UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DIRETORIA DE PESQUISA E PÓS GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

JÔNATAS TRABUCO BELOTTI

IMPLEMENTAÇÃO PROJETO PRÁTICO 8.5: REDE DE *KOHONEN*

RELATÓRIO

PONTA GROSSA
2017

JÔNATAS TRABUCO BELOTTI

IMPLEMENTAÇÃO PROJETO PRÁTICO 8.5: REDE DE *KOHONEN*

Relatório apresentado como requisito parcial à obtenção de nota na disciplina de Fundamentos de Redes Neurais Artificiais do Programa de Pós-Graduação em Engenharia Elétrica, da Universidade Tecnológica Federal do Paraná– Campus Ponta Grossa.

Professor: Prof. Dr. Sérgio Okida

PONTA GROSSA
2017

SUMÁRIO

1 INTRODUÇÃO

Diversas aplicações possuem apenas o conjunto de entradas disponível, de forma que a saída desejada para cada entrada não é conhecida. Apesar de não possuírem as saídas esperadas, essas amostras possuem informações relevantes sobre o comportamento do sistema, de forma que as mesmas devem ser consideradas no mapeamento do sistema (SILVA; SPATTI; FLAUZINO, 2010).

Para este tipo de aplicação Silva, Spatti e Flauzino (2010) relatam a necessidade da utilização de redes com capacidade de se auto-organizar por meio de métodos de treinamento competitivos, métodos esses que detectam similaridades, regularidades e correlações entre as amostras de treinamento sem a necessidade da saída esperada. Mediante tais métodos de treinamento as redes separam o conjunto de entradas em classes, cada uma possuindo características e comportamentos próprios que a identificam dentre as demais.

No contexto de redes treinadas de forma auto-organizada uma das mais difundidas é a rede auto-organizada de *Kohonen*, ou simplesmente rede de *Kohonen*. Sua inspiração vem do córtex cerebral, onde a ativação de uma determinada região é a resposta a um estimulo sensorial específico, como estimulo motor, visual, auditivo ou de tato.

O principio básico da aprendizagem competitiva é a competição entre os neurônios da rede, de forma que apenas os neurônios vencedores serão estimulados através do ajuste de seus pesos sinápticos. Existem variações do aprendizado competitivo onde apenas um neurônio é vencedor ou ainda onde além do neurônio vencedor, os neurônios próximos ao vencedor também tem seus pesos sinápticos atualizados.

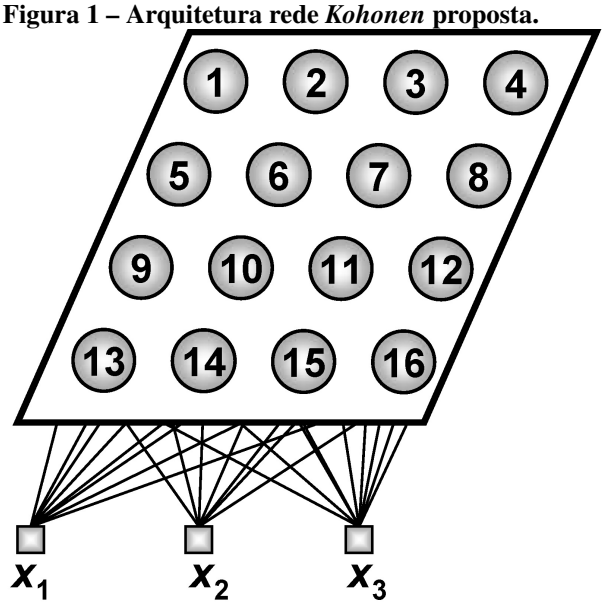
Esse relatório tem como objetivo descrever o desenvolvimento do Projeto Prático 8.5 do livro Redes neurais artificias para engenharia e ciências aplicadas de Silva, Spatti e Flauzino (2010). O projeto consiste na implementação de uma rede neural artificial de *Kohonen* para ser utilizada na classificação de amostras de borracha para a fabricação de pneus.

