Machine Learning - Octave/Matlab Tutorial

赵燕

目录

1 Octave/Matlab Tutorial

1.1 Basic Operations

1.1.1 赋值语句

```
>> a=3
a =  3
>> a=3;     #不会打印出a=3
>> a=3
a =  3
>> b='hi';
>> b
b = hi
>> c=(3>=1);
>> c
c = 1       #输出为真
```

1.1.2 打印和输出变量

```
>> a=pi;
>> a
a =  3.1416
>> disp(a);
 3.1416            #disp命令输出
>> disp(sprintf('2 decimals:%0.2f',a))
2 decimals:3.14    #打印字符串，保留两位小数
>> disp(sprintf('6 decimals:%0.6f',a))
6 decimals:3.141593  #sprintf是打印生成字符串
>> a
a =  3.1416
>> format long
>> a
a =  3.14159265358979
>> format short
>> a
a =  3.1416
>>
```

1.1.3 向量，矩阵和集合

```
>> A=[1 3;3,4;5,6]
A =        #矩阵

   1   3
   3   4
   5   6

>> v=[1,2,3]
v =       #行向量

   1   2   3

>> v=[1;2;3]
v =       #列向量

   1
   2
```

```
   3

>> v=1:0.1:2
v =      #集合，从1开始，增量（步长）为0.1，直到2

 Columns 1 through 4:

    1.0000    1.1000    1.2000    1.3000

 Columns 5 through 8:

    1.4000    1.5000    1.6000    1.7000

 Columns 9 through 11:

    1.8000    1.9000    2.0000
>> v=1:6
v =

   1   2   3   4   5   6

>>
```

1.1.4 生成矩阵

```
>> ones(2,3)
ans =   #元素都为1矩阵

   1   1   1
   1   1   1

>> C=2*ones(2,3)
C =      #元素都为2的矩阵

   2   2   2
   2   2   2

>> w=ones(1,3)
w =      #1行3列

   1   1   1

>> w=zeros(1,3)
w =      #0矩阵

   0   0   0

>> w=rand(1,3)
#随机矩阵，元素随机，数值在0到1之间
w =

   0.056270    0.270442    0.232801

>> rand(3,3)
#随机矩阵，元素随机，数值在0到1之间
ans =
```

```
   0.42812   0.94129   0.32911
   0.37266   0.52775   0.89005
   0.43005   0.61385   0.76779

>> w=randn(1,3)
 #高斯随机矩阵（正态分布），元素随机，平均值为0的高斯分布
w =

  -1.11347   0.73961  -0.43813

>> w=randn(1,3)
#高斯随机矩阵（正态分布），元素随机，平均值为0的高斯分布
w =

  -0.20530   1.09960  -1.53719

>>

>> w=-6+sqrt(10)*(randn(1,10000))
w =Columns 1 through 3:

  -5.0452e+00  -3.6748e+00  -9.9375e+00

 Columns 4 through 6:

  -2.4220e+00  -9.0436e+00  -5.9153e+00

 Columns 7 through 9:

  -8.9856e+00  -7.3453e+00  -7.7757e+00

 Columns 10 through 12:

  -7.7120e+00  -4.2215e+00  -9.6187e+00

 Columns 13 through 15:

  -4.5269e+00  -3.2191e+00  -2.3526e+00

 Columns 16 through 18:

  -4.7875e+00  -6.7731e+00  -6.5302e+00

 Columns 19 through 21:

  -6.9177e+00  -5.0446e+00  -8.6510e+00

 Columns 22 through 24:

  -2.6468e+00  -4.2173e+00  -9.5689e+00
warning: broken pipe
>> hist(w)   #绘制直方图
>> hist(w,50)
>>>
```

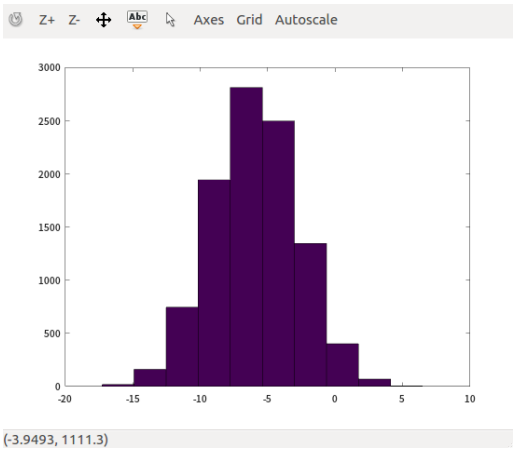

图 1: 集合w的直方图

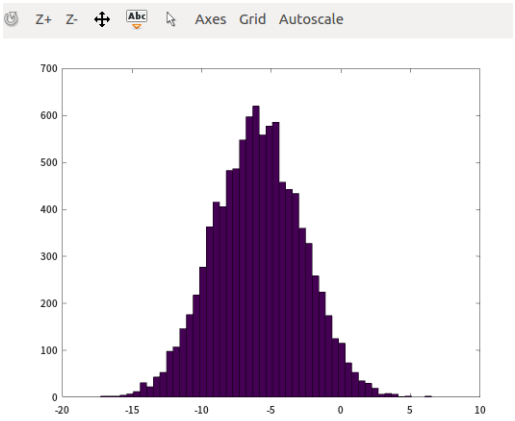

图 2: 集合w的直方图(50条)

