01 and 02: Introduction, Regression Analysis, and Gradient Descent

Next Index

Introduction to the course

- We will learn about
  - State of the art
  - How to do the implementation
- Applications of machine learning include
  - Search
  - Photo tagging
  - Spam filters
- The AI dream of building machines as intelligent as humans
  - Many people believe best way to do that is mimic how humans learn
- What the course covers
  - Learn about state of the art algorithms
  - But the algorithms and math alone are no good
  - Need to know how to get these to work in problems
- Why is ML so prevalent?
  - Grew out of AI
  - Build intelligent machines
    - You can program a machine how to do some simple thing
      - For the most part hard-wiring AI is too difficult
    - Best way to do it is to have some way for machines to learn things themselves
      - A mechanism for learning - if a machine can learn from input then it does the hard work for you

Examples

- Database mining
  - Machine learning has recently become so big party because of the huge amount of data being generated
  - Large datasets from growth of automation web
  - Sources of data include
    - Web data (click-stream or click through data)
      - Mine to understand users better
      - Huge segment of silicon valley
    - Medical records
      - Electronic records -> turn records in knowledges
    - Biological data
      - Gene sequences, ML algorithms give a better understanding of human genome
    - Engineering info
      - Data from sensors, log reports, photos etc
- Applications that we cannot program by hand
  - Autonomous helicopter
  - Handwriting recognition
    - This is very inexpensive because when you write an envelope, algorithms can automatically route envelopes through the post
  - Natural language processing (NLP)
    - AI pertaining to language
  - Computer vision
    - AI pertaining vision
- Self customizing programs
  - Netflix
  - Amazon
  - iTunes genius
  - Take users info
    - Learn based on your behavior
- Understand human learning and the brain
  - If we can build systems that mimic (or try to mimic) how the brain works, this may push our own understanding of the associated neurobiology

What is machine learning?

- Here we...
  - Define what it is

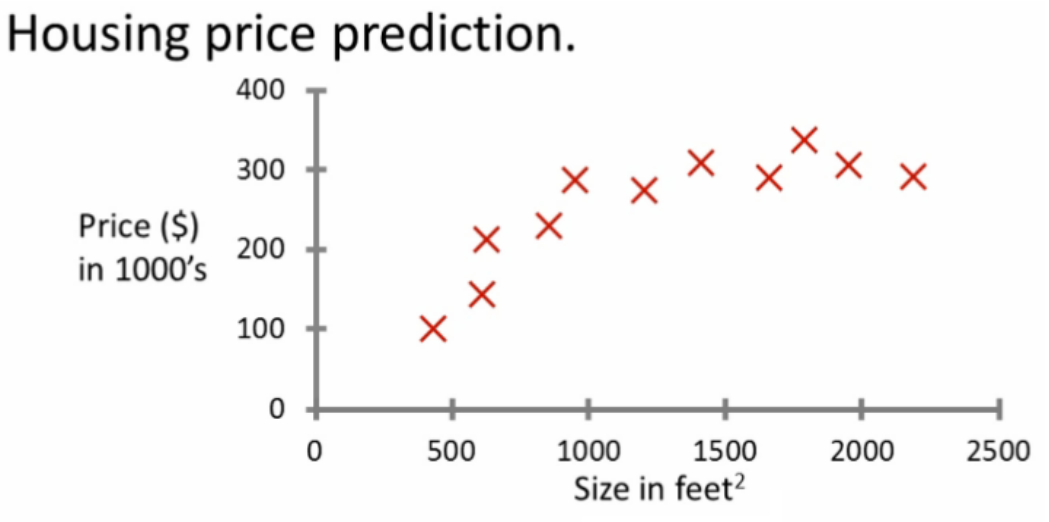
    - When to use it
- Not a well defined definition
    - Couple of examples of how people have tried to define it
- Arthur Samuel (1959)
    - ***Machine learning:*** **"Field of study that gives computers the ability to learn without being explicitly programmed"**
        - Samuels wrote a checkers playing program
            - Had the program play 10000 games against itself
            - Work out which board positions were good and bad depending on wins/losses

- Tom Michel (1999)
    - ***Well posed learning problem:*** **"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."**
        - The checkers example,
            - E = 10000s games
            - T is playing checkers
            - P if you win or not

- Several types of learning algorithms
    - **Supervised learning**
        - Teach the computer how to do something, then let it use it;s new found knowledge to do it
    - **Unsupervised learning**
        - Let the computer learn how to do something, and use this to determine structure and patterns in data
    - Reinforcement learning
    - Recommender systems
- This course
    - Look at practical advice for applying learning algorithms
    - Learning a set of tools and **how** to apply them

Supervised learning - introduction

- Probably the most common problem type in machine learning
- Starting with an example
    - How do we predict housing prices
        - Collect data regarding housing prices and how they relate to size in feet

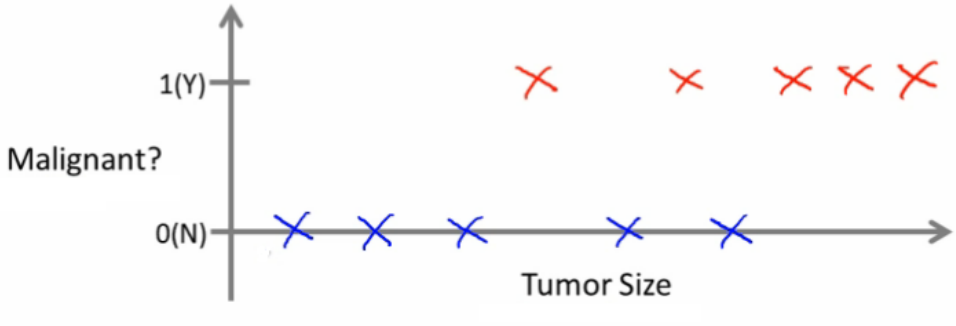


- **Example problem:** "Given this data, a friend has a house 750 square feet - how much can they be expected to get?"

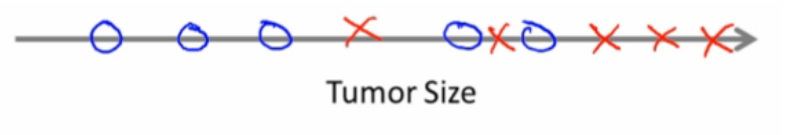
- What approaches can we use to solve this?
    - Straight line through data
        - Maybe $150 000
    - Second order polynomial
        - Maybe $200 000
    - One thing we discuss later - how to chose straight or curved line?
    - Each of these approaches represent a way of doing supervised learning
- *What does this mean?*
    - We gave the algorithm a data set where a "right answer" was provided
    - So we know actual prices for houses

        - The idea is we can learn what makes the price a certain value from the **training data**
        - The algorithm should then produce more right answers based on new training data where we don't know the price already
            - i.e. predict the price
- We also call this a **regression problem**
    - Predict continuous valued output (price)
    - No real discrete delineation

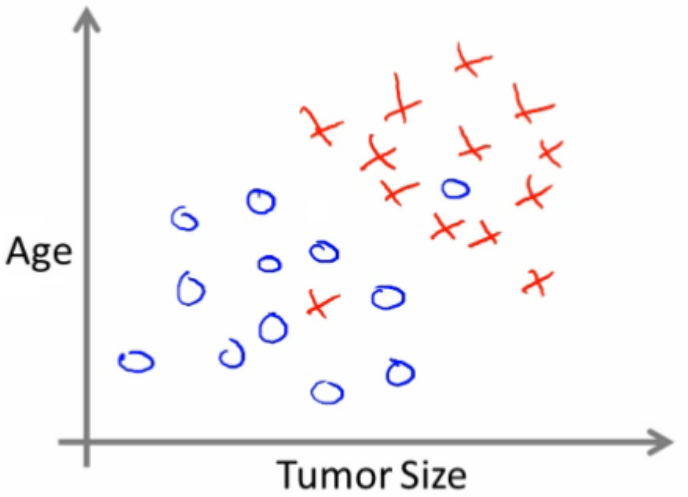
- Another example
    - Can we definer breast cancer as malignant or benign based on tumour size