1.1 ESTUDO DE CASO

No projeto prático 8.5 do livro Redes neurais artificias para engenharia e ciências aplicadas de Silva, Spatti e Flauzino (2010) é apresentado um processo de fabricação de pneus onde determinadas imperfeições na borracha dos pneus impedem que os mesmos possam ser vendidos. Diversas amostras de borracha contendo tais anomalias foram coletadas, de forma que para cada amostra foram medidas 3 grandezas x_1, x_2 e x_3. Entretanto a equipe de engenheiros e cientistas não tem percepção técnica de como essas 3 variáveis podem estar relacionadas, desse modo não há uma regra que diga de acordo com essas 3 variáveis se a borracha irá gerar um pneu que poderá ser vendido ou não.

Com o objetivo de resolver esse problema pretende-se aplicar uma rede de *Kohonen*, constituída de 16 neurônios, com o objetivo de detectar as eventuais similaridades e correlações existentes entre essas variáveis. A Figura 1 apresenta a arquitetura proposta para a rede, sendo composta de 16 neurônios alimentados por 3 entradas.

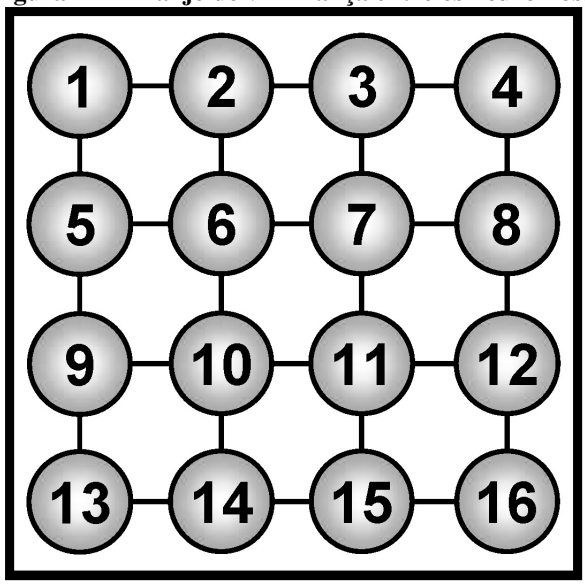
Figura 1 – Arquitetura rede *Kohonen* proposta.



Fonte: (SILVA; SPATTI; FLAUZINO, 2010).

Portanto baseado no conjunto de treinamento, pede-se que seja treinado um mapa auto-organizado de *Kohonen*, conforme a estrutura espacial apresentada na Figura 2, utilizando uma taxa de aprendizagem de $0,001$ e um raio de vizinhança unitário ($R = 1$).

Figura 2 – Arranjo de vizinhança entre os neurônios.



Fonte: (SILVA; SPATTI; FLAUZINO, 2010).

2 DESENVOLVIMENTO DO PROJETO

O mapa auto-organizável de *Kohonen* proposto na Seção 1.1 foi desenvolvido na linguagem Java, a classe *Kohonen* é a responsável por implementar o funcionamento da rede, seu código fonte está disponível no Apêndice A. Com o objetivo de facilitar o acesso ao código fonte, o mesmo juntamente com o programa já compilado foram disponibilizados em um repositório do *GitHub*[1] que pode ser acessado pelo link <https://github.com/jonatastbelotti/redeKohonen>.

2.1 TREINAMENTO DA REDE

O treinamento da rede foi realizado para o conjunto de treinamento fornecido pelo livro, tal conjunto está disponível no Anexo A. A cada época de treinamento os pesos sinápticos dos neurônios vencedores foram atualizados de acordo com a Equação 2.1, enquanto que os pesos sinápticos dos neurônios vizinhos foram atualizados pela Equação 2.2.

$$\mathbf{w}^{(v)} \leftarrow \mathbf{w}^{(v)} + \eta \cdot (\mathbf{x}^{(k)} - \mathbf{w}^{(v)}) \tag{2.1}$$

$$\mathbf{w}^{(\Omega)} \leftarrow \mathbf{w}^{(\Omega)} + \frac{\eta}{2} \cdot (\mathbf{x}^{(k)} - \mathbf{w}^{(\Omega)}) \tag{2.2}$$

onde $\mathbf{w}^{(v)}$ é o vetor de pesos sinápticos do neurônio vencedor, η é a taxa de aprendizagem, $\mathbf{x}^{(k)}$ é o vetor de entradas da k-ésima amostra de treinamento e $\mathbf{w}^{(\Omega)}$ é o vetor de pesos sinápticos dos neurônios vizinhos do neurônio vencedor.

