This does not seem to affect the classification errors - as the graphs in Figure 11 demonstrate. The only noticeable fact is that the new algorithm is performing consistently better than Lang's. The two sets of graphs from these tests in Appendix A (Figures 24 and 25) show that a_r appears to have no noticeable effect on the new algorithm (perhaps multiplicative ageing means that far higher values of a_r need to be tested). For high values of a_r Lang's algorithm appears to never forget class 2 (the stopping class), as might be expected - this can be seen in the graph of the number of classes present during the algorithm .

3.3.5 Parameter b_a

The graph for Lang's algorithm in Figure 12 shows sudden changes in behaviour. These occur at $b_a \approx 0.24$, and again at $b_a \approx 0.32$. The value normally used is 0.01, so at these points we are a considerable distance from the standard value, meaning that links are aged very quickly. The changes are probably due to the ageing process occurring so quickly that classes are forgotten before their next presentation.

The number of iterations between classes is at least 2, and can be as high as 4 iterations in our test data. From our tests we calculate that the average link length within a class during our algorithms is about 0.04. Using this value for the shortest link connected to a node c_{sj} in equation (2.8), and a value of 0.24 for b_a gives us a temporal persistence of around 4 iterations. So it is at exactly this point that we'd expect to see a poor performance, since there are 4 classes present during most of the algorithm. Of course, since some of the data is irregular, some classes are seen more often than that and so will still be remembered. The next jump in the graph occurs at $b_a \approx 0.32$, which



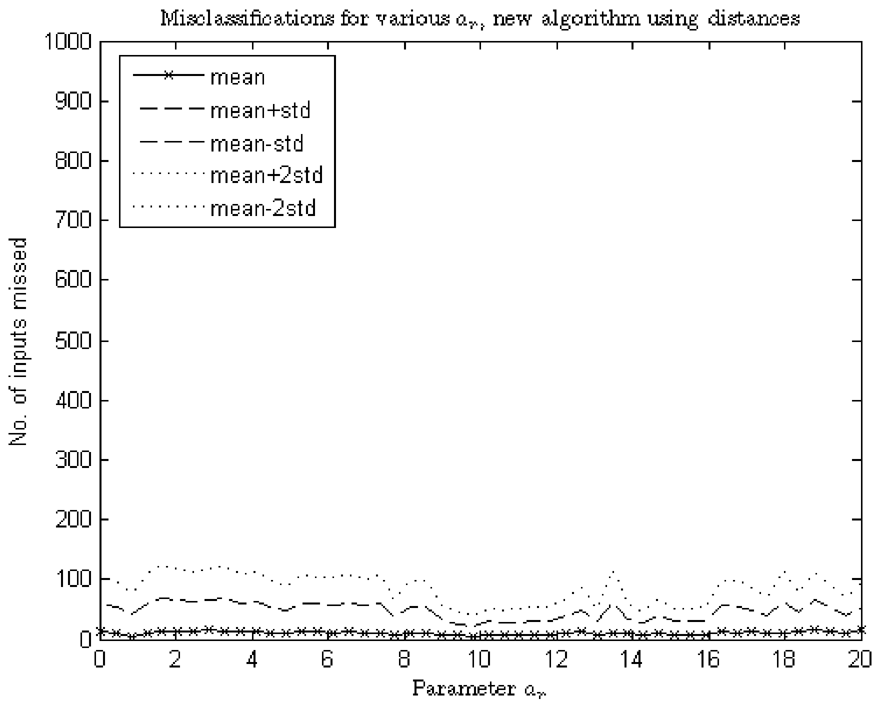
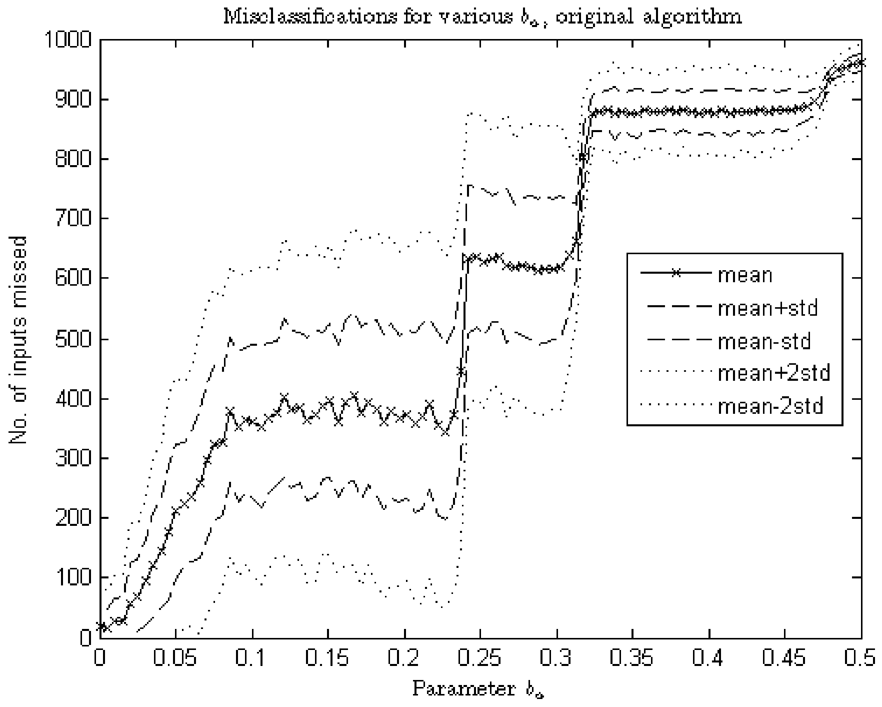


Figure 11: Performance of the algorithms as a_r is changed. The top graph is Lang's algorithm and the bottom graph is the new algorithm



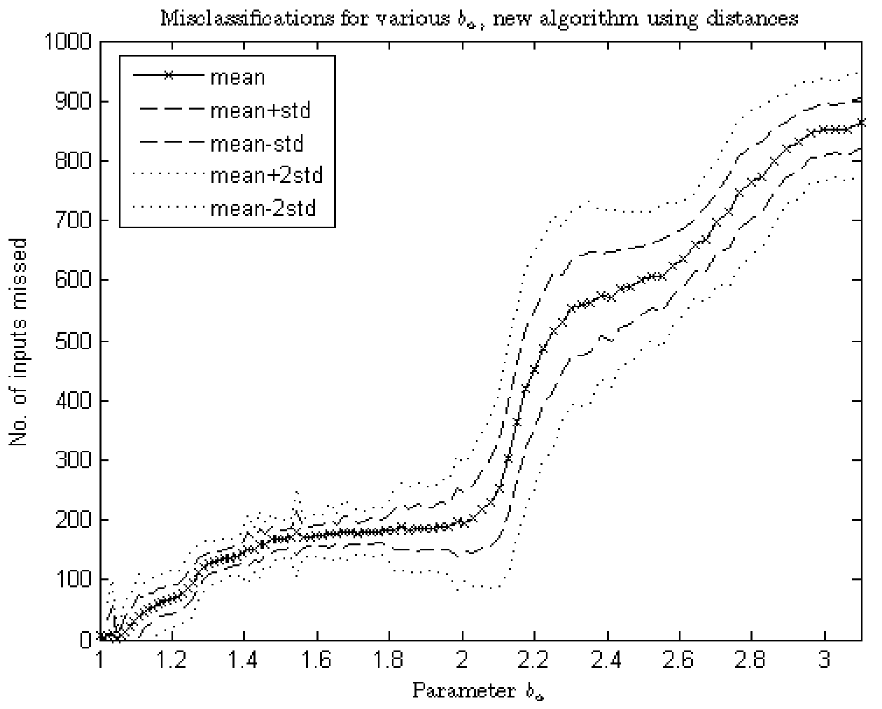


Figure 12: Performance of the algorithms as b_a is changed. The top graph is Lang's algorithm and the bottom graph is the new algorithm. Note that different x-axis scales are used since the operation of the ageing process has changed.

corresponds to a temporal persistence of only 3 iterations - which confirms this as the explanation for the abrupt changes in performance.

The new algorithm has less catastrophic changes, but still a sharp rise in missed data occurs at about b_a=2.2, which corresponds (using equation (2.15)) to a temporal persistence of almost exactly 4 iterations once again. In this case the 3 iteration threshold occurs at about $b_a = 2.9$. The degradation occurs slightly before and continues after these values, because link lengths are not always exactly the same value, and only on average around 0.04.

3.3.6 Noise

Figure 13 shows how the algorithms slow down when noise is added to the data set, while Figure 14 shows the missed classifications (false-negatives). The x-axis shows the number of noisy entries that we added, up to a maximum of 27000 extra entries, beyond the original 1000 in the test data set. For each experiment we generated one test data set for both algorithms to use, and then added more noise on successive runs. We repeated this 10 times on different data sets to get an average of the results as shown in the diagrams. Both algorithms cope fairly well with noise at these high levels, although the new algorithm appears to perform better again. The new algorithm is noticeably slower, but both algorithms demonstrate a strong linear relationship between the amount of noise and the computational time. The standard deviations are not shown on the time graph because they are so small (typically a only few seconds). The times were produced on a Dell Inspiron 530s, 3.0 GHz Intel Core2 Duo CPU, with 3326MB RAM, running Vista.

