SCHOOL OF COMPUTER SCIENCE | THE UNIVERSITY OF MANCHESTER

# Handwritten Digit Recognition

Leonardo Claudio de Paula e Silva
October 28, 2015

# SUMMARY

# 0 - PREPARATION

The aim of this part is simple, it aims to prepare the student and the framework with the necessary information. The Matlab command **load** loads data from file (postaldata.mat in these case). This will load **data** and **labels** to the memory. Data is a matrix[5000x256] and contains images of the US postal office; label contains a array of integers identifying wich value is the image e.g. 0,1,2,..,9.

To test and set up the environment a simple code is made (Listing 1) and the result is shown in figure 1.1. The Matlab commands **close all** and **clear all** are not necessary, there are just a good practice to close all open windows and clear the environment avoiding preexisting variables to compromise the generated data and future training and testing. The following code extracts the *(i*500)+10* row and show the digit in it. [$i \in 0,1,..,9$]

```
close all;
clear all;

load postaldata

for i=0:9
    d = data((i*500)+10,:);
    showdigit(d);
end
```

Listing 1: Part 0 implementation

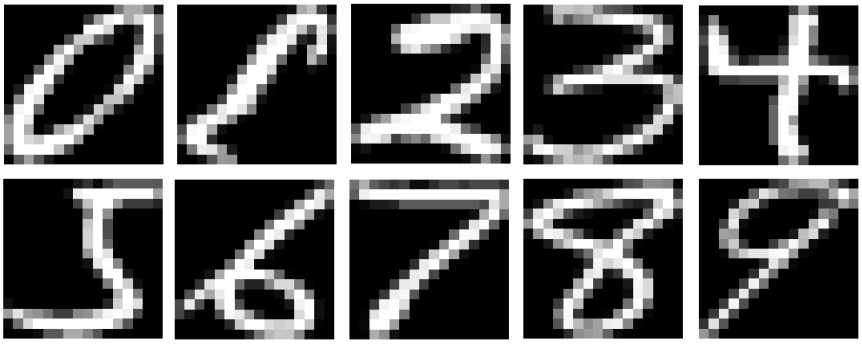

Figure 0.1: Samples gicen in the postaldata.mat file

# 1 - TRAINING AND TESTING

Using the k-nearest neighbors algorithm (**KNN**) and the data provided by the postaldata.mat the aim of this part is to construct a dataset with '3', '8' and '6' digits, classify it using the KNN algorithm, plot the testing accuracy and average accuracy graphs.

It is worth explain the KNN algorithm before the discussion about the procedures.
The KNN algorithm is one of the simplest machine learning algorithm. It is a pattern recognition algorithm used for classification and regression. To the handwritten digit recognition problem just the classification is important. The KNN input consist of the number of neighbors to compare with the analysed element, this number is called **k** and it returns the class in which the analysed element better matches. Basically, the algorithm compare the class of the k nearest neighbors for the element and classify the element as the class the most appeared. As k gets bigger the number os neighbors to compare increases the consistency increases since it reduces the effect of noises on the classification, however the boundaries between classes becomes less distinct. When k is equal to one it is called the nearest neighbor algorithm and it probably leads to a over-fitting.

To define the value of k that best fits the problem is not straightforward, actually, as the data is generated randomly the best value for k may not be the same at all interaction, hence it is needed an average accuracy to obtain the best value.

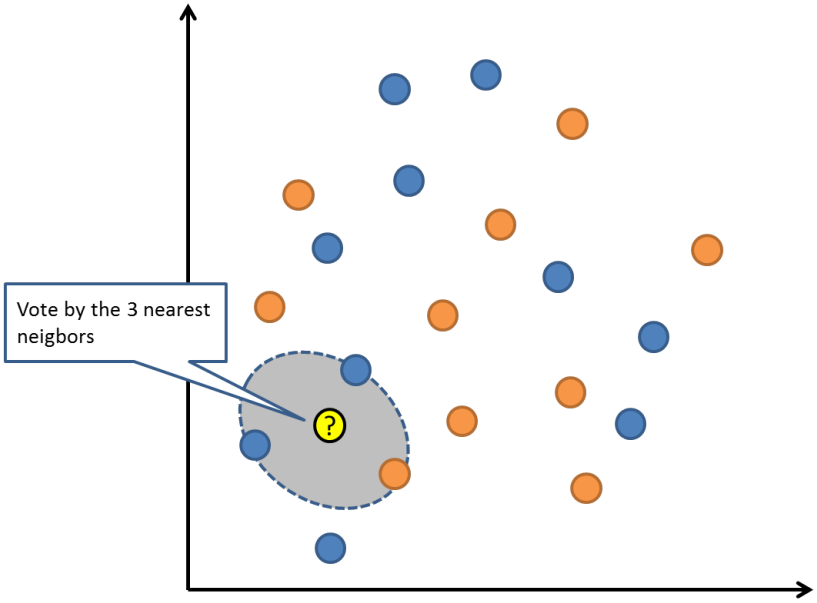


Figure 1.1: KNN Model

## 1.1 Constructing Training Dataset

In order to construct the dataset and facilitate the future use of the dataset a **gatherdata** function (Listing 2) was made *(can be found in gatherdata.m)*. Therefore to analyse another another samples is straightforward.

```
% The function gathers data from postaldata and returns the matrix and the
% labels. It uses the '3', '6' and '8' digits.
function [D, labels] = gatherdata()
    load postaldata

    %% Gathering data to analyse
    D = zeros(200,256);
    labels = zeros(200,1);
    r = randi([1001, 4000],1,1500);
    count = 0;
    loop = 0;

    while(count < 200)
        if((r(loop+1) > 1000 && r(loop+1) < 1501))
            D(count+1,:) = data(r(loop+1),:);
            labels(count+1) = 3;
            count = count + 1;
        elseif (r(loop+1) > 3500 && r(loop+1) < 4001)
            D(count+1,:) = data(r(loop+1),:);
            labels(count+1) = 8;
            count = count + 1;
        elseif (r(loop+1) > 2500 && r(loop+1) < 3001)
            D(count+1,:) = data(r(loop+1),:);
            labels(count+1) = 6;
            count = count + 1;
        end
        loop = loop+1;
    end
end
```

Listing 2: Gatherdata function

The function returns a matrix D[200x256] and a array[200x1]. The matrix contains 200 '3','6' and '8' digits samples and the array is filled with integers identifying the number in the respective row of the matrix. To accomplish that a while loop is created with a if statement for each number (3,6 and 8), if the randomly generated row number falls in any of the range for the 3, 6 or 8 the row is stored in the D matrix, the label in labels and count is increased by one. the loop stops when count is equals to 200.

## 1.2 KNN Training classification and Accuracy Report

To classify the accurcacy using the KNN algorithm it was used the professors **knearest.m** file. Varying the ***k*** and comparing the accuracy. the accuracy is measured my comparing the labels from the vectors. For that it was used the **accuracy** function *(can be found in gatherdata.m)*.

```
% numK is the range of k to test eg. numK = 3 will test for k = 1,2 and 3.
% test is the test data
% truelabelsTest is the labels of the test data
% data is the training data
% truelabels is the labels of the training data
% acc is a numK size vector containing the accuracy using k neighbors.
function acc = accuracy(numK, test, truelabelsTest, data, truelabels)
    examples = 200;
    acc = zeros(numK,1);
    for k=1:numK
        for i = 1:examples
            Y = knearest(k, test(i,:), data, truelabels);
            if Y == truelabelsTest(i)
                acc(k) = acc(k) + 1;
            end
        end
        acc(k) = acc(k)/examples;
    end
end
```

Listing 3: Accuracy function

```
k = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,...
30,31];
acc = accuracy(31, D, myLabels, D, myLabels);
figure;
p1 = plot(k, acc,'-o');
title('Training Accuracy varying k')
xlabel('k')
ylabel('accuracy')

%% Gathering another 200 data
[D2 myLabels2] = gatherdata();

%% Verifing the accuracy using KNN for the testing data
acc = accuracy(31, D2, myLabels2, D, myLabels);
figure;
p2 = plot(k, acc,'r-x');
title('Testing Accuracy varying k')
xlabel('k')
ylabel('accuracy')
```

Listing 4: Verifying accuracy using KNN algorithm comparing the training set with the testing set

After running those scripts we got the following graphs shown in Figure 1.1:

## 1.3 KNN Testing classification and Accuracy Report

The methodology and procedures used here are the same used in the subsection 1.1 the difference is in the second and third parameters of accuracy. Now the test is between a randomly generated matrix against the previous training matrix.

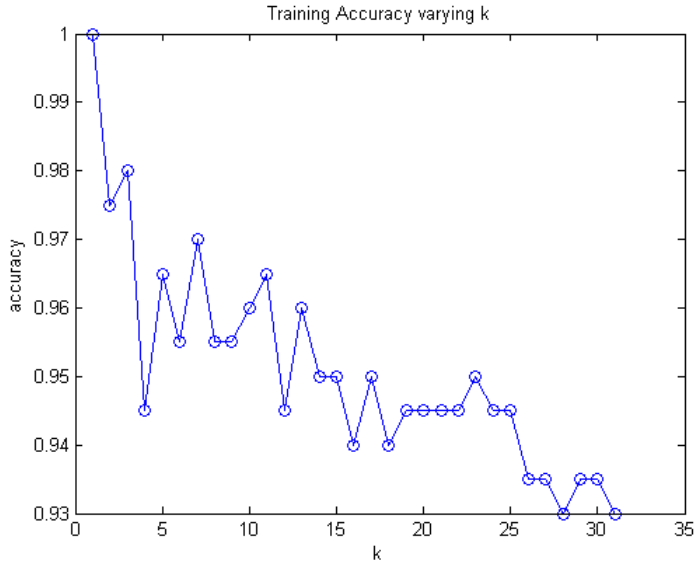


Figure 1.2: Training Accuracy Graph varying the k value

```
%% Gathering another 200 data
[D2 myLabels2] = gatherdata();

%% Verifing the accuracy using KNN for the testing data
acc = accuracy(31, D2, myLabels2, D, myLabels);
figure;
p2 = plot(k, acc,'r-x');
title('Testing Accuracy varying k')
xlabel('k')
ylabel('accuracy')
```

Listing 5: Verifying accuracy using KNN algorithm comparing the training set with the testing set

## 1.4 Average KNN Testing classification and Accuracy Report

In order to take a deep lock in the data and understand how the randomly picking of rows affect the result an average accuracy is measured from a total of 10 interactions. The methodology and procedures are the same presented in subsections 1.2 and 1.3.

```
%% Calculating Average for training and testing
figure;
averageTraining = average(0);
e = std(averageTraining)*ones(size(averageTraining));
p4 = errorbar(averageTraining, e);
title('Training Average Accuracy varying k')
xlabel('k')
ylabel('accuracy')
```

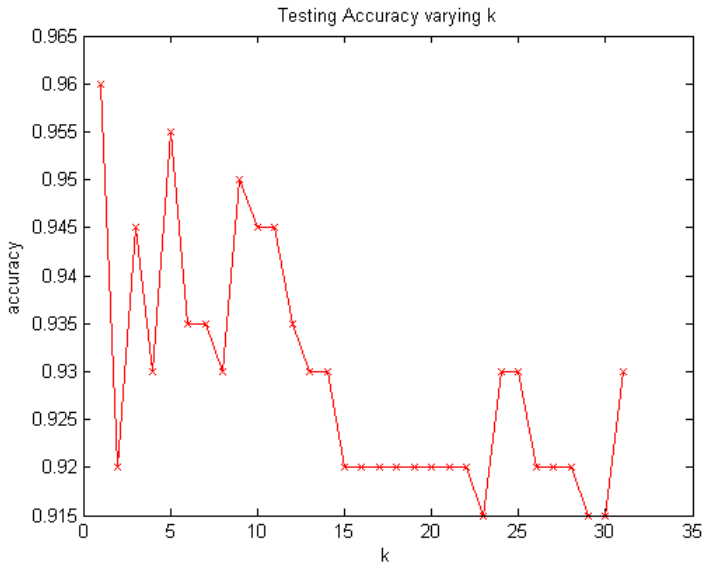


Figure 1.3: Testing Accuracy Graph varying the k value

```
9
10 figure;
11 averageTesting = average(1);
12 e = std(averageTesting)*ones(size(averageTesting));
13 p5 = errorbar(averageTesting, e, 'r-x');
14 title('Testing Average Accuracy varying k')
15 xlabel('k')
16 ylabel('accuracy')
```

Listing 6: Verifying the average accuracy using KNN algorithm for training and testing

For that the results are shown in Figures 1.3 and 1.4.

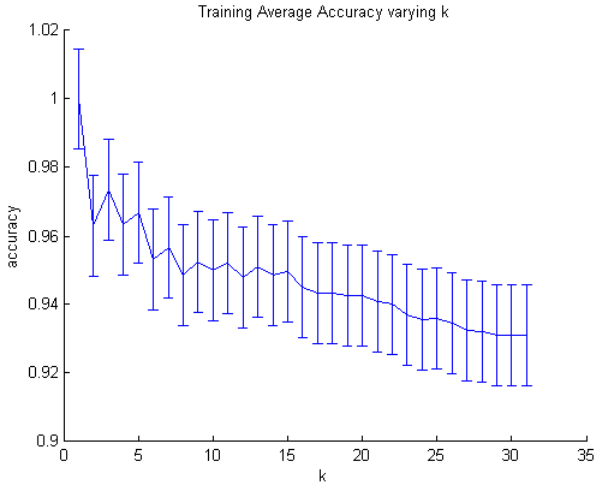


Figure 1.4: Average Training Accuracy Graph varying the k value


Figure 1.5: Average Training Accuracy Graph varying the k value

## 2 - Feature Extractors for Digit Recognition

The aim of this part is to edit the **extractfeatures.m** file to get a lower dimension representation of the features contained in postaldata trying to get a higher score. Score is a methodology to compare the amount of information (memory) necessary to get a accuracy. If the score is higher it means that less memory is necessary to obtain a certain accuracy which is the goal.

Therefore, a simple score definition is:

$$score = \frac{accuracy \quad on \quad benchmark \quad set}{number \quad of \quad features \quad used} \tag{2.1}$$

Changing the extractfeatures file to get a 128 featured result (Listing 7):

As it seems, the accuracy goes down to 0.30 with 16 features counting the number of gaps in each row. Therefore, if I use less features my accuracy is lower but the amount of data to process and to store is much lower too.

```
%
% EXTRACTFEATURES( digdata )
%
% Arguments: 'digdata' is a vector, size 256.
%
% Process the supplied vector to generate a lower dimensional
% feature vector, to be used in a learning algorithm.
%
% The returned vector is the sum of gaps in each of the 16 columns.
%
%
%
function dextracted = extractfeatures( data )

        gap = zeros(16,1);
        tmp = char(zeros(1,256));
        % Get vector 1x256 and change any number > 0 to '|' and every 0 to
        % ' '.
        for i = 1:256
            if(data(i) > 0 )
                tmp(i) = '|';
            else tmp(i) = ' ';
            end
        end
        % Transform the vector into a matrix 16x16 and rotate it to be
        % visualy equal to the number seem by showdigit.
        tmp = vec2mat(tmp,16);
        tmp = rot90(tmp,-1);
        tmp = fliplr(tmp);

        % A 'gap' is the difference of the black background to the white
        % ink. There fore, when there is a change of character there is a
        % gap.
        for i = 1:16
```

```
            for j = 2:16
                if(tmp(i,j) ~= tmp(i,j-1))
                    gap(i) = gap(i) + 1;
                end
            end
        end
        % As gap is a column the transpose of it is a row.
        gap = gap.';
        dextracted = gap;
end
```

Listing 7: Extractfeatures function

```
close all
clear all

%% PART 2 - Feature Extractors for Digit Recognition
%% Calculate 256 Features Score
% Test accuracy using k = 1 to k = 31.
[train256, trainlabels256] = gatherdata();
[test256, testlabels256] = gatherdata();
acc256 = accuracy(31, test256, testlabels256, train256, trainlabels256);
score256 = (acc256./256)*100;

%% Calculate 16 features scores
% Gather data to change number of features
% Training data
[train16, trainlabels16] = gatherdata();
dextracted = zeros(200,16);
for i = 1:200
    dextracted(i,:) = extractfeatures( train16(i,:) );
end

% Testing data
[test16, testlabels16] = gatherdata();
testextracted = zeros(200,16);
for i = 1:200
    testextracted(i,:) = extractfeatures( test16(i,:) );
end

acc16 = accuracy(31, testextracted, testlabels256, dextracted, trainlabels16);
score16 = (acc16./16)*100;

%% Plotting data
k = 1:31;
figure;
plot(k, score256,'b-o');
title('256 Featured Score varying k')
xlabel('k')
ylabel('score')
figure;
plot(k, score16,'r-x');
title('16 Featured Score varying k')
xlabel('k')
```

```
ylabel('score')

%% confusion Matrix
% Testing for k = 10
samples = 200;
predicted = zeros(1,200);
% 256 Features
for i = 1:samples
    predicted(i) = knearest(10, test256(i,:), train256, trainlabels256);
end
CM256 = confusionmatrix(trainlabels256, predicted, 200)

% 16 Features
for i = 1:samples
    predicted(i) = knearest(10, test16(i,:), train16, trainlabels16);
end
CM16 = confusionmatrix(trainlabels16, predicted, 200)
```

Listing 8: Part 2 implementation

To create a confusion matrix is straightforward using a nested if-else statement. To make the code simpler it was made a function **confusionmatrix** that takes three parameters as arguments: trueLabels (original labels), testLabels (to be compared with the original ones) and size (the number of elements to be compared).

```
% Size is the number of elements in the vector.
% Return the confuion matrix 3x3
function CM = confusionmatrix(trueLabels, testLabels, size)

% Confusion matrix
CM = zeros(3,3);

for i=1:size
    actual = trueLabels(i);
    predicted = testLabels(i);

    if(actual == 3)
        if(predicted == 3)
            CM(1,1) = CM(1,1)+1;
        elseif(predicted == 6)
            CM(1,2) = CM(1,2)+1;
        elseif(predicted == 8)
            CM(1,3) = CM(1,3)+1;
        end
    elseif(actual == 6)
        if(predicted == 3)
            CM(2,1) = CM(2,1)+1;
        elseif(predicted == 6)
            CM(2,2) = CM(2,2)+1;
        elseif(predicted == 8)
            CM(2,3) = CM(2,3)+1;
        end
    elseif(actual == 8)
        if(predicted == 3)
```

```
            CM(3,1) = CM(3,1)+1;
        elseif(predicted == 6)
            CM(3,2) = CM(3,2)+1;
        elseif(predicted == 8)
            CM(3,3) = CM(3,3)+1;
        end
    end
end

end
```

Listing 9: Confusionmatrix function

$$CM = \begin{bmatrix} 27 & 19 & 19 \\ 30 & 24 & 17 \\ 24 & 29 & 11 \end{bmatrix} \tag{2.2}$$

$$CM = \begin{bmatrix} 24 & 24 & 18 \\ 27 & 24 & 17 \\ 30 & 20 & 16 \end{bmatrix} \tag{2.3}$$

Table of confusion for the '3' digit using 256 features. The methodology to create the confusion table for the '6' and '8' digit is the same.

- 27 true positives
  - Actual '3' digits that were correctly classified as '3' digit
- 54 false positive
  - '6's and '8's that were incorrectly labeled as '3'
- 38 false negative
  - '3's that were incorrectly marked as '6's or '8's
- 81 true negatives
  - all the remaining digits correctly classified as non-'3'

Table of confusion for the '3' digit using 16 features. The methodology to create the confusion table for the '6' and '8' digit is the same.

- 24 true positives
- 57 false positive
- 42 false negative
- 77 true negatives

Analysing the whole picture for the 256 features, there is 27, 24 and 11 true positives for '3','6' and '8' digits respectively. Therefore the '8' digit has the greatest error. For the 16 features data we have 24, 24 and 16. Once again, the '8' digit has the greatest error.

# 3 - PERCEPTRONS

The Perceptron is a machine learning supervised algorithm to classify 2 classes. It is a linear classifier using weights to make predictions about the class. Basicaly, the net is calculated by

$$Net = \sum \omega_i x_i + bias \tag{3.1}$$

Then, the net value is evaluated by the sigmoid function to define 0 or 1 (binary classifier) and the wight's values are updated. The process is repeated as many times as wanted. More interactions leads to better classification but demands more time.

The KNN algorithm is a instant training algorithm, in that case there is no training there is just a comparison between the element and the k nearest neighbors. Despite of this great advantage of a instant result the amount of memory that is needed is huge. It is necessary to store the whole dataset in memory in order to use this method. When using the perceptrons, it is necessary to really train the machine, so there is a time cost, however it is not necessary to store the dataset once the machine is trained, hence there is gain in memory.

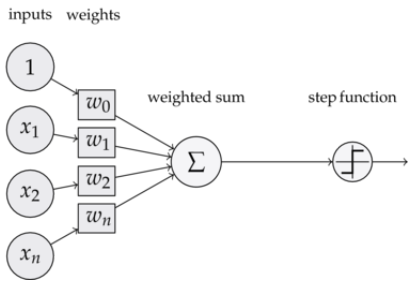


Figure 3.1: Perceptron Model

To implement the perceptron it was necessary to modify the gatherdata function because it was classifying 3 classes ('3','6' and '8'). Hence, a gatherdata2 function was created just removing the '6' digit part, as follows:

```
% The function gathers data from postaldata and returns the matrix and the
% labels. It uses the '3' and '8' digits.
function [D, labels] = gatherdata2()
    load postaldata

    %% Gathering data to analyse
    D = zeros(200,256);
    labels = zeros(200,1);
    r = randi([1001, 4000],1,1500);
    count = 0;
    loop = 0;

    while(count < 200)
```

```
        if((r(loop+1) > 1000 && r(loop+1) < 1501))
            D(count+1,:) = data(r(loop+1),:);
            labels(count+1) = 3;
            count = count + 1;
        elseif (r(loop+1) > 3500 && r(loop+1) < 4001)
            D(count+1,:) = data(r(loop+1),:);
            labels(count+1) = 8;
            count = count + 1;
        end
        loop = loop+1;
    end
end
```

Listing 10: Gatherdata2 function

```
close all
clear all

%% PART 4 - PERCEPTRON
%% Training Perceptron
interactions = 20;
accPerceptron = zeros(1,interactions);
for in = 1:interactions
    [input output] = gatherdata2();

    samples = 200;
    bias = -1;
    coeff = 0.75;
    weights = -1*2.*rand(256+1,1);
    int = in;

    % Binary classifier - sigmoid function turns normalizes to 0-1
    % 3 = 0 and 8 = 1
    for i = 1:200
        if(output(i) == 3)
            output(i) = 0;
        else output(i) = 1;
        end
    end

    for i = 1:int
        out = zeros(samples,1);
        for j = 1:samples
            y = bias*weights(1,1);
            for b = 1:256
                y = y + input(j,b)*weights(b+1,1);
            end
            out(j) = 1/(1+exp(-y));
            delta = output(j)-out(j);
            weights(1,1) = weights(1,1)+coeff*bias*delta;
            for b = 1:256
                weights(b+1,1) = weights(b+1,1)+coeff*input(j,b)*delta;
            end
        end
```

```
    end

    %% Accuracy

    % gathering test data
    [testdata, testlabels] = gatherdata2();

    % 3 = 0 and 8 = 1
    for i = 1:200
        if(testlabels(i) == 3)
            testlabels(i) = 0;
        else testlabels(i) = 1;
        end
    end

    % Testing data
    for j = 1:samples
        y = bias*weights(1,1);
        for b = 1:256
            y = y + testdata(j,b)*weights(b+1,1);
        end
        out(j) = 1/(1+exp(-y));
    end


    % Accuracy calculation
    for i = 1:200
        if(testlabels(i) == out(i))
            accPerceptron(in) = accPerceptron(in)+1;
        end
    end

    accPerceptron(1,in) = accPerceptron(in)/200;
end

figure;
plot(1:interactions, accPerceptron,'b-o');
title('Perceptron Accuracy varying the number of interactions')
xlabel('interactions')
ylabel('accuracy')
```

Listing 11: Perceptron implementation

Now the implementation follows the description above. There is 256 inputs, therefore 256 weights and only one perceptron. In addition, there are 200 samples to test. It was tested the accuracy of the perceptron varying the number of interactions (1-20).

Therefore, both algorithm are great and one may be more suitable than another depending on de problem. In the case of the handwritten recognition, now it is possible to compare the accuracy of these methods:

- *KNN* **94.5%** *Max accuracy value for the testing average.* $k = 9$
- *Perceptrons* **96.20%** *average of 20 interactions*

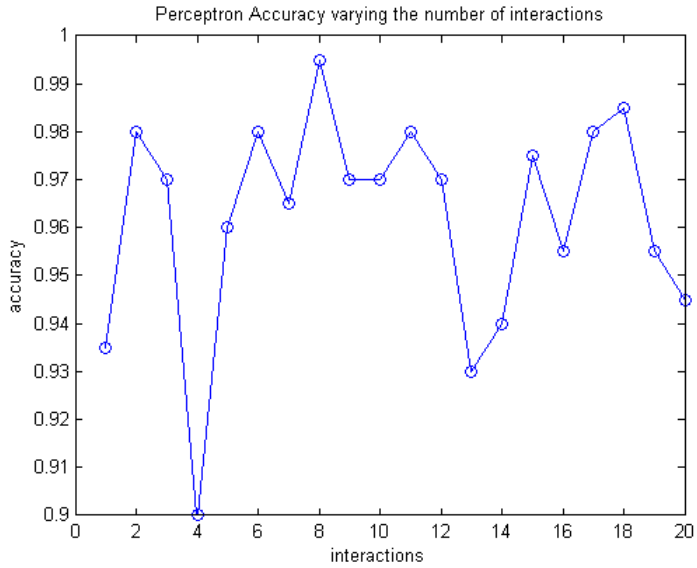


Figure 3.2: Perceptron Accuracy varying the number of interactions

# LIST OF FIGURES