Introduction to Neural Computing and Applications (INCA)
Open Examination

Exam Number: Y1403115

Section A

1 Architecture Discussion

1.1 Problem

The problem is a two class classification problem - occupancy detection based on sensor readings. The recorded sensor data is temperature, humidity, light and carbon dioxide. The features include an additional derived feature, humidity ratio, which was derived from the original recorded variables. Each point in the data set is also timestamped. In [3] the time stamp was used to derive two additional features - week status(WS) and number of seconds from midnight(NSM). A certain amount of noise is expected in the recorded data so a suitable architecture should have good generalisation. Since data was measured in a controlled environment we expect the range of training variables to roughly match the range of expected data.

1.2 Architectures

There are several neural network architectures suitable for this kind of classification problem. We will consider Multilayer perceptron and Radial basis function network.

1.2.1 Multilayer Perceptron (MLP)

MLP is a feedforward neural network with multiple neurons composed in multiple layers. All neurons are the same in the case of classification and they compute a weighted sum of inputs and apply an activation function (sigmoid or tanh). There is an input layer, which distributes the inputs, zero or more hidden layers and an output layer. Each layer feeds into the next one. This type of neural network uses and error correcting learning rule and a backpropagation training algorithm. Backpropagation consists of two stages - a forward pass, where outputs are computed and a backward pass where weights are adjusted according to the error in outputs. In the backward pass the output layer weights are adjusted according to the local gradient of the error function and the hidden layer weights are adjusted according to the local gradients of the nodes in the subsequent layer.

1.2.2 MLP Advantages and disadvantages

The multilayer perceptron is suiatable for nonlinearly seperable classification problems. Once a good aproximate solution is found generalisation can be easyly improved by adjusting the stopping criteria of the training algorithm or using regularisation. MLP is a global approximator (because it split's the entire pattern space using lines), which can be an advantage if dataset

we have doesn't describe the full range of expected data.
Determining the number of hidden layers and number of neurons in each layer may require a lot of experimentation. The training algorithm uses local gradients and can often get stuck in a local minimum of the error function.

1.2.3 Radial Basis Function Network (RBFN)

RBFN is a feedforward network, consisting of single hidden layer and an input and output layer. Input layer just distributes inputs to the hidden layer. Each neuron in the hidden layer computes a radially symmetric basis activation function (typically a Gaussian) from the inputs. The output layer neurons compute a weighted sum of the outputs of the hidden layer. Each neuron in the hidden layer has two associated parameters - a point in the feature space, which is called a center, and a spread. It computes it's response to a given input based on the euclidean distance between the data point and it's center. Training is typically done in two stages - center selection and weight estimation, where wighgts can be cimputed in a single matrix vector calculation. RBFNs can also be trained in a supervised manner (like MLP) or by selecting centers at each data point and using regularisation, but these aproaches are much slower.

1.2.4 RBFN Advantages and Disadvantages

There is only one hidden layer so we only need to experiment with it's size. When using unsupervised center selection, weights can be computed in a single calculation. There are much less architecture parameters that can be varied when searching for an architecture compared to the MLP. The RBFN is a local approximator, which is an advantage when we expect new data to be in a similar range to the training data or when abnormal inputs need to be ignored.
It is difficult to determine center locations. When data isn't clustered might need a very large hidden layer. Supervised center selection and regularisation training approaches are very slow.

1.2.5 Choice of network architecture

Comparing the two architectures stated above we choose an RBFN network architecture. Compared to the MLP the RBFN has a simpler training process. Furthermore, using our knowledge of the problem we can see that the data we have should more or less be like the expected data after training. Because of this an RBFM would be more suitable as a local classifier rather than an MLP. There are far less training parameters that have to be discovered experimentally when using an RBFN

2 Creation and Application

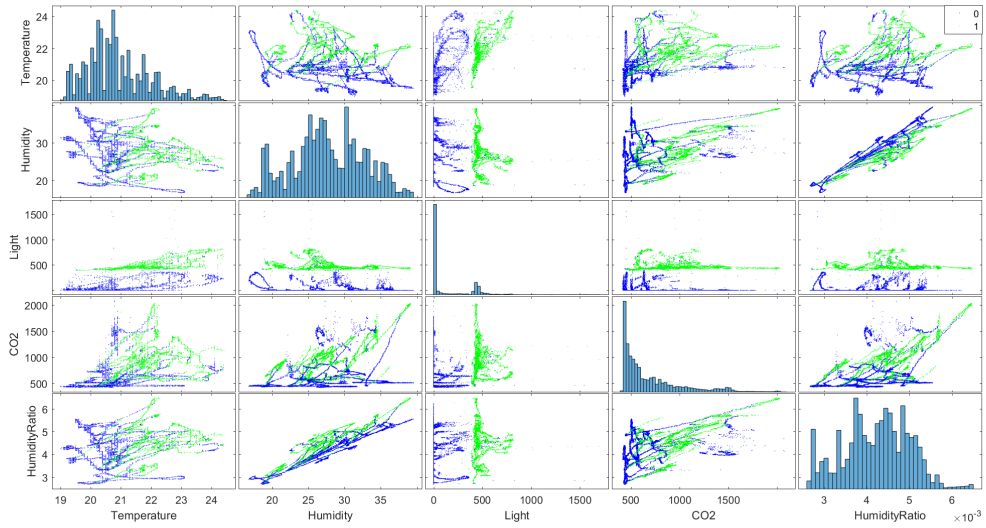
2.1 Data

Three data sets are provided for training, validation and testing. The test sets are merged together according to the chronological order of the timestamps. The composite test set has 20,560 data points The timestamps themselves are not used as features. The features are Temperature, Humidity, Light, CO2, Humidity Ratio and are all real valued. The target outputs are binary. Hummidity Ratio has not been measured, it has been derived from Humidity measurements [3].

In figure 1 the MATLAB *gplotmatrix* function is used to plot all input variables against each other and create histograms of each variable along the diagonal and classes are color coded. We

can make several observations based on this plot. The plots of light against all other features show a clear separation of the two classes, so light is probably an important feature. On the other hand the histogram of light is bimodal and clearly the examples around the second peak are much less than around the first, which means they are under-represented and might not be properly incorporated in the model. Looking at CO2 we see that it's distribution is skewed, which also might cause problems for our model. The other distributions of variables look normal.