1.1.5 单位阵

```
>> eye(4)
ans =

Diagonal Matrix

   1   0   0   0
   0   1   0   0
   0   0   1   0
   0   0   0   1

>> I=eye(4)
I =

Diagonal Matrix

   1   0   0   0
   0   1   0   0
   0   0   1   0
   0   0   0   1
```

```
>> I=eye(6)
I =

Diagonal Matrix

   1   0   0   0   0   0
   0   1   0   0   0   0
   0   0   1   0   0   0
   0   0   0   1   0   0
   0   0   0   0   1   0
   0   0   0   0   0   1

>> eye(3)
ans =

Diagonal Matrix

   1   0   0
   0   1   0
   0   0   1

>>
```

1.1.6 help命令

```
>> help eye
'eye' is a built-in function from the file libinterp/corefcn/data.cc

 -- eye (N)
 -- eye (M, N)
 -- eye ([M N])
 -- eye (..., CLASS)
     Return an identity matrix.

     If invoked with a single scalar argument N,return a square NxN
     identity matrix.

     If supplied two scalar arguments (M, N), 'eye' takes them to be the
     number of rows and columns.  If given a vector with two elements,
     'eye' uses the values of the elements as the number of rows and
     columns, respectively.  For example:

          eye (3)
           =>  1  0  0
               0  1  0
               0  0  1

     The following expressions all produce the same result:
     #q退出该命令
>> help rand
>> help help
```

1.2 Moving Data Around

1.1.1 size()命令

返回矩阵的尺寸

```
>> A=[1,2;3,4;5,6]
A =

   1   2
   3   4
   5   6

>> size(A)
#size()命令返回一个1*2的矩阵，返回矩阵的尺寸
ans =

   3   2

>> sz=size(A)
#这个矩阵用sz来存放，所以sz就是一个1*2的矩阵
sz =

   3   2

>> size(sz)
#计算矩阵的维度
ans =

   1   2

>> size(A,1)
#返回A矩阵的第一个元素3，行数
ans =  3
>> size(A,2)
#返回A矩阵的第2个元素2，列数
ans =  2
```

1.2.2 length命令

```
>> v=[1 2 3 4]
#向量v
v =

   1   2   3   4

>> length(v)
#返回最大维度的大小
ans =  4
>> length(A)
#矩阵A的最大维度是3
ans =  3
>> length([1;2;3;4;5])
#一般只是给向量用length命令
ans =  5
>>
```

1.2.3 在系统中加载和寻找数据

```
>> pwd
#显示出Octave当前所处路径
ans = /home/zhaozhao
>> cd #改变路径 'C:\Users\ang\Desktop
```

```
>> ls #列出所有的路径
courses-learning   Notes Octave
>> load features.dat  #加载了features文件
>> load priceY.dat
>> load ('featuresX.dat')
>> who
#显示出当前Octave所存储的变量
Variables in the current scope:

A    I    ans  c    v
C    a    b    sz   w

>> size(featuresX)
>> size(PriceY)
>> whos
#同时会列出维度
Variables in the current scope:

   Attr Name        Size                     Bytes  Class
   ==== ====        ====                     =====  =====
        A           3x2                         48  double
        C           2x3                         48  double
        I           6x6                         48  double
        a           1x18  double
        ans         1x14                        14  char
        b           1x22  char
        c           1x11  logical
        sz          1x2                         16  double
        v           1x4                         32  double
        w           1x10000                  80000  double

Total is 10072 elements using 80217 bytes
>> v=priceY(1:10)
#存储数据
>> save hello.mat v;
#将v存储为hello.mat
>> save hello.txt v -ascii
#save as text(ASCII)
```

