

# Classification with recursive vectors and scalars

Gonzalo Reynaga García  
February 2019

This article is an example of using recursive vectors to classify 4 types of images with only 4 test samples and other example to solve “xor” problem. The procedure could provide ideas how solve the classification problem in different ways including to scalars.

## Notation

The notation in this document to define recursive vector of scalars

$$\begin{aligned}[1] &= [1\ 0\ 0\ \dots\ 0\ 0] \\ [1_1] &= [0\ 1\ 0\ \dots\ 0\ 0] \\ [1_2] &= [0\ 0\ 1\ \dots\ 0\ 0]\end{aligned}$$

“ ^ ” represent recursive vector multiplication.

## Classification example with recursive vectors

In this problem we provide 4 test images to test and then compare with others. This is intended to be an example how to use recursive vectors(revectors) for this kind of problem.

The objective is to find a function that give us a value of what type of image correspond. We are going to use 4 images to test so, we have 4 inputs and we set desired output for that inputs creating linear system of equations.

So, the images will be divided in four pieces reducing the dimension of the vectors.

The function to find in this case will be:

$$Y = A_0 \wedge X_0 + A_1 \wedge X_1 + A_2 \wedge X_2 + A_3 \wedge X_3 \quad \text{Equation.1}$$

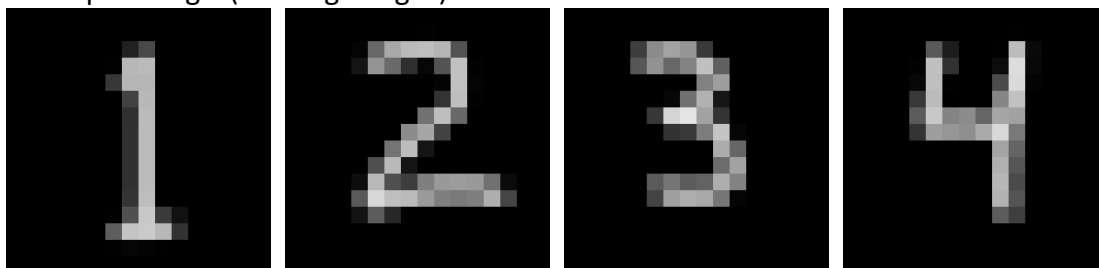
Where:

$Y$  is the output

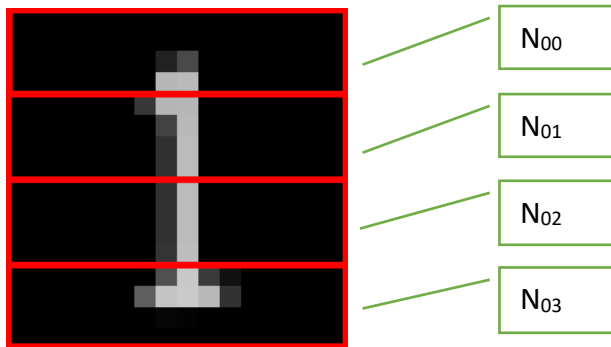
$A_k$  are unknown variables to find

$X_k$  it is the input image(the images are divided in 4 parts).

The input images(training images):



The image is divided as:



And replace the  $X_k$  values to fill the linear equation.





$$\begin{bmatrix} [1] \\ [1_1] \\ [1_2] \\ [1_3] \end{bmatrix} = \begin{bmatrix} N_{00} & N_{01} & N_{02} & N_{03} \\ N_{10} & N_{11} & N_{12} & N_{13} \\ N_{20} & N_{21} & N_{22} & N_{23} \\ N_{30} & N_{31} & N_{32} & N_{33} \end{bmatrix} \wedge \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix}$$

And we solve the for  $A$  like any matrix









$$\begin{bmatrix} N_{00} & N_{01} & N_{02} & N_{03} \\ N_{10} & N_{11} & N_{12} & N_{13} \\ N_{20} & N_{21} & N_{22} & N_{23} \\ N_{30} & N_{31} & N_{32} & N_{33} \end{bmatrix}^{-1} \wedge \begin{bmatrix} [1] \\ [1_1] \\ [1_2] \\ [1_3] \end{bmatrix} = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix}$$

### Testing

When setting the training images as input in the equation 1, we get the expected output.

Input = 	output = <span style="border: 1px solid green; padding: 2px;">[1</span> -1.35924e-09 -2.12221e-09 -8.15746e-10 ...
Input = 	output = [1.35924e-09 <span style="border: 1px solid green; padding: 2px;">1</span> 8.15746e-10 -2.12221e-09 ...
Input = 	output = [2.12221e-09 8.15745e-10 <span style="border: 1px solid green; padding: 2px;">1</span> -1.35924e-09 ...
Input = 	output = [-8.15746e-10 2.12221e-09 1.35924e-09 <span style="border: 1px solid green; padding: 2px;">1</span> ...