O ajuste dos pesos sinápticos foi realizado enquanto a soma dos ajustes de todos os neurônios vencedores era maior que $0,001$. Por sua vez a soma dos ajustes de todos os neurônios vencedores é dada pela Equação 2.3.

$$\text{Ajustes} = \sum_{k=1}^{|x|} \eta \cdot (\mathbf{x}^{(k)} - \mathbf{w}^{(v)}) \tag{2.3}$$

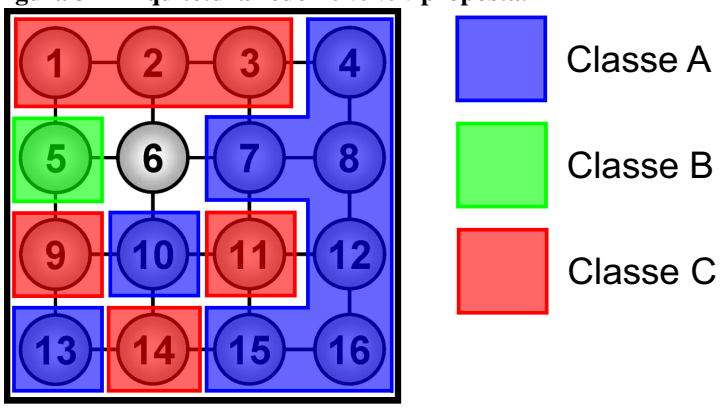
onde $|x|$ é o tamanho do conjunto de treinamento, η é a taxa de aprendizagem, $\mathbf{x}^{(k)}$ é o vetor de entradas da k-ésima amostra de treinamento.

Ao final do treinamento da rede de *Kohonen* cada neurônio representa uma classe do sistema mapeado, de forma que o neurônio ativado representa a classe da entrada da rede. O treinamento da rede desenvolvida foi realizado ao final de 42 épocas de treinamento. A Figura 3 apresenta a classe que cada neurônio representa ao final do treinamento da rede.

Pela Figura 3 nota-se que apenas o neurônio 6 não representa uma classe das 3 classes A, B ou C. Isso se deve ao fato que ao final do treinamento da rede de *Kohonen* cada neurônio

[1] No repositório do GitHub o caminho para acessar a classe *Kohonen* é './redeKohonen/src/Modelo/Kohonen.java'.

Figura 3 – Arquitetura rede *Kohonen* proposta.



Fonte: (SILVA; SPATTI; FLAUZINO, 2010).

representa uma classe de classificação, como nesse problema em específico são 16 neurônios para 3 classes, apenas 3 neurônios são necessários. Quanto mais épocas de treinamento forem realizadas menos neurônios serão representantes de uma classe, até que apenas 3 neurônios presentem as 3 classes.

2.2 EXECUÇÃO DA REDE

Após o treinamento descrito na Seção 2.1 a mesma foi executada para a classificação de 12 amostras de borracha, essas 12 amostras de borracha estão disponíveis no Anexo B. As amostras juntamente com a classificação obtida pela rede são apresentadas na Tabela 1.

Tabela 1 – Resultado execução da rede

Amostra	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	**Classe**
1	$0,2471$	$0,1778$	$0,2905$	A
2	$0,8240$	$0,2223$	$0,7041$	B
3	$0,4960$	$0,7231$	$0,5866$	C
4	$0,2923$	$0,2041$	$0,2234$	A
5	$0,8118$	$0,2668$	$0,7484$	B
6	$0,4837$	$0,8200$	$0,4792$	C
7	$0,3248$	$0,2629$	$0,2375$	A
8	$0,7209$	$0,2116$	$0,7821$	B
9	$0,5259$	$0,6522$	$0,5957$	C
10	$0,2075$	$0,1669$	$0,1745$	A
11	$0,7830$	$0,3171$	$0,7888$	B
12	$0,5393$	$0,7510$	$0,5682$	C

Fonte: Autoria própria.

3 CONCLUSÃO

Concluí-se que os mapas auto-organizáveis de *Kohonen* são uma excelente opção para aplicações onde o treinamento da rede não pode ser supervisionado devido a ausência das saídas esperadas para o conjunto de treinamento. Visto que a rede de *Kohonen* desenvolvida nesse trabalho obteve excelentes resultados para a classificação da borracha.

REFERÊNCIAS

SILVA, Ivan Nunes da; SPATTI, Danilo Hernane; FLAUZINO, Rogério Andrade. **Redes Neurais Artificiais Para Engenharia e Cincias Aplicadas - Curso Pratico**. 1. ed. São Paulo: ARTLIBER, 2010. ISBN 978-85-88098-53-4.