1.2.4 在系统中操作数据

```
A=[1 2;3 4;5 6]
A =

   1   2
   3   4
   5   6

>> A(3,2)
ans =  6
>> A(2,:)
ans =

   3   4

# ":"means every element along that row/column
```

```
>> A(:,2)
ans =

   2
   4
   6
>> A([1,3],:)
ans =

   1   2
   5   6

>> A
A =

   1   2
   3   4
   5   6

>> A(:,2)
ans =

   2
   4
   6

>> A(:,2)=[10;11;12]
A =

    1   10
    3   11
    5   12
#第2列被替换为[10;11;12]
>>>> A=[A,[100;101;102]];
>> A
A =

     1     2   100
     3     4   101
     5     6   102
#在原来的矩阵A右边附上一个新的列矩阵
>> A=[A,[100;101;102]]
A =

     1     2   100   100
     3     4   101   101
     5     6   102   102

>> size(A)
ans =

   3   4

>> A(:)
ans =
```

```
     1
     3
     5
     2
     4
     6
   100
   101
   102
   100
   101
   102
#把A中的所有元素放入一个单独的列向量，得到一个12*1的向量，这些元素都是A中元素排列起来的
>>
>>>> A=[1 2;3 4;5 6]
A =

   1   2
   3   4
   5   6

>> B=[11 12;13 14;15 16]
B =

   11   12
   13   14
   15   16

>> C=[A B]#与[A,B]一样
C =

    1    2   11   12
    3    4   13   14
    5    6   15   16
#把两个矩阵直接连接起来，A在左边，B在右边，组成了矩阵C
>> C=[A;B]
C =

    1    2
    3    4
    5    6
   11   12
   13   14
   15   16
#用\: "隔开，A在B的上面
>>
```

1.3 Computing on Data

1.3.1 如何对数据进行运算

```
>> A=[1 2;3 4;5 6]
A =

   1   2
   3   4
   5   6
```

```
>> B=[11 12;13 14;15 16]
B =

   11   12
   13   14
   15   16

>> C=[1 1;2 2]
C =

   1   1
   2   2

>> A*C
ans =

    5    5
   11   11
   17   17

>> A .* B
#两个矩阵的元素位运算，点号一般来表示元素的位运算
ans =

   11   24
   39   56
   75   96

>> A .^ 2
#矩阵A中的每一个元素平方
ans =

    1    4
    9   16
   25   36
>> v=[1;2;3]
v =

   1
   2
   3

>> 1 ./ v
#每一个元素的倒数
ans =

   1.00000
   0.50000
   0.33333

>> 1 ./ A
#A中每一个元素的倒数
ans =

   1.00000   0.50000
```

```
   0.33333   0.25000
   0.20000   0.16667

>> log(v)
#每个元素求对数
ans =

   0.00000
   0.69315
   1.09861

>> exp(v)
#自然数e的幂次运算，以e为底，以这些元素为幂的运算
ans =

    2.7183
    7.3891
   20.0855

>> v
v =

   1
   2
   3

>> abs(v)
#求每个元素的绝对值
ans =

   1
   2
   3

>> abs([-1;2;-3])
ans =

   1
   2
   3

>> -v
ans =

  -1
  -2
  -3

>> -1 * v
ans =

  -1
  -2
  -3
>> v+ones(length(v),1)
#每一个元素+1
```

```
ans =

   2
   3
   4

>> v+1
ans =

   2
   3
   4

>>
```