Figure 1: Plot matrix



There seems to be a big correlation between Hummidity and HummidityRatio, which makes sense since the later is derived from the first one. It might be a good idea to discard HummidityRatio. However to make sure that we are really removing unimportant features and be able to measure their contribution to the dataset we use Principal component analysis (PCA). PCA does two things. It orthogonalises input components to remove correlation between them. It can also be used to remove principal components which have very little contribution towards the total variance of the data. Since PCA is sensitive to big variances in the data we first preprocess the data to have a variance of 1 (using Matlab *mapstd* function) and then we apply PCA (Matlab *pca* function). PCA gives us the eigenvectors and eigenvalues of the covariance matrix of the data. The eigenvectors are the transformed inputs (prinicpal components) and the eigenvalues tell us how much each component contributes to the overall variance. Looking at figure 2 we can see that the eigenvalue for the fifth principal component is only 0.001 and in terms of percentage, the contribution of this component to overall variance is 0.03%. This means that the fifth principal component hardly has any contribution on the variance of the data. Removing it wouldn't affect the separability of the data but might make network training more efficient. The forth principal component has a much greater contribution to overall variance (6.3%) so we decide keep it. The threashhold for eigenvalues of principal components has to be between the value of the forth and fifth. We decide to set it at 0.02.

Furthermore, looking at figure 3 we see that the distributions of all 4 principal components are more or less normal, which means we don't have to worry about potential problems arising from the skewed CO2 distribution or asymetric bimodal distribution of Light.

In the end, preprocessing of the input data consist of normalising variances of features to 1.

Figure 2: Eigenvalues and percent contribution to overall variance of each principal components

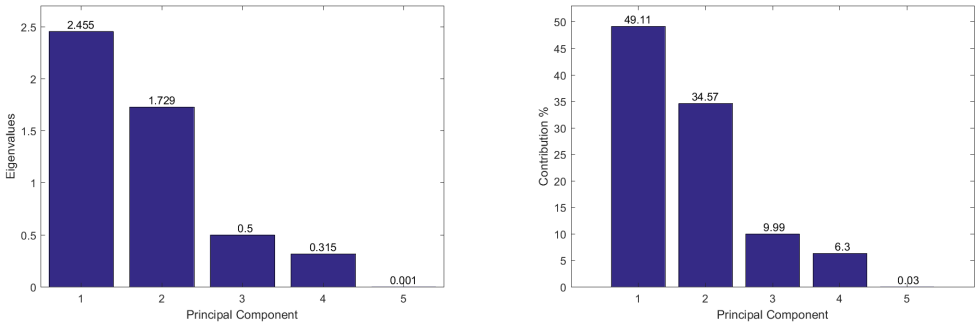
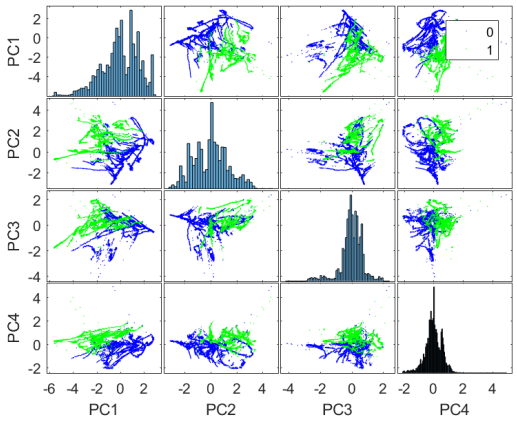


Figure 3: Plot matrix of principal components



And applying a transformation using principal component analysis, removing principal components with eigenvalues less than 0.02. We use the Matlab functions $x1, trans1 = mapstd(x)$ and $x2, trans2 = processpca(x1, 0.02)$. The transformation settings $trans1$ and $trans2$ are provided in the appendix in table 5 and table 6.

2.2 Network

2.2.1 Inputs

Inputs are preprocessed before they are presented to the network. The data set is then randomly split into a training set and a test set with ratio 75%/25%.

2.2.2 Input Layer

The input layer distributes inputs, after preprocessing, to each neuron in the hidden layer. The size of the input layer is therefore the size of the input vector $N = 4$ (after PCA). It is fully connected to the hidden layer.

2.2.3 Hidden Layer

There is a single hidden layer fully connected to the output layer. The optimal size of the hidden layer found by experimentation is $M = 20$. The activation function of each neuron is an

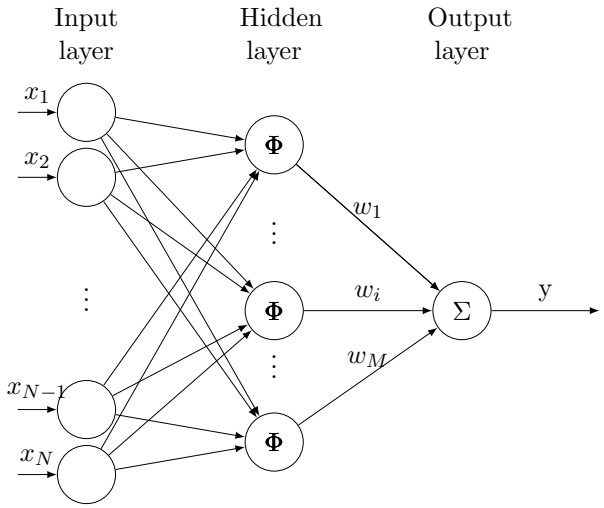


Figure 4: RBFN architecture diagram.

isotropic Gaussian:

$$\Phi_i(\boldsymbol{x}) = \exp\left(-\frac{1}{2\sigma^2}||\boldsymbol{x} - \boldsymbol{c_i}||^2\right) \quad \text{where} \quad \sigma^2 = \frac{1}{2}(\epsilon \times d_{max})^2 \tag{1}$$

Where m is the number of hidden neurons (number of centres), d_{max} is the maximum distance between centres, ϵ is a parameter used to vary the spread of the Gaussian ($0 < \epsilon < 2$) and $\boldsymbol{c_i}$ is the center vector of neuron i. The centres for each hidden layer neuron are found using a k-means clustering algorithm. The number of centres (c_i) and spread parameter ϵ were found through experimentation.

2.2.4 Output Layer

The output layer calculates a weighted sum of hidden layer outputs:

$$y(\boldsymbol{x}) = \sum_{i=1}^{m} w_i \Phi_i(\boldsymbol{x}) \tag{2}$$

In order to obtain hard classification output a threshold is applied to $y(\boldsymbol{x})$ - if $y > 0.5$ output is 1, else output is 0. The outputs use a boolean encoding, where 1 signifies occupancy.