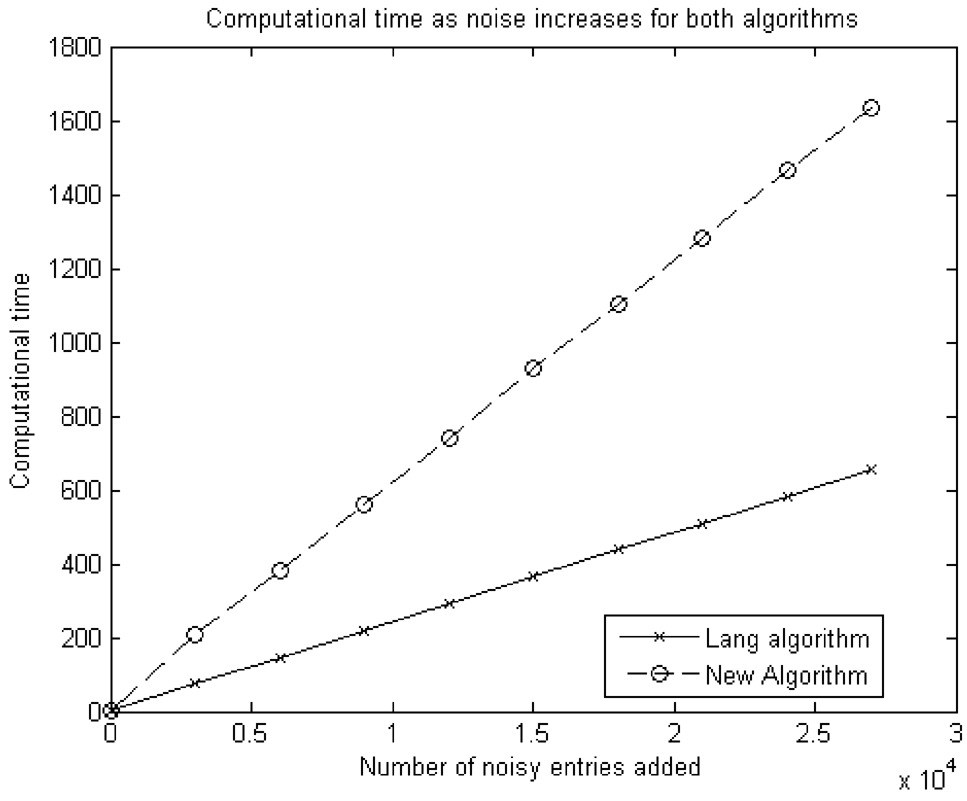
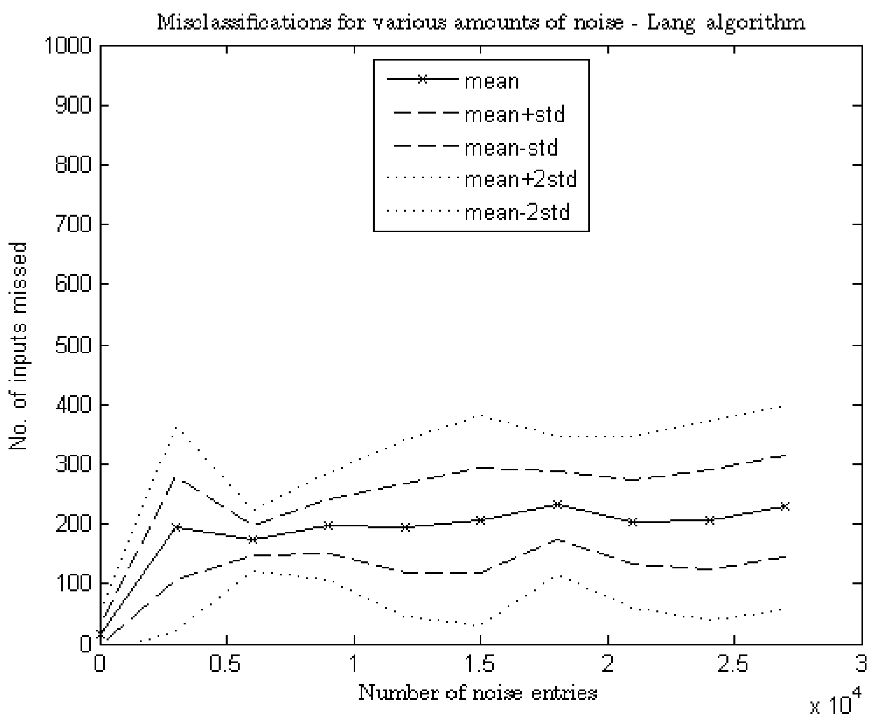


Figure 13: Computational time (in seconds) of the algorithms as the amount of noise is increased from 0 to about 30000



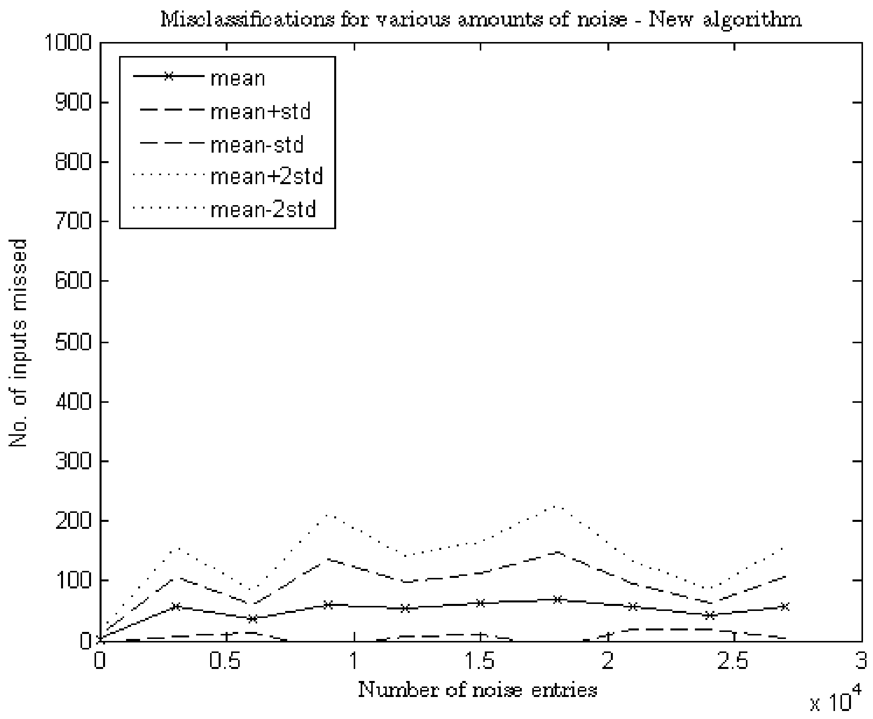


Figure 14: Performance of the algorithms as the amount of noise is increased from 0 to about 30000

4 Application to THALES Data

4.1 Data Format

THALES supplied us with a data set of 42156 signals, each of which had 5 variables, labelled RF (Frequency), PW (Pulse Width), AMP (Amplitude), BRG (Bearing) and TOA (Time of arrival). Figure 15 is a representation of all the data, note the 3D plot in the lower left, which shows the clusters in the three dimensions we will ultimately use.

The last dimension of the data - the time of arrival of the signal - is, in its current format, un-usable as an input to the PSOM, since the TOA is not a constant value for a certain emitter. The AMP dimension is also problematic - the incoming signals are often from rotating radars, so they are typically picked up faintly at first, more strongly as they aim directly at us, and then lose strength as they turn away. As a result the AMP value for each emitter changes in a sinusoidal manner, and is not directly useful for the PSOM algorithm, since again it is not constant for each emitter.

That said, both of these dimensions of data can - and should - be used to further classify and separate out the emitters, if additions are made to the classification code beyond the pure PSOM function. In this project we focus on what can be achieved before any such modifications are made, but implementing them to improve the performance would be a natural next step.

Using the RF, PW and BRG dimensions of the data we import the data as a 42156×3 matrix. The first step is to normalise it. Without any knowledge of the context of these dimensions we can proceed to normalise all the dimensions separately and identically weighted by simply calculating the range of values in each dimension, dividing by this range and subtracting the minimum value. This puts all data into values into the range $[0, 1]$. Obviously if we wished one dimension to have more effect than the others then we can adjust this process, see Section 5.3 for more discussion on this.

With the data normalised it just remains to set the parameters for the algorithm and let it run. If nothing is known about the data structure or cluster sizes then estimates will have to be used at first, and the parameters tuned on subsequent tests.

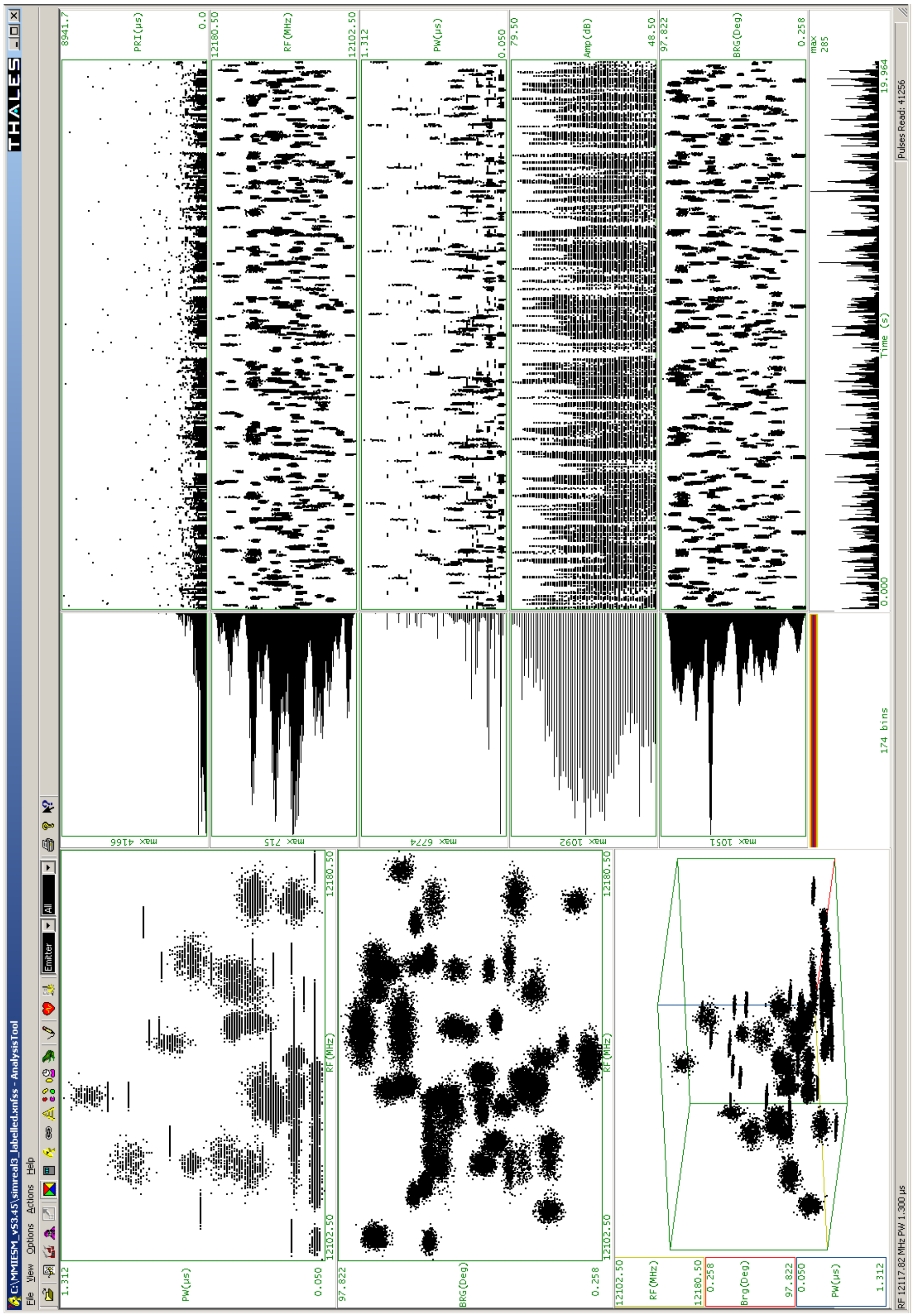


Figure 15: THALES simulated data set, used with permission of THALES Aerospace.