1.3.2 更多操作

```
>> A
A =

   1   2
   3   4
   5   6

>> A'
ans =

   1   3   5
   2   4   6

>> (A')'
ans =

   1   2
   3   4
   5   6

>> a=[1 15 2 0.5]
a =

    1.00000   15.00000    2.00000    0.50000

>> val=max(a)
val =  15
>> [val,ind]=max(a)
val =  15
ind =  2
>> max(A)
ans =

   5   6

>> A
A =

   1   2
```

```
   3   4
   5   6

>> a
a =

    1.00000   15.00000    2.00000    0.50000

>> a<3
ans =

  1  0  1  1

>> find(a<3)
ans =

   1   3   4

>> A=magic(3)
A =

   8   1   6
   3   5   7
   4   9   2

>> help magic
'magic' is a function from the file /usr/share/octave/4.2.1/m/special-matrix/magic.m

 -- magic (N)

     Create an N-by-N magic square.

     A magic square is an arrangement of the integers '1:n^2' such that
     the row sums, column sums, and diagonal sums are all equal to the
     same value.

     Note: N must be a scalar greater than or equal to 3.  If you supply
     N less than 3, magic returns either a nonmagic square, or else the
     degenerate magic squares 1 and [].

Additional help for built-in functions and operators is
available in the online version of the manual.  Use the command
'doc <topic>' to search the manual index.

Help and information about Octave is also available on the WWW
at http://www.octave.org and via the help@octave.org
mailing list.
>> A=magic(3)
A =

   8   1   6
   3   5   7
   4   9   2

>> [r,c]=find(A>=7)
r =
```

```
   1
   3
   2

c =

   1
   2
   3

>> A(2,3)
ans =  7
>> help find
'find' is a built-in function from the file libinterp/corefcn/find.cc

 -- IDX = find (X)
 -- IDX = find (X, N)
 -- IDX = find (X, N, DIRECTION)
 -- [i, j] = find (...)
 -- [i, j, v] = find (...)
     Return a vector of indices of nonzero elements of a matrix, as a
     row if X is a row vector or as a column otherwise.

     To obtain a single index for each matrix element, Octave pretends
     that the columns of a matrix form one long vector (like Fortran
     arrays are stored).  For example:

          find (eye (2))
            => [ 1; 4 ]

     If two inputs are given, N indicates the maximum number of elements
     to find from the beginning of the matrix orvector.

     If three inputs are given, DIRECTION shouldbe one of "first" or
     "last", requesting only the first or last Nindices, respectively.
     However, the indices are always returned in>> a
a =

    1.00000   15.00000    2.00000    0.50000

>> sum(a)
ans =  18.500
>> prod*a)
parse error:

  syntax error

>>> prod*a)
          ^

>> prod(a)
ans =  15
>> floor(a)
ans =
```

```
    1   15    2    0

>> ceil(a)
ans =

    1   15    2    1

>> rand(a)
error: rand: conversion to integer value failed
error: rand: dimensions must be a scalar or array of integers
>> rand(3)
ans =

   0.36575   0.15743   0.28353
   0.35116   0.58297   0.89024
   0.52420   0.59669   0.94361

>> max(rand(3),rand(3))
ans =

   0.49349   0.84108   0.75945
   0.84417   0.65052   0.61740
   0.87132   0.98382   0.54329

>> A
A =

   8   1   6
   3   5   7
   4   9   2

>> max(A,[],1)
ans =

   8   9   7

>> max(A,[],2)
ans =

   8
   7
   9

>> max(A)
ans =

   8   9   7

>> max(max(A))
ans =  9
>> A(:)
ans =

   8
   3
   4
```

```
   1
   5
   9
   6
   7
   2

>> max(A(:))
ans =  9
>> A=magic(9)
A =

   47   58   69   80    1   12   23   34   45
   57   68   79    9   11   22   33   44   46
   67   78    8   10   21   32   43   54   56
   77    7   18   20   31   42   53   55   66
    6   17   19   30   41   52   63   65   76
   16   27   29   40   51   62   64   75    5
   26   28   39   50   61   72   74    4   15
   36   38   49   60   71   73    3   14   25
   37   48   59   70   81    2   13   24   35

>> sum(A,1)
ans =

 Columns 1 through 8:

   369   369   369   369   369   369   369   369

 Column 9:

   369

>> sum(A,2)
ans =

   369
   369
   369
   369
   369
   369
   369
   369
   369

>> eye(9)
ans =

Diagonal Matrix

   1   0   0   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0
   0   0   1   0   0   0   0   0   0
   0   0   0   1   0   0   0   0   0
   0   0   0   0   1   0   0   0   0
```

```
   0   0   0   0   0   1   0   0   0
   0   0   0   0   0   0   1   0   0
   0   0   0   0   0   0   0   1   0
   0   0   0   0   0   0   0   0   1

>> A
A =

   47   58   69   80    1   12   23   34   45
   57   68   79    9   11   22   33   44   46
   67   78    8   10   21   32   43   54   56
   77    7   18   20   31   42   53   55   66
    6   17   19   30   41   52   63   65   76
   16   27   29   40   51   62   64   75    5
   26   28   39   50   61   72   74    4   15
   36   38   49   60   71   73    3   14   25
   37   48   59   70   81    2   13   24   35

>> A .* eye(9)
ans =

   47    0    0    0    0    0    0    0    0
    0   68    0    0    0    0    0    0    0
    0    0    8    0    0    0    0    0    0
    0    0    0   20    0    0    0    0    0
    0    0    0    0   41    0    0    0    0
    0    0    0    0    0   62    0    0    0
    0    0    0    0    0    0   74    0    0
    0    0    0    0    0    0    0   14    0
    0    0    0    0    0    0    0    0   35

>> sum(sum(A.*eye(9)))
ans =  369
>> sum(sum(A.*flipud(eye(9))))
ans =  369
>> eye(9)
ans =

Diagonal Matrix

   1   0   0   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0
   0   0   1   0   0   0   0   0   0
   0   0   0   1   0   0   0   0   0
   0   0   0   0   1   0   0   0   0
   0   0   0   0   0   1   0   0   0
   0   0   0   0   0   0   1   0   0
   0   0   0   0   0   0   0   1   0
   0   0   0   0   0   0   0   0   1

>> flipud(eye(9))
ans =

Permutation Matrix

   0   0   0   0   0   0   0   0   1
```

```
   0   0   0   0   0   0   0   1   0
   0   0   0   0   0   0   1   0   0
   0   0   0   0   0   1   0   0   0
   0   0   0   0   1   0   0   0   0
   0   0   0   1   0   0   0   0   0
   0   0   1   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0
   1   0   0   0   0   0   0   0   0

>> A=magic(3)
A =

   8   1   6
   3   5   7
   4   9   2

>> pinv(A)
ans =

   0.147222  -0.144444   0.063889
  -0.061111   0.022222   0.105556
  -0.019444   0.188889  -0.102778

>> temp=pinv(A)
temp =

   0.147222  -0.144444   0.063889
  -0.061111   0.022222   0.105556
  -0.019444   0.188889  -0.102778

>> temp*A
ans =

   1.0000e+00   2.0817e-16  -3.1641e-15
  -6.1062e-15   1.0000e+00   6.2450e-15
   3.0531e-15   4.1633e-17   1.0000e+00

>>
```

