Ảnh có chứa bánh, LEGO, đồ chơi, bàn

Mô tả được tạo với mức tin cậy rất cao MACHINE LEARNING

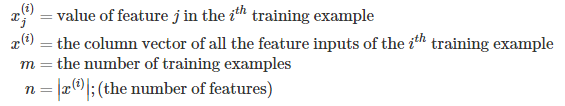
Standford University

**Lecture 2:**

Chapter 1:Linear Regression with Multiple Variables

Linear regression with multiple variables is also known as "multivariate linear regression".

We now introduce notation for equations where we can have any number of input variables.

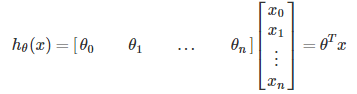


Now define the multivariable form of the hypothesis function as follows, accommodating these multiple features:



In order to develop intuition about this function, we can think about *θ*0 as the basic price of a house, *θ*1 as the price per square meter, *θ*2 as the price per floor, etc. *x*1 will be the number of square meters in the house, *x*2 the number of floors, etc.

Using the definition of matrix multiplication, our multivariable hypothesis function can be concisely represented as:

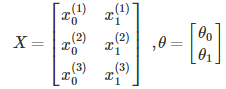


This is a vectorization of our hypothesis function for one training example; see the lessons on vectorization to learn more.

Remark: Note that for convenience reasons in this course Mr. Ng assumes 

[**Note**: So that we can do matrix operations with theta and x, we will set *x*(*i*)0 = 1, for all values of i. This makes the two vectors 'theta' and *x*(*i*)match each other element-wise (that is, have the same number of elements: n+1).]

The training examples are stored in X row-wise, like such:



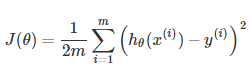
You can calculate the hypothesis as a column vector of size (m x 1) with:

*hθ*(*X*)=*Xθ*

**For the rest of these notes, and other lecture notes, X will represent a matrix of training examples***x*(*i*)**stored row-wise.**

**Cost function**

For the parameter vector θ (of type R*n*+1 or in R(*n*+1)×1, the cost function is:



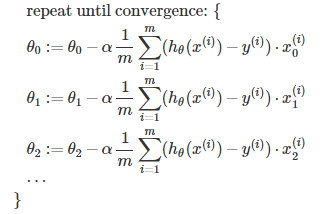
The vectorized version is:



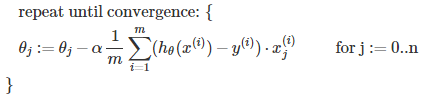
Where *y*⃗  denotes the vector of all y values.

**Gradient Descent for Multiple Variables**

The gradient descent equation itself is generally the same form; we just have to repeat it for our 'n' features:



In other words:

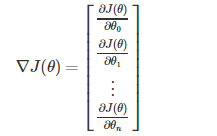


Matrix Notation

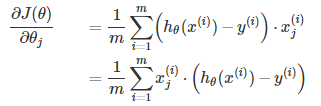
The Gradient Descent rule can be expressed as:



Where ∇*J*(*θ*) is a column vector of the form:



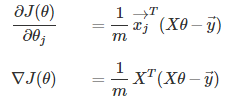
The j-th component of the gradient is the summation of the product of two terms:



Sometimes, the summation of the product of two terms can be expressed as the product of two vectors.

Here, *x*(*i*)*j*, for i = 1,...,m, represents the m elements of the j-th column, *xj*→ , of the training set X.

The other term (*hθ*(*x*(*i*))−*y*(*i*)) is the vector of the deviations between the predictions *hθ*(*x*(*i*)) and the true values *y*(*i*). Re-writing , we have:



Finally, the matrix notation (vectorized) of the Gradient Descent rule is:



Feature Normalization

We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. Ideally:

−1 ≤ *x*(*i*) ≤ 1

or

−0.5 ≤ *x*(*i*) ≤ 0.5

These aren't exact requirements; we are only trying to speed things up. The goal is to get all input variables into roughly one of these ranges, give or take a few.

Two techniques to help with this are **feature scaling** and **mean normalization**. Feature scaling involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable, resulting in a new average value for the input variable of just zero. To implement both of these techniques, adjust your input values as shown in this formula:



Where *μi* is the **average** of all the values for feature (i) and *si* is the range of values (max - min), or *si* is the standard deviation.

Note that dividing by the range, or dividing by the standard deviation, give different results. The quizzes in this course use range - the programming exercises use standard deviation.

Example: *xi* is housing prices with range of 100 to 2000, with a mean value of 1000. Then

.

Quiz question #1 on Feature Normalization (Week 2, Linear Regression with Multiple Variables)

Your answer should be rounded to exactly two decimal places. Use a '.' for the decimal point, not a ','. The tricky part of this question is figuring out which feature of which training example you are asked to normalize. Note that the mobile app doesn't allow entering a negative number (Jan 2016), so you will need to use a browser to submit this quiz if your solution requires a negative number.

Gradient Descent Tips

**Debugging gradient descent.** Make a plot with *number of iterations* on the x-axis. Now plot the cost function, J(θ) over the number of iterations of gradient descent. If J(θ) ever increases, then you probably need to decrease α.

**Automatic convergence test.** Declare convergence if J(θ) decreases by less than E in one iteration, where E is some small value such as 10−3. However in practice it's difficult to choose this threshold value.

It has been proven that if learning rate α is sufficiently small, then J(θ) will decrease on every iteration. Andrew Ng recommends decreasing α by multiples of 3.

Features and Polynomial Regression

We can improve our features and the form of our hypothesis function in a couple different ways.

We can **combine** multiple features into one. For example, we can combine *x*1 and *x*2 into a new feature *x*3 by taking *x*1⋅*x*2.

**Polynomial Regression**

Our hypothesis function need not be linear (a straight line) if that does not fit the data well.

We can **change the behavior or curve** of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).

For example, if our hypothesis function is *hθ*(*x*)=*θ*0+*θ*1*x*1 then we can create additional features based on *x*1, to get the quadratic function *hθ*(*x*)=*θ*0+*θ*1*x*1+*θ*2*x*21 or the cubic function *hθ*(*x*)=*θ*0+*θ*1*x*1+*θ*2*x*21+*θ*3*x*31

In the cubic version, we have created new features *x*2 and *x*3 where *x*2=*x*21 and *x*3=*x*31.

To make it a square root function, we could do:



Note that at 2:52 and through 6:22 in the "Features and Polynomial Regression" video, the curve that Prof Ng discusses about "doesn't ever come back down" is in reference to the hypothesis function that uses the sqrt() function (shown by the solid purple line), not the one that uses *size*2(shown with the dotted blue line). The quadratic form of the hypothesis function would have the shape shown with the blue dotted line if *θ*2 was negative.

One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

eg. if *x*1 has range 1 - 1000 then range of *x*21 becomes 1 - 1000000 and that of *x*31 becomes 1 - 1000000000.

Normal Equation

The "Normal Equation" is a method of finding the optimum theta **without iteration.**

*θ*=(*XTX*)−1*XTy*

There is **no need** to do feature scaling with the normal equation.

Mathematical proof of the Normal equation requires knowledge of linear algebra and is fairly involved, so you do not need to worry about the details.

Proofs are available at these links for those who are interested:

<https://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)>

<http://eli.thegreenplace.net/2014/derivation-of-the-normal-equation-for-linear-regression>

The following is a comparison of gradient descent and the normal equation:

|  |  |
| --- | --- |
| **Gradient Descent** | **Normal Equation** |
| Need to choose alpha | No need to choose alpha |
| Needs many iterations | No need to iterate |
| O (*kn*2) | O (*n*3), need to calculate inverse of *XTX* |
| Works well when n is large | Slow if n is very large |

With the normal equation, computing the inversion has complexity O(*n*3). So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

**Normal Equation Noninvertibility**

When implementing the normal equation in octave we want to use the 'pinv' function rather than 'inv.'

*XTX* may be **noninvertible**. The common causes are:

* Redundant features, where two features are very closely related (i.e. they are linearly dependent)
* Too many features (e.g. m ≤ n). In this case, delete some features or use "regularization" (to be explained in a later lesson).

Solutions to the above problems include deleting a feature that is linearly dependent with another or deleting one or more features when there are too many features.

ML:Octave Tutorial

Basic Operations



PS1('>> ');

%% Change working directory in windows example:

cd 'c:/path/to/desired/directory name'

%% Note that it uses normal slashes and does not use escape characters for the

  empty spaces.

%% elementary operations

5+6

3-2

5\*8

1/2

2^6

1 == 2 % false

1 ~= 2 % true. note, not "!="

1 && 0

1 || 0

xor(1,0)

%% variable assignment

a = 3; % semicolon suppresses output

b = 'hi';

c = 3>=1;

% Displaying them:

a = pi

disp(a)

disp(sprintf('2 decimals: %0.2f', a))

disp(sprintf('6 decimals: %0.6f', a))

format long

a

format short

a

%% vectors and matrices

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

v = [1 2 3]

v = [1; 2; 3]

v = 1:0.1:2 % from 1 to 2, with stepsize of 0.1. Useful for plot axes

v = 1:6 % from 1 to 6, assumes stepsize of 1 (row vector)

C = 2\*ones(2,3) % same as C = [2 2 2; 2 2 2]

w = ones(1,3) % 1x3 vector of ones

w = zeros(1,3)

w = rand(1,3) % drawn from a uniform distribution

w = randn(1,3)% drawn from a normal distribution (mean=0, var=1)

w = -6 + sqrt(10)\*(randn(1,10000)); % (mean = -6, var = 10) - note: add the

  semicolon

hist(w) % plot histogram using 10 bins (default)

hist(w,50) % plot histogram using 50 bins

% note: if hist() crashes, try "graphics\_toolkit('gnu\_plot')"

I = eye(4) % 4x4 identity matrix

% help function

help eye

help rand

help help

Moving Data Around

*Data files used in this section*: [featuresX.dat](https://raw.githubusercontent.com/tansaku/py-coursera/master/featuresX.dat), [priceY.dat](https://raw.githubusercontent.com/tansaku/py-coursera/master/priceY.dat)



%% dimensions

sz = size(A) % 1x2 matrix: [(number of rows) (number of columns)]

size(A,1) % number of rows

size(A,2) % number of cols

length(v) % size of longest dimension

%% loading data

pwd % show current directory (current path)

cd 'C:\Users\ang\Octave files' % change directory

ls % list files in current directory

load q1y.dat % alternatively, load('q1y.dat')

load q1x.dat

who % list variables in workspace

whos % list variables in workspace (detailed view)

clear q1y % clear command without any args clears all vars

v = q1x(1:10); % first 10 elements of q1x (counts down the columns)

save hello.mat v; % save variable v into file hello.mat

save hello.txt v -ascii; % save as ascii

% fopen, fread, fprintf, fscanf also work [[not needed in class]]

%% indexing

A(3,2) % indexing is (row,col)

A(2,:) % get the 2nd row.

% ":" means every element along that dimension

A(:,2) % get the 2nd col

A([1 3],:) % print all the elements of rows 1 and 3

A(:,2) = [10; 11; 12] % change second column

A = [A, [100; 101; 102]]; % append column vec

A(:) % Select all elements as a column vector.

% Putting data together

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

B = [11 12; 13 14; 15 16] % same dims as A

C = [A B] % concatenating A and B matrices side by side

C = [A, B] % concatenating A and B matrices side by side

C = [A; B] % Concatenating A and B top and bottom

Computing on Data



%% initialize variables

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

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

C = [1 1;2 2]

v = [1;2;3]

%% matrix operations

A \* C % matrix multiplication

A .\* B % element-wise multiplication

% A .\* C or A \* B gives error - wrong dimensions

A .^ 2 % element-wise square of each element in A

1./v % element-wise reciprocal

log(v) % functions like this operate element-wise on vecs or matrices

exp(v)

abs(v)

-v % -1\*v

v + ones(length(v), 1)

% v + 1 % same

A' % matrix transpose

%% misc useful functions

% max (or min)

a = [1 15 2 0.5]

val = max(a)

[val,ind] = max(a) % val - maximum element of the vector a and index - index

  value where maximum occur

val = max(A) % if A is matrix, returns max from each column

% compare values in a matrix & find

a < 3 % checks which values in a are less than 3

find(a < 3) % gives location of elements less than 3

A = magic(3) % generates a magic matrix - not much used in ML algorithms

[r,c] = find(A>=7) % row, column indices for values matching comparison

% sum, prod

sum(a)

prod(a)

floor(a) % or ceil(a)

max(rand(3),rand(3))

max(A,[],1) - maximum along columns(defaults to columns - max(A,[]))

max(A,[],2) - maximum along rows

A = magic(9)

sum(A,1)

sum(A,2)

sum(sum( A .\* eye(9) ))

sum(sum( A .\* flipud(eye(9)) ))

% Matrix inverse (pseudo-inverse)

pinv(A) % inv(A'\*A)\*A'

Plotting Data



%% plotting

t = [0:0.01:0.98];

y1 = sin(2\*pi\*4\*t);

plot(t,y1);

y2 = cos(2\*pi\*4\*t);

hold on; % "hold off" to turn off

plot(t,y2,'r');

xlabel('time');

ylabel('value');

legend('sin','cos');

title('my plot');

print -dpng 'myPlot.png'

close; % or, "close all" to close all figs

figure(1); plot(t, y1);

figure(2); plot(t, y2);

figure(2), clf; % can specify the figure number

subplot(1,2,1); % Divide plot into 1x2 grid, access 1st element

plot(t,y1);

subplot(1,2,2); % Divide plot into 1x2 grid, access 2nd element

plot(t,y2);

axis([0.5 1 -1 1]); % change axis scale

%% display a matrix (or image)

figure;

imagesc(magic(15)), colorbar, colormap gray;

% comma-chaining function calls.

a=1,b=2,c=3

a=1;b=2;c=3;

Control statements: for, while, if statements



v = zeros(10,1);

for i=1:10,

v(i) = 2^i;

end;

% Can also use "break" and "continue" inside for and while loops to control

  execution.

i = 1;

while i <= 5,

v(i) = 100;

i = i+1;

end

i = 1;

while true,

v(i) = 999;

i = i+1;

if i == 6,

break;

end;

end

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

Functions

To create a function, type the function code in a text editor (e.g. gedit or notepad), and save the file as "functionName.m"

Example function:



function y = squareThisNumber(x)

y = x^2;

To call the function in Octave, do either:

1) Navigate to the directory of the functionName.m file and call the function:



% Navigate to directory:

cd /path/to/function

% Call the function:

functionName(args)

2) Add the directory of the function to the load path and save it:**You should not use addpath/savepath for any of the assignments in this course. Instead use 'cd' to change the current working directory. Watch the video on submitting assignments in week 2 for instructions.**



% To add the path for the current session of Octave:

addpath('/path/to/function/')

% To remember the path for future sessions of Octave, after executing

      addpath above, also do:

savepath

Octave's functions can return more than one value:



function [y1, y2] = squareandCubeThisNo(x)

y1 = x^2

y2 = x^3

Call the above function this way:



[a,b] = squareandCubeThisNo(x)

Vectorization

Vectorization is the process of taking code that relies on **loops** and converting it into **matrix operations**. It is more efficient, more elegant, and more concise.

As an example, let's compute our prediction from a hypothesis. Theta is the vector of fields for the hypothesis and x is a vector of variables.

With loops:



prediction = 0.0;

for j = 1:n+1,

prediction += theta(j) \* x(j);

end;

With vectorization:



prediction = theta' \* x;

If you recall the definition multiplying vectors, you'll see that this one operation does the element-wise multiplication and overall sum in a very concise notation.

Working on and Submitting Programming Exercises

1. Download and extract the assignment's zip file.
2. Edit the proper file 'a.m', where a is the name of the exercise you're working on.
3. Run octave and cd to the assignment's extracted directory
4. Run the 'submit' function and enter the assignment number, your email, and a password (found on the top of the "Programming Exercises" page on coursera)

Video Lecture Table of Contents

**Basic Operations**



0:00 Introduction

3:15 Elementary and Logical operations

5:12 Variables

7:38 Matrices

8:30 Vectors

11:53 Histograms

12:44 Identity matrices

13:14 Help command

**Moving Data Around**



0:24 The size command

1:39 The length command

2:18 File system commands

2:25 File handling

4:50 Who, whos, and clear

6:50 Saving data

8:35 Manipulating data

12:10 Unrolling a matrix

12:35 Examples

14:50 Summary

**Computing on Data**



0:00 Matrix operations

0:57 Element-wise operations

4:28 Min and max

5:10 Element-wise comparisons

5:43 The find command

6:00 Various commands and operations

**Plotting data**



0:00 Introduction

0:54 Basic plotting

2:04 Superimposing plots and colors

3:15 Saving a plot to an image

4:19 Clearing a plot and multiple figures

4:59 Subplots

6:15 The axis command

6:39 Color square plots

8:35 Wrapping up

**Control statements**



0:10 For loops

1:33 While loops

3:35 If statements

4:54 Functions

6:15 Search paths

7:40 Multiple return values

8:59 Cost function example (machine learning)

12:24 Summary

**Vectorization**



0:00 Why vectorize?

1:30 Example

4:22 C++ example

5:40 Vectorization applied to gradient descent

9:45 Python