2.2.5 Best found architecture

The best architecture found through experimentation uses k-means algorithm to find center locations. The experimentally discovered parameter values are $m = 20, \epsilon = 0.8$. The network architecture and parameters are given below in table 1.

2.2.6 Implementation

The RBF network was implemented in Matlab following the solution from practical session. The code is given in the appendix (RBFN.m). For performing experiments an additional script was used it is also provided in the appendix (runRBFN.m)

Network type	Radial basis function network
Data features/inputs	Temperature, Humidity, Light, CO2, HumidityRatio
Data preprocessing	Normalize standard deviation to 1 Principal component analysis (PCA)
PCA variance threshold	0.02
Data train/test set split	Random 75%/25%
Hidden layer activation function (RBF)	$\Phi_i(\boldsymbol{x}) = \exp\left(-\frac{1}{(\epsilon \times d_{max})^2}\|\|\boldsymbol{x} - \boldsymbol{c_i}\|\|^2\right)$
Hidden layer size (m)	20
RBF spread σ^2	$\frac{1}{2}(0.8 \times d_{max})^2$
Output layer activation function	Linear: $y(\boldsymbol{x}) = \sum_{i=1}^{m} w_i \Phi_i(\boldsymbol{x})$
Output layer size	1
Training algorithm	1. Unsupervised center selection 2. One step weight calculation using matrix pseudo inverse
Centre selection algorithm	k-means clustering
Backprop performance function	Mean Squared Error (MSE)

Table 1: Best found RBFN architecture.

2.3 Training

Three types of RBFN training algorithms were considered - regularisation with number of hidden neurons equal to the number of data points, supervised center selection and unsupervised center selection. Unsupervised center selection was chosen out of the three because of it's performance in terms of speed. This is a two step training algorithm. The first step is selecting the center points for each hidden neuron. The second step is calculating the output layer weights based on the outputs of the hidden layer in order to minimise an error function (MSE in this case). After having selected the center points and preprocessed the data, the training algorithms goes through the following steps, assuming center points are in a matrix c, preprocessed training data is in a matrix x, the target data is stored in a binary vector D and the predefined number of centres is m:

1 Calculate the euclidean distance from each of the centres (columns of c) to each of the data points (columns in x) and store them in a matrix *distance*. Calculate the distances from centres to centres and store them in a matrix *dc*.

2 Find the maximum center to center distance and store it in d_{max}. d_{max} is then used to calculate the spread of all radial basis functions in the hidden layer: $\sigma^2 = \frac{1}{2}(\epsilon \times d_{max})^2$, where ϵ is a user defined parameter.

3 Calculate a matrix of responses from each basis function to each data point. Using equation 1 this calculation would be:

$$F = \Phi_i(\boldsymbol{x}) = \exp\left(-\frac{m}{(\epsilon \times d_{max})^2} distances^2\right) \tag{3}$$

4 Calculate the weights of the output layer W using the pseudo inverse of F^+ and the target data D: $W = F^+ D$.

For the center selection step of the training algorithm two options were considered - random center selection and k-means clustering. The kmeans function from Matlab was used for the

second approach, which uses k-means++ algorithm [1] for generating initial center locations and then runs the k-means clustering algorithm [5] to iteratively search for the true center locations. In order to decide which one to use an experiment was performed using two networks, structurally the same as the one in table 1,but with number of centres $m = 10, 15, 20, 25$ and $\epsilon = 1$. The two networks use the two different center selection algorithms. They were ran 10 times for each number of centres. The results are given below in table 2

Number of centres (m)	10	15	20	25
Random center selection Test MSE mean	0.0735	0.0785	0.0832	0.0649
Random center selection Test MSE Std	0.0306	0.0523	0.0499	0.0304
K-means Test MSE mean	0.0331	0.0237	0.0190	0.0187
K-means Test MSE Std	0.0032	0.0044	0.0043	0.0041

Table 2: Performance results for random centre selection and k-means

Clearly the performance (in terms of MSE) of k-means is much better than random center selection. So this is the algorithm we will use.

2.4 Evaluation

We have made decisions about the network structure, training algorithm, data preprocessing and splitting into training and test sets. The parameters that need to be discovered experimentally are the number of centres (size of hidden layer) and basis function spread parameter ϵ, which we introduced in the Network section.

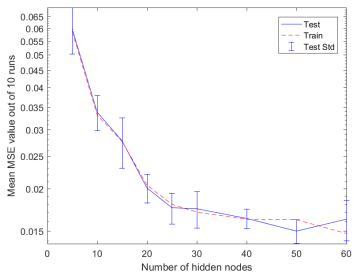
2.4.1 Metrics

MSE was used as a metric of performance. However MSE on it's own is not sufficient, because it will keep decreasing as we increase the number of nodes in the network, but if we add two many nodes the generalisation of the network will be poor. This is why in addition to MSE, we also look at the percentage of errors in each of the two classes. Ideally we want our model to make an approximately equal amount of errors in both classes. If the model develops a bias against a particular class this means that we are probably over fitting the data or not capturing the structure of the data.

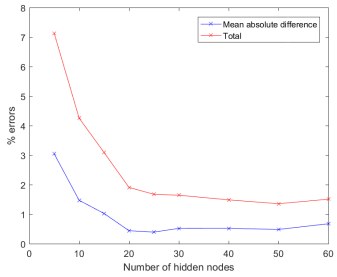
2.4.2 Experimentation for determining final network

Initially we consider only the number of centres m and search for networks with low MSE and approximately equal percentage of errors in both classes. We set the spread parameter of the network $\epsilon = 1$ and run for different hidden layer sizes $m = 5, 10, 15, 20, 25, 30, 40, 50, 60$. For each value of m we train and evaluate the network 10 times and record the mean value of MSE, mean percentage of errors and mean absolute difference of errors for both the training and test sets. Results are plotted in figure 5.

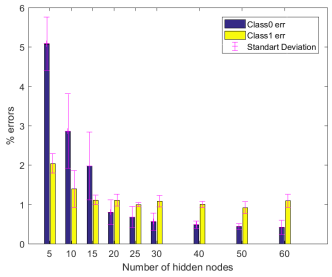
From figure 5a we can see that the MSE decreases when increasing the number of nodes, as we expected. After 20 hidden nodes MSE decreases at a slower rate. Looking at figure 5b we see that after around 20 hidden nodes the mean absolute difference in errors doesn't change very much and the total number of errors decreases very slowly. These two plots give us an understanding of how performance increases by increasing number of neurons, but tells us little about how much we are overfitting the data. Looking at figure 5c we can see that after 25 hidden nodes the errors in class 1 remain the same, while the errors in class 0 decrease. After that point the network is developing a bias against class 0. This is a hint that we are overfitting after this



(a) Mean MSE of Test and Train sets.



