Machine Learning

Week 1

I. Introduction

Types of machine learning

- Supervised learning (the “right answer” is provided as input, in the “training set”)
  - Regression problem (expected output is real-value)
  - Classification problem (answer is a class, such as yes or no)
- Unsupervised learning

II. Linear Regression with One Variable

<u>**Notation**</u>

m: Number of training examples
x’s: “input” variables (features)
y’s: “output” (targets) variables
(x, y): one training example
($x^{(i)}$, $y^{(i)}$): i th training example
h(): function (hypothesis) found by the learning algorithm
θ: (theta) the “parameters” used in h() together with the features x
J(θ): the cost function of h_θ
n: number of features (inputs)
$x^{(i)}$: inputs (features) of the i th training example
$x_j^{(i)}$: the value of feature j in the i th training example
λ: (lambda) regularization parameter
:= means assignment in algorithm, rather than mathematical equality

Possible h (“hypothesis”) functions

- Linear regression with one variable (a.k.a. univariate linear regression):
  - $h_\theta(x) = \theta_0 + \theta_1 x$ Shorthand: h(x)   Where θ_0 and θ_1 are “parameters”
- Linear regression with multiple variables (a.k.a. multivariate linear regression):
  - $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 \ldots \theta_n x_n$ (for n features)
- Polynomial regression
  - $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 (x_1)^2 + \theta_3 (x_1)^3$ (e.g. by just making up new features that are the square and cube of an existing feature)

Cost function

The cost function J() evaluates how close h(x) match y given the parameters finding the parameters θ used by h(). For linear regression (here with *one* feature x)
pick θ_0 and θ_1 so that $h_\theta(x)$ is close to y for our training examples (x,y)

i.e. minimize the “**sum of square errors**” cost function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

By taking the *square* of the error (i.e. $h_\theta(x) - y$), we avoid having too small results from h_θ cancelling out too large results (as $-1^2 == 1$), thus yielding a truer “cost” of the errors.

N.B. $\frac{1}{2m}$ ”makes some of the math easier” (see explanation why).

“Squared error” cost function: a reasonable choice for cost function. The most common one for regression problems.

Gradient descent
Iterative algorithm for finding a local minimum for the cost function. Works for linear regression with any number of parameters, but also other kinds of “hypotheses” functions. Scales better for cases with large number of features than *solving* for the optimal min J()
$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ (for j=0 and j=1, repeat until convergence)
where
θ_j is the “parameter” in h_θ that is used for feature j
α is the learning rate
$\frac{\partial}{\partial \theta_j}$ is the partial derivative (slope) at the current point θ_j
$J(\theta_0, \theta_1)$ is the cost function (in this case with two parameters, for cases with only one feature)
N.B. update θ_0 and θ_1 *simultaneously! (i.e. as one atomic operation)*

Learning rate
The size α of each step when iterating to find a solution. N.B. no need to vary α between iterations. Gradient descent will naturally take smaller and smaller steps the closer we get to a solution.

Gradient descent for two-parameter linear regression (repeat until convergence)
for j = 0 and j = 1

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} \left(\frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \right)$$

simplifies to

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

“Batch” gradient descent
Just means each iteration of the gradient is applied to all the training examples (in a “batch”).

III. Linear Algebra Revision (optional)

[...]

Week 2

IV. Linear Regression with Multiple Variables

Large number of feature

For problems involving many "features" (i.e. x_0,x_1,x_2,x_3 ... x_n) linear algebra vector notation is more efficient.

Multi-parameter linear regression in vector notation

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 \ldots \theta_n x_n$

for convenience of notation, and to allow use of vector multiplication, define a 0th feature x_0 = 1, thus we can write

$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 \ldots \theta_n x_n$

Now, defining a ((n+1) × 1) vector x containing all the features and a ((n+1) × 1) vector θ containing all the parameters for the hypothesis function h_θ, we can efficiently multiply the two (yielding a scalar result) if we first transpose (rotate) the θ into θ^T

$h_\theta(x) = \theta^T x$ in Octave: **theta' * x**

Picking features

Use your domain insights and intuitions to pick features. E.g. deriving a combined feature might help. There are also automatic algorithms for picking features.