1.4 Plotting Data

1.4.1 绘制和可视化数据

```
>> t=[0:0.01:0.98];
>> t
t =

 Columns 1 through 4:

   0.00000   0.01000   0.02000   0.03000

 Columns 5 through 8:

   0.04000   0.05000   0.06000   0.07000

 Columns 9 through 12:
```

```
   0.08000   0.09000   0.10000   0.11000

 Columns 13 through 16:

   0.12000   0.13000   0.14000   0.15000

 Columns 17 through 20:

   0.16000   0.17000   0.18000   0.19000

 Columns 21 through 24:

   0.20000   0.21000   0.22000   0.23000

 Columns 25 through 28:

   0.24000   0.25000   0.26000   0.27000

 Columns 29 through 32:

>> y1=sin(2*pi*4*t);
>> plot(t,y1);
>> y2=cos(2*pi*4*t);
>> plot(t,y2);
>> plot(t,y1);
>> hold on;
>> plot(t,y2,'r');
>> xlabel('time')
>> ylabel('value')
>> legend('sin','cos')
>> title('my plot')
>> close
>> clf;#清除一幅图像
>> figure(1):plot(t,y1)
ans = [](1x0)
>>  figure(1):plot(t,y2)
ans = [](1x0)
>> subplot(1,2,1)#把图像分为1*2的格子，也就是前两个参数，然后他使用第一个格子，也就是第一个参数
>> plot(t,y1)
>> subplot(1,2,2);#把图像分为1*2的格子，也就是前两个参数，然后他使用第二个格子，也就是第二个参数
>> plot(t,y2)
>> axis([0.5 1 -1 1])
>>
>> A=magic(5)
A =

   17   24    1    8   15
   23    5    7   14   16
    4    6   13   20   22
   10   12   19   21    3
   11   18   25    2    9

>>
>> imagesc(A)#绘制一个5*5的矩阵，一个5*5的彩色格图，不同的颜色对应A矩阵中的不同值
>> imagesc(A),colorbar,colormap gray;#生成一个灰度分布图，并在右边也加入一个颜色条，所以这个颜色显示不同深
>> A(1,2)
```

```
ans =  24
>> A(4,5)
ans =  3
>> imagesc(magic(15)),colorbar,colormap gray;
>> a=1,b=2,c=3
a =  1
b =  2
c =  3
>> a=1;b=2;c=3;
#用分号代替逗号不会输出任何东西
>> a=1,b=2,c=3
#用逗号连接是另一种Octave中更便捷的方式
a =  1
b =  2
c =  3
>>
```