- Looking at data
    - Five of each
    - Can you estimate prognosis based on tumor size?
    - This is an example of a **classification problem**
        - Classify data into one of two discrete classes - no in between, either malignant or not
        - In classification problems, can have a discrete number of possible values for the output
            - e.g. maybe have four values
                - 0 - benign
                - 1 - type 1
                - 2 - type 2
                - 3 - type 4
- In classification problems we can plot data in a different way



- Use only one attribute (size)
    - In other problems may have multiple attributes
    - We may also, for example, know age and tumor size



- Based on that data, you can try and define separate classes by
    - Drawing a straight line between the two groups
    - Using a more complex function to define the two groups (which we'll discuss later)
    - Then, when you have an individual with a specific tumor size and who is a specific age, you can hopefully use that

information to place them into one of your classes
- You might have many features to consider
  - Clump thickness
  - Uniformity of cell size
  - Uniformity of cell shape
- The most exciting algorithms can deal with an infinite number of features
  - How do you deal with an infinite number of features?
  - Neat mathematical trick in support vector machine (which we discuss later)
    - If you have an infinitely long list - we can develop and algorithm to deal with that

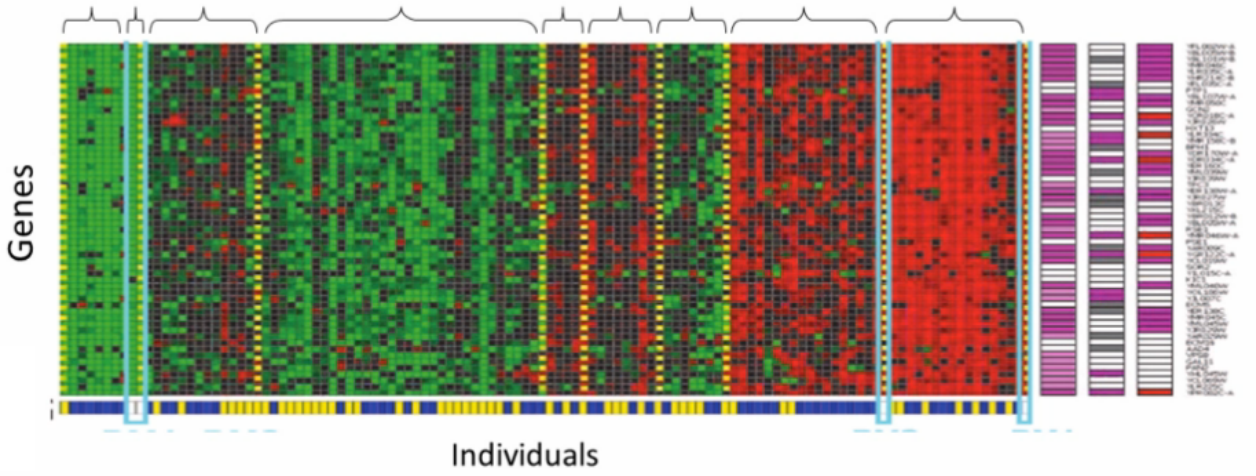
- ***Summary***
  - Supervised learning lets you get the "right" data a
  - Regression problem
  - Classification problem

Unsupervised learning - introduction

- Second major problem type
- In unsupervised learning, we get unlabeled data
  - Just told - here is a data set, can you structure it
- One way of doing this would be to cluster data into to groups
  - This is a **clustering algorithm**

Clustering algorithm

- Example of clustering algorithm
  - Google news
    - Groups news stories into cohesive groups
  - Used in any other problems as well
    - Genomics
    - Microarray data
      - Have a group of individuals
      - On each measure expression of a gene
      - Run algorithm to cluster individuals into types of people

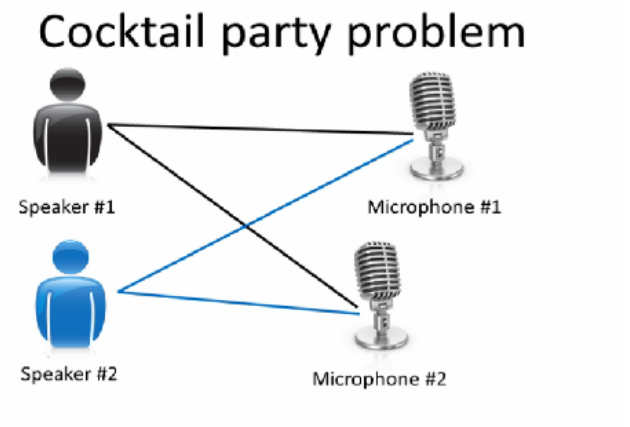


    - Organize computer clusters
      - Identify potential weak spots or distribute workload effectively
    - Social network analysis
      - Customer data
    - Astronomical data analysis
      - Algorithms give amazing results

- Basically
  - Can you automatically generate structure
  - Because we don't give it the answer, it's unsupervised learning

Cocktail party algorithm

- Cocktail party problem
  - Lots of overlapping voices - hard to hear what everyone is saying
    - Two people talking
    - Microphones at different distances from speakers



- Record sightly different versions of the conversation depending on where your microphone is
  - But overlapping none the less
- Have recordings of the conversation from each microphone
  - Give them to a cocktail party algorithm
  - Algorithm processes audio recordings
    - Determines there are two audio sources
    - Separates out the two sources
- Is this a very complicated problem
  - Algorithm can be done with one line of code!
  - **[W,s,v] = svd((repmat(sum(x.*x,1), size(x,1),1).*x)*x');**
    - Not easy to identify
    - But, programs can be short!
    - Using octave (or MATLAB) for examples
      - Often prototype algorithms in octave/MATLAB to test as it's very fast
      - Only when you show it works migrate it to C++
      - Gives a much faster agile development
- Understanding this algorithm
  - **svd** - linear algebra routine which is built into octave
    - In C++ this would be very complicated!
  - Shown that using MATLAB to prototype is a really good way to do this

Linear Regression

- Housing price data example used earlier
  - Supervised learning regression problem
- What do we start with?
  - Training set (this is your data set)
  - Notation (*used throughout the course*)
    - m = number of **training examples**
    - x's = input variables / features
    - y's = output variable "target" variables
      - (x,y) - single training example
      - (x^i, y^j) - specific example (i^{th} training example)
        - i is an index to training set

Size in feet² (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
...	...

m = 47

- With our training set defined - how do we used it?
  - Take training set

    - Pass into a learning algorithm
    - Algorithm outputs a function (denoted *h*) (h = **hypothesis**)
        - This function takes an input (e.g. size of new house)
        - Tries to output the estimated value of Y
- How do we represent hypothesis *h* ?
    - Going to present h as;
        - $h_\theta(x) = \theta_0 + \theta_1 x$
            - h(x) (shorthand)

$$h_\theta(x) = \theta_0 + \theta_1 x$$

- What does this mean?
    - Means Y is a linear function of x!
    - θ_i are **parameters**
        - θ_0 is zero condition
        - θ_1 is gradient
- This kind of function is a linear regression with one variable
    - Also called **univariate linear regression**

- So in summary
    - A hypothesis takes in some variable
    - Uses parameters determined by a learning system
    - Outputs a prediction based on that input

Linear regression - implementation (cost function)

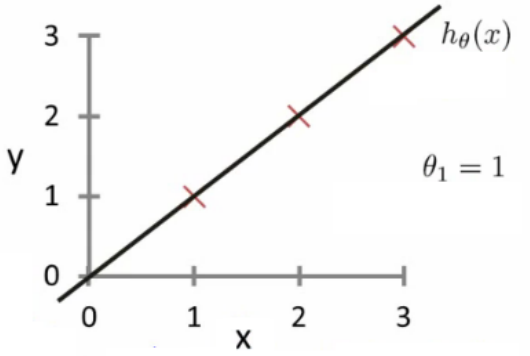
- A cost function lets us figure out how to fit the best straight line to our data
- Choosing values for θ_i (parameters)
    - Different values give you different functions
    - If θ_0 is 1.5 and θ_1 is 0 then we get straight line parallel with X along 1.5 @ y
    - If θ_1 is > 0 then we get a positive slope
- Based on our training set we want to generate parameters which make the straight line
    - Chosen these parameters so $h_\theta(x)$ is close to y for our training examples
        - Basically, uses xs in training set with $h_\theta(x)$ to give output which is as close to the actual y value as possible
        - Think of $h_\theta(x)$ as a "y imitator" - it tries to convert the x into y, and considering we already have y we can evaluate how well $h_\theta(x)$ does this
- To formalize this;
    - We want to want to solve a **minimization problem**
    - Minimize $(h_\theta(x) - y)^2$
        - i.e. minimize the difference between h(x) and y for each/any/every example
    - Sum this over the training set