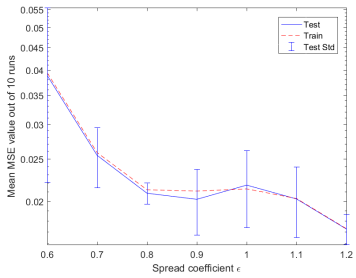
(b) Mean absolute difference of errors in class 0 and class 1 and mean percentage of errors in both classes, measured on test set.



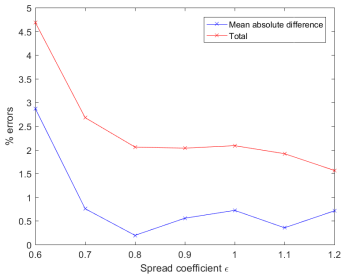
(c) Bar plot of mean percentage of errors in class 0 and class 1 measured on test set.

Figure 5: Performance of RBFN with parameters $m = 5, 15, 20, 25, 30, 40, 50, 60$ and $\epsilon = 1$, ran 10 times for each parameter combination.

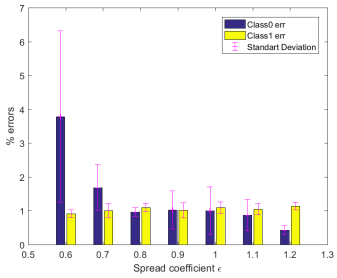
point. On the other hand, when we use fewer than 20 nodes the bias of the network is against the other class. This probably means we don't have enough neurons to capture the structure of the data. So we decide to use 20 hidden nodes the hidden layer and perform experiments to determine the spread coefficient ϵ, while keeping $m = 20$. The results are shown in figure 6.



(a) Mean MSE of Test and Train sets.



(b) Mean absolute difference of errors in class 0 and class 1 and mean percentage of errors in both classes, measured on test set.



(c) Bar plot of mean percentage of errors in class 0 and class 1 measured on test set.

Figure 6: Performance of RBFN with parameters $m = 20$ and $\epsilon = 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2$, ran 10 times for each parameter combination.

Looking at figure 6a we see that MSE reaches a local minimum around $\epsilon = 0.9$ and at that point the test set MSE is lower than the training set MSE. After $\epsilon = 1$ it starts decreasing, which indicates we might be ofverfitting again. Looking at 6b, we see that at $\epsilon = 0.8$ we get very low mean absolute difference in errors between the two classes. Furthermore, looking at 6c we see that at the same point the mean percentage of errors for both classes is around 1% with a very small standard deviation. We choose $\epsilon = 0.8$ for our final architecture, as it achieves a low MSE, a balance between errors in the two classes and stable results.

2.5 Results

The final chosen architecture achieves an average MSE of 0.025. In tables 3 and 4 the mean results for the training and test sets out of 20 runs of the architecture are shown. We can see that the distribution of errors in the two classes isn't as balanced as when we ran the network

10 times. This could mean that the experiments performed were insufficient.

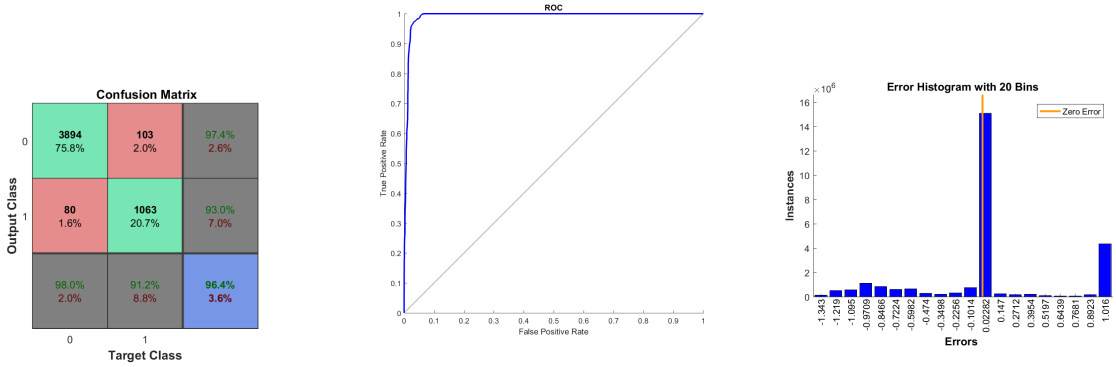
mse	std mse	err0	std err0	err1	std err1	err diff	std err diff
0.025075	0.0065629	1.5438	0.90081	1.0428	0.19202	0.68385	0.82889

Table 3: Mean test results for $m = 20$, $\epsilon = 0.8$ out of 20 runs.

mse	std mse	err0	std err0	err1	std err1	err diff	std err diff
0.025452	0.0062647	1.6051	0.85912	1.0639	0.10549	0.67672	0.80653

Table 4: Mean train results for $m = 20$, $\epsilon = 0.8$ out of 20 runs.

On figure 7 the confusion matrix ROC plot and error histograms are shown for a single run of the network.



(a) Confusion matrix. (b) ROC plot. (c) Error histogram.

Figure 7: Performance of RBFN with parameters $m = 20$ and $\epsilon = 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2$, ran 10 times for each parameter combination.

From the confusion matrix we see that the network achieves 96.4% accuracy with a relatively equal distribution of errors in both classes. The ROC plot indicates a very good classifier as well. However on the error histogram we see that there are a lot of errors at the far right. This indicates that the network wasn't even close to classifying these data points correctly. These maybe points located near near a class boundary which are misclassified due to noise or outliers, which are not in range of any of the RBFs. We also notice that the network shows a relatively small MSE even with a very small hidden layer. This could mean that the data we have isn't representative of the problem. Comparing the performance of the RBFN to that of the models proposed in [3] we see that we get approximately the same accuracy as the models using the same input variables.

Section B

3 Building energy consumption prediction

The problem of predicting energy consumption of a building can be very important for places where energy consumption goes up in colder months, while energy production goes down. The

chosen network is a Multilayer Perceptron. A similar problem solved with neural networks can be seen in [4].