APÊNDICE A - IMPLEMENTAÇÃO DA CLASSE *HOPFIELD* EM JAVA

```
1  package Modelo;
2
   import Controle.Comunicador;
4  import Recursos.Arquivo;
   import Recursos.Numero;
6  import java.util.ArrayList;
   import java.util.HashMap;
8  import java.util.List;

10 /**
    *
12  * @author Jônatas Trabuco Belotti [jonatas.t.belotti@hotmail.com]
    */
14 public class Kohonen {

16   public static final int NUM_ENTRADAS = 3;
     private final int NUM_LINHAS = 4;
18   private final int NUM_COLUNAS = 4;
     private final int NUM_NEURONIOS = NUM_LINHAS * NUM_COLUNAS;
20   private final double MUDANCA_MINIMA = 0.01;
     private final int NUM_MAXIMO_EPOCAS = 10000;
22
     private final double TAXA_APRENDIZAGEM = 0.001;
24   private final double RAIO_VIZINHANCA = 1D;

26   private int mapaNeuronios[][];
     private List[] conjuntoVizinhos;
28   private double pesos[][];
     private double entrada[];
30   private HashMap<Integer, String> classeNeuronio;

32   private int numEpocas;

34   /**
      * Método construtor, responsável por instanciar os objetos,
          criar o mapa de neurônios e os conjuntos de vizinhos
36    */
     public Kohonen() {
38     int neuronio, linhaNeuronio, colunaNeuronio;
```

```
        double distancia;
40
        linhaNeuronio = colunaNeuronio = 0;
42
        mapaNeuronios = new int[NUM_LINHAS][NUM_COLUNAS];
44      conjuntoVizinhos = new List[NUM_NEURONIOS];
        pesos = new double[NUM_NEURONIOS][NUM_ENTRADAS];
46      entrada = new double[NUM_ENTRADAS];
        classeNeuronio = new HashMap<>();
48
        //Criando o mapa dos neurônios
50      neuronio = 0;
        for (int linha = 0; linha < NUM_LINHAS; linha++) {
52        for (int coluna = 0; coluna < NUM_COLUNAS; coluna++) {
            classeNeuronio.put(neuronio, "-");
54          mapaNeuronios[linha][coluna] = neuronio++;
          }
56      }

58      //Montando conjuntos de vizinhos
        for (neuronio = 0; neuronio < NUM_NEURONIOS; neuronio++) {
60        conjuntoVizinhos[neuronio] = new ArrayList();

62        for (int linha = 0; linha < NUM_LINHAS; linha++) {
            for (int coluna = 0; coluna < NUM_COLUNAS; coluna++) {
64            if (mapaNeuronios[linha][coluna] == neuronio) {
                linhaNeuronio = linha;
66              colunaNeuronio = coluna;
                coluna = NUM_COLUNAS;
68              linha = NUM_LINHAS;
              }
70          }
          }
72
          for (int linha = 0; linha < NUM_LINHAS; linha++) {
74          for (int coluna = 0; coluna < NUM_COLUNAS; coluna++) {
              if (mapaNeuronios[linha][coluna] != neuronio) {
76              distancia = Math.sqrt(Math.pow((double) (linhaNeuronio
                    - linha), 2D) + Math.pow((double) (colunaNeuronio -
                    coluna), 2D));

78              if (distancia <= RAIO_VIZINHANCA) {
```

```
                conjuntoVizinhos[neuronio].add(mapaNeuronios[linha][
                    coluna]);
80            }
            }
82        }
        }
84    }
  }
86
  /**
88   * Método que realiza o treinamento da rede.
   * @param arquivoTreinamento arquivo contendo o conjunto de
      amostras de treinamento
90   * @return retorna verdadeiro se o ttreinamento foi realizado com
       sucesso, falso caso contrário
   */
92 public boolean treinar(Arquivo arquivoTreinamento) {
    String classe;
94  double mudanca;
    int neuronioMenorDistancia;
96  int numAmostra;

98  Comunicador.iniciarLog("Iniciando treinamento da rede...");

100 //Iniciando vetores de pesos já normalizados
    for (int neuronio = 0; neuronio < NUM_NEURONIOS; neuronio++) {
102   separarEntradas(arquivoTreinamento.lerArquivo().split("\n")[
        neuronio]);
      normalizarVetor(entrada);
104   copiarVetor(entrada, pesos[neuronio]);
    }
106
    //Iniciando número de epocas
108 this.numEpocas = 0;

110 //Atualizar pesos até que não haja mudança significativa
    do {
112   mudanca = 0D;

114   //Para cada amostra de treinamento do arquivo
      for (String linha : arquivoTreinamento.lerArquivo().split("\n
        ")) {
```

```
116         separarEntradas(linha);
            normalizarVetor(entrada);
118
            //Determinando qual o neurônio mais próximo da amostra
120         neuronioMenorDistancia = calcNeuronioMenorDistancia();

122         //Ajustando peso do neuronio vencedor
            for (int i = 0; i < NUM_ENTRADAS; i++) {
124           pesos[neuronioMenorDistancia][i] += TAXA_APRENDIZAGEM * (
                 entrada[i] - pesos[neuronioMenorDistancia][i]);
              mudanca += Math.abs(TAXA_APRENDIZAGEM * (entrada[i] -
                 pesos[neuronioMenorDistancia][i]));
126         }
            normalizarVetor(pesos[neuronioMenorDistancia]);
128
            //Ajustanndo pesos dos vizinhos do vencedor
130         for (Object obj : conjuntoVizinhos[neuronioMenorDistancia])
                {
              int neuronio = (int) obj;
132
              for (int i = 0; i < NUM_ENTRADAS; i++) {
134             pesos[neuronio][i] += (TAXA_APRENDIZAGEM / 2D) * (
                   entrada[i] - pesos[neuronioMenorDistancia][i]);
              }
136           normalizarVetor(pesos[neuronio]);
            }
138       }

140       Comunicador.adicionarLog(String.format("%d %.8f", numEpocas,
             mudanca));

142       numEpocas++;
        } while (mudanca >= MUDANCA_MINIMA && numEpocas <=
           NUM_MAXIMO_EPOCAS);
144
        Comunicador.adicionarLog("Fim do treinamento!");