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

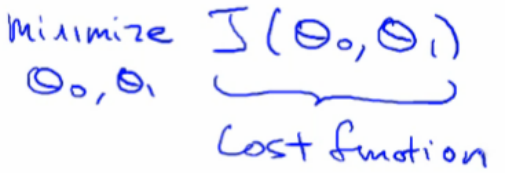
- Minimize squared different between predicted house price and actual house price
    - 1/2m
        - 1/m - means we determine the average
        - 1/2m the 2 makes the math a bit easier, and doesn't change the constants we determine at all (i.e. half the smallest value is still the smallest value!)
    - Minimizing θ_0/θ_1 means we get the values of θ_0 and θ_1 which find on average the minimal deviation of x from y when we use those parameters in our hypothesis function
- More cleanly, this is a cost function

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

- And we want to minimize this cost function
  - Our cost function is (because of the summartion term) inherently looking at ALL the data in the training set at any time
- **So to recap**
  - **Hypothesis** - is like your prediction machine, throw in an *x* value, get a putative *y* value



  - **Cost** - is a way to, using your training data, determine values for your θ values which make the hypothesis as accurate as possible



    - This cost function is also called the squared error cost function
      - This cost function is reasonable choice for most regression functions
      - Probably most commonly used function
  - In case $J(\theta_0,\theta_1)$ is a bit abstract, going into what it does, why it works and how we use it in the coming sections

Cost function - a deeper look

- Lets consider some intuition about the cost function and why we want to use it
  - The cost function determines parameters
  - The value associated with the parameters determines how your hypothesis behaves, with different values generate different
- Simplified hypothesis
  - Assumes $\theta_0 = 0$

$$h_\theta(x) = \theta_1 x$$

$\theta_0 = 0$

θ_1

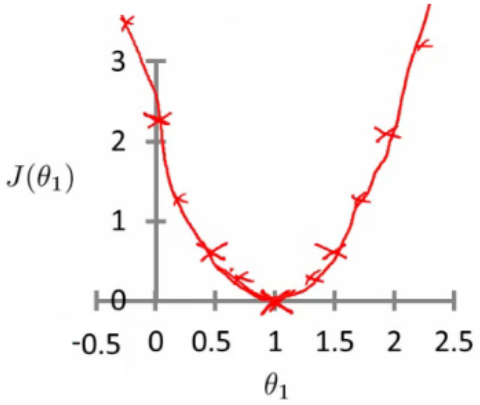
$$J(\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

$$\underset{\theta_1}{\text{minimize}}\ J(\theta_1)$$

- Cost function and goal here are very similar to when we have θ_0, but with a simpler parameter
  - Simplified hypothesis makes visualizing cost function J() a bit easier

- So hypothesis pass through 0,0
- Two key functins we want to understand
  - $h_\theta(x)$
    - Hypothesis is a function of x - function of what the size of the house is

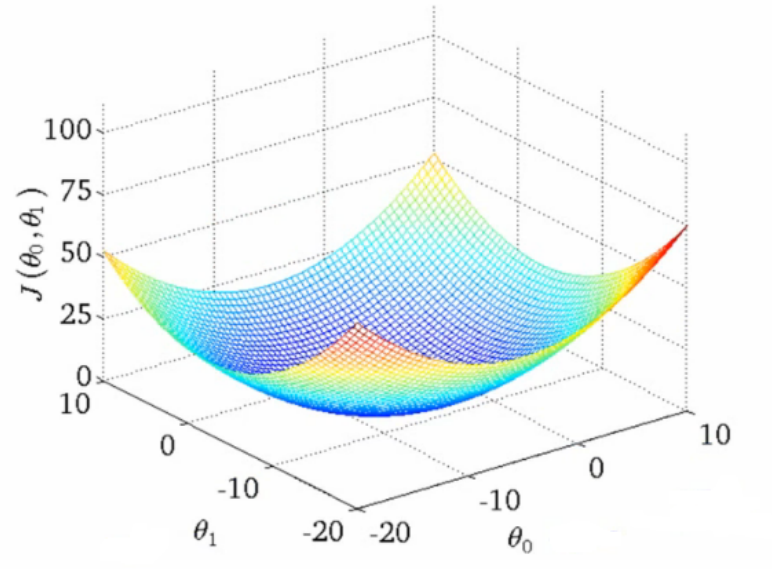
- $J(\theta_1)$
  - Is a function of the parameter of θ_1
- So for example
  - $\theta_1 = 1$
  - $J(\theta_1) = 0$
- Plot
  - θ_1 vs $J(\theta_1)$
  - Data
    - 1)
      - $\theta_1 = 1$
      - $J(\theta_1) = 0$
    - 2)
      - $\theta_1 = 0.5$
      - $J(\theta_1) = \sim 0.58$
    - 3)
      - $\theta_1 = 0$
      - $J(\theta_1) = \sim 2.3$
- If we compute a range of values plot
  - $J(\theta_1)$ vs θ_1 we get a polynomial (looks like a quadratic)



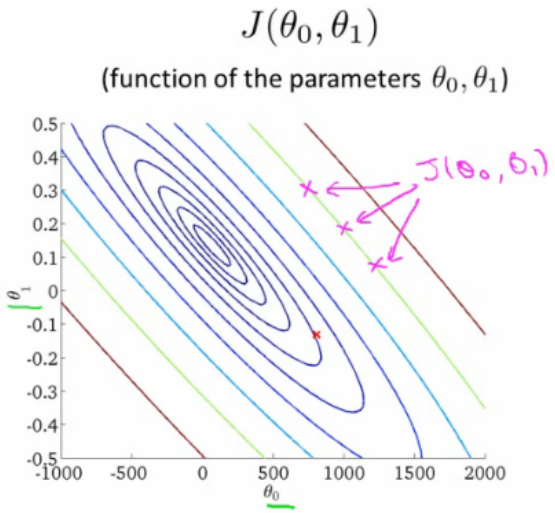
- The optimization objective for the learning algorithm is find the value of θ_1 which minimizes $J(\theta_1)$
  - So, here $\theta_1 = 1$ is the best value for θ_1

A deeper insight into the cost function - simplified cost function

- Assume you're familiar with contour plots or contour figures
  - Using same cost function, hypothesis and goal as previously
  - It's OK to skip parts of this section if you don't understand cotour plots
- Using our original complex hyothesis with two pariables,
  - So cost function is
    - $J(\theta_0, \theta_1)$
- Example,
  - Say
    - $\theta_0 = 50$
    - $\theta_1 = 0.06$
  - Previously we plotted our cost function by plotting
    - θ_1 vs $J(\theta_1)$
  - Now we have two parameters
    - Plot becomes a bit more complicated
    - Generates a 3D surface plot where axis are
      - $X = \theta_1$
      - $Z = \theta_0$
      - $Y = J(\theta_0,\theta_1)$



- We can see that the height (y) indicates the value of the cost function, so find where y is at a minimum
- Instead of a surface plot we can use a **contour figures/plots**
  - Set of ellipses in different colors
  - Each colour is the same value of $J(\theta_0, \theta_1)$, but obviously plot to different locations because θ_1 and θ_0 will vary
  - Imagine a bowl shape function coming out of the screen so the middle is the concentric circles



- Each point (like the red one above) represents a pair of parameter values for Θ0 and Θ1
  - Our example here put the values at
    - θ_0 = ~800
    - θ_1 = ~-0.15
  - Not a good fit
    - i.e. these parameters give a value on our contour plot far from the center
  - If we have
    - θ_0 = ~360
    - θ_1 = 0
    - This gives a better hypothesis, but still not great - not in the center of the countour plot
  - Finally we find the minimum, which gives the best hypothesis
- Doing this by eye/hand is a pain in the ass
  - What we really want is an efficient algorithm fro finding the minimum for θ_0 and θ_1