3.1 Technical features of architecture

MLP is a feedforward, multilayer architecture consisting of one input layer, one output layer and one or more hidden layers. The input layer distributes the inputs. Each neuron in the hidden layers computes a weighted sum of inputs, applies an activation function and produces an output. Each layer is fully connected to the next one. The output layer produces the outputs. The MLP uses an error correction learning rule, where it minimises an error function defined by the difference of it's output and the desired output (typically the error function is mean squared error). The activation functions for all hidden layer neurons are either hyperbolic tangent functions or a sigmoid functions. In the case of regression, the output layer has a linear activation function. The training algorithm for MLP is backpropagation. It consists of a forward pass, where the network computes it's output and a backward pass, where the weights are adjusted. Weight changes at each layer are computed based on the local gradient of the next layer. The MLP partitions the feature space into regions (using straight lines in the 2D case).

3.2 Suitability of the architecture for your application

The multilayer perceptron is a universal aproximator. Given the right architecture and enough data it can approximate any function. The training process is slow and has to be done before the network is deployed. MLP also requires a large amount of data to be trained and the data needs to be representative of the problem. However once deployed the network can produce outputs very quickly. MLP is suitable for this particular problem because there is already a lot of historical energy consumption data available to train the network. It might be susceptible to changes in building, depending on the data used as inputs. Data about lots of different buildings might be required in order to achieve good generalisation.

3.3 Source of data, and how the network might be trained

As mentioned already, there is plenty of historical data to use as inputs to our network. Since the problem is time series prediction we will be using past energy consumption as inputs. Since energy consumption is often dependant on the time of year, the first three input variables will be the energy consumption for the month before, this month and next month last year (if we are predicting for July 2017, inputs are energy consumption for June, July, August 2016). We also take the average temperature from the weather forecast for these three past months as an input. Another sensible input value would be the energy efficiency score of the building(this might be country specific). The network is trained using backpropagation. The desired values are obtained from historical data from past years. The activation function of hidden layer neurons will be sigmoid, so normalizing input variables to the range 0-1 so they match the dynamic range of the activation function will speed up training. The input data will be split into training, validation and test sets (50/75/75). During training the error on the validation set will be monitored and used as a stopping criteria. Size and number of hidden layers need to be determined experimentally. If desired accuracy isn't achieved then regularisation can be used in training.

3.4 How the network would handle significant changes to building or environment

The energy consumption inputs of the network depend on the building they were measured in, so changes in building might affect the performance of the network significantly. The data we have for training should cover the range of buildings and climate zones, where we want the network to be deployed afterwards. The energy efficiency input, as mentioned might be country specific and might have different ranges, or categories. We can overcome this by normalising the values for different efficiency scales to the range 0-1. However the scales might differ in their criterion for determining energy efficiency. If this is the case the network won't generalise well in the different countries. If none of these approaches gives sensible results, more domain knowledge of the problem might be required to select more suitable inputs.

4 Abnormality detection in the operation of HVAC systems

Early detection of faults in an HVAC system could significantly decrease maintenance costs. Research in abnormality detection in HVAC sensor data has previously shown good results [6]. The proposed neural network architecture follows the one used in [2].

4.1 Technical features of the architecture

Binary CMMs are single layer neural networks, which use the Hebbian learning rule. They consist of a single weight matrix which stores correlations between input and output patterns. Learning is done by computing the outer product of the binary input and output and adding it to the outer products of all other stored patterns to form the weight matrix. Recall of a stored pattern is done by multiplying the weight matrix and the input. The result from a recall is real valued. In order to obtain the associated binary output pattern, the output needs to be threasholded. A Wilshaw threashold is chosen for this architecture where each value in the recalled vector is compared to the number of bits, set to one in the input, and is set to one if it's greater and to 0 otherwise. Real valued inputs also need to be encoded. We to this by a binning procedure, where we choose bins for certain ranges of the input and associate a binary pattern to that bin. After all inputs have been binned they are concatenated to form the input to the CMM.

4.2 Suitability of the architecture for your application

A hardware implementation of CMM can be very fast. Store and recall are both very fast operations. This means we can monitor HVAC systems operation in real time and discover abnormal behaviour as soon as it occurs. This makes our chosen architecture very suitable for the particular problem.

4.3 Source of data, and how the network might be trained

The source of data will be sensors, part of the building managment system, which monitor different aspects of the HVAC system operation. Sensors could be placed in each individual room in order to help localise faults easier. The sensor measurements we will use as inputs are energy consumption, airflow, supply water temperature. Sensor readings are taken at discrete time intervals to form our training input vectors. A significant amount of data might be required in order to achieve good generalisation and performance. We decide to monitor normal operation of the HVAC system for 24 hours during 4 days of the year in different weather conditions and

temperatures. The network is trained by applying the described binning and concatenation operation to different input vectors and storing them in the CMM. Detecting abnomalities is done by recalling patterns in real time and comparing how similar they are to known patterns using k-nearest neighbour algorithm. The tolerated level of simmilarity can be adjusted by using a smaller threashold at the outputs during recall and requires either domain knowledge or experimentation. If the achieved performance is not satisfactory domain knowledge can also be used in determining appropriate input variables and sizes of bins.

4.4 How the network would handle significant changes to building or environment

The CMMs susceptibility to change of building or environment depends on the training examples. Given enough training data for different buildings in different environmental conditions the network will show good generalisation. On the other hand network can be trained over time by storing input patterns which prove to be false alarms. These patterns have to be stored manually by maintenance staff of the building for example.

References

[1] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[2] Jim Austin, Tom Jackson, Victoria Jane Hodge, and Grant Brewer. Aura-alert: The use of binary associative memories for condition monitoring applications. In *CM 2010 The Seventh International Conference on Condition Monitoring and Machinery Failure Prevention Technologies*, pages 699–711, June 2010.

[3] Luis M. Candanedo and Vronique Feldheim. Accurate occupancy detection of an office room from light, temperature, humidity and {CO2} measurements using statistical learning models. *Energy and Buildings*, 112:28 – 39, 2016.

[4] Hamid R Khosravani, María Del Mar Castilla, Manuel Berenguel, Antonio E Ruano, and Pedro M Ferreira. A comparison of energy consumption prediction models based on neural networks of a bioclimatic building. *Energies*, 9(1):57, 2016.

[5] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[6] Balakrishnan Narayanaswamy, Bharathan Balaji, Rajesh Gupta, and Yuvraj Agarwal. Data driven investigation of faults in hvac systems with model, cluster and compare (mcc). In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, BuildSys '14, pages 50–59, New York, NY, USA, 2014. ACM.