146     Comunicador.adicionarLog("Iniciando identificação da classe de
           cada neurônio...");

148     //Identificando a que classe cada neurônio pertence
        numAmostra = 0;
150
```

```
    for (String linha : arquivoTreinamento.lerArquivo().split("\n")
        ) {
      numAmostra++;
      separarEntradas(linha);
      normalizarVetor(entrada);

      neuronioMenorDistancia = calcNeuronioMenorDistancia();

      classe = "A";
      if (numAmostra >= 21 && numAmostra <= 60) {
        classe = "B";
      }
      if (numAmostra >= 61 && numAmostra <= 120) {
        classe = "C";
      }

      classeNeuronio.put(neuronioMenorDistancia, classe);
    }

    imprimirClasses();

    return true;
  }

  /**
   * Método que realiza a execução da rede, classifica um conjunto
      de amostras nas classes estabelecidas (A, B e C)
   * @param arquivoTeste arquivo contendo o conjunto de amostra a
      serem classificadas
   */
  public void testar(Arquivo arquivoTeste) {
    String texto;
    int numAmostra;
    int neuronio;

    Comunicador.iniciarLog("Iniciando execução da rede");
    numAmostra = 0;

    for (String linha : arquivoTeste.lerArquivo().split("\n")) {
      numAmostra++;
      separarEntradas(linha);
      normalizarVetor(entrada);
```

```
190
        neuronio = calcNeuronioMenorDistancia();
192
        texto = "" + numAmostra + " ";
194
        for (int i = 0; i < NUM_ENTRADAS; i++) {
196       texto += String.format("%f ", entrada[i]);
        }
198
        texto += String.format("-> %s", classeNeuronio.get(neuronio))
           ;
200
        Comunicador.adicionarLog(texto);
202     }

204   }

206   /**
       * Método que recura os valores da entrada da rede de uma linha
          de texto.
208    * @param linha String (texto) contendo os valores das entradas (
          x1, x2, x3, ...)
       */
210   private void separarEntradas(String linha) {
        String[] vetor;
212     int i;

214     vetor = linha.split("\\s+");
        i = 0;
216
        if (vetor[0].equals("")) {
218       i++;
        }
220
        //preenche o vetor de entradas a partir da linha lida do
           arquivo
222     for (int j = 0; j < NUM_ENTRADAS; j++) {
          entrada[j] = Numero.parseDouble(vetor[i++]);
224     }
      }
226
      /**
```

```
228     * Método que normaliza um vetor.
        * @param vetor vetor a ser normalizado.
230     */
    private void normalizarVetor(double vetor[]) {
232     double modulo;

234     //Calculando modulo do vetor
        modulo = 0D;
236     for (int i = 0; i < vetor.length; i++) {
          modulo += Math.pow(vetor[i], 2D);
238     }
        modulo = Math.sqrt(modulo);
240
        //Normalizando cada elemento do vetor
242     for (int i = 0; i < vetor.length; i++) {
          vetor[i] /= modulo;
244     }
    }
246
    /**
248     * Calcula a distância euclidiana entre dois vetores.
        * @param vetor1 vetor da origem do cálculo da distância
            euclidiana.
250     * @param vetor2 vetor destino do cálculo da distância euclidiana
            .
        * @return Retorna o valor (double) da distância eucliadiana
            entre os dois vetores.
252     */
    private double distanciaEuclidiana(double vetor1[], double vetor2
        []) {
254     double distancia;

256     distancia = 0D;

258     for (int i = 0; i < vetor1.length; i++) {
          distancia += Math.pow(vetor1[i] - vetor2[i], 2D);
260     }

262     distancia = Math.sqrt(distancia);

264     return distancia;
    }
```

```
266
    /**
268  * Realiza a copia dos valores de um vetor para outro.
     * @param origem vetor de onde os valores são copiados.
270  * @param destino vetor para onde os valores são copiados.
     */
272 private void copiarVetor(double[] origem, double[] destino) {
      for (int i = 0; i < origem.length && i < destino.length; i++) {
274     destino[i] = origem[i];
      }
276 }

278 /**
     * Calcula qual o neurônio com menor distância euclidiana até a
        entrada da rede.
280  * @return retorna o neurônio (de 0 até n-1) mais próximo do
        vetor de entrada da rede.
     */
282 private int calcNeuronioMenorDistancia() {
      double distancia;
284   double menorDistancia;
      int neuronioMenorDistancia;
286
      menorDistancia = Double.MAX_VALUE;
288   neuronioMenorDistancia = 0;

290   for (int neuronio = 0; neuronio < NUM_NEURONIOS; neuronio++) {
        distancia = distanciaEuclidiana(pesos[neuronio], entrada);
292
        if (distancia < menorDistancia) {
294       menorDistancia = distancia;
          neuronioMenorDistancia = neuronio;
296     }
      }
298
      return neuronioMenorDistancia;
300 }

302 /**
     * Imprime o mapa de contexto e a classe de cada neurônio.
304  */
    private void imprimirClasses() {
```

```
306       String texto;

308       Comunicador.adicionarLog("Mapa de contexto:");

310       for (int linha = 0; linha < NUM_LINHAS; linha++) {
            texto = "";
312
            for (int coluna = 0; coluna < NUM_COLUNAS; coluna++) {
314           texto += classeNeuronio.get(mapaNeuronios[linha][coluna]) +
                  "  ";
            }
316
            Comunicador.adicionarLog(texto);
318       }

320       Comunicador.adicionarLog("Classe de cada neurônio:");
          for (int neuronio = 0; neuronio < NUM_NEURONIOS; neuronio++) {
322         Comunicador.adicionarLog(String.format("%d -> %s", neuronio
               +1, classeNeuronio.get(neuronio)));
          }
324     }
    }
```