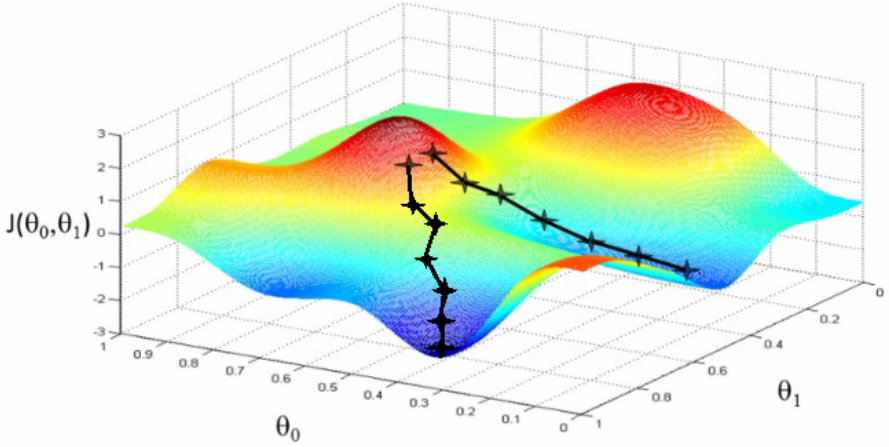
Gradient descent algorithm

- Minimize cost function J

- Gradient descent
  - Used all over machine learning for minimization
- Start by looking at a general J() function
- Problem
  - We have $J(\theta_0, \theta_1)$
  - We want to get **min $J(\theta_0, \theta_1)$**
- Gradient descent applies to more general functions
  - $J(\theta_0, \theta_1, \theta_2 .... \theta_n)$
  - min $J(\theta_0, \theta_1, \theta_2 .... \theta_n)$

How does it work?

- Start with initial guesses
  - Start at 0,0 (or any other value)
  - Keeping changing θ_0 and θ_1 a little bit to try and reduce $J(\theta_0,\theta_1)$
- Each time you change the parameters, you select the gradient which reduces $J(\theta_0,\theta_1)$ the most possible
- Repeat
- Do so until you converge to a local minimum
- Has an interesting property
  - Where you start can determine which minimum you end up



  - Here we can see one initialization point led to one local minimum
  - The other led to a different one

A more formal definition

- Do the following until covergence

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(for } j = 0 \text{ and } j = 1\text{)}$$

- What does this all mean?
  - Update θ_j by setting it to (θ_j - α) times the partial derivative of the cost function with respect to θ_j
- Notation
  - :=
    - Denotes assignment
    - NB a = b is a *truth assertion*
  - α (alpha)
    - Is a number called the **learning rate**
    - Controls how big a step you take
      - If α is big have an aggressive gradient descent
      - If α is small take tiny steps

- Derivative term

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

    - Not going to talk about it now, derive it later
- There is a subtly about how this gradient descent algorithm is implemented
    - Do this for θ_0 and θ_1
    - For j = 0 and j = 1 means we **simultaneously** update both
    - How do we do this?
        - Compute the right hand side for both θ_0 and θ_1
            - So we need a temp value
        - Then, update θ_0 and θ_1 at the same time
        - We show this graphically below

$$\begin{aligned}
&\text{temp0} := \theta_0 - \alpha \tfrac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\
&\text{temp1} := \theta_1 - \alpha \tfrac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\
&\theta_0 := \text{temp0} \\
&\theta_1 := \text{temp1}
\end{aligned}$$

- If you implement the non-simultaneous update it's not gradient descent, and will behave weirdly
    - But it might look sort of right - so it's important to remember this!

Understanding the algorithm

- To understand gradient descent, we'll return to a simpler function where we minimize one parameter to help explain the algorithm in more detail
    - min θ_1 $J(\theta_1)$ where θ_1 is a real number
- Two key terms in the algorithm
    - Alpha
    - Derivative term
- Notation nuances
    - Partial derivative vs. derivative
        - Use partial derivative when we have multiple variables but only derive with respect to one
        - Use derivative when we are deriving with respect to all the variables
- Derivative term

$$\frac{\partial}{\partial \theta_j} J(\theta_1)$$

    - Derivative says
        - Lets take the tangent at the point and look at the slope of the line
        - So moving towards the mimum (down) will greate a negative derivative, alpha is always positive, so will update $j(\theta_1)$ to a smaller value
        - Similarly, if we're moving up a slope we make $j(\theta_1)$ a bigger numbers
- Alpha term (α)
    - What happens if alpha is too small or too large
    - Too small
        - Take baby steps
        - Takes too long
    - Too large
        - Can overshoot the minimum and fail to converge
- When you get to a local minimum
    - Gradient of tangent/derivative is 0
    - So derivative term = 0
    - alpha * 0 = 0
    - So $\theta_1 = \theta_1 - 0$
    - So θ_1 remains the same

- As you approach the global minimum the derivative term gets smaller, so your update gets smaller, even with alpha is fixed
    - Means as the algorithm runs you take smaller steps as you approach the minimum
    - So no need to change alpha over time

Linear regression with gradient descent

- Apply gradient descent to minimize the squared error cost function $J(\theta_0, \theta_1)$
- Now we have a partial derivative

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_j} \cdot \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

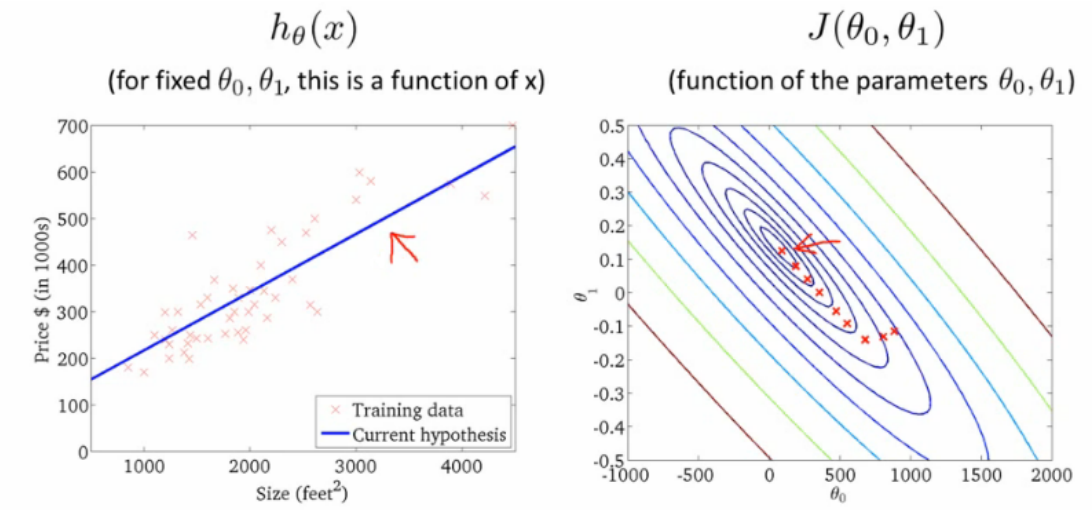
$$= \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$

- So here we're just expanding out the first expression
  - $J(\theta_0, \theta_1) = 1/2m....$
  - $h_\theta(x) = \theta_0 + \theta_1 * x$
- So we need to determine the derivative for each parameter - i.e.
  - When j = 0
  - When j = 1
- Figure out what this partial derivative is for the θ_0 and θ_1 case
  - When we derive this expression in terms of j = 0 and j = 1 we get the following

$$j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

$$j = 1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

- To check this you need to know multivariate calculus
  - So we can plug these values back into the gradient descent algorithm
- How does it work
  - Risk of meeting different local optimum
  - The linear regression cost function is always a **convex function** - always has a single minimum
    - Bowl shaped
    - One global optima
      - So gradient descent will always converge to global optima
  - In action
    - Initialize values to
      - $\theta_0 = 900$
      - $\theta_1 = -0.1$



- End up at a global minimum
- This is actually **Batch Gradient Descent**
  - Refers to the fact that over each step you look at all the training data
    - Each step compute over m training examples
  - Sometimes non-batch versions exist, which look at small data subsets

        - We'll look at other forms of gradient descent (to use when m is too large) later in the course
- There exists a numerical solution for finding a solution for a minimum function
    - **Normal equations** method
    - Gradient descent scales better to large data sets though
    - Used in lots of contexts and machine learning