4.2 Results and Analysis

Applying the algorithm to the THALES data, the main output we get is the list of classes that the data signals have been assigned. This is the classification that the algorithm has provided. More usefully for judging the results we have the list of `classmean`s and the other class statistics (see Section 2.2.5) which identify where each algorithm class is located in the input space. After an initial run on the data we sent our results to THALES for them to compare with the actual emitter data they held. This was compared by means of a spreadsheet, and they tried to pair each algorithm class with an emitter. As a rough method of assessing the success they assigned each of the algorithm classes a description. All quotes in this section are from e-mail communication with THALES [9] .

In a successful example the mean values of the data assigned to a certain algorithm class were exactly the same as an emitter in their input set, and the number of signals there was also identical. More commonly the mean values of a class in the algorithm were very close to those of a certain emitter, with a slight discrepancy in the number of signals there. This occurs when a few data signals are misclassified into this class, but not enough to make the class un-representative of the emitter. In either of these cases the algorithm class is labelled **'good'** and judged to be *'a good representation of the emitter, without significant elements of other emitters.'*

Sometimes the algorithm mistakenly incorporates significant numbers of signals from two or more emitters into the same algorithm class. If the main emitter is still recognisable, then the algorithm class is labelled as **'OK'** - *'It is possible to tell the main emitter, but there is evidence of the track containing other emitter(s)'.*

If no obvious pairing with an emitter is seen then it is likely that the algorithm class has combined multiple emitters into one class. These classes are labelled **'poor'** - and are *'basically a combination of multiple emitters, what we call an Over Merge (OM)'.*

Finally the algorithm can generate a class which represents the same emitter as another class in the algorithm - these are labelled as **'Multiple Tracks'** - where *'the class is a repeat of another class which has already been assigned to a specific emitter'.*

In the first run on the THALES data set we used the parameter values stated in Table 8 (we still used the a_{cl} parameter at this point), but we also included the AMP dimension of data, and normalised this in the same way as the other dimensions.

Parameter	**Value**
a_n	0.14
a_{cl}	80% of $a_n = 0.112$
a_r	1
b_c	1
b_a	1.0001
b_v	0.2

Table 8: Parameters for first THALES run

After this run THALES revealed that 55 emitters were present in the data set - our initial run had identified 57 classes. However, on comparison with the mean values for the emitters, as described above, THALES judged that 20 of these classes were ‘good’, 16 ‘OK’ and 9 ‘poor’ with the remaining 12 being ’Multiple Tracks’ [9].

We then removed the AMP dimension from the classification procedure for the reasons described in section 4.1, and we reduced the a_n parameter (to 0.11) since the 9 poor classes were evidence that some classes were being merged.

A more precise way of scoring the results is to make a direct count of the mis-classified results as we did with the test data, but this is only possible when we know which data signals are from which emitters, which THALES only revealed late on in the testing process. With this information not only can we score the results more easily, but we can set some of the parameters more effectively. The incoming data can be visualised by class as in Figure 16. From this we can see that the longest time between class presentations is certainly less than 10,000 iterations, and so we can set our b_a value, (using equation 2.14 and a short link value of 0.04), at about 1.0003.

We can also calculate how wide a radius the emitter classes have in our normalised space. We find a mean distance of about 0.09 from the class centres to furthest signals, and so setting $a_n = 0.1$ seems reasonable.

The best results we obtained on the THALES data set were with the values in Table 9. This run gave us a score of 5,524 false-negatives (and 9,525 false-positives, although

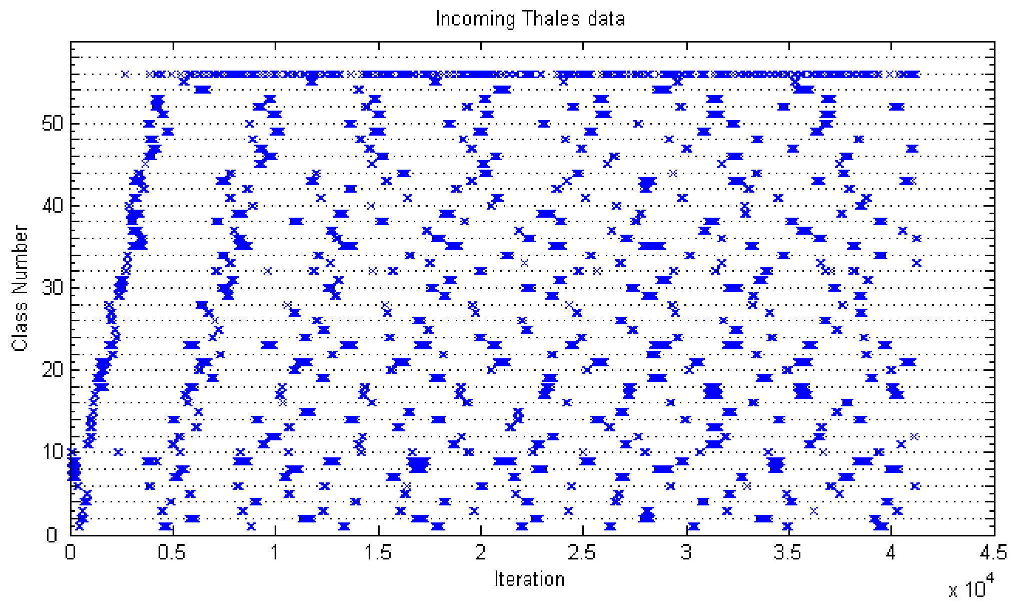


Figure 16: Incoming data plotted by class from the THALES simulated data set. The data from each emitter comes in bursts - a sweeping radar would be detectable only when it points towards us. Used with permission of THALES Aerospace.

Parameter	**Value**
a_n	0.11
a_{cl}	equal to a_n
a_r	1
b_c	1
b_a	1.0002
b_v	0.15

Table 9: Parameters for best THALES run

this is a less useful statistic - see discussion in Section 3.2) out of a total of 42,156 signals. From the false-negatives score (the data that was missed from its class) we can say that about 87% of the data signals were classified correctly together, although the false-positive score indicates that some of these classes were merged. We can check on this by copying the process that THALES used to assess these results. We calculate that we have 34 'good' classes, (10 of which we labelled 'excellent' since they had only missed or gained one or two inputs), 5 'OK' classes, and 9 'poor' ones, where multiple emitters were merged as one class. No multiple tracks were obvious, but in most of the poor classes it can be easily deduced which two emitters have been merged. The spreadsheet of the analysis is available in the online store of files associated with this project.

It is almost certain that still better results can be obtained by further tests and tweaking of the parameters, and also that using the two neglected dimensions will aid the separation process enormously.

5 Further Work

5.1 Algorithm Changes

In Section 2.2 we discuss ways of changing Lang's algorithm - in particular we suggest new ways of performing the neighbourhood update, which have proved useful to counter some of the undesirable effects of Lang's version. However, there are many other variations that we could have suggested. It is perhaps a result of the less than rigorous nature of the PSOM algorithm as it stands that there is no clear method to choose, but a rather a wide array of slightly different options. Having said that, there is nothing to prevent these untried methods potentially producing better results.

For example - the neighbourhood update procedure was altered in this project to prevent the 'neuron pushing' behaviour observed in Lang's algorithm. We changed the update of 'far' neighbours (outside the clustering threshold but still linked to the focus) by no longer pushing them away, just ageing the link to them. Lang's idea of pushing them away makes sense to provide the separation between classes. An option which may retain this feature would be to push the far neurons away, but not in proportion to their link length. Even a simple inversely proportional relationship may work better, so that very distant neurons are altered only a small amount, whereas neurons just outside the clustering threshold are altered enough to provide the separation effect Lang was after. This is just one of many possibilities that could be usefully trialled.

Another suggestion which should perhaps be looked at in future work on this algorithm would be the initialisation process. Do we need to introduce a random element into the initial group of neurons? If the network is designed to create new groups of neurons in a location representative of a new class appearing in the data then why shouldn't it start the whole process off in a similar way? The difference in the long-term running of the network will almost certainly prove trivial, but it may aid testing procedures by removing an arbitrary random element from the process, which can effect the subsequent classification, particularly over a small data set.

5.2 Parameters

We mention in Section 2.2.1 the possibility of introducing a new parameter to control the neighbourhood update. Depending on the actual process used to update the neighbourhood (Section 5.1 discusses some options) we could use this parameter in a similar way to b_v which is used to control how drastically the focus is updated. At present we only use the link length to determine the 'strength' of the neighbourhood update (that is - how drastically we update the neighbours to be like the focus). This parameter could be used instead of using link length, or it could be used in conjunction with that current link length method. Yet again we see that there are many variations that can (and should) be tried to further explore the experimental working of this algorithm.

If we could find a way to determine and optimise parameters dynamically, during the algorithm then much of the necessity of finding optimal values beforehand is removed. This is obviously a desirable outcome and so this should be a priority for future study.

5.3 Changes for THALES Data

The THALES data set provides us with 5 dimensions of data that are potentially useful to separate the signals into clusters. However the PSOM as an algorithm is only designed to deal with clusters that exhibit consistently similar values in the data dimensions, and we left two of the dimensions (time of arrival (TOA) and amplitude (AMP)) out of our input data since they could not be described in that way. One clear property of the emitters THALES have simulated is a 'pulse time', where the signals from a certain emitter tend to come at regular intervals (as a radar device might be expected to send out regular signals as it scans an area). These regular timings would be a constant feature that could help further distinguish between clusters. The problem is that until we know where to classify a signal we can't know how long it has been since the last signal from that emitter, and so we can't know the pulse time. As a result the TOA has not been incorporated into the classification process in this project, but in order to enhance the PSOM for use on this type of data we can envision a simple further step in the implementation that can make use of the TOA to compare pulse times. It is to be hoped that this enhancement would only improve the classification performance of the simple PSOM. In a slightly different way the AMP dimension is not constant, but tends to vary in a sinusoidal way over a the space of time that the signal is detected (see Figure 17 for an example of just two emitters from the THALES data, where the shape of the AMP dimension can be clearly seen in the 4th box down on the right hand side). So expecting AMP to remain a constant feature in the cluster is an error, but we would expect subsequent signals from an emitter to have similar AMP values, and that these would progress in a predictable (sinusoidal) manner. Again it is simple enough to see ways to add to the implementation to take this into account. Both of these changes move the classification away from the simple use of a pure PSOM, and as a result are beyond the scope of this short project, but the classification of this sort of data can only be enhanced if they could be incorporated.