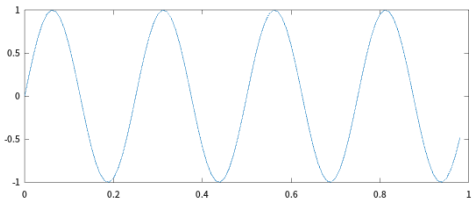

图 3: sin函数y1

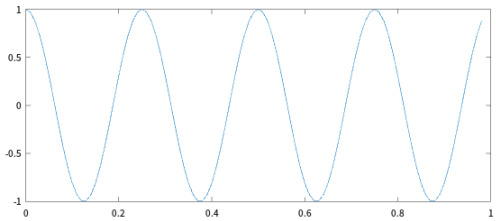

图 4: cos函数y2

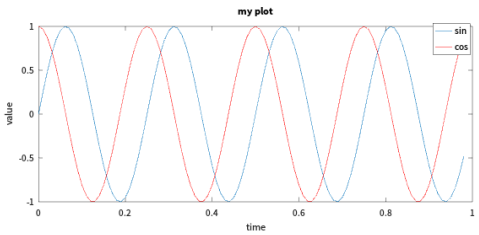

图 5: 两个函数图像画在一起（hold on）

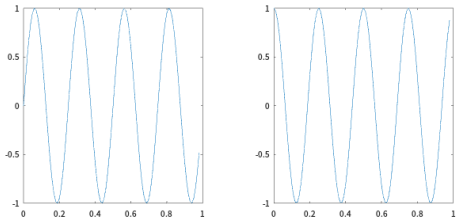

图 6: subplot

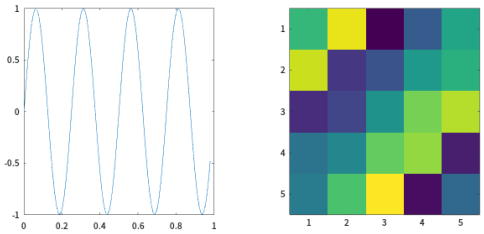

图 7: imagesc（A）

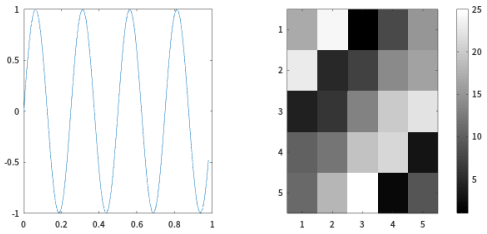

图 8: imagesc(A),colorbar,colormap gray;

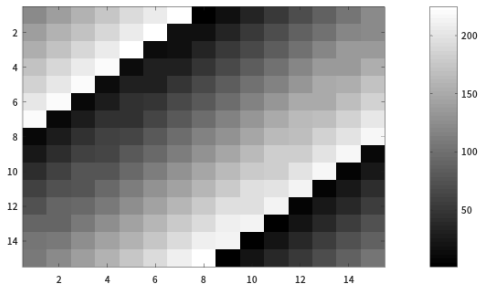

图 9: imagesc(magic(15)),colorbar,colormap gray;

1.5 Control Statements:for,while,if statement

1.5.1 for

```
>> v=zeros(10,1)
v =

   0
   0
   0
```

```
   0
   0
   0
   0
   0
   0
   0

>> for i=1:10,
v(i)=2^i;
end;
>> v
v =

      2
      4
      8
     16
     32
     64
    128
    256
    512
   1024

>> indices=1:10;
#通过设置你的 indices (索引) 等于 1 一直到 10,来做到这一点。这时indices 就是一个从 1 到 10 的序列。
>> indices
indices =

 Columns 1 through 9:

    1    2    3    4    5    6    7    8    9

 Column 10:

   10

>> for i=indices,
disp(i);
end;
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
>>
```

1.5.2 while

```
>> i=1;
>> while i<=5,
v(i)=100;
i=i+1;
end;
>> v
v =

    100
    100
    100
    100
    100
     64
    128
    256
    512
   1024

>> i=1;
>> while true,
v(i)=999;
i=i+1;
 if i==6,
   break;
 end;
end;
>> v
v =

    999
    999
    999
    999
    999
     64
    128
    256
    512
   1024

>>
```

1.5.3 if

```
>> i=1;
>> while true,
v(i)=999;
i=i+1;
 if i==6,
   break;
 end;
end;
>> v
v =
```

```
    999
    999
    999
    999
    999
     64
    128
    256
    512
   1024

>>
```

1.5.4 if-else

```
>> v(1)
ans =  999
>> v(1)=2;
>> if v(1)==1,
disp('The value is one');
elseif v(1)==2,
disp('The value is two');
else
disp('The value is not one or two.');
end;
The value is two
>>
>> quit #退出Octave
```

1.5.5 Function

1.如何定义和调用函数

```
function y = squareThisNumber(x)
#Octave 这个函数有一个参数,就是参数 x,还有定义的函数体,也就是 y 等于 x 的平方。
```

2.search path (搜索路径)
3.addpath 命令添加路径
4.更复杂的例子
有一个数据集,像这样,数据点为[1,1], [2,2], [3,3],要求计算不同θ值所对应的代价函数J。

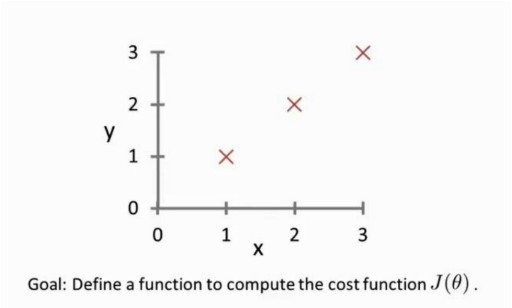

图 10: 数据集

首先让我们把数据放到 Octave 里,我把我的矩阵设置为 X = [1 1; 1 2; 1 3];

函数的定义:

```
function J = costFunctionJ(X, y, theta)

% X is the "design matrix" containing our training examples.
% y is the class labels

m = size(X,1);                 % number of training examples
predictions = X*theta;  % predictions of hypothesis on all m
examples
sqrErrors = (predictions-y).^2;  % squared errors


J = 1/(2*m) * sum(sqrErrors);
```

图 11: 函数的定义

当我在 Octave 里运行时,我键入 j = costFunctionJ(X, y, theta),它就计算出 j 等于0,这是因为如果我的数据集 x 为 [1;2;3], y 也为 [1;2;3] 然后设置 θ 0 等于 0, θ 1 等于1,这给了我恰好 45 度的斜线,这条线是可以完美拟合我的数据集的。

而相反地,如果我设置 theta 等于[0; 0],那么这个假设就是 0 是所有的预测值,和刚才一样,设置 θ 0 = 0, θ 1 也等于 0,然后我计算的代价函数,结果是 2.333。实际上,他就等于 1 的平方,也就是第一个样本的平方误差,加上 2 的平方,加上 3 的平方,然后除以 2m,也就是训练样本数的两倍,这就是 2.33。

```
>> (1^2+2^2+3^2)/(2*3)
ans= 2.3333
>>
```

1.6 Vectorization

Vectorization example:

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^T x \tag{1}$$

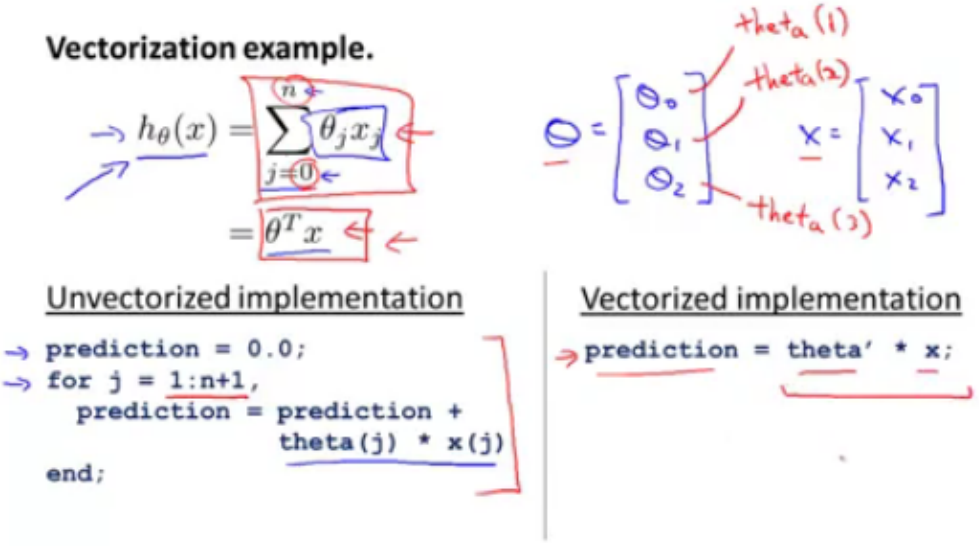

图 12: 向量化

如果你想要计算$h_\theta(x)$注意到右边是求和,那么你可以自己计算 j = 0 到 j = n 的和。但换另一种方式来想想,把$h_\theta(x)$ 看作$\theta^T x$,那么你就可以写成两个向量的内积,其中θ就是θ_0,θ_1,θ_2如果你有两个特征量,如果n =2,并且如果你把 x 看作x_0,x_1,x_2,这两种思考角度,会给你两种不同的实现方式.

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \tag{2}$$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \tag{3}$$

未向量化的代码实现方式：用一个 for 循环对 n 个元素进行加和。

Unvectorized implementation

```
prediction = 0.0;
for j = 1:n+1,
  prediction = prediction +
               theta(j) * x(j)
end;
```

图 13: 未向量化代码实现

计算$h_\theta(x)$是未向量化的,我们可能首先要初始化变量 prediction 的值为 0.0,而这个变 量 prediction 的最终结果就是$h_\theta(x)$,然后我要用一个 for 循环,j 取值 0 到 n+1,变量 prediction 每次就通过自身加上 theta(j) 乘以 x(j) 更新值,这个就是算法的代码实现。

提醒：这里的向量我用的下标是 0,所以我有 θ_0,θ_1,θ_2，但因为 MATLAB 的下标从 1 开始,在 MATLAB 中θ_0,我们可能会用 theta(1) 来表示,这第二个元素最后就会变成,theta(2) 而第三个元素,最终可能就用 theta(3) 表示,因为 MATLAB 中的下标从 1开始,这就是为什么这里我的 for 循环,j 取值从 1 直到 n+1,而不是从 0 到 n。

向量化的代码实现:

Vectorized implementation

```
prediction = theta' * x;
```

图 14: 向量化代码实现

把 x 和 θ 看做向量,而你只需要令变量 prediction 等于 theta 转置乘以 x,你就可以这样计算。与其写所有这些 for 循环的代码,你只需要一行代码,这行代码就是利用Octave 的高度优化的数值,线性代数算法来计算两个向量 θ 以及 x 的内积,这样向量化的实现更简单,它运行起来也将更加高效。

这就是 Octave 所做的而向量化的方法,在其他编程语言中同样可以实现。让我们来看 一个 C++ 的例子:

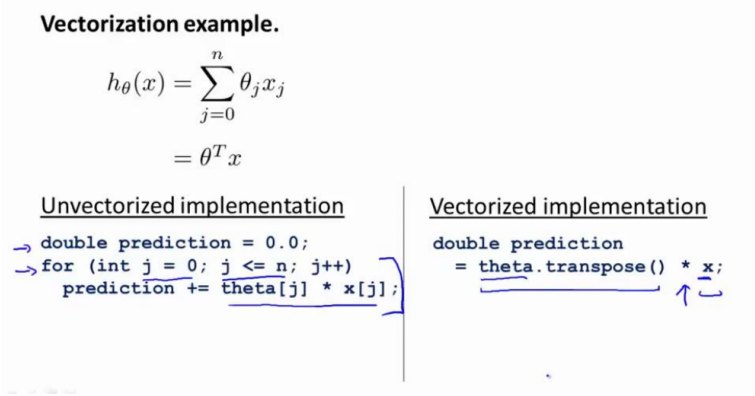

图 15: 向量化

与此相反,使用较好的 C++ 数值线性代数库,你可以写出像右边这样的代码,因此取 决于你的数值线性代数库的内容。你只需要在 C++ 中将两个向量相乘,根据你所使用的数 值和线性代数库的使用细节的不同,你最终使用的代码表达方式可能会有些许不同,但是通 过一个库来做内积,你可以得到一段更简单、更有效的代码。

现在,让我们来看一个更为复杂的例子,这是线性回归算法梯度下降的更新规则:

$$\begin{cases} \theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\ \theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)} \\ \theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)} \end{cases}$$
$$(n = 2)$$

图 16: 线性回归算法梯度下降规则

我们用这条规则对 j 等于 0、1、2 等等的所有值,更新对象 θ_j,我只是用 θ_0,θ_1,θ_2来写方程,假设我们有两个特征量,所以 n 等于 2,这些都是我们需要对 θ_0,θ_1,θ_2 进行更新,这些都应该是同步更新,我们用一个向量化的代码实现,这里是和之前相同的三个方程,只不过写得小一点而已。

可以想象实现这三个方程的方式之一,就是用一个 for 循环,就是让 j 等于 0、等于 1、等于 2,来更新θ_j。

我们用向量化的方式来实现,看看我们是否能够有一个更简单的方法。基本上用三行代码或者一个 for 循环,一次实现这三个方程。

怎样能用这三步,并将它们压缩成一行向量化的代码来实现。做法如下:

(1) 把 θ 看做一个向量,然后我用 θ - α 乘以某个别的向量 δ 来更新 θ。

$$\delta = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \tag{4}$$

实现过程：我要把 θ 看作一个向量,有一个 n+1 维向量,α 是一个实数,δ 在这里是一个向量。

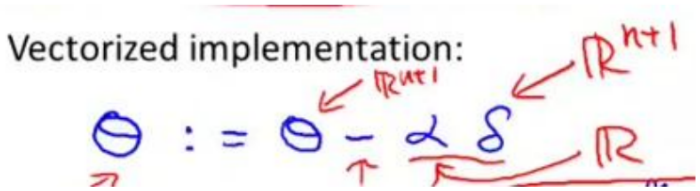

图 17: 线性回归算法梯度下降规则

所以这个减法运算是一个向量减法,因为 α 乘以 δ 是一个向量,所以 θ 就是 θ - α δ 得到的向量。

向量δ:

$$\delta = \begin{bmatrix} \delta_0 \\ \delta_1 \\ \delta_2 \end{bmatrix} \tag{5}$$

$x^{(i)}$是一个向量:

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \end{bmatrix} \tag{6}$$

就会得到这些不同的式子,然后作加和

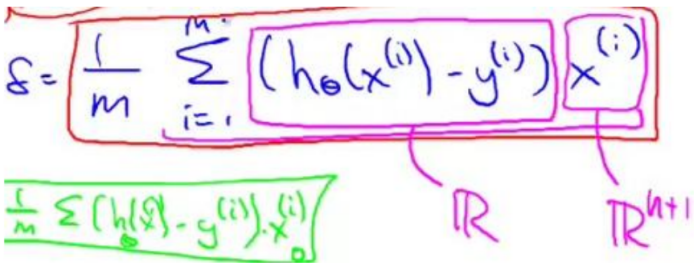

图 18: 表达式

实际上,在以前的一个小测验,如果你要解这个方程,我们说过为了向量化这段代码, 我们会令 u = 2v +5w 因此,我们说向量 u 等于 2 乘以向量 v 加上 5 乘以向量 w。用这个例子说明,如何对不同的向量进行相加,这里的求和是同样的道理。

$$u(j) = 2v(j) + 5w(j)(forall j) \tag{7}$$

$$u = 2v + 5w \tag{8}$$

这就是为什么我们能够向量化地实现线性回归。