ANEXO A - CONJUNTO DE TREINAMENTO

x1	x2	x3
0.2417	0.2857	0.2397
0.2268	0.2874	0.2153
0.1975	0.3315	0.1965
0.3414	0.3166	0.1074
0.2587	0.1918	0.2634
0.2455	0.2075	0.1344
0.3163	0.1679	0.1725
0.2704	0.2605	0.1411
0.1871	0.2965	0.1231
0.3474	0.2715	0.1958
0.2059	0.2928	0.2839
0.2442	0.2272	0.2384
0.2126	0.3437	0.1128
0.2562	0.2542	0.1599
0.1640	0.2289	0.2627
0.2795	0.1880	0.1627
0.3463	0.1513	0.2281
0.3430	0.1508	0.1881
0.1981	0.2821	0.1294
0.2322	0.3025	0.2191
0.7352	0.2722	0.6962
0.7191	0.1825	0.7470
0.6921	0.1537	0.8172
0.6833	0.2048	0.8490
0.8012	0.2684	0.7673
0.7860	0.1734	0.7198
0.7205	0.1542	0.7295
0.6549	0.3288	0.8153
0.6968	0.3173	0.7389
0.7448	0.2095	0.6847
0.6746	0.3277	0.6725
0.7897	0.2801	0.7679
0.8399	0.3067	0.7003
0.8065	0.3206	0.7205
0.8357	0.3220	0.7879
0.7438	0.3230	0.8384