### Testing with different images

Input = 	output = <span style="border: 1px solid green; padding: 2px;">0.519613</span> 0.0199563 0.0981575 0.458052 ...
Input = 	output = <span style="border: 1px solid red; padding: 2px;">0.0944424</span> 0.155258 0.234159 0.419556 ...
Input = 	output = 0.227338 <span style="border: 1px solid green; padding: 2px;">0.830017</span> -0.551057 0.0622568 ...
Input = 	output = 0.539572 <span style="border: 1px solid green; padding: 2px;">0.64462</span> -0.223176 -0.00981938 ...
Input = 	output = 0.142507 -0.524978 <span style="border: 1px solid green; padding: 2px;">0.484608</span> 0.439013 ...
Input = 	output = 0.108017 0.468673 <span style="border: 1px solid red; padding: 2px;">0.331925</span> 0.257975...
Input = 	output = 0.726491 0.606634 -0.581886 <span style="border: 1px solid green; padding: 2px;">1.25332</span> ...
Input = 	output = 0.700708 0.77677 0.0989637 <span style="border: 1px solid green; padding: 2px;">0.891672</span> ..

It can be seen taking the maximum value as the selection, it selects right 6 of 8.

Due technical difficulties it wasn't possible to test with bigger images because the method used to invert the matrix with determinants was causing floating point errors. The matrix inverse and recursive vector inverse should be resolved with iteration methods, in this article only shows the procedure. Other options we can test, is splitting the image with overlapping sections.

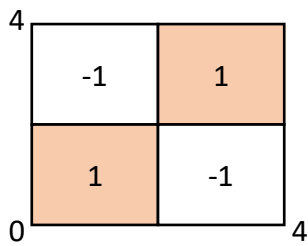
Other option could be not split the image at all and create  $N \times x$  rearranging the order of the vector values (flip vertically, horizontally, conjugate, etc.), because for the math point of view are different vectors and still creating the linear equation.

Reorder the vector allow us, when the  $A$  values are found, to merge equation 1 into a classic matrix vector multiplication  $Y = M \cdot X$ .

The script used for this example can be found in the appendix A.

## Classification with scalars

In this example we are going to try to solve the “xor” problem in a similar way we tried to solve the images classification with just linear functions. The input values are from 0 to 4.



The input are two values and the output is one value.

So, the input could be considered as a complex number  $(v_0 + v_1i)$

If we split the vector, we have  $v_0, v_1$  and use the function  $y = a_0 \cdot x_0 + a_1 \cdot x_1$ , we know is the rotation of a point and doesn't solve the “xor” function.

We are going to try a to find a function:

$$y = a_0x_0 + a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5x_5 + a_6x_6 + a_7x_7 \quad \text{Equation.2}$$

And the input is just 1 vector(with 2 scalars), the we can split each scalar by four by its digits. To simplify, we are going to use binary base and extend the input scalar to 16(has 4 bits in binary) and each digit will be the input of the linear equation.

Example, the input number (3, 2.5) will be transformed into [0,0,1,1, 0,0,1,0.5]

With only 8 inputs to create a square matrix and solve  $a_k$ .

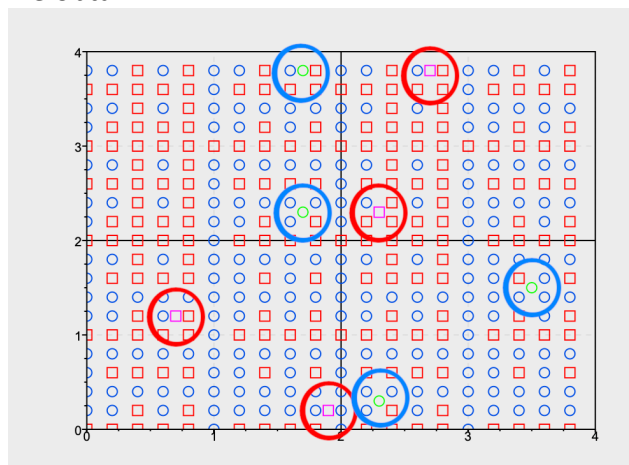
$x_0 = [ 1.7 \ 3.5 \ 2.3 \ 1.7 ]$ ; // for -1

$y_0 = [ 2.3 \ 1.5 \ 0.3 \ 3.8 ]$ ;