Cost function for multiple features

For n+1 features (where x_0 = 1) the combined cost function over m training samples will be

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

which really means (note that i starts from 1 and j starts from 0)

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} ((\sum_{j=0}^{n} \theta_j x_j^{(i)}) - y^{(i)})^2$$

Gradient descent for the cost function of multi-parameter linear regression

with the number of features n >= 1 and x_0 = 1, one iteration j of the gradient descent is

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

in Octave: **theta = theta - alpha * (1/m) * sum((theta' * x - y) * x)** thus (atomically updating θ_j for j = 0, ..., n)

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)}$$

...

$$\theta_n := \theta_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_n^{(i)}$$

Practical considerations for gradient descent

Feature scaling

Make sure the values in each group of properties x_n are on the same order (i.e. scale some groups if necessary), or the gradient descent will take a long time to converge (because the contour plot is too elongated). Typically, make all groups contain values $-1 \le x_i \le 1$

Mean normalization

To make sure the values for feature x_i range between -1 and +1, replace x_i with $x_i - \mu_i$ where μ_i is the mean of all values in the training set for feature x_i *(excluding x_0 as it is always 1)*

So together, $x_i := \frac{x_i - \mu_i}{s_i}$ where μ_i is the mean of all values for the property in the training set and s_i is the range of values (i.e. max(x) - min(x)) for the property i.

How to pick learning rate α

The number of iterations before linear descent converges can vary a lot (anything between 30 and 3 million is normal).

To make sure the linear descent works, plot the running-minimum of the cost function J(θ) and make sure it decrease for each iteration.

Automatic convergence test; e.g. declare convergence if J(θ) change by less than 10^{-3} in one iteration. However looking at the plot is usually better.

If J(θ) is increasing rather than decreasing (or oscillating) then the usual reason is that α is too big.

Too small α will result is too slow change in J(θ).

Trying to determine α heuristically, try steps of ≈ x3, so 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, ...

Polynomial regression

If a simple straight line does not fit the training data well, then polynomial regression can be used.

Just define some new feature that are the square and cube of an existing feature.
E.g. $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 (x_1)^2 + \theta_3 (x_1)^3$

Note that feature scaling is extra important when using squared and cubed features.

In addition to squaring and cubing a features, also consider adding the root of the feature.

“Normal equation”: Using matrix multiplication to solve for θ that gives min J(θ)
For n features and m training examples, create a (m × n+1) matrix X (the “design matrix”) from the features x in the training set, where the first column is for feature x_0 that is always 1, and a (m × 1) vector y containing the y values of the training set. Then this will give the parameters θ that minimize the cost function J for h_θ:

$\theta = (X^T X)^{-1} X^T y$ in Octave: **pinv(X’ * X) * X’ * y**

N.B. unlike when using gradient descent for finding θ that gives min J(θ), feature scaling is not necessary when solving for θ as shown above.

Gradient descent vs Solving for θ (“normal equation”)
Gradient descent
Pros: 1) Works well even for large number of features,
Cons: 1) Need to choose learning rate, 2) Needs many iterations (may be slow)
Solving for θ
Pros: 1) No need to choose learning rate, 2) No need to iterate
Cons: 1) Need to compute inverse of n×n matrix, which is done in $O(n^3)$ time, so slow for very large number of features (> 1,000-10,000 features), 2) $X^T X$ might be non-invertible (rarely)

Using matrices to process multiple training cases at once
By replacing the vector-based operations in the formulas mentioned above with matrix-based operations, all the training cases can be considered at once (instead of iterating over the set one case at the time). This allows making use of Octave’s much faster low-level matrix multiplication implementation.

To calculate h for multiple training cases, make X a (m × n+1) matrix populated with the *transposed* feature vectors $x^{(i)}$ one per row, and make θ a (n+1 × 1) vector (as before). The result h_θ shall be a (m × 1) vector. Now, because of the non-commutative nature of matrix multiplication, the $h_\theta(x) = \theta^T x$ formula *won’t work for X as is*. To yield a (m × 1) vector we need to multiply the (m × n+1) matrix X with the (n+1 × 1) vector θ in that order. I.e. the matrix version of the $h_\theta()$ functions is

$h\theta(X) = X\theta$ in Octave: **X * theta**

To calculate J(θ) for multiple training cases, use the same version of h above. Note that to take the power of $(h_\theta - y)$ when h_θ is a matrix we need expand it and transpose one of the terms, i.e. $(X\theta - y)^2 \Rightarrow (X\theta - y)^T(X\theta - y)$, and when using matrixes to hold multiple training cases the summation goes away!

$J(\theta) = \frac{1}{2m}(X\theta - y)^T(X\theta - y)$

in Octave:

J = (1/(2*m)) * (X*theta - y)' * (X*theta - y)

To perform one iteration in gradient descent that considers all training cases at once,

$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

needs to be changed into

$\theta_j := \theta_j - \alpha\frac{1}{m}X^T(h_\theta(X) - y)$

Note that with a matrix holding all features for all training sets, again the need to sum goes away!

in Octave:

d = (1/m) * X' * (X*theta - y); % sum i = 0 to m goes away!

theta = theta - alpha * d

V. Octave Tutorial

Comments

%

Assignment

a = 4

a = 4; % Suppress the output

Display

Print the value of a: **disp(a)**

Format string: **sprintf('6 decimals: %0.6f', a)**

Type formatting of subsequent outputs: **format short** or **format long**

Histogram: **hist(x, 50)**% matrix x in 50 bins

List variables in current scope: **who**

Detailed list of variables in current scope: **whos**

Plot

Plotting the graph for a function y1: **t = [0:0.01:0.98]; y1 = sin(2*pi*4*t); plot(t, y1);**

Incrementally add to the current graph: **hold on**

Label the axes: **xlabel('time'); ylabel('value')**

Add a legend: **legend('sin', 'cos')**

Save plot to file: **print -dpng 'myPlot.png'**

Close the figure window: **close**

Switch between figures: **figure(1); figure(2)**

Divide plot into 1x2 grid, access first element: **subplot(1,2,1)**
Change scale of axes: **axis([0.5 1 -1 1])**

Matrices and vectors
1 x 3 matrix: **[1 2 3]**
3 x 3 matrix: **[1 2 3; 4 5 6; 7 8 9]**
3 x 1 matrix: **[1; 2; 3]**
row matrix with values from 1 to 2 in increments of 0.1: **[1:0.1:2]**

2 x 3 matrix of all 0: **zeros(2,3)**
2 x 3 matrix of all 1: **ones(2,3)**
2 x 3 matrix of all 2: **ones(2,3) * 2**
3 x 3 matrix of all 1: **ones(3)**

Identity matrices
3 x 3 identity matrix: **eye(3)**

Random
3 x 4 matrix with all random numbers (between 0 and 1): **randn(3, 4)**

Magic squares
n x n matrix where sums of all columns, rows, diagonals are the same: **magic(5)**

Transpose
transpose of X = [1 2 3]: **X’** results in [1; 2; 3]

Flip Up-Down
flipud(A)

Size
Size of 2 x 3 matrix A: **size(A)**   results in a vector [2 3]
Number of rows in 2 x 3 matrix A: **size(A, 1)**   results in 2
Number of columns in 2 x 3 matrix A: **size(A, 2)**   results in 3
The longest dimension of 2 x 3 matrix A: **length(A)**   results in 3

Importing data
Display current path: **pwd**
Change path: **cd**
Load data from file feature.dat: **load(‘features.dat’)**
Clear variable A: **clear A**
Assign items 10 through 20 of vector v to vector a: **a = v(10:20)**
Save variable v to binary format file: **save hello.mat v**
Loading variable v back from file: **load hello.mat**   % automatically assigned to v!
Save to text file: **save hello.txt v -ascii**

Indexing and assignment
Get element $A_{2,3}$: **A(2,3)**
Get 2nd row vector: **A(2,:)**
Get 2nd column vector: **A(:,2)**
Get 1st and 3rd row: **A([1 3], :)**
Assign vector to 2nd column: **A(:,2) = [10; 11; 12]**
Append another column vector to a matrix: **A = [A, [1; 2; 3]]**
Put all elements in matrix A into a single column: **A(:)**
Concatenate matrix B to the right of A into C: **C = [A B]**
Concatenate matrix B below A into C: **C = [A; B]**

Mathematical operations
Multiply matrix A with matrix B: **A * B**
Multiply matrix A *element-wise* with matrix B: **A .* B**
Square every element in matrix A: **A .^ 2**
Element-wise inverse of A: **1 ./ A**
Absolute values of A: **abs(A)**
Max value of elements in v: **max(v)**
Check which elements pass the condition: **v < 3**   results in e.g. [1 0 1 1]
Return the values of the elements that pass the condition: **find(v < 3)**
Indices of all elements that pass the condition: **[i, j] = find(A < 3)**
Sum of all elements of A: **sum(A)**
Per-column sum of all elements of A: **sum(A,1)**
Per-row sum of all elements of A: **sum(A,2)**
Product of all elements of A: **prod(A)**
Round elements of A: **round(A)**
Round elements of A down: **floor(A)**
Round elements of A up: **ceil(A)**
Per-column maximum of matrix A: **max(A, [], 1)**
Per-row maximum of matrix A: **max(A, [], 2)**
Max of all the values in A: **max(max(A))** or **max(A(:))**

Flow control
For-loop from 1 to 10: **for i = 1:10, v(i) = 2^i; end;**
While-loop: **while i <= 5, v(i) = 100; i = i + 1; end;**
If-else statement: **if something==true, do(); elseif other ~= false, do2(); else do3(); end;**

Functions
Create a file with the extension .m, using the name of the function as the name of the file. State the function signature (including named return value!) on the first line in the file (prefixed by “function”), then leave a blank like, and implement the function below it.

Call **addpath(...)** with the path to the directory containing your function files.

Week 3

VI. Logistic Regression

Classification problems
These are problems to which the answer is one of a number of classes, e.g

- Online transactions: Fraudulent / Legit
- Tumor: Malignant / Benign
- Email classification: Work / Friends / Family / Hobby groups

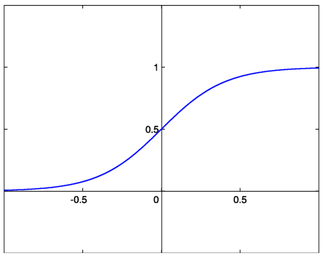
Two-class (or binary-class) classification
Formally $y \in \{0, 1\}$ where 0 is the "Negative class", and 1 is the "Positive class"

The *linear regression* technique as used earlier might work reasonably well for some training sets, but suffers badly in many cases (e.g. when the "spread" of the groups are not equal). It may also output answers not even in the 0..1 range, which is at least inappropriate.

Logistic regression
This is a better method for classification than pure linear regression. Note that it is a classification technique, despite being called logistic *"regression"*, but also that it is a modification of linear regression (i.e. filtered by g(), see below) rather than a completely different technique.

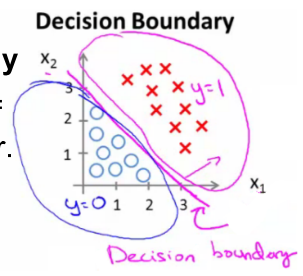
For logistic regression, the hypothesis function is

$h_\theta(x) = g(\theta^T x)$ where $g(z) = \frac{1}{1+e^{-z}}$ thus $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$
and g() is called the "Sigmoid" or "logistic" function (two interchangeable names for the same thing).



Note that since y must be either 0 or 1, the i output of $h_\theta(x)$ should be interpreted as the probability that y is 1. Formally $h_\theta(x) = P(y = 1 \mid x; \theta)$ i.e. "the probability that y = 1, given x, parameterized by θ".

Decision boundary
Clamping the output of $h_\theta(x) = g(\theta^T x)$ hard to y, such that **y = 1 if h ≥ 0.5** and **y = 0 if h < 0.5** creates a *decision boundary*, such that drawn in a graph, all y = 0 should fall on one side of the boundary and all y = 1 should fall on the other. In terms of $\theta^T x$ (rather than $g(\theta^T x)$), **y = 1 if $\theta^T x \geq 0$**.



Note that the decision boundary is a property of the hypothesis function (provided its parameters), not of the training set (though the training set can be used to fit the parameters to achieve the correct boundary).

Non-linear decision boundaries
As for the polynomial regression seen earlier, adding additional higher-order parameters (such as $(x_1)^2$ and $(x_2)^2$) we can achieve non-linear and complex decision boundaries for logistic regression too.

Cost-function for logistic regression
The “squared error” cost function used for linear regression is not suitable for logistic regression, because the non-linear nature of g() will very likely produce a non-convex plot of the cost function, i.e. with many local minima.

A much better cost function for logistic regression is this

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost\left(h_\theta(x^{(i)}), y^{(i)}\right)$$

where **$Cost(h_\theta(x),y) = -\log(h_\theta(x))$ if $y = 1$** *but* **$-\log(1 - h_\theta(x))$ if $y = 0$**
This is a convex function that has the desirable properties that its cost is zero when y = 1 and $h_\theta(x) = 1$, but as $h_\theta(x) \rightarrow 0$ the cost $\rightarrow \infty$. This means that if $h_\theta(x)$ is wrong, this cost function will penalize the learning algorithm by a very large amount.

Because y is always either 0 or 1 for two-class logistic regression, we can write the Cost function without the conditional statements, like this:

$$Cost(h\theta(x),y) = -y\, log(h\theta(x)) - (1-y)\, log(1 - h\theta(x))$$

Note that one of the terms on either side of the central minus will always be zero, because y is either 1 or 0.

Thus in full (moving the minus out front):

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost\left(h_\theta(x^{(i)}), y^{(i)}\right) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\, log\left(h\theta\left(x^{(i)}\right)\right) + (1-y^{(i)})\, log\left(1 - h\theta\left(x^{(i)}\right)\right)\right]$$

Gradient descent for logistic regression
After solving the partial derivative with respect to the new cost function, the formula for logistic gradient descent looks just like the one for linear gradient descent (minus the ½ which was only used before to simplify the derivative), i.e.

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$ *(N.B. 1/m missing from lecture video in error)*

but *__remember__* that the $h_\theta(x^{(i)})$ has changed from $h_\theta(x) = \theta^T x$ to $h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$

Apply and iterate the gradient descent algorithm for logistic regression the same way as for linear regression. Applying feature scaling (also in the same way) may help gradient descent run faster.

The matrix version is XXXXXXXXX

Advanced optimization algorithms and concepts
Gradient is a fairly simple algorithm, but somewhat slow and for large problem sets more advanced and efficient algorithms can be used, such as Conjugate gradient, BFGS, and L-BFGS. These are much more complex, but also faster, and with no need to manually pick α. These "have a clever inner loop" called *line-search algorithm*. N.B. they can be used just fine without being fully understood; just use a library implementation (e.g. see fminunc()).

Multi-class classification
Classification of data into more-than-two groups (not be confused with multi-label classification). Examples include classification of emails into Work / Friends / Family / Hobby groups.

There is no genuine multi-class classification algorithm. Instead, for n classes we can split the problem into n separate "one-vs-all" (or one-vs-rest) binary logistic regression problems, and afterwards assign the class for which the binary logistic regressions return the highest probability.

VII. Regularization - the problem of overfitting

Underfitting vs. Overfitting
With too few features we risk underfitting, or "high bias" (e.g. fitting a straight line through a second-degree curve), but with too many we risk overfitting, or "high variance" (e.g. fitting a high-order function perfectly to the training data but fail to be generic enough to fit any new data).

Addressing overfitting
Either the number for features can be manually reduced, or an automatic "model selection algorithm" can be used, or we can try regularization.

Regularization
Means keeping all features, but reducing the magnitude of the parameters θ, thus reducing each features' contribution to the hypothesis function. Works well when we have a lot of features that each contribute a little to the prediction of y, but the weakness is that we don't know which features actually correlate with y, so we just have to treat all the same.

To regularize e.g. the "square error" cost function we add a regularization term, with a regularization parameter λ (i.e. lambda), like this

$$J(\theta) = \frac{1}{2m}\left[\left(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2\right) + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

Lambda regulates the trade-off between the goal of fitting the data well and the goal of keeping the parameters small to avoid overfitting.

Too big lambda will result in underfitting.

Linear gradient descent with regularization
Working out the partial derivative with respect to the new cost function we get (note that θ_0 must be handled separately so not to penalize it too):

```
Repeat {
    $\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$
    $\theta_j := \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$
}
```

Normal equation with regularization

$\theta = \left(X^T X + \lambda M\right)^{-1} X^T y$ where M is a (n+1 × n+1) identity matrix but with a zero at 0,0 and lambda is a scalar.

Logistic gradient descent with regularization
Adding the normalization term to the logistic cost function we get

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log\left(h\theta\left(x^{(i)}\right)\right) + (1 - y^{(i)}) \log\left(1 - h\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

and solving the partial derivative with respect to this, we get

```
Repeat {
    $\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$
    $\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}\right] + \frac{\lambda}{m}\theta_j$
}
```

Remember that although it looks similar to linear gradient descent, here $h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$

Week 4

Useful resources

List of 40+ Machine Learning APIs

Free books:

- The Elements of Statistical Learning: Data Mining, Inference, and Prediction (Second

Edition), by Trevor Hastie, Robert Tibshirani, Jerome Friedman. February 2009
- A First Encounter with Machine Learning, by Max Welling, Donald Bren. November 4, 2011
- Information Theory, Inference, and Learning Algorithms, by David J.C. MacKay. Version 7.2 (fourth printing) March 28, 2005
- Bayesian Reasoning and Machine Learning, by David Barber. DRAFT March 29, 2013
- Pattern Recognition and Machine Learning, by Christopher M. Bishop. 2006
- Introduction to Machine Learning (Second Edition). by Ethem Alpaydın. 2010
- Pattern Classification and Machine Learning, by Matthias Seeger. April 8, 2013

Teaching slides etc:
- Statistical Learning. By Federica Giummol
- Stanford CS 229, Machine Learning Course Materials