[7] Ping Zhou and Jim Austin. *A Binary Correlation Matrix Memory k-NN Classifier*, pages 251–256. Springer London, London, 1998.

Appendices

A Data import script

Listing 1: getData.m

```
[dataTable, ~, ~, ~] = getDataTable();
x = dataTable{:,{'Temperature','Humidity','Light','CO2','HumidityRatio'}}';
t = dataTable{:, {'Occupancy'}}';

function [table1, tableTest1, tableTrain, tableTest2] = getDataTable()
datatest1 = getDataset('data\datatest.txt');
datatraining = getDataset('data\datatraining.txt');
datatest2 = getDataset('data\datatest2.txt');

dates = [datatest1.Date; datatraining.Date; datatest2.Date];
temp = [datatest1.Temperature; datatraining.Temperature; datatest2.Temperature];
hum = [datatest1.Humidity; datatraining.Humidity; datatest2.Humidity];
light = [datatest1.Light; datatraining.Light; datatest2.Light];
co2 = [datatest1.CO2; datatraining.CO2; datatest2.CO2];
humr = [datatest1.HumidityRatio; datatraining.HumidityRatio; datatest2.HumidityRatio];
occ = [datatest1.Occupancy; datatraining.Occupancy; datatest2.Occupancy];

tableTest1 = table(datatest1.Date, datatest1.Temperature, datatest1.Humidity,...
 datatest1.Light, datatest1.CO2, datatest1.HumidityRatio, datatest1.Occupancy,...
 'VariableNames', {'Date','Temperature','Humidity','Light','CO2','HumidityRatio','Occupancy'});

tableTest2 = table(datatraining.Date, datatraining.Temperature, datatraining.Humidity,...
 datatraining.Light, datatraining.CO2, datatraining.HumidityRatio, datatraining.Occupancy,...
 'VariableNames', {'Date','Temperature','Humidity','Light','CO2','HumidityRatio','Occupancy'});

tableTrain = table(datatest2.Date, datatest2.Temperature, datatest2.Humidity,...
 datatest2.Light, datatest2.CO2, datatest2.HumidityRatio, datatest2.Occupancy,...
 'VariableNames', {'Date','Temperature','Humidity','Light','CO2','HumidityRatio','Occupancy'});
```

```
29
30      table1 = table(dates, temp, hum, light, co2, humr, occ, ...
31          'VariableNames', {'Date','Temperature','Humidity','Light','
                CO2','HumidityRatio','Occupancy'});
32  end
33
34
35  function datatraining = getDataset(filename, startRow, endRow)
36  %IMPORTFILE Import numeric data from a text file as a matrix.
37  %   DATATRAINING = IMPORTFILE(FILENAME) Reads data from text file
        FILENAME
38  %   for the default selection.
39  %
40  %   DATATRAINING = IMPORTFILE(FILENAME, STARTROW, ENDROW) Reads data
        from
41  %   rows STARTROW through ENDROW of text file FILENAME.
42  %
43  % Example:
44  %   datatraining = importfile('datatraining.txt', 2, 8144);
45  %
46  %    See also TEXTSCAN.
47
48  % Auto-generated by MATLAB on 2017/03/17 14:26:37
49
50  %% Initialize variables.
51  delimiter = ',';
52  if nargin<=2
53      startRow = 2;
54      endRow = inf;
55  end
56
57  %% Format for each line of text:
58  %   column2: datetimes (%{yyyy-MM-dd HH:mm:ss}D)
59  %       column3: double (%f)
60  %   column4: double (%f)
61  %       column5: double (%f)
62  %   column6: double (%f)
63  %       column7: double (%f)
64  %   column8: double (%f)
65  % For more information, see the TEXTSCAN documentation.
66  formatSpec = '%*q%{yyyy-MM-dd HH:mm:ss}D%f%f%f%f%f%f%[^\n\r]';
67
68  %% Open the text file.
69  fileID = fopen(filename,'r');
70
71  %% Read columns of data according to the format.
72  % This call is based on the structure of the file used to generate
        this
```

```
% code. If an error occurs for a different file, try regenerating the code
% from the Import Tool.
dataArray = textscan(fileID, formatSpec, endRow(1)-startRow(1)+1, 'Delimiter', delimiter, 'EmptyValue' ,NaN,'HeaderLines', startRow(1)-1, 'ReturnOnError', false, 'EndOfLine', '\r\n');
for block=2:length(startRow)
    frewind(fileID);
    dataArrayBlock = textscan(fileID, formatSpec, endRow(block)-startRow(block)+1, 'Delimiter', delimiter, 'EmptyValue' ,NaN,'HeaderLines', startRow(block)-1, 'ReturnOnError', false, 'EndOfLine', '\r\n');
    for col=1:length(dataArray)
        dataArray{col} = [dataArray{col};dataArrayBlock{col}];
    end
end

%% Close the text file.
fclose(fileID);

%% Post processing for unimportable data.
% No unimportable data rules were applied during the import, so no post
% processing code is included. To generate code which works for
% unimportable data, select unimportable cells in a file and regenerate the
% script.

%% Create output variable
datatraining = table(dataArray{1:end-1}, 'VariableNames', {'Date','Temperature','Humidity','Light','CO2','HumidityRatio','Occupancy'});

% For code requiring serial dates (datenum) instead of datetime, uncomment
% the following line(s) below to return the imported dates as datenum(s).

% datatraining.Temperature=datenum(datatraining.Temperature);

end
```

B Network creation script

Listing 2: RBFN.m

```
%% Set parameters of the RBF network
% 1. Center selection algorithm - kmeans = 1 for kmeans algorithm; random
% center selection otherwise
% 2. Number of centers/hidden nodes -
% 3. Scaling coefficient for the standard deviation of each hidden layer
% neuron - if coefficient = 1, use maximum distance between centers.
% 4. Input data matrix
% 5. Target class vector
use_kmeans = use_kmeans;
m1 = m1;        % number of centres
std_coef = std_coef;
x = x;
t = t;

%% Train and evaluate Network
% 1. Preprocess inputs
x1 = mapstd(x);
x2 = processpca(x1, 0.02);
inputs = x2;


% 2. Split Data into training and test sets
N = size(x,2); % Calculate number of data points
training_set_part = 75;
perm_idx = randperm(N);
X1 = inputs(:, perm_idx(1:(N*training_set_part/100)));
D1 = t(:, perm_idx(1:(N*training_set_part/100)));
X2 = inputs(:, perm_idx((N*training_set_part/100)+1:end));
D2 = t(:, perm_idx((N*training_set_part/100)+1:end));


% 3. Select centers using either Kmeans algorithm or randomly select points from data set.
if use_kmeans
    [~,c] = kmeans(X1',m1);      % find m1 points as centres
    c = c';
else
    N = size(X1,2);              % calculate number of data points
    sample = randsample(N,m1);   % generate m1 random numbers
    c = X1(:,sample');           % sample m1 points from N as centres
end;


```

```
% 4. Calculate the distance from each of these centres to all of these data points.
% Use the function pdist() to generate a matrix of distances, then select that part of
% the matrix containing the distances required.
N = size(X1,2);  % calculate number of data points
points = [X1 c]; % append the centres to the data
dist = squareform(pdist(points')); % calculate the distance matrix, from
                                   % every point or centre to
                                   % every other point or centre
distance = dist(1:N,N+1:N+m1); % extract the section with distances
                               % from points to centres
dc = dist(N+1:end,N+1:end); % extract the section with distances
                            % from centre to centre


% 5. Find the maximum centre to centre distance times a coeeficient to use as the
% standard deviation of the basis functions.
% The matrix  dc  will contain the distance from every centre (in rows) to every other centre (in
% columns).
dmax = std_coef * max(max(dc)); % find the maximum of these

% 6. Calculate the matrix F composed of the individual basis functions responses to each
% data point.
% Take 'dmax' as an estimate of the standard deviation of the RBF
% nodes.
% The response from each RBF node to each data point depends on the distance
% between the node centre and the data point. The operation dist.^2 squares all these
% distances. The calculation of F is the exp() function applied to each element of the matrix.
F = exp((-m1/(dmax*dmax))*(distance.^2)); % using these distances, calculate the
                                         % response from each
                                            centre


% 7. Using the pseudo-inverse of F and the target data, find the weights.
pF = pinv(F); % calculate the pseudoinverse of F
W = pF * D1'; % use pF instead of F-1 to calculate the weights
              % required to give the outputs D


```

```
% 8. Check how well the network classifies the training data
% Compare the calculated and desired outputs. Apply a threshold on the
% output to turn the low responses into a no classification.
Dcalc = F*W;              % calculated outputs based on F and W
Dclass = Dcalc > 0.5;     % and compare to find the bigger output
MSE1 = sum((D1 - Dcalc').^2)/numel(D1);


% 9. Evaluate performance on a separate set of data.
% Use the same basis functions (same number, centres, standard
% deviation) and weights, but recalculate the distances and therefore F.
% Calculate F but using the same centres
% and dmax. This means calculating a new distance matrix for X2
N = size(X2,2);           % calculate number of data points
points = [X2 c];          % append the centres to the data
dist = squareform(pdist(points')); % calculate the distance matrix, from
                                   % every point or centre to
                                   % every other point or centre
dist = dist(1:N,N+1:N+m1); % extract the section with distances
                           % from points to centres


% 10. Once you have the distance matrix, calculate the new F matrix using the same expression, and
% then calculate classes using the same W as before:
F2 = exp((-m1/(dmax*dmax))*(dist.*dist)); % using new distances, calculate F2
Dcalc = F2*W;             % calculated outputs based on F2 and W
Dthresh = Dcalc > 0.5;    % and compare
MSE2 = sum((D2 - Dcalc').^2)/numel(D2);
```

C Network experimentation script

Listing 3: runRBFN.m

```
use_kmeans = 1;
nodes = [20];
std_coefitients = [0.8];
reps = 1;

% Create structure to hold mean training results
s = length(nodes) * length(std_coefitients);
mtrr.mse = zeros(1, s);
mtrr.std_mse = zeros(1, s);
mtrr.err0 = zeros(1, s);
mtrr.std_err0 = zeros(1, s);
mtrr.err1 = zeros(1, s);
mtrr.std_err1 = zeros(1, s);
mtrr.err_diff = zeros(1, s);
mtrr.std_err_diff = zeros(1, s);
mtrr.nodes = nodes;
mtrr.reps = reps;
mtrr.std_coefitients = std_coefitients;

% Create structure to hold mean test results
mtr.mse = zeros(1, s);
mtr.std_mse = zeros(1, s);
mtr.err0 = zeros(1, s);
mtr.std_err0 = zeros(1, s);
mtr.err1 = zeros(1, s);
mtr.std_err1 = zeros(1, s);
mtr.err_diff = zeros(1, s);
mtr.std_err_diff = zeros(1, s);
mtr.nodes = nodes;
mtr.reps = reps;
mtr.std_coefitients = std_coefitients;

% Create structure to hold intermediate training results
train_res.mse = zeros(1, reps);
train_res.err0 = zeros(1, reps);
train_res.err1 = zeros(1, reps);
train_res.err_diff = zeros(1, reps);

% Create structure to hold intermediate test results
test_res.mse = zeros(1, reps);
test_res.err0 = zeros(1, reps);
test_res.err1 = zeros(1, reps);
test_res.err_diff = zeros(1, reps);

count = 1;

```

```
47 for m1 = nodes
48 for std_coef = std_coefitients
49
50     fprintf('Hidden Nodes = %i; std coefficient = %5.3f\n', m1,
       std_coef)
51
52     for i=1:reps
53
54         rbnet % Create, train and evalueate network
55
56         temp = D1-Dclass';
57         train_res.mse(i) = MSE1;
58         train_res.err0(i) = sum(temp(temp>0))*100/numel(temp);
59         train_res.err1(i) = abs(sum(temp(temp<0)))*100/numel(temp);
60         train_res.err_diff(i) = abs(train_res.err0(i) - train_res.
           err1(i));
61
62         temp = D2-Dthresh';
63         test_res.mse(i) = MSE2;
64         test_res.err0(i) = sum(temp(temp>0))*100/numel(temp);
65         test_res.err1(i) = abs(sum(temp(temp<0)))*100/numel(temp);
66         test_res.err_diff(i) = abs(test_res.err0(i) - test_res.err1(
           i));
67
68         plotconfusion(D1, Dclass', 'Training', D2, Dthresh', 'Test')
           ;
69         fprintf('MSE1 = %6.4f; MSE2 = %6.4f; Err0 = %4.2f; Err1 =
           %4.2f \n', MSE1, MSE2, test_res.err0(i), test_res.err1(i)
           );
70
71     end;
72
73     mtrr.mse(count) = mean(train_res.mse);
74     mtrr.std_mse(count) = std(train_res.mse);
75     mtrr.err0(count) = mean(train_res.err0);
76     mtrr.std_err0(count) = std(train_res.err0);
77     mtrr.err1(count) = mean(train_res.err1);
78     mtrr.std_err1(count) = std(train_res.err1);
79     mtrr.err_diff(count) = mean(train_res.err_diff);
80     mtrr.std_err_diff(count) = std(train_res.err_diff);
81
82     mtr.mse(count) = mean(test_res.mse);
83     mtr.std_mse(count) = std(test_res.mse);
84     mtr.err0(count) = mean(test_res.err0);
85     mtr.std_err0(count) = std(test_res.err0);
86     mtr.err1(count) = mean(test_res.err1);
87     mtr.std_err1(count) = std(test_res.err1);
88     mtr.err_diff(count) = mean(test_res.err_diff);
89     mtr.std_err_diff(count) = std(test_res.err_diff);
```

```
90
91      count = count+1;
92  end;
93  end;
94  clear temp s count
```