$x_1 = [ 2.3 \ 0.7 \ 2.7 \ 1.9 ]$ ; // for 1

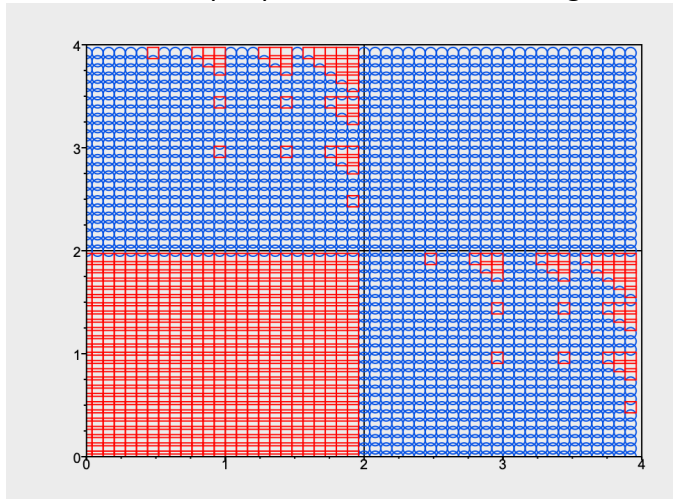
$y_1 = [ 2.3 \ 1.2 \ 3.8 \ 0.2 ]$ ;

we obtain:



The red markers are positive output the blue markers are the negative. We can see the 8 point is not enough for solve the “xor”. However, it is possible to create a function no linear due the input transformation and solve a linear matrix, and the input values (“training” values) has the correct result if we substitute in the equation 2.

If we use 324 input points and use linear regression we obtain:



Still is not an xor, but the shape is closer to xor.

is possible to create the “xor” function adding more  $a_k$  elements?

Continuing splitting the input scalar has inconvenient we need to define at least one point containing each bit, otherwise the matrix will have a column of zeros and will be singular.

The script used for this example can be found in the appendix B.

### Conclusion

In case the recursive vectors, I think is possible to create a binary search using the product as comparison, and then a more specialized comparison until find the result.

From the convolution point of view, I think it is possible to use only one filter to find multiples features, because It only change the filter values order and the signs without changing the value itself and could “compress” the filter information, better than use a filter for each feature.

For the scalar problem, as was suggested with the recursive vectors, we can divide the digits with overlapping digits to avoid singularities. And, if we use the input digits as a recursive vector, we can scramble the digits to create a new input. If you think about it, the polynomial square, cubic, etc. seems a way to scramble the input digits.

I believe the brain doesn't manage numbers, but manage level of intensities of input, and the digits of a number represent its intensities at different levels, that is why I think that manage digits as input is similar as a neuron could work than the whole number in problems like the “xor”.

I hope this article could be interesting and could help provide ideas to solve this kind of problems.

### Contact:

Any questions or suggestions, you can contact me at [gonzaloreynaga@yahoo.com](mailto:gonzaloreynaga@yahoo.com)

## Appendix A

Script file used for image classification example with recursive vectors.

**Note:** In the program menu set the "Math library folder" from "Math-c->preferences..." to be able to find the "import" files and images files.