What's next - important extensions
Two extension to the algorithm

- **1) Normal equation for numeric solution**
    - To solve the minimization problem we can solve it [min $J(\theta_0, \theta_1)$] exactly using a numeric method which avoids the iterative approach used by gradient descent
    - Normal equations method
    - Has advantages and disadvantages
        - Advantage
            - No longer an alpha term
            - Can be much faster for some problems
        - Disadvantage
            - Much more complicated
    - We discuss the normal equation in the **linear regression with multiple features** section
- **2) We can learn with a larger number of features**
    - So may have other parameters which contribute towards a prize
        - e.g. with houses
            - Size
            - Age
            - Number bedrooms
            - Number floors
        - x1, x2, x3, x4
    - With multiple features becomes hard to plot
        - Can't really plot in more than 3 dimensions
        - Notation becomes more complicated too
            - Best way to get around with this is the notation of linear algebra
            - Gives notation and set of things you can do with matrices and vectors
            - e.g. Matrix

$$X = \begin{bmatrix} 2104 & 5 & 1 & 45 \\ 1416 & 3 & 2 & 40 \\ 1534 & 3 & 2 & 30 \\ 852 & 2 & 1 & 36 \end{bmatrix} \qquad y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 172 \end{bmatrix}$$

- We see here this matrix shows us
    - Size
    - Number of bedrooms
    - Number floors
    - Age of home
- All in one variable
    - Block of numbers, take all data organized into one big block
- Vector
    - Shown as *y*
    - Shows us the prices
- Need linear algebra for more complex linear regression modles
- Linear algebra is good for making computationally efficient models (as seen later too)
    - Provide a good way to work with large sets of data sets
    - Typically vectorization of a problem is a common optimization technique

03: Linear Algebra - Review

Previous Next Index

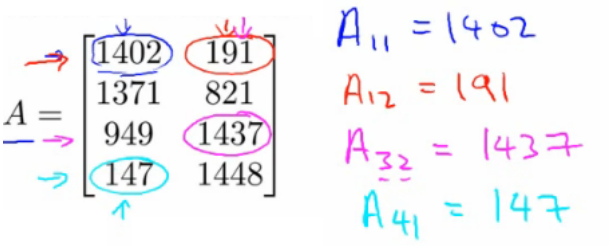
Matrices - overview

- Rectangular array of numbers written between square brackets
  - 2D array
  - Named as capital letters (A,B,X,Y)
- Dimension of a matrix are [Rows x Columns]
  - Start at top left
  - To bottom left
  - To bottom right
  - $R^{[r \times c]}$ means a matrix which has r rows and c columns

$$A = \begin{bmatrix} 1402 & 191 \\ 1371 & 821 \\ 949 & 1437 \\ 147 & 1448 \end{bmatrix}$$

    - Is a [4 x 2] matrix

- Matrix elements
  - $A_{(i,j)}$ = entry in i^{th} row and jth column



- Provides a way to organize, index and access a lot of data

Vectors - overview

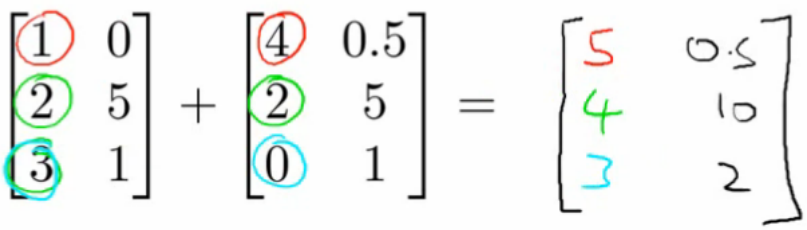
- Is an n by 1 matrix
  - Usually referred to as a lower case letter
  - n rows
  - 1 column
  - e.g.

$$y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

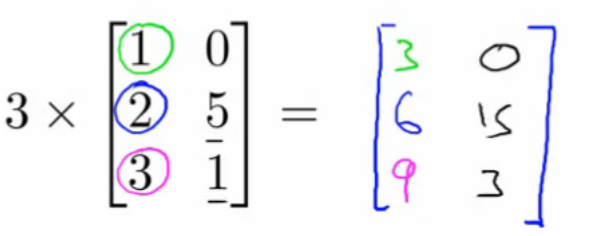
- Is a 4 dimensional vector
  - Refer to this as a vector R4
- Vector elements
  - v_i = i^{th} element of the vector
  - Vectors can be 0-indexed (C++) or 1-indexed (MATLAB)
  - In math 1-indexed is most common
    - But in machine learning 0-index is useful
  - Normally assume using 1-index vectors, but be aware sometimes these will (explicitly) be 0 index ones

Matrix manipulation

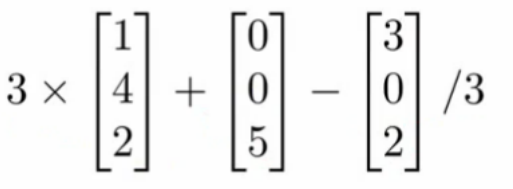
- ***Addition***
  - Add up elements one at a time
  - Can only add matrices of the *same dimensions*
    - Creates a new matrix of the same dimensions of the ones added



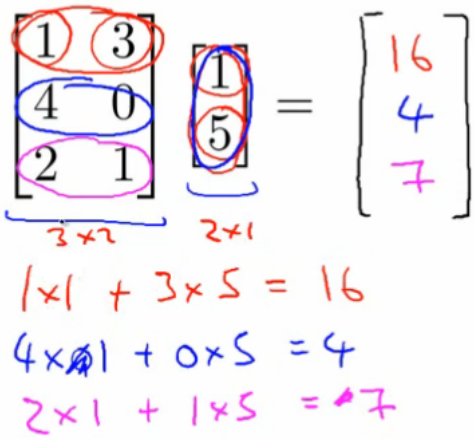
- ***Multiplication by scalar***
  - Scalar = real number
  - Multiply each element by the scalar
  - Generates a matrix of the same size as the original matrix



- ***Division by a scalar***
  - Same as multiplying a matrix by 1/4
  - Each element is divided by the scalar
- ***Combination of operands***
  - Evaluate multiplications first