0.8172	0.3319	0.7628
0.8248	0.2614	0.8405
0.6979	0.2142	0.7309
0.6804	0.3181	0.7017
0.6973	0.3194	0.7522
0.7910	0.2239	0.7018
0.7052	0.2148	0.6866
0.8088	0.1908	0.7563
0.7640	0.1676	0.6994
0.7616	0.2881	0.8087
0.8188	0.2461	0.7273
0.7920	0.3178	0.7497
0.7802	0.1871	0.8102
0.7332	0.2543	0.8194
0.6921	0.1529	0.7759
0.6833	0.2197	0.6943
0.7860	0.1745	0.7639
0.8009	0.3082	0.8491
0.7793	0.1935	0.6738
0.7373	0.2698	0.7864
0.7048	0.2380	0.7825
0.8393	0.2857	0.7733
0.6878	0.2126	0.6961
0.6651	0.3492	0.6737
0.4856	0.6600	0.4798
0.4114	0.7220	0.5106
0.5671	0.7935	0.5929
0.4875	0.7928	0.5532
0.5172	0.7147	0.5774
0.5483	0.6773	0.4842
0.5740	0.6682	0.5335
0.4587	0.6981	0.5900
0.5794	0.7410	0.4759
0.4712	0.6734	0.5677
0.5126	0.8141	0.5224
0.5557	0.7749	0.4342
0.4916	0.8267	0.4586
0.4629	0.8129	0.4950
0.5850	0.7358	0.5107
0.4435	0.7030	0.4594

0.4155	0.7516	0.5524
0.4887	0.7027	0.5886
0.5462	0.7378	0.5107
0.5251	0.8124	0.5686
0.4635	0.7339	0.5638
0.5907	0.7144	0.4718
0.4982	0.8335	0.4597
0.5242	0.7325	0.4079
0.4075	0.8372	0.4271
0.5934	0.8284	0.5107
0.5463	0.6766	0.5639
0.4403	0.8495	0.4806
0.4531	0.7760	0.5276
0.5109	0.7387	0.5373
0.5383	0.7780	0.4955
0.5679	0.7156	0.5022
0.5762	0.7781	0.5908
0.5997	0.7504	0.5678
0.4138	0.6975	0.5148
0.5490	0.6674	0.4472
0.4719	0.7527	0.4401
0.4458	0.8063	0.4253
0.4983	0.8131	0.5625
0.5742	0.6789	0.5997
0.5289	0.7354	0.4718
0.5927	0.7738	0.5390
0.5199	0.7131	0.4028
0.5716	0.6558	0.4451
0.5075	0.7045	0.4233
0.4886	0.7004	0.4608
0.5527	0.8243	0.5772
0.4816	0.6969	0.4678
0.5809	0.6557	0.4266
0.5881	0.7565	0.4003
0.5334	0.8446	0.4934
0.4603	0.7992	0.4816
0.5491	0.6504	0.4063
0.4288	0.8455	0.5047
0.5636	0.7884	0.5417
0.5349	0.6736	0.4541

```
0.5569    0.8393    0.5652
0.4729    0.7702    0.5325
0.5472    0.8454    0.5449
0.5805    0.7349    0.4464
```

ANEXO B - CONJUNTO DE TESTE

x1	x2	x3
0.2471	0.1778	0.2905
0.8240	0.2223	0.7041
0.4960	0.7231	0.5866
0.2923	0.2041	0.2234
0.8118	0.2668	0.7484
0.4837	0.8200	0.4792
0.3248	0.2629	0.2375
0.7209	0.2116	0.7821
0.5259	0.6522	0.5957
0.2075	0.1669	0.1745
0.7830	0.3171	0.7888
0.5393	0.7510	0.5682