File: numbers.mc

```
////////////////////////////////////
//vectorize an array of numbers WITHOUT
//imaginary part (scalars were not merged)
//example: vectorize([3 4]) = 3+4i
function vectorize(a)
    vSize = max(size(a)) (0);

    Ra = a(0:(vSize/2-1));
    Ia = a((vSize/2):vSize-1);

    if vSize == 2
        if isArray(a(0)) == 0
            return a(0) + 1i*a(1);
        end
        Ra = Ra(0);
        Ia = Ia(0);
    end

    Ra = vectorize(Ra);
    Ia = vectorize(Ia);
    return [Ra Ia];
end

////////////////////////////////////
//remove the brackets to convert it to an array of numbers.
function unbracket(a)
    vSize = max(size(a)) (0);
    if max(size(a(0))) (0) == 1
        return [real(a(0)) imag(a(0)) real(a(1)) imag(a(1))];
    end
    vectorArray = [];
    for i=0:vSize-1
        vectorArray = vectorArray <-> unbracket(a(i));
    end
    return vectorArray;
end

function Det4x4(m)
    result = m(0,0)^Det3x3(m(1:,1:)) - m(0,1)^Det3x3(m(1:[0 2 3])) ...
            +m(0,2)^Det3x3(m(1:[0 1 3])) - m(0,3)^Det3x3(m(1:[0 1 2]));
    return result;
end

function Det3x3(m)
    result = m(0,0)^m(1,1)^m(2,2) + m(0,1)^m(1,2)^m(2,0) + m(0,2)^m(1,0)^m(2,1) ...
            - m(0,2)^m(1,1)^m(2,0) - m(0,1)^m(1,0)^m(2,2) - m(0,0)^m(1,2)^m(2,1);
    return result;
end

function imgVectorize(file)
    img1 = imread(file);
    img1 = (imgchannel(img1,1)/255);
    s = size(img1) (0);

    imgvec=[];
    for i = 0:$s
        imgvec = imgvec <-> img1(i,:);
    end
    vecsize = max(size(imgvec)) (0)/4;
    imgvec = vectorize(imgvec);

    //split in 4 pieces
    N0 = imgvec(0) (0);
    N1 = imgvec(0) (1);
    N2 = imgvec(1) (0);
    N3 = imgvec(1) (1);
    return [N0 N1 N2 N3 [s vecsize]];
end

function vector2img(v,imSizeWidth)
    F = unbracket(v)*255;
    t0 = F(0:$imSizeWidth);
    for i = 1:$imSizeWidth/4
        idx = ((imSizeWidth)*i):((imSizeWidth)*(i+1)-1);
        t0 = t0 <-> F(idx);
    end
    return t0;
end

clear();
clc();
//Loading images

[N00 N01 N02 N03 [s vector_size]] = imgVectorize("one.png");
[N10 N11 N12 N13 [s vector_size]] = imgVectorize("two.png");
[N20 N21 N22 N23 [s vector_size]] = imgVectorize("three.png");
[N30 N31 N32 N33 [s vector_size]] = imgVectorize("four.png");

////////////////////////////////////
//create the desired results

result_one = zeros(1, vector_size);
result_one(0)=1;
result_one = vectorize(result_one);

result_two = zeros(1, vector_size);
result_two(1)=1;
result_two = vectorize(result_two);

result_three = zeros(1, vector_size);
```

```

result_three(2)=1;
result_three = vectorize(result_three);

result_four = zeros(1, vector_size);
result_four(3)=1;
result_four = vectorize(result_four);
//////////
// The equation we try to solve is:
// | result | = A0*X0 + A1*X1 + A2*X2 + A3*X3
// We have the data, we need to solve the 4x4 matrix to find
// the parameters.

// | result_one | = | N00 N01 N02 N03 | ^ | A0 |
// | result_two | = | N10 N11 N12 N13 | | A1 |
// | result_three | = | N20 N21 N22 N23 | | A2 |
// | result_four | = | N30 N31 N32 N33 | | A3 |

Nmat = [N00 N01 N02 N03;N10 N11 N12 N13;N20 N21 N22 N23;N30 N31 N32 N33];
results=[result_one; result_two; result_three; result_four];

/// <-> symbol is merge horizontally
/// <:> symbol is merge vertically

A0mat = results <-> Nmat(:,1:3);
A1mat = Nmat(:,0) <-> results <-> Nmat(:,2:3);
A2mat = Nmat(:,0:1) <-> results <-> Nmat(:,3);
A3mat = Nmat(:,0:2) <-> results;

A0det = Det4x4(A0mat);
A1det = Det4x4(A1mat);
A2det = Det4x4(A2mat);
A3det = Det4x4(A3mat);
Ndet = inv(Det4x4(Nmat));

A0 = A0det ^ Ndet;
A1 = A1det ^ Ndet;
A2 = A2det ^ Ndet;
A3 = A3det ^ Ndet;
print("end\n");

//////////
R1 = A0^N00 + A1^N01 + A2^N02 + A3^N03;
R2 = A0^N10 + A1^N11 + A2^N12 + A3^N13;
R3 = A0^N20 + A1^N21 + A2^N22 + A3^N23;
R4 = A0^N30 + A1^N31 + A2^N32 + A3^N33;

R1 = unbracket(R1)
R2 = unbracket(R2)
R3 = unbracket(R3)
R4 = unbracket(R4)
print("\ntesting image one\n");

[T0 T1 T2 T3 b] = imgVectorize("one_0.png");
Q10 = A0^T0 + A1^T1 + A2^T2 + A3^T3;
Q10 = unbracket(Q10)
[T0 T1 T2 T3 b] = imgVectorize("one_1.png");
Q11 = A0^T0 + A1^T1 + A2^T2 + A3^T3;
Q11 = unbracket(Q11)

print("\ntesting image two\n");
[T0 T1 T2 T3 b] = imgVectorize("two_0.png");
Q20 = A0^T0 + A1^T1 + A2^T2 + A3^T3;
Q20 = unbracket(Q20)
[T0 T1 T2 T3 b] = imgVectorize("two_1.png");
Q21 = A0^T0 + A1^T1 + A2^T2 + A3^T3;
Q21 = unbracket(Q21)

print("\ntesting image three\n");
[T0 T1 T2 T3 b] = imgVectorize("three_0.png");
Q30 = A0^T0 + A1^T1 + A2^T2 + A3^T3;
Q30 = unbracket(Q30)
[T0 T1 T2 T3 b] = imgVectorize("three_1.png");
Q31 = A0^T0 + A1^T1 + A2^T2 + A3^T3;
Q31 = unbracket(Q31)

print("\ntesting image four\n");
[T0 T1 T2 T3 b] = imgVectorize("four_0.png");
Q40 = A0^T0 + A1^T1 + A2^T2 + A3^T3;
Q40 = unbracket(Q40)
[T0 T1 T2 T3 b] = imgVectorize("four_1.png");
Q41 = A0^T0 + A1^T1 + A2^T2 + A3^T3;
Q41 = unbracket(Q41)

print("\nA0 image\n");
show0 = vector2img(N00,s);
show1 = vector2img(N01,s);
show2 = vector2img(N02,s);
show3 = vector2img(N03,s);
imshow(show0 <:> show1 <:> show2 <:> show3);