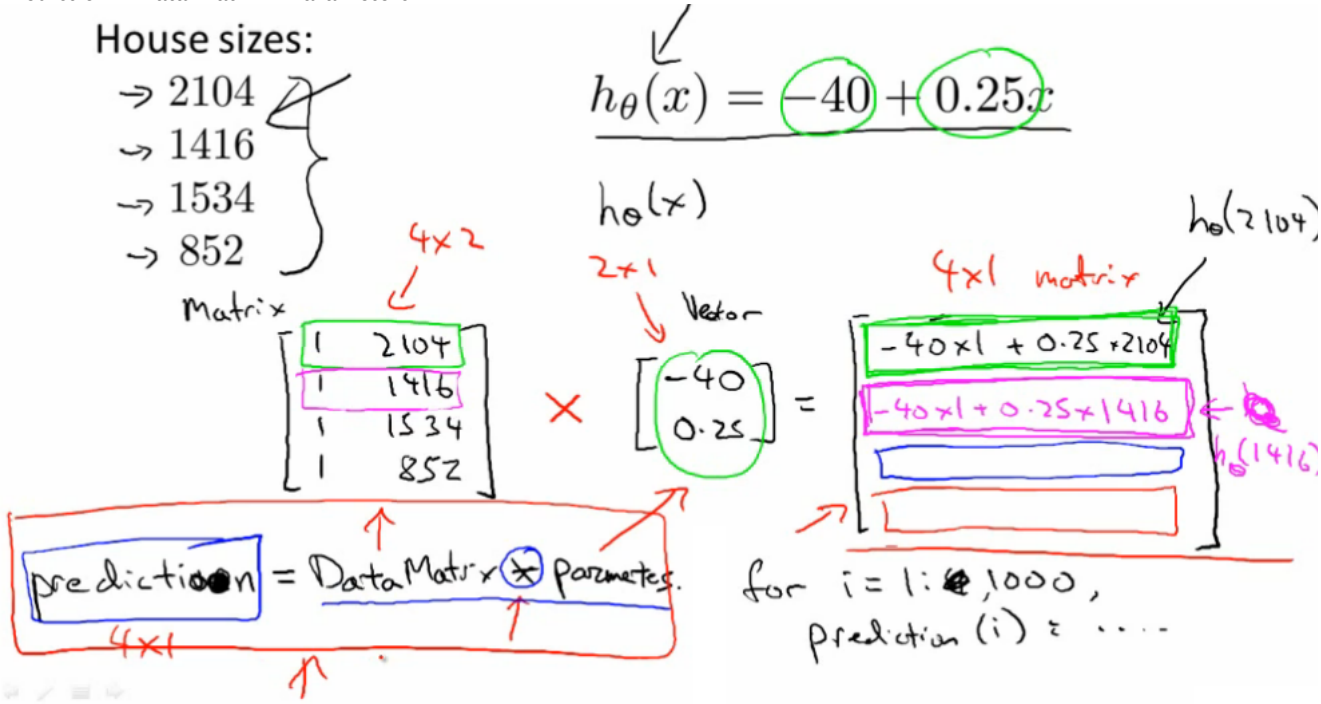


- **Matrix by vector multiplication**
  - [3 x 2] matrix * [2 x 1] vector
    - New matrix is [3 x 1]
      - More generally if [a x b] * [b x c]
        - Then new matrix is [a x c]
    - How do you do it?
      - Take the two vector numbers and multiply them with the first row of the matrix
        - Then add results together - this number is the first number in the new vector
      - The multiply second row by vector and add the results together
      - Then multiply final row by vector and add them together



- Detailed explanation
  - A * x = y
    - A is m x n matrix

        - x is n x 1 matrix
        - n must match between vector and matrix
            - i.e. inner dimensions must match
        - Result is an m-dimensional vector
    - To get y_i - multiply A's i^{th} row with all the elements of vector x and add them up
- Neat trick
    - Say we have a data set with four values
    - Say we also have a hypothesis $h_\theta(x) = -40 + 0.25x$
        - Create your data as a matrix which can be multiplied by a vector
        - Have the parameters in a vector which your matrix can be multiplied by
    - Means we can do
        - Prediction = Data Matrix * Parameters



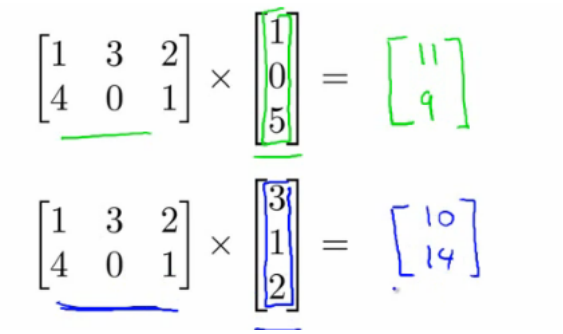
        - Here we add an extra column to the data with 1s - this means our θ_0 values can be calculated and expressed

- The diagram above shows how this works
    - This can be far more efficient computationally than lots of for loops
    - This is also easier and cleaner to code (assuming you have appropriate libraries to do matrix multiplication)

- ***Matrix-matrix multiplication***
    - General idea
        - Step through the second matrix one column at a time
        - Multiply each column vector from second matrix by the entire first matrix, each time generating a vector
        - The final product is these vectors combined (not added or summed, but literally just put together)
    - Details
        - A x B = C
            - A = [m x n]
            - B = [n x o]
            - C = [m x o]
                - With vector multiplications o = 1
        - Can only multiply matrix where columns in A match rows in B
    - Mechanism
        - Take column 1 of B, treat as a vector
        - Multiply A by that column - generates an [m x 1] vector
        - Repeat for each column in B
            - There are o columns in B, so we get o columns in C
    - Summary
        - *The i^{th} column of matrix C is obtained by multiplying A with the i^{th} column of B*
    - Start with an example
    - A x B

$$\begin{bmatrix}1 & 3 & 2\\4 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 3\\0 & 1\\5 & 2\end{bmatrix}$$

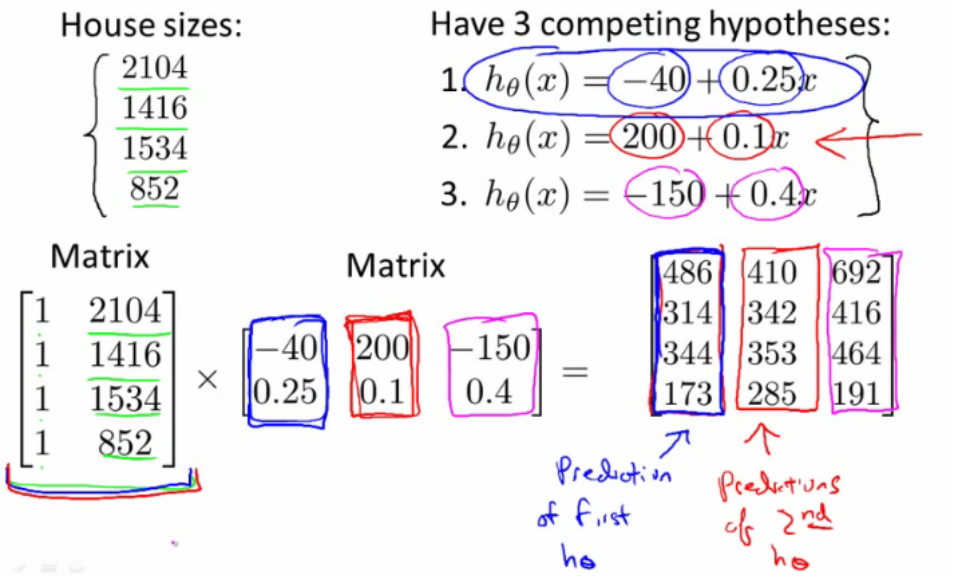
- Initially
  - Take matrix A and multiply by the first column vector from B
  - Take the matrix A and multiply by the second column vector from B



- 2 x 3 times 3 x 2 gives you a 2 x 2 matrix

Implementation/use

- House prices, but now we have three hypothesis and the same data set
- To apply all three hypothesis to all data we can do this efficiently using matrix-matrix multiplication
  - Have
    - Data matrix
    - Parameter matrix
  - Example
    - Four houses, where we want to predict the prize
    - Three competing hypotheses
    - Because our hypothesis are one variable, to make the matrices match up we make our data (houses sizes) vector into a 4x2 matrix by adding an extra column of 1s



- What does this mean
  - Can quickly apply three hypotheses at once, making 12 predictions
  - Lots of good linear algebra libraries to do this kind of thing very efficiently

Matrix multiplication properties

- Can pack a lot into one operation
  - However, should be careful of how you use those operations

- Some interesting properties
- **Commutativity**
  - When working with raw numbers/scalars multiplication is commutative
    - 3 * 5 == 5 * 3
  - This is not true for matrix
    - A x B != B x A
    - **Matrix multiplication is not commutative**
- **Associativity**
  - 3 x 5 x 2 == 3 x 10 = 15 x 2
    - Associative property
  - **Matrix multiplications is associative**
    - A x (B x C) == (A x B) x C
- **Identity matrix**
  - 1 is the identity for any scalar
    - i.e. 1 x z = z
      - for any real number
  - In matrices we have an identity matrix called *I*
    - Sometimes called $I_{\{n \times n\}}$

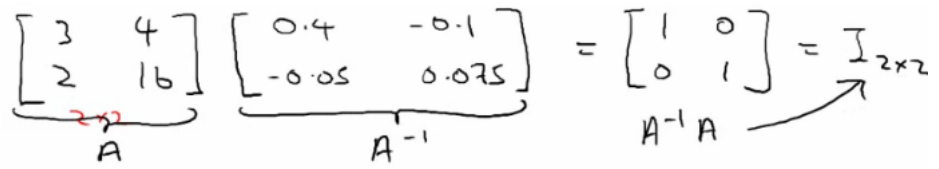
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}_{2 \times 2} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}_{3 \times 3} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{4 \times 4}$$

- See some identity matrices above
  - Different identity matrix for each set of dimensions
  - Has
    - 1s along the diagonals
    - 0s everywhere else
  - 1x1 matrix is just "1"