Another data-specific change would be to offer some ranking of importance of the dimensions - are some expected to vary more than others? Dimensions that are known to be more rigid within a cluster are more helpful to distinguish different clusters. When the data is normalised these dimensions can be given a larger range in the space than other dimensions so as to increase their influence on the clustering procedure. This is not done in this project for reasons of simplicity - but anyone wishing to use the PSOM on data which has properties like this can usefully use this technique.

5.4 Mathematical Foundation

A different, but perhaps even more vital, direction for further work on the PSOM would be to try to set it on a more rigorous mathematical foundation. Many of the complications in the algorithm at present, and the wide variety of possible changes, are a result of not really knowing how to frame the algorithm mathematically. This project has reduced the 'black-box' nature of the algorithm to a certain extent by trying to justify each part of its working experimentally and numerically. Any further progress in this direction really needs to be more analytical, and may have to involve a fundamental re-working of the algorithm from a new stand-point. Philip Bond, while supervising this project, has suggested a new way of viewing the SOM and potentially the PSOM in the form of a fibre bundle, and from this viewpoint analysing the behaviour geometrically. A useful contribution to this work would be to re-code the PSOM in this different format

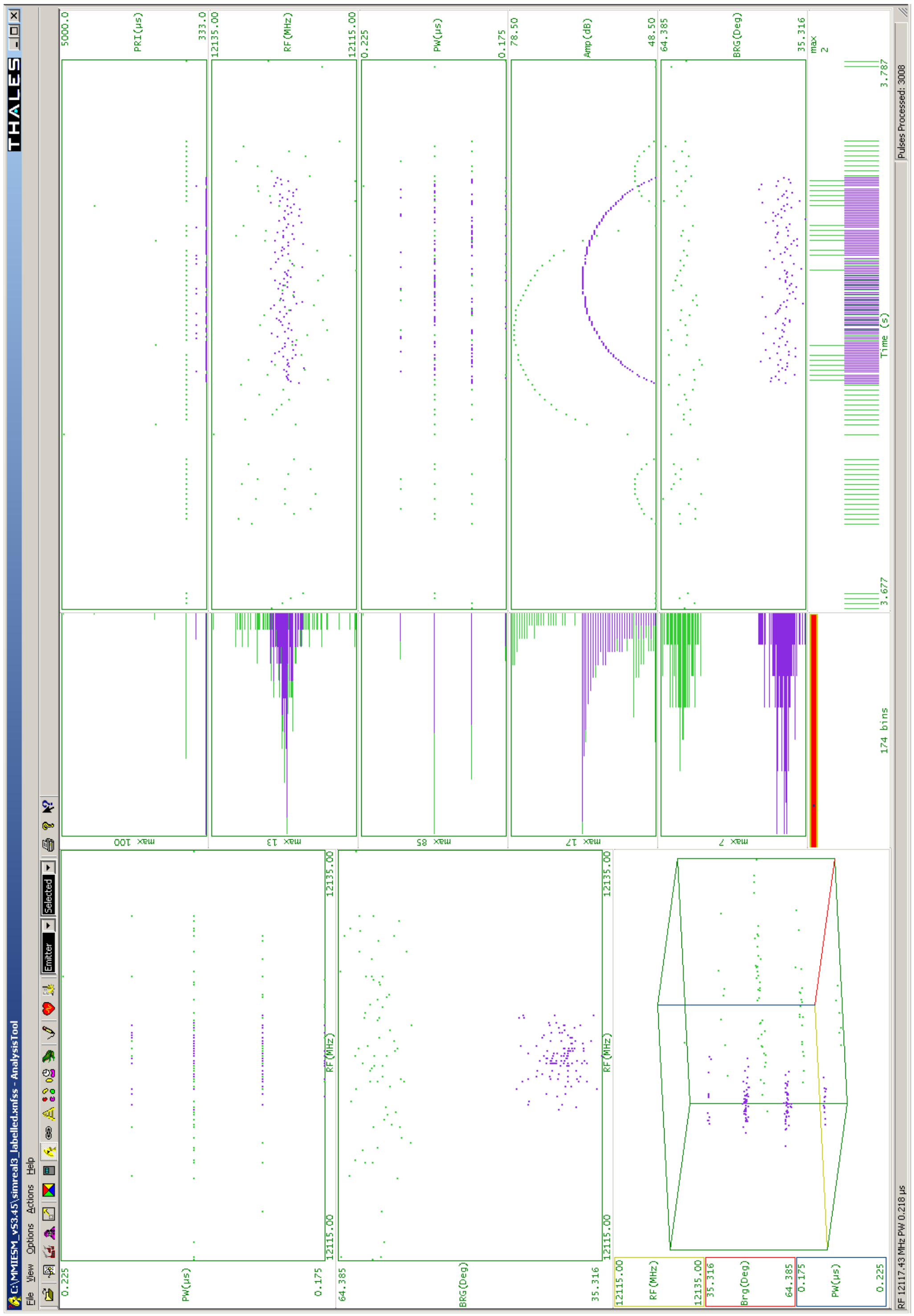

C:\MMIESM_v53.45\simreal3_labelled.xmfss - AnalysisTool
THALES
File View Options Actions Help
Emitter
Selected
PRI(µs)
RF(MHz)
PW(µs)
Amp(dB)
BRG(Deg)
Time (s)
174 bins
Pulses Processed: 3008
RF 12117.43 MHz PW 0.218 µs

Figure 17: Two emitters from the THALES simulated data set

so providing a concrete implementation of this version of the PSOM. This would enable numerical results to be compared with the theoretical analysis that becomes possible.

6 Conclusions

6.1 Summary

This project provides a new implementation of the plastic self organising map (PSOM) algorithm in Matlab - see Appendix C for the pseudo code. The full code and other files used in the project can be found online at

`http://www.maths.ox.ac.uk/∼porterm/research/PSOM`.

We offer several improvements to the algorithm detailed in Lang's original thesis [8]. These improvements are shown to markedly improve the performance of the algorithm on test data, and we explain the rationale behind the modifications and justify their inclusion by numerical experiments. The THALES data set provides additional evidence that the PSOM is functioning effectively, with an initial run on the data (with minimal knowledge of the data structure) correctly classifying about 70% of the emitters, and subsequent runs improving this to above 80% - without using two significant dimensions of the data (TOA and AMP). We also describe various directions for further work on this project, noting in particular ways to further modify the algorithm, but also emphasising the need for the PSOM to be set on a more rigorous mathematical footing.

6.2 The New PSOM

Advantages and disadvantages of the PSOM are detailed in Lang's original thesis [8], but are updated here, with references specific to this project:

6.2.1 Advantages

- The PSOM successfully deals with data of various natures, as shown in the test data sets generated for the project. It can classify regular and irregular data, forget old classes, and learn new ones in a continuous manner, without the need to stop the algorithm and retrain - this is the primary purpose for choosing the PSOM above a static neural network. It can continue classification indefinitely, without the need for stopping criteria, since it treats every signal in the same way, and no part of the algorithm depends on the total time run - this makes the algorithm ideal for a continuously changing and ongoing situation.

- It is easy to choose parameter values effectively given enough information about the data coming in. We have reduced the number of parameters to make this even easier. Choices and heuristics for them are justified by numerical testing as far as possible, and they prove fairly robust - in that they do not require highly accurate values since they change the performance of the algorithm in a mostly continuous way.
- The PSOM works in a noisy environment and successfully classifies data from our test data sets even with a large noise to signal ratio. Even with ratios as high as 30:1 the algorithm typically successfully classifies about 90% of the signal data.
- The PSOM provides a quick and largely successful classification on the THALES data set, which could easily be improved with the modifications discussed in Section 5.3. The success of the algorithm, in the absence of these modifications and with no weighting of the data dimensions is very promising, and provides good justification for pursuing this approach further.

6.2.2 Disadvantages

- Some prior information is still needed to successfully set the remaining parameters, notably the potential size of clusters (after normalisation), and the longest time between presentations of a class. Exploring the dynamic setting of parameters (as discussed in Section 5.2) may help this problem.
- The topology of the network can sometimes be unsatisfactory. As a result of links being cut and never reformed it is entirely possible for nodes in the network to end up very close to each other, but not linked in any way that the network can discern. Lang also points this out ([8] p. 173), and it means that the PSOM should not be used to present topologically reliable representations of the data - if just used as a classification device then this should not be a problem.
- There is no clear method of optimising the working of the algorithm [8] or what measurement to use for this. Recognition error (for example) provides a useful measure of the success of the algorithm, but it is unclear how to automatically

tune the parameters to reduce this, and the recognition error will of course have to be occasionally high if new classes are to be detected.

- Many of the above concerns are perhaps due to the fact that the PSOM is not (at present) built on a rigorous mathematical foundation, and suffers from a somewhat *ad-hoc* feel, although we have reduced this to a certain extent. As a result there are many possible variations to the specific working of the algorithm, and no real way yet of justifying which to choose, other than by numerical evidence of experiments.

A Graphs From Test Data

This appendix contains two sets of graphs for each parameter we vary in our experiments, one set for each version of the algorithm. Each graph has three axes, which may change orientation in the diagrams so as to give the best viewpoint. The vertical axis is always the size of the variable that we are measuring, while the two horizontal axes show the iteration number as it progresses through the algorithm, and the parameter as it is varied. The best way to view these diagrams is to follow along the iteration axis, as this represents moving through the algorithm in time. See Section 3.2 for descriptions of these measurements, but we briefly describe them here

Top Left - Recognition Error. How far each input is from the focus. We expect this to be initially high as the algorithm creates new nodes to represent the data, and we also expect a spike at just before 800 iterations, when class 5 is introduced.

Top Right - Number of Classes. How many classes are currently being used by the algorithm. Ideally this will be 4 classes for the most part. If class 2 is forgotten before class 5 arrives then we will see it drop to 3, before returning to 4. Otherwise it will increase to 5 classes as class 5 enters near 800 iterations.

Bottom Left - Number of Nodes. A measure of the size of the network - this is useful to check on efficiency. If each class is represented by 2 or 3 neurons we will expect this to take values between 8 and 15. Higher or lower values probably mean less then optimal performance.

Bottom Right - Number of Links. Similar to the previous graph, this is a useful alternative measure of size (and so efficiency). This time we just count the links rather than the nodes. When the graph appears stable it probably means the network has settled to a good representation of the incoming data.

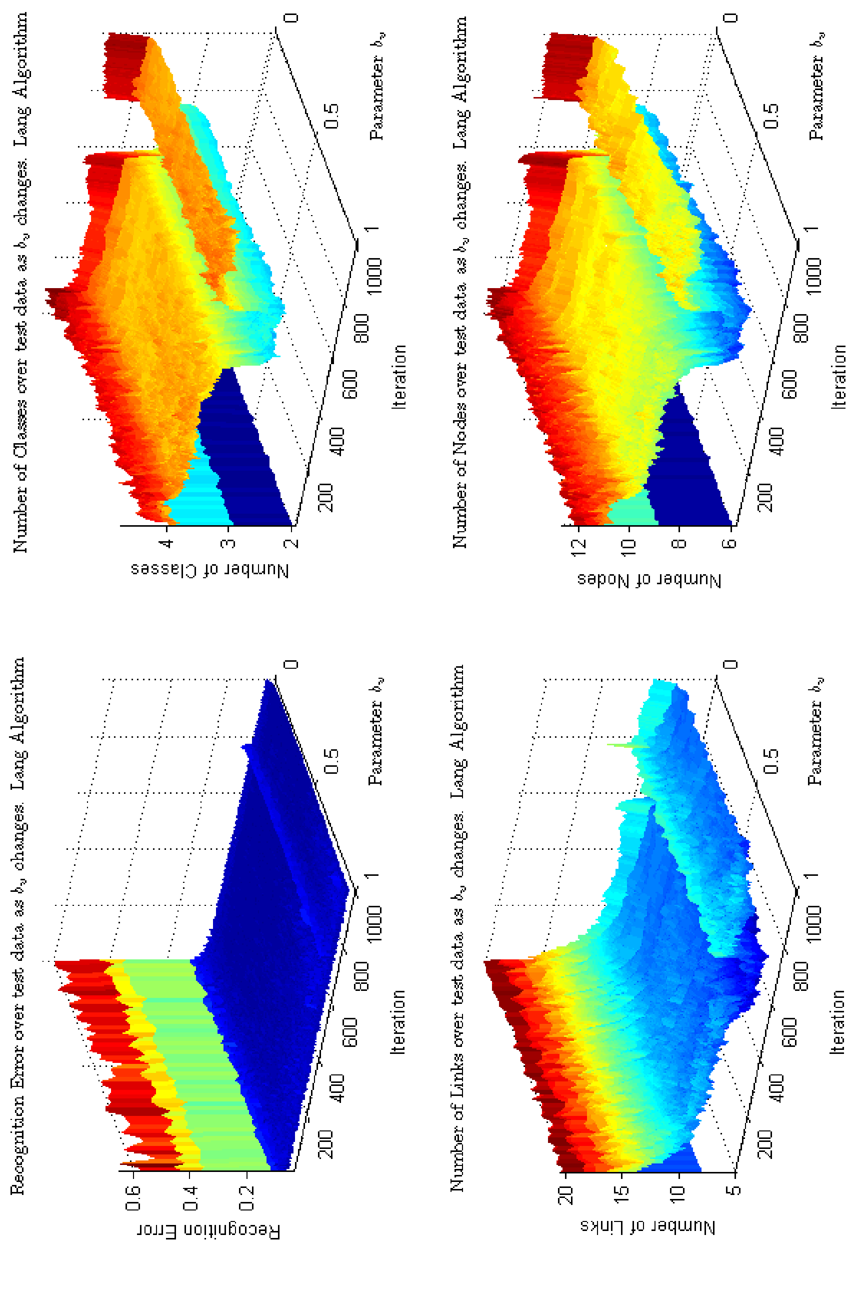


Figure 18: Other statistics for b_v, Lang's algorithm

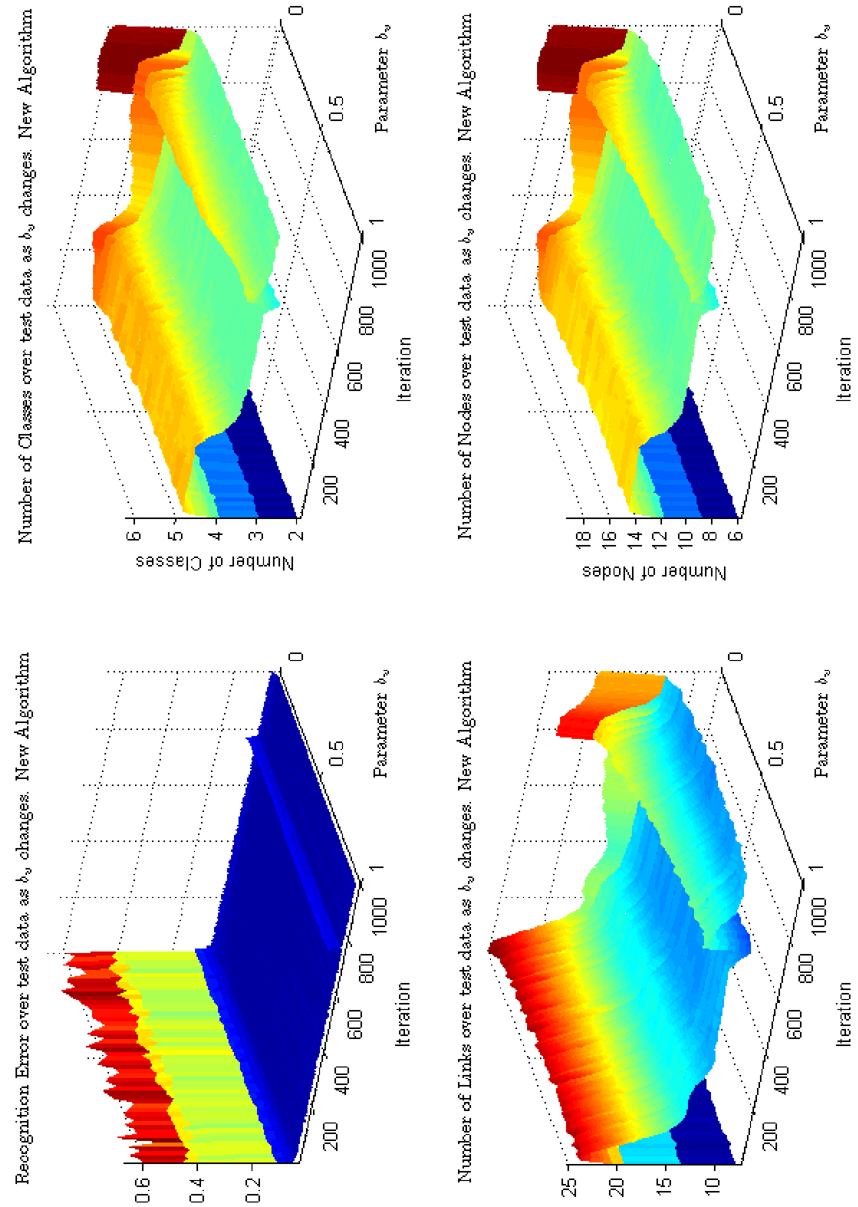


Figure 19: Other statistics for b_v, new algorithm

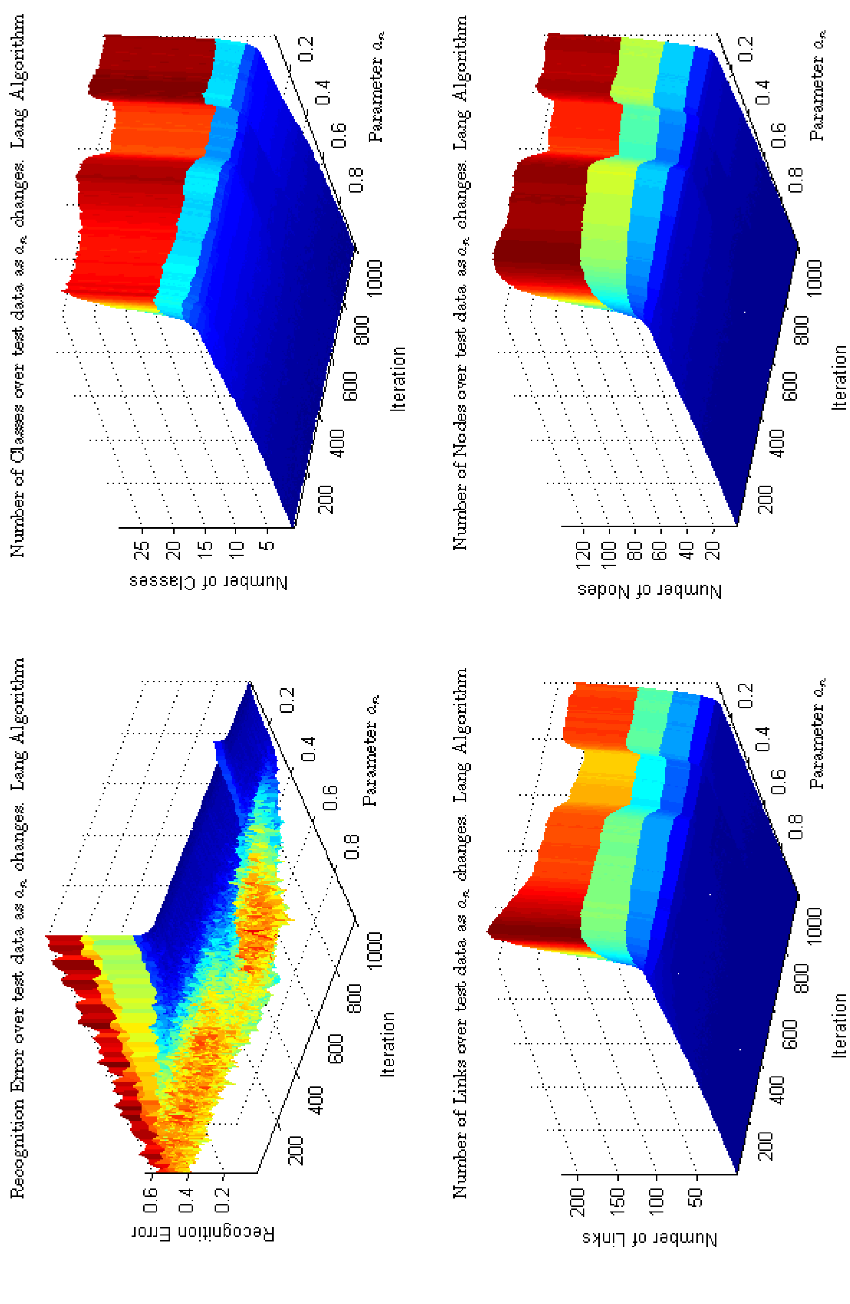


Figure 20: Other statistics for a_n, Lang's algorithm

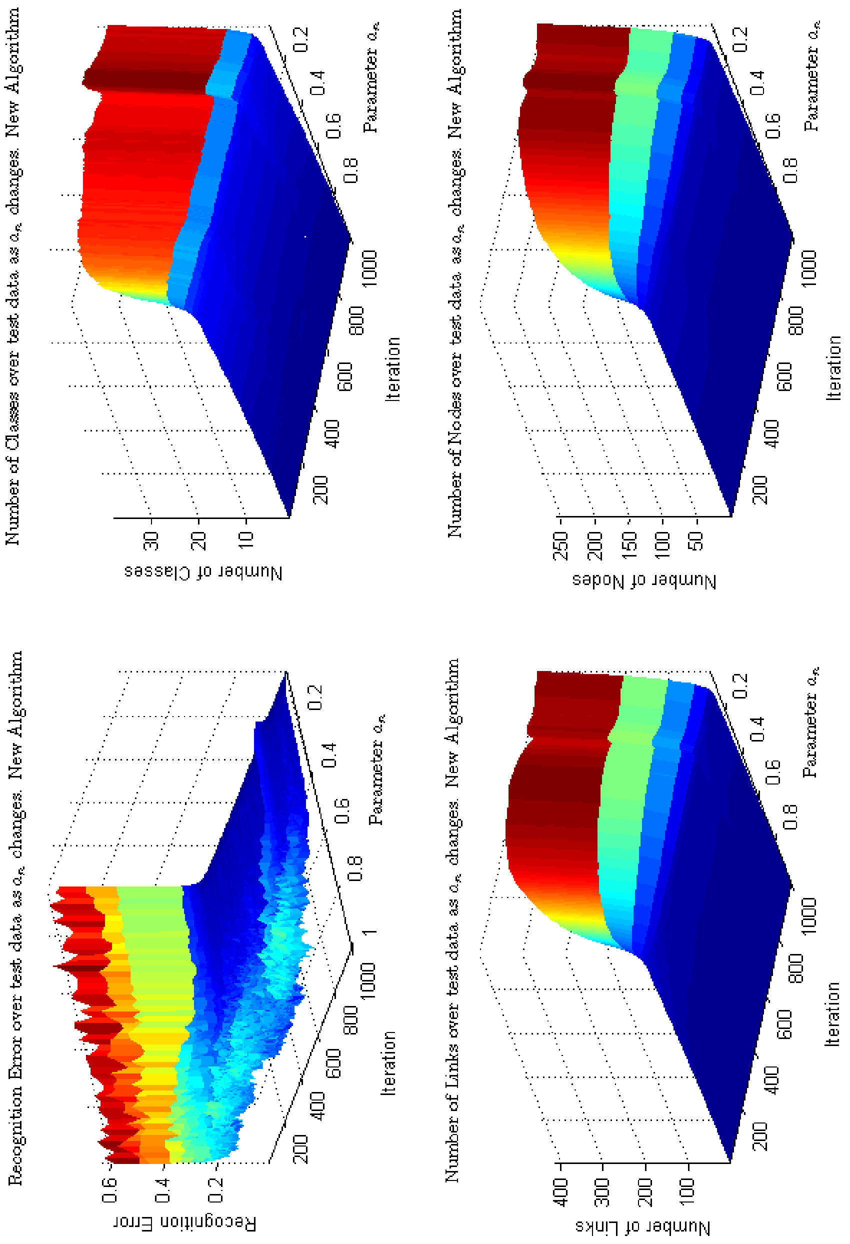


Figure 21: Other statistics for a_n, new algorithm

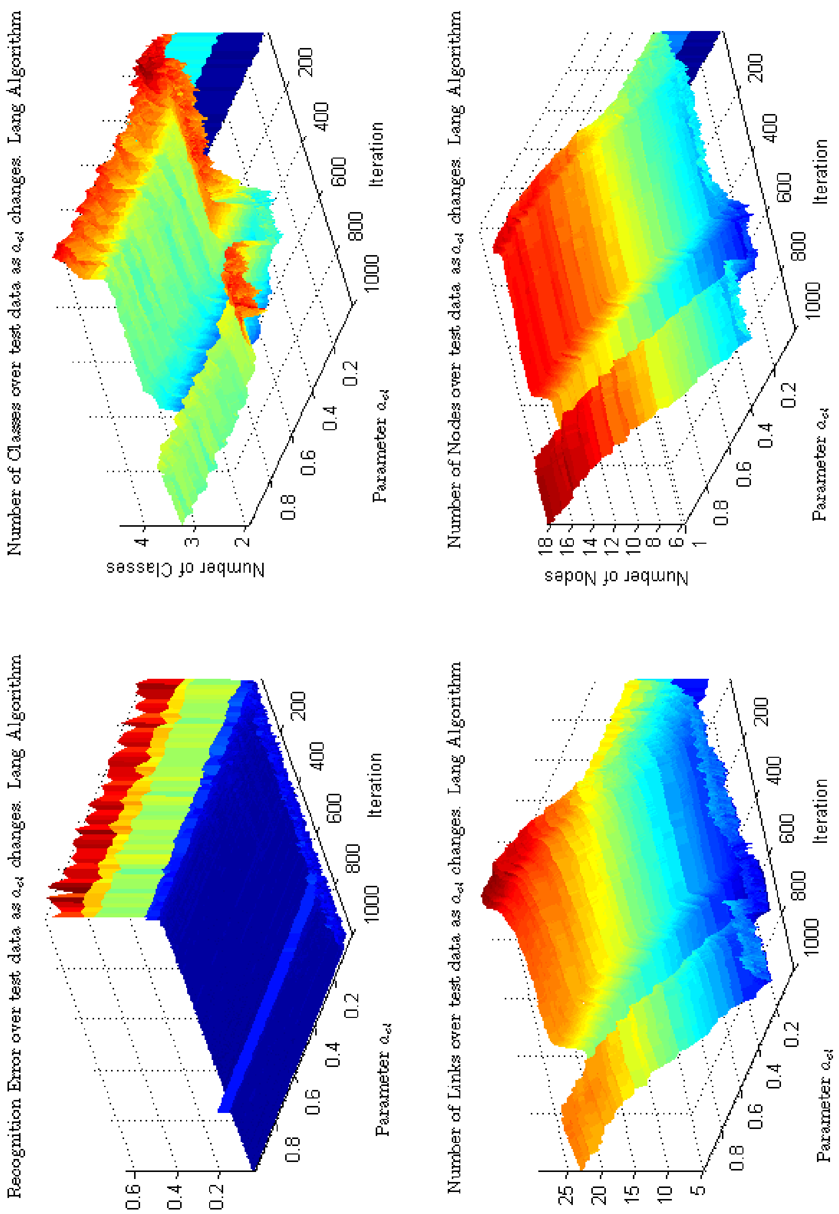


Figure 22: Other statistics for a_{cl}, Lang's algorithm

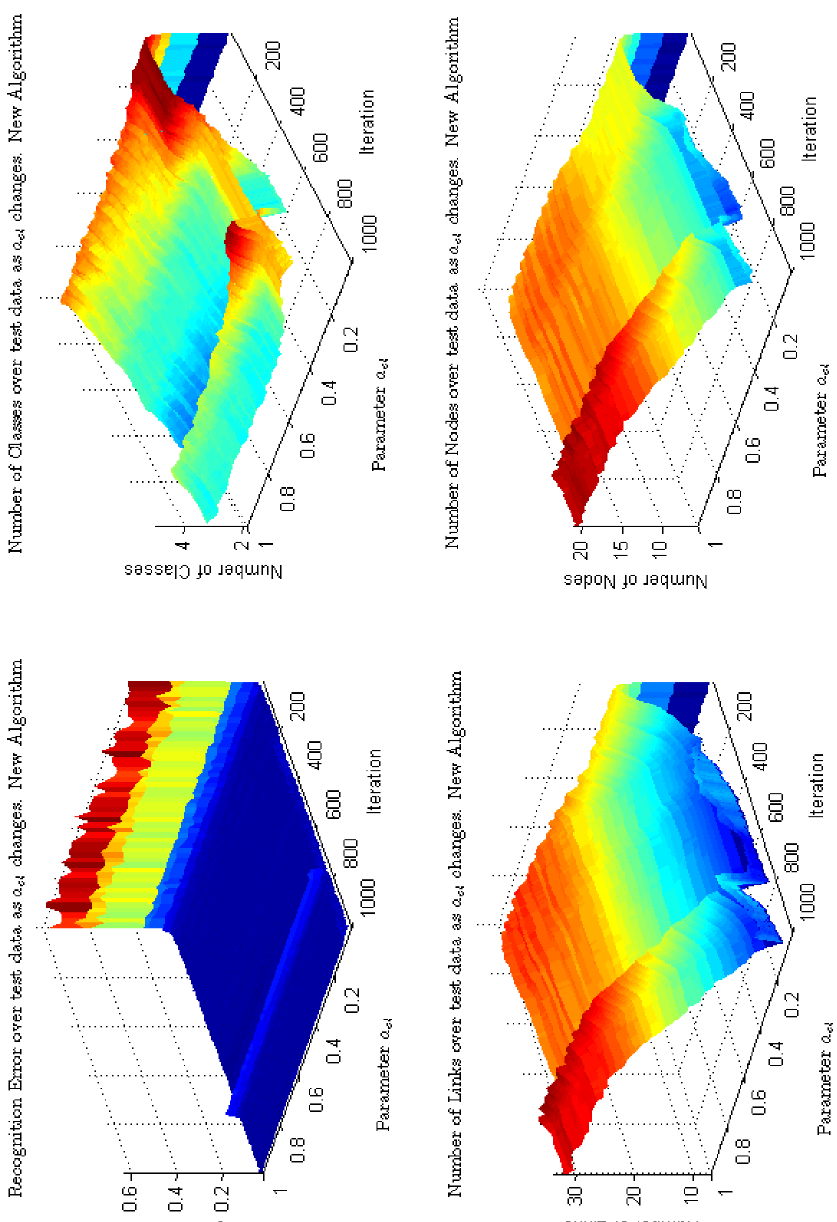


Figure 23: Other statistics for a_{cl}, new algorithm

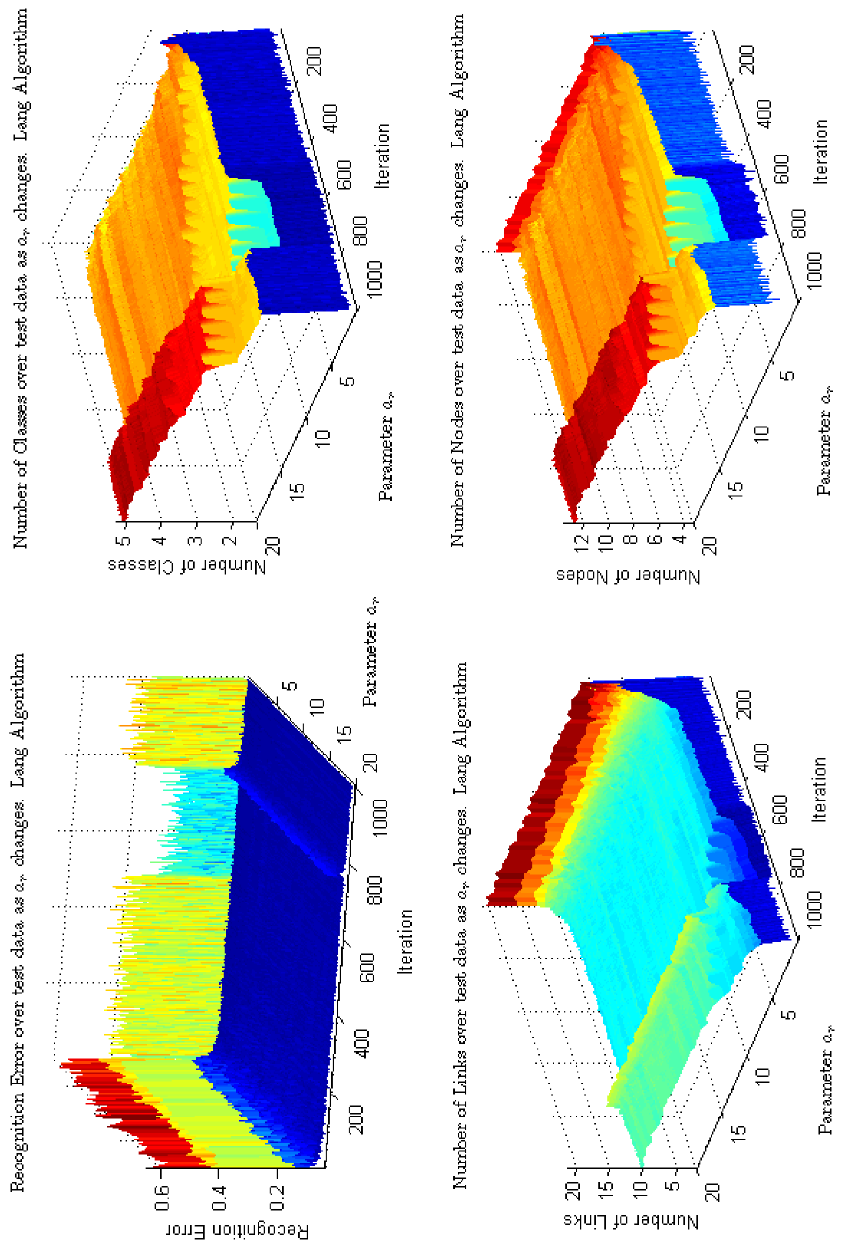


Figure 24: Other statistics for a_r, Lang's algorithm

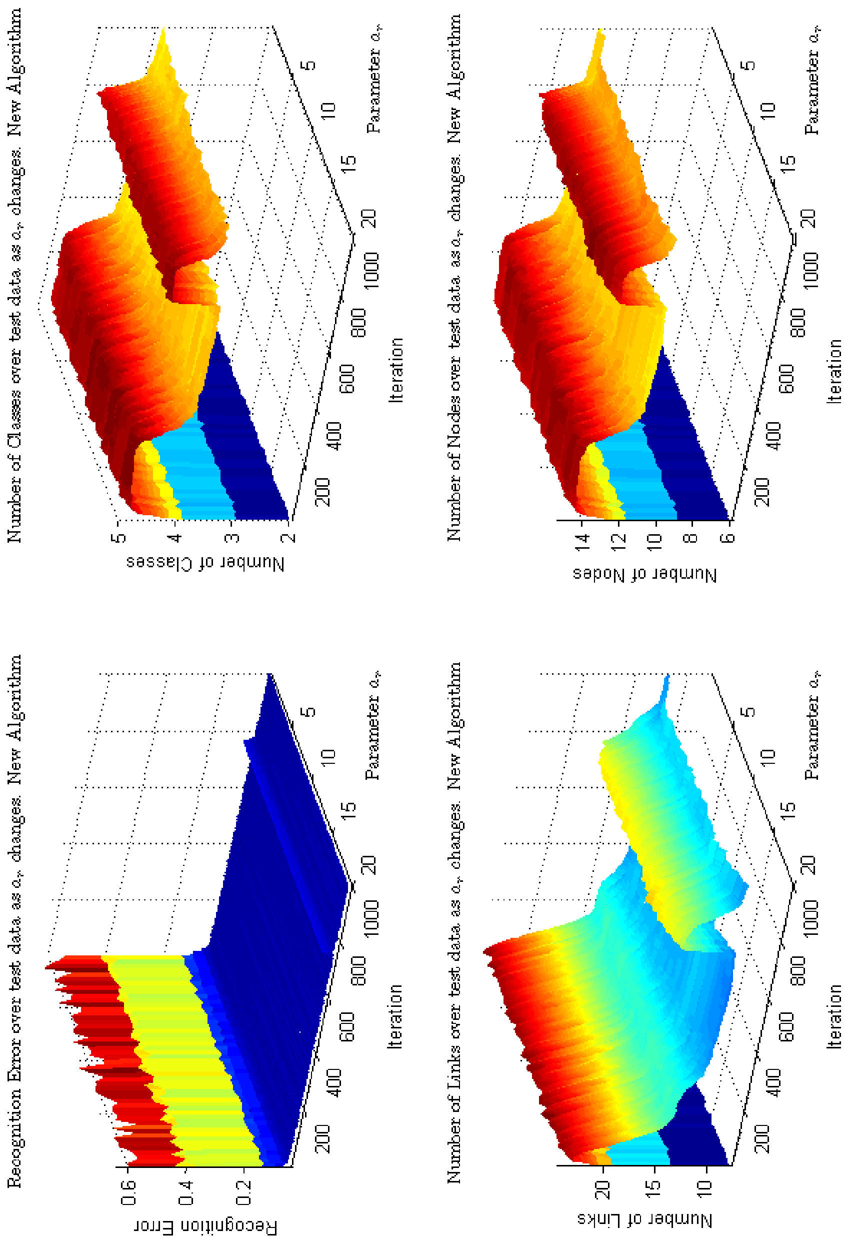


Figure 25: Other statistics for a_r, new algorithm

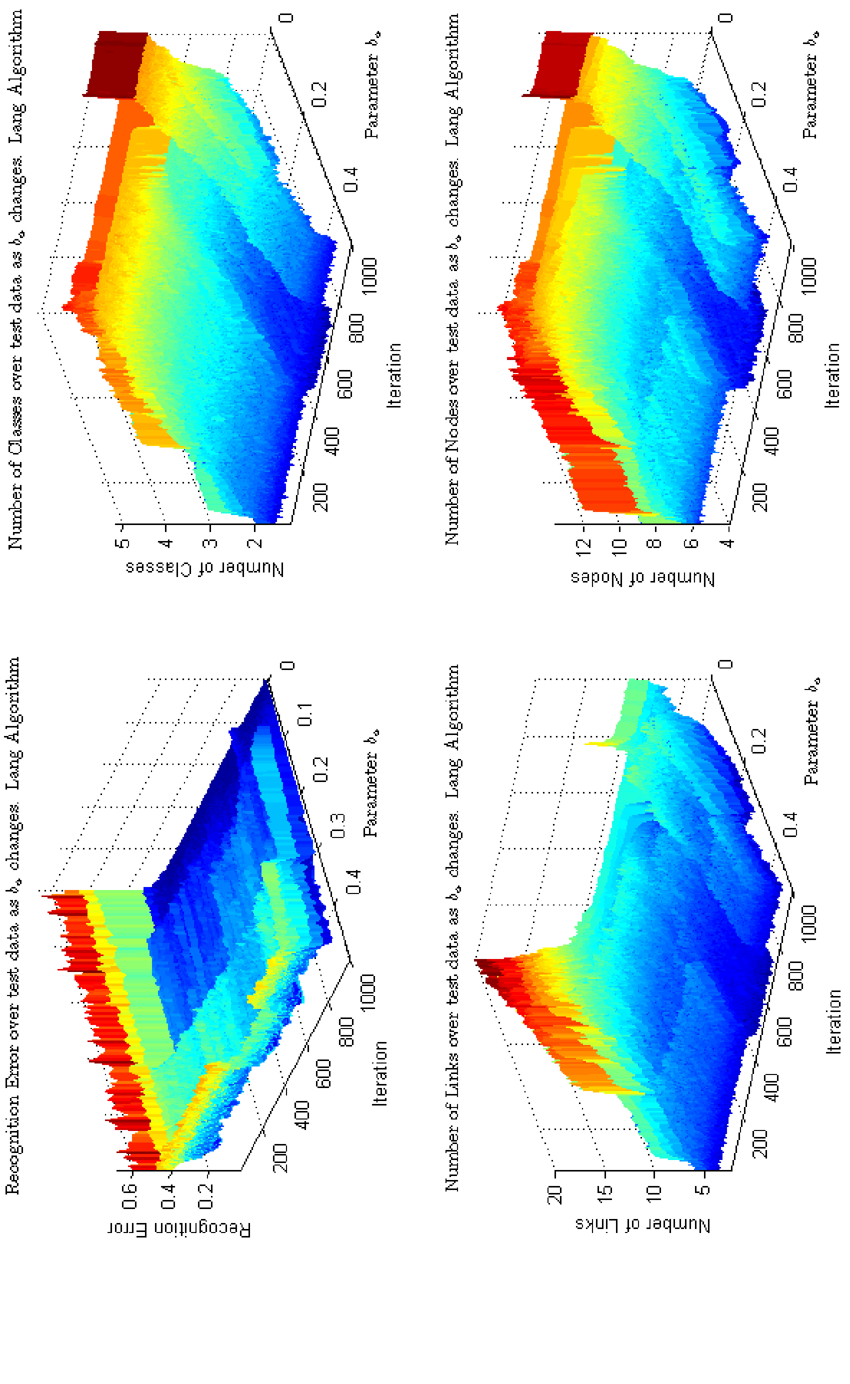


Figure 26: Other statistics for b_a, Lang's algorithm

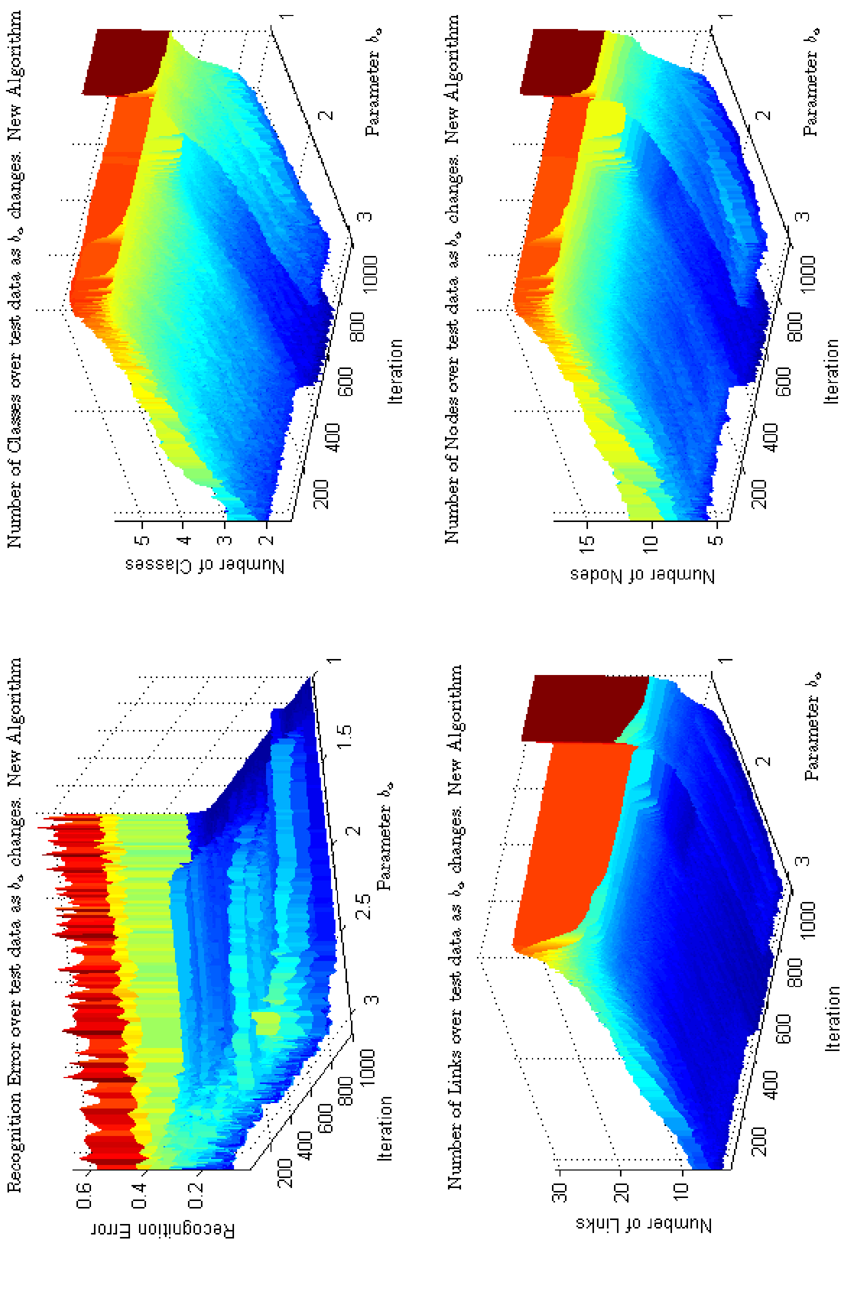


Figure 27: Other statistics for b_a, new algorithm

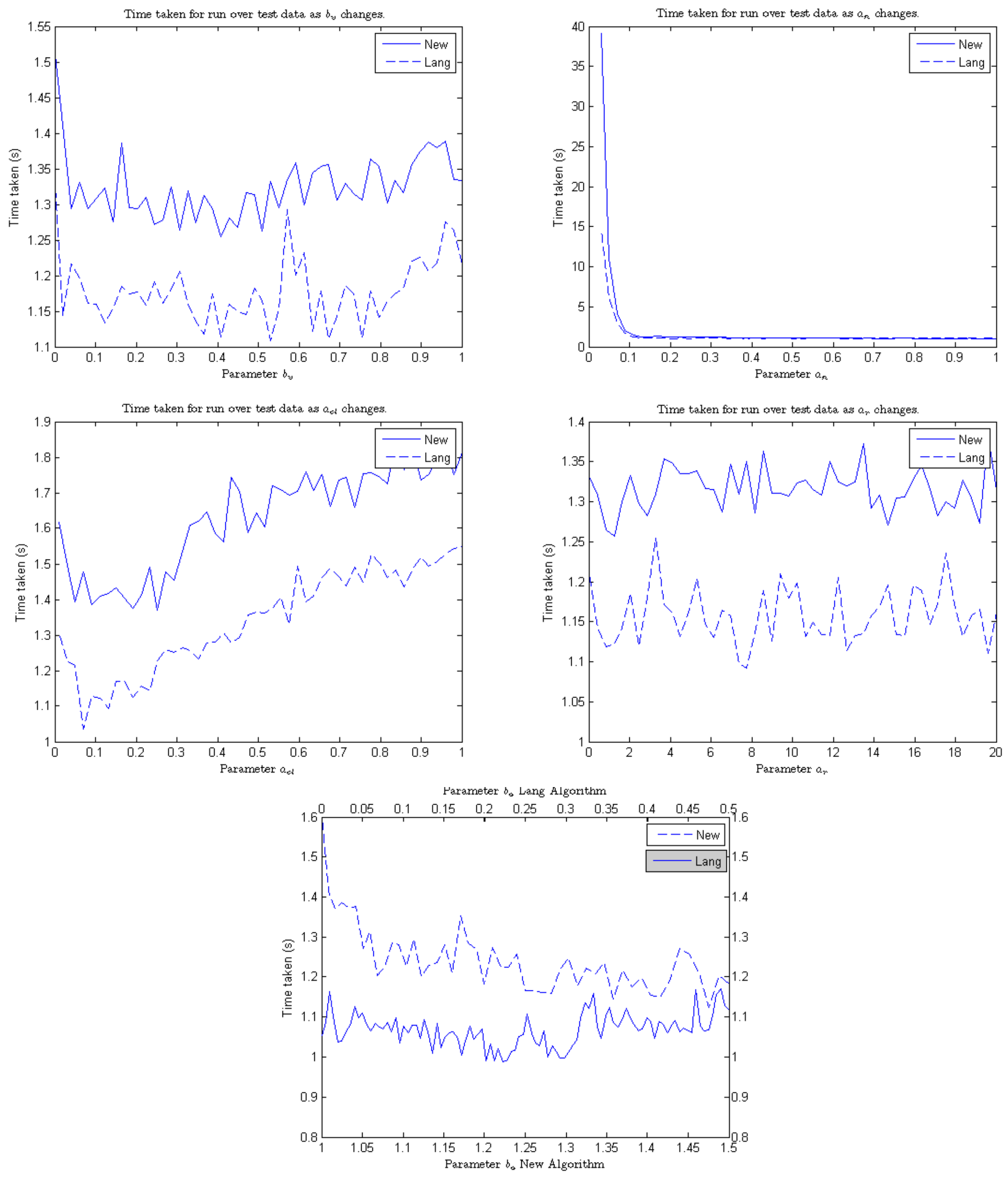


Figure 28: Times taken for different values of parameters. The times should not be taken as useful speed measurements as they were done on computers being accessed remotely at the university Computing Laboratory

B Comparison with k-means Clustering

The k-means clustering technique is a simple, effective and widely used one. Given a specific number of clusters, k, the algorithm forms data into clusters by the following process:

1. Initialise - choose (somehow) k cluster centres - i.e. the centre of all the potential cluster regions. This can be done randomly, or more effectively by using some of the first few data points in the data set.

2. Classify - assign each data point to the 'closest' cluster. Any one of several distance measures can be used to determine the closest, but it is simplest to think of this as the Euclidean distance between the data point and the cluster centre.

3. Compute cluster means - calculate the mean position of all points allocated to each cluster, these are the new cluster means.

4. Repeat - from step 2 using the new list of cluster means, until no changes of cluster membership occur anymore.

The k-means algorithm is very simple, and perhaps the most often used clustering algorithm. It is efficient - its complexity is linearly proportional to the size of the data set [3] - but it probably unsuitable for our situation. To cluster the data this algorithm needs to know all the data at the beginning - it is unable to function in 'real time' as data arrives, and so is unsuitable for a continuous clustering process.

The results in Table 10 illustrate the problems with k-means clustering on this type of data. Both false-positives and false negatives are present, and the classification is unpredictable as the three wildly different results show.

	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
1	0	200	99	100	0
2	258	0	0	0	0
3	0	0	0	0	49
4	242	0	1	0	0
5	0	0	0	0	51

	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
1	0	0	0	0	100
2	0	200	99	100	0
3	161	0	0	0	0
4	155	0	0	0	0
5	184	0	1	0	0

	Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
1	244	0	0	0	0
2	0	0	100	0	0
3	0	0	0	0	100
4	0	200	0	100	0
5	256	0	0	0	0

Table 10: Three clustering attempts using `kmeans(dataset,5)` in Matlab, compare with Table 4 for the PSOM results

C PSOM Pseudo-code

We here lay out the structure of the algorithm as it currently stands:

```
Require: data set (and the dimension of the data), parameter values for $a_n$, $a_r$, $b_v$ &
    $b_a$.
 1: initialise network with 3 random nodes connected by links
 2: repeat
 3:   input next entry from data set
 4:   find focus - closest (Euclidean) node
 5:   if distance to focus $< a_n$ then
 6:     update focus (use $b_v$)
 7:     update neighbours
 8:     assign class of focus to input
 9:   else
10:     compare with forgotten classes
11:     if distance to nearest old class < standard deviation of old class then
12:       create new group of neurons with old class number
13:       assign old class number to input
14:     else
15:       create new group of neurons with new class number
16:       assign new class number to input
17:     end if
18:   end if
19:   age all links (multiply by $b_a$)
20:   delete links longer than $a_r$
21:   delete nodes with no links
22: until data set exhausted
```

For all full code and other files see the website at

`http://www.maths.ox.ac.uk/~porterm/research/PSOM`

References

[1] Beale, J; Jackson, T. (1991) *Neural Computing: an introduction*, Reprint with corrections, Institute of Physics Publishing, Bristol.

[2] Duda, Richard O.; Hart, Peter E.; Stork, David G. (2001) *Pattern Classification - 2nd Edition*, John Wiley & Sons, Inc., New York.

[3] Gan, Guojun; Chaoqun Ma; Jianhong Wu (2007) *Data Clustering: Theory, Algorithms, and Applications*, ASA-SIAM Series on Statistics and Applied Probability, SIAM, Philadelphia, ASA, Alexandria, VA.

[4] Haykin, S. (1999) *Neural Networks, a Comprehensive Foundation*, 2nd ed, Prentice Hall, London.

[5] Keim, D and Hinneburg, A. (1999) "Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering." *Proceedings of the 25th international conference on very large data bases (VLDB '99)*, pp. 506-517.

[6] Kohonen, T. (1982) "Self-organised formation of topologically correct feature maps" *Biological Cybernetics*, **43**, pp. 59 - 69.

[7] Kohonen, T. (1990) "The Self-Organising Map" *Proceedings of the Institute of Electrical and Electronic Engineers,* **78**, issue 9, pp. 1464 - 1480.

[8] Lang, R. (2007) *Ph.D. Thesis*, University of Reading.

[9] Lavery, R. (2008), Personal e-communication regarding results of THALES data set test, 23rd July 2008