```

## Appendix B

### Script use for the "xor" problem

#### File: xor.mc

```
function numberTo8(a,b)
    result = [0 0 0 0];
    q=a^4;
    result(0) = floor(q/8);
    q=q-result(0)*8;
    result(1) = floor(q/4);
    q=q-result(1)*4;
    result(2) = floor(q/2);
    q=q-result(2)*2;
    result(3) = q;
    out = result;

    q=b^4;
    result(0) = floor(q/8);
    q=q-result(0)*8;
    result(1) = floor(q/4);
    q=q-result(1)*4;
    result(2) = floor(q/2);
    q=q-result(2)*2;
    result(3) = q;

    return out <-> result;
end

clear();
clc();
plot([2,2],[0,4],"k",[0,4],[2,2],"k");
/// <-> symbol is merge horizontally
/// <-> symbol is merge vertically

graphsetprop("addData on");

x0 = [ 1.7 3.5 2.3 1.7 ];
y0 = [ 2.3 1.5 0.3 3.8 ];
x1 = [ 2.3 0.7 2.7 1.9 ];
y1 = [ 2.3 1.2 3.8 0.2 ];
plot(x0,y0,"go",x1,y1,"ms");

graphsetprop("axis",[0 4 0 4]);

d=[];
m=[];

for i=0:$4
    d = d<:-1;
    m = m <-> numberTo8(x0(i),y0(i));
    d = d<:1;
    m = m <-> numberTo8(x1(i),y1(i));
end

//train with 8 points, a square matrix
r = inv(m) * d;

X0=[];
X1=[];
Y0=[];
Y1=[];
total = 20;
for i=0:$total
    for j=0:$total
        t = numberTo8(i*4/total,j*4/total);
        s = dot(r,t);
        if s < 0
            X0 = X0 <-> i*4/total;
            Y0 = Y0 <-> j*4/total;
        else
            X1 = X1 <-> i*4/total;
            Y1 = Y1 <-> j*4/total;
        end
    end
end

plot(X0,Y0,"bo",X1,Y1,"rs");

graphsetprop("axis",[0 4 0 4]);
graphsetprop("addData off");

d=[];
m=[];
X0=[];
X1=[];
Y0=[];
Y1=[];
total =20;
for i=1:$total
    for j=1:$total
        t = numberTo8(i*4/total,j*4/total);
        s=-1;
        if (i*4/total)>2 && (j*4/total)>2
            s=1;
        end
        if (i*4/total)<2 && (j*4/total)<2
            s=1;
        end
        if s < 0
            d = d<:-1;
        else
            d = d<:1;
        end
        m = m <-> t;
    end
end

//linear regression of the 324(18*18)points
r=inv(m'*m)*m'*d;

X0=[];
X1=[];
Y0=[];
Y1=[];
total = 50;
for i=0:$total
```



```
for j=0:$total
    t = numberTo8(i*4/total,j*4/total);
    s = dot(r,t);
    if s < 0
        X0 = X0 <-> i*4/total;
        Y0 = Y0 <-> j*4/total;
    else
        X1 = X1 <-> i*4/total;
        Y1 = Y1 <-> j*4/total;
    end
end
end

graphnew();
graphsetprop("addData on");
plot([2,2],[0,4],"k",[0,4],[2,2],"k");
plot(X0,Y0,"bo",X1,Y1,"rs");

graphsetprop("axis",[0 4 0 4]);
graphsetprop("addData off");
```