- Has the property that any matrix A which can be multiplied by an identity matrix gives you matrix A back
  - So if A is [m x n] then
    - A * I
      - I = n x n
    - I * A
      - I = m x m
    - (To make inside dimensions match to allow multiplication)
- Identity matrix dimensions are implicit
- Remember that matrices are not commutative AB != BA
  - Except when B is the identity matrix
  - Then AB == BA

Inverse and transpose operations

- **Matrix inverse**
  - How does the concept of "the inverse" relate to real numbers?
    - 1 = "identity element" (as mentioned above)
      - Each number has an inverse
        - This is the number you multiply a number by to get the identify element
        - i.e. if you have x, x * 1/x = 1
    - e.g. given the number 3
      - $3 * 3^{-1} = 1$ (the identity number/matrix)
    - In the space of real numbers **not everything has an inverse**
      - e.g. 0 does not have an inverse
  - What is the inverse of a matrix
    - If A is an m x m matrix, then A inverse = A^{-1}
    - So $A*A^{-1} = I$
    - Only matrices which are m x m have inverses
      - Square matrices only!
  - Example
    - 2 x 2 matrix



    - How did you find the inverse
        - Turns out that you can sometimes do it by hand, although this is very hard
        - Numerical software for computing a matrices inverse
            - Lots of open source libraries
    - If A is all zeros then there is no inverse matrix
        - Some others don't, intuition should be matrices that don't have an inverse are a singular matrix or a degenerate matrix (i.e. when it's too close to 0)
        - So if all the values of a matrix reach zero, this can be described as reaching singularity

- **Matrix transpose**
    - Have matrix A (which is [n x m]) how do you change it to become [m x n] while keeping the same values
        - i.e. swap rows and columns!
    - How you do it;
        - Take first row of A - becomes 1st column of A^T
        - Second row of A - becomes 2nd column...
    - A is an m x n matrix
        - B is a transpose of A
        - Then B is an n x m matrix
        - $A_{(i,j)} = B_{(j,i)}$

$$\underline{A} = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 5 & 9 \end{bmatrix} \quad \underline{A^T} = \begin{bmatrix} 1 & 3 \\ 2 & 5 \\ 0 & 9 \end{bmatrix}$$

04: Linear Regression with Multiple Variables

Previous Next Index

Linear regression with multiple features

New version of linear regression with multiple features

- Multiple variables = multiple features
- In original version we had
  - X = house size, use this to predict
  - y = house price
- If in a new scheme we have more variables (such as number of bedrooms, number floors, age of the home)
  - x_1, x_2, x_3, x_4 are the four features
    - x_1 - size (feet squared)
    - x_2 - Number of bedrooms
    - x_3 - Number of floors
    - x_4 - Age of home (years)
  - y is the output variable (price)
- More notation
  - **n**
    - number of features (n = 4)
  - **m**
    - number of examples (i.e. number of rows in a table)
  - $\mathbf{x^i}$
    - vector of the input for an example (so a vector of the four parameters for the i^{th} input example)
    - i is an index into the training set
    - So
      - x is an n-dimensional feature vector
      - x^3 is, for example, the 3rd house, and contains the four features associated with that house
  - $\mathbf{x_j^i}$
    - The value of feature j in the ith training example
    - So
      - x_2^3 is, for example, the number of bedrooms in the third house
- Now we have multiple features
  - What is the form of our hypothesis?
  - Previously our hypothesis took the form;
    - $h_\theta(x) = \theta_0 + \theta_1 x$
      - Here we have two parameters (theta 1 and theta 2) determined by our cost function
      - One variable x
  - Now we have multiple features
    - $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$
  - For example
    - $h_\theta(x) = 80 + 0.1x_1 + 0.01x_2 + 3x_3 - 2x_4$
      - An example of a hypothesis which is trying to predict the price of a house
      - Parameters are still determined through a cost function
  - For convenience of notation, $x_0 = 1$
    - For every example i you have an additional 0th feature for each example
    - So now your **feature vector** is n + 1 dimensional feature vector indexed from 0
      - This is a column vector called x
      - Each example has a column vector associated with it
      - So let's say we have a new example called "X"
    - **Parameters** are also in a 0 indexed n+1 dimensional vector
      - This is also a column vector called θ
      - This vector is the same for each example

- Considering this, hypothesis can be written
  - $h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$
- If we do
  - $h_\theta(x) = \theta^T X$
    - θ^T is an [1 x n+1] matrix
    - In other words, because θ is a column vector, the transposition operation transforms it into a row vector
    - So before
      - θ was a matrix [n + 1 x 1]
    - Now
      - θ^T is a matrix [1 x n+1]
    - Which means the inner dimensions of θ^T and X match, so they can be multiplied together as
      - [1 x n+1] * [n+1 x 1]
      - = $h_\theta(x)$
      - So, in other words, the transpose of our parameter vector * an input example X gives you a predicted hypothesis which is [1 x 1] dimensions (i.e. a single value)
  - This $x_0 = 1$ lets us write this like this
- This is an example of multivariate linear regression

Gradient descent for multiple variables

- Fitting parameters for the hypothesis with gradient descent
  - Parameters are θ_0 to θ_n
  - Instead of thinking about this as n separate values, think about the parameters as a single vector (θ)
    - Where θ is n+1 dimensional
- Our cost function is

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

- Similarly, instead of thinking of J as a function of the n+1 numbers, J() is just a function of the parameter vector
  - J(θ)
- **Gradient descent**

Repeat {

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

} (simultaneously update for every $j = 0, \ldots, n$)

- Once again, this is
  - $\theta_j = \theta_j$ - learning rate (α) times the partial derivative of J(θ) with respect to $\theta_{J(...)}$
  - We do this through a **simultaneous update** of every θ_j value
- Implementing this algorithm
  - When n = 1

Repeat {

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

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

(simultaneously update θ_0, θ_1) }

- Above, we have slightly different update rules for θ_0 and θ_1
  - Actually they're the same, except the end has a previously undefined $x_0^{(i)}$ as 1, so wasn't shown
- We now have an almost identical rule for multivariate gradient descent

New algorithm $(n \geq 1)$:

Repeat { $\leftarrow \frac{\partial}{\partial\theta_j}J(\theta)$

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

(simultaneously update θ_j for $j = 0, \ldots, n$) }

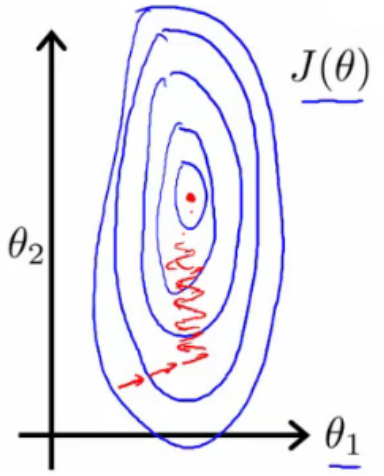
- What's going on here?
  - We're doing this for each j (0 until n) as a simultaneous update (like when n = 1)
  - So, we re-set θ_j to
    - θ_j minus the learning rate (α) times the partial derivative of of the θ vector with respect to θ_j
    - In non-calculus words, this means that we do
      - Learning rate
      - Times 1/m (makes the maths easier)
      - Times the sum of
        - The hypothesis taking in the variable vector, minus the actual value, times the j-th value in that variable vector for EACH example
  - It's important to remember that

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

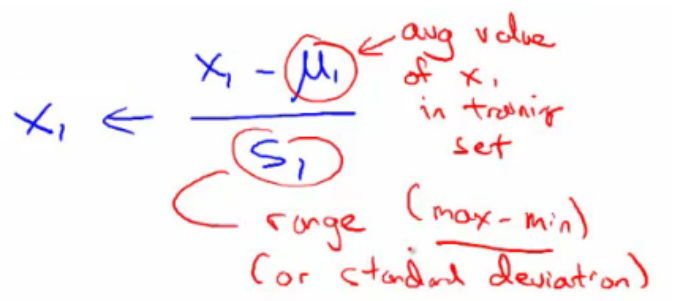
- These algorithm are highly similar

Gradient Decent in practice: 1 Feature Scaling

- Having covered the theory, we now move on to learn about some of the practical tricks
- Feature scaling
  - If you have a problem with multiple features
  - You should make sure those features have a similar scale
    - Means gradient descent will converge more quickly
  - e.g.
    - x1 = size (0 - 2000 feet)
    - x2 = number of bedrooms (1-5)
    - Means the contours generated if we plot θ_1 vs. θ_2 give a very tall and thin shape due to the huge range difference
  - Running gradient descent on this kind of cost function can take a long time to find the global minimum