D Transformation settings

xmean	xstd	xrows	yrows	ymean	ystd	gain	xoffset	nochange
20.9062	1.0553	5	5	0	1	0.94758	20.9062	0
27.6559	4.9822					0.20072	27.6559	
130.7566	210.4309					0.0047522	130.7566	
690.5533	311.2013					0.0032134	690.5533	
0.0042283	0.00076787					1302.3054	0.0042283	

Table 5: mapstd transformation settings

properties	transform					inverseTransform			
name = processpca	-0.37955	-0.39547	-0.40712	-0.49929	-0.53339	-0.37955	0.52236	0.21998	-0.69077
xrows = 5	0.52236	-0.58823	0.4643	0.10901	-0.392	-0.39547	-0.58823	0.1784	0.065158
maxfrac = 0.02	0.21998	0.1784	0.37132	-0.85446	0.22761	-0.40712	0.4643	0.37132	0.6934
yrows = 4	-0.69077	0.065158	0.6934	0.091391	-0.17156	-0.49929	0.10901	-0.85446	0.091391
nochange = 0						-0.53339	-0.392	0.22761	-0.17156

Table 6: processpca transformation settings

E Results from experimentation

mse	std mse	err0	std err0	err1	std err1	err diff	std err diff	nodes
0.059716	0.0093742	5.0856	0.67757	2.0409	0.24889	3.0447	0.74965	5
0.033823	0.0040718	2.8619	0.94546	1.393	0.46739	1.4689	1.1801	10
0.027771	0.0047367	1.9767	0.86396	1.1128	0.12229	1.0233	0.64423	15
0.020121	0.0019786	0.79767	0.30993	1.1089	0.15014	0.43969	0.22506	20
0.017575	0.0018574	0.67899	0.27136	0.99416	0.048918	0.393	0.081413	25
0.017462	0.0021756	0.56226	0.22001	1.0798	0.14909	0.51751	0.28567	30
0.016336	0.0010998	0.48249	0.093888	1.0019	0.07632	0.51946	0.13605	40
0.01498	0.0012328	0.4358	0.069967	0.91829	0.14697	0.48249	0.18813	50
0.016253	0.0022396	0.41634	0.18093	1.0914	0.16341	0.6751	0.19107	60

Table 7: Test results for $m = 5, 15, 20, 25, 30, 40, 50, 60$ and $\epsilon = 1$, 10 runs

mse	std mse	err0	std err0	err1	std err1	err diff	std err diff	nodes
0.058733	0.0091118	5.0564	0.63631	1.9514	0.24993	3.1051	0.74387	5
0.033152	0.0038992	2.7808	0.81748	1.3437	0.35573	1.4371	0.93089	10
0.027671	0.0057495	2.0104	0.9814	1.0837	0.14054	1.0499	0.78676	15
0.020522	0.0023281	0.90856	0.408	1.1291	0.076455	0.47471	0.10834	20
0.017991	0.001829	0.65499	0.3314	1.0772	0.035316	0.51686	0.055958	25
0.017054	0.0012634	0.52529	0.16249	1.035	0.094582	0.50973	0.18991	30
0.016236	0.00054387	0.46498	0.073118	1.0156	0.064793	0.55058	0.10566	40
0.016195	0.0012942	0.48768	0.094859	1.0875	0.092021	0.59987	0.14418	50
0.014741	0.00090728	0.40856	0.094127	0.98184	0.056304	0.57328	0.098266	60

Table 8: Train results for $m = 5, 15, 20, 25, 30, 40, 50, 60$ and $\epsilon = 1$, 10 runs

mse	std mse	err0	std err0	err1	std err1	err diff	std err diff	std coef
0.038798	0.016715	3.7802	2.532	0.90856	0.11046	2.8716	2.589	0.6
0.025469	0.0039952	1.6809	0.67949	0.99805	0.20344	0.75681	0.70723	0.7
0.020859	0.0011533	0.95914	0.12939	1.0992	0.12245	0.19455	0.18204	0.8
0.0202	0.0034591	1.0214	0.55894	1.0136	0.22342	0.55642	0.43554	0.9
0.021771	0.0043611	0.99805	0.70792	1.0895	0.16786	0.72179	0.32869	1
0.020238	0.0037262	0.86965	0.46231	1.0486	0.16872	0.35409	0.23162	1.1
0.017267	0.0013571	0.42607	0.13555	1.1381	0.11129	0.71206	0.1527	1.2

Table 9: Test results for $m = 20$ and $\epsilon = 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2$, 10 runs

mse	std mse	err0	std err0	err1	std err1	err diff	std err diff	std coef
0.039359	0.016031	3.8301	2.5963	0.93061	0.19077	2.8995	2.7467	0.6
0.025842	0.0046411	1.7354	0.80573	1.0467	0.13525	0.79507	0.7904	0.7
0.021243	0.0018937	0.98249	0.18012	1.1226	0.07359	0.17899	0.1276	0.8
0.021095	0.0042415	1.0311	0.57676	1.085	0.066911	0.50519	0.29447	0.9
0.021342	0.0055074	1.0266	0.79916	1.0447	0.072438	0.70687	0.37628	1
0.020281	0.0027321	0.7847	0.46976	1.1336	0.082926	0.51881	0.28693	1.1
0.017302	0.00069705	0.38975	0.10109	1.1537	0.047405	0.76394	0.10585	1.2

Table 10: Train results for $m = 20$ and $\epsilon = 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2$, 10 runs