- Pathological input to gradient descent
  - So we need to rescale this input so it's more effective
  - So, if you define each value from x1 and x2 by dividing by the max for each feature
  - Contours become more like circles (as scaled between 0 and 1)
- May want to get everything into -1 to +1 range (approximately)
  - Want to avoid large ranges, small ranges or very different ranges from one another
  - Rule a thumb regarding acceptable ranges
    - -3 to +3 is generally fine - any bigger bad
    - -1/3 to +1/3 is ok - any smaller bad
- Can do **mean normalization**
  - Take a feature x_i
    - Replace it by (x_i - mean)/max
    - So your values all have an average of about 0



- Instead of max can also use standard deviation

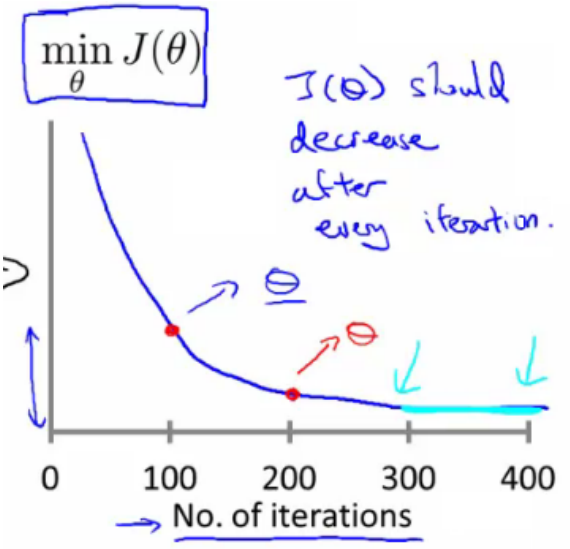
Learning Rate α

- Focus on the learning rate (α)
- Topics
  - Update rule
  - Debugging
  - How to chose α

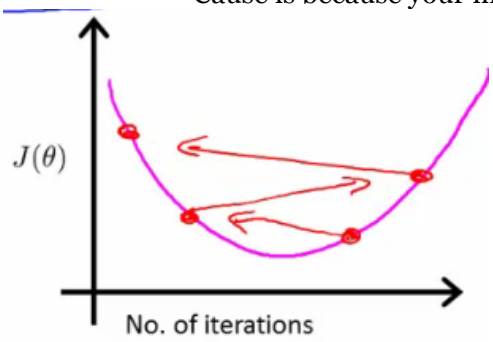
Make sure gradient descent is working

- Plot min J(θ) vs. no of iterations
  - (i.e. plotting J(θ) over the course of gradient descent
- If gradient descent is working then J(θ) should decrease after every iteration
- Can also show if you're not making huge gains after a certain number
  - Can apply heuristics to reduce number of iterations if need be
  - If, for example, after 1000 iterations you reduce the parameters by nearly nothing you could chose to only

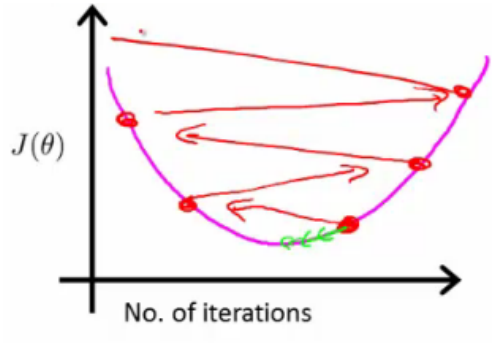
run 1000 iterations in the future
- Make sure you don't accidentally hard-code thresholds like this in and then forget about why they're their though!



- Number of iterations varies a lot
  - 30 iterations
  - 3000 iterations
  - 3000 000 iterations
  - Very hard to tel in advance how many iterations will be needed
  - Can often make a guess based a plot like this after the first 100 or so iterations
- Automatic convergence tests
  - Check if J(θ) changes by a small threshold or less
    - Choosing this threshold is hard
    - So often easier to check for a straight line
      - Why? - Because we're seeing the straightness in the context of the whole algorithm
      - Could you design an automatic checker which calculates a threshold based on the systems preceding progress?
- Checking its working
  - If you plot J(θ) vs iterations and see the value is increasing - means you probably need a smaller α
    - Cause is because your minimizing a function which looks like this



- But you overshoot, so reduce learning rate so you actually reach the minimum (green line)

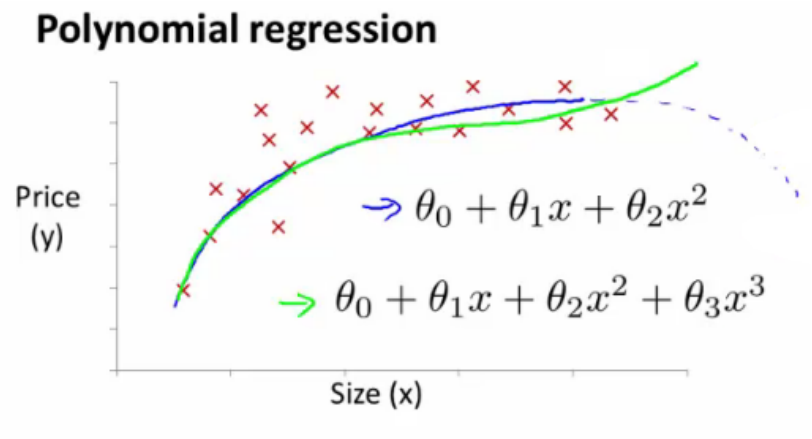


    - So, use a smaller α
- Another problem might be if J(θ) looks like a series of waves
    - Here again, you need a smaller α
- However
    - If α is small enough, J(θ) will decrease on every iteration
    - BUT, if α is too small then rate is too slow
        - A less steep incline is indicative of a slow convergence, because we're decreasing by less on each iteration than a steeper slope
- Typically
    - Try a range of alpha values
    - Plot J(θ) vs number of iterations for each version of alpha
    - Go for roughly threefold increases
        - 0.001, 0.003, 0.01, 0.03. 0.1, 0.3

Features and polynomial regression

- Choice of features and how you can get different learning algorithms by choosing appropriate features
- Polynomial regression for non-linear function

- Example
    - House price prediction
        - Two features
            - Frontage - width of the plot of land along road (x_1)
            - Depth - depth away from road (x_2)
    - You don't have to use just two features
        - **Can create new features**
    - Might decide that an important feature is the land area
        - So, create a new feature = frontage * depth (x_3)
        - $h(x) = \theta_0 + \theta_1 x_3$
            - Area is a better indicator
    - Often, by defining new features you may get a better model
- Polynomial regression
    - May fit the data better
    - $\theta_0 + \theta_1 x + \theta_2 x^2$ e.g. here we have a quadratic function
    - For housing data could use a quadratic function
        - But may not fit the data so well - inflection point means housing prices decrease when size gets really big
        - So instead must use a cubic function



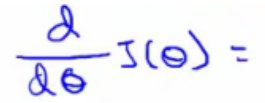
- How do we fit the model to this data
    - To map our old linear hypothesis and cost functions to these polynomial descriptions the easy thing to do is set
        - $x_1 = x$

        - $x_2 = x^2$
        - $x_3 = x^3$
    - By selecting the features like this and applying the linear regression algorithms you can do polynomial linear regression
    - Remember, feature scaling becomes even more important here
- Instead of a conventional polynomial you could do variable ^(1/something) - i.e. square root, cubed root etc
- Lots of features - later look at developing an algorithm to chose the best features

Normal equation

- For some linear regression problems the normal equation provides a better solution
- So far we've been using gradient descent
    - Iterative algorithm which takes steps to converse
- Normal equation solves θ analytically
    - Solve for the optimum value of theta
- Has some advantages and disadvantages

How does it work?

- Simplified cost function
    - $J(\theta) = a\theta^2 + b\theta + c$
        - θ is just a real number, not a vector
    - Cost function is a quadratic function
    - How do you minimize this?
        - Do
        - 
            - Take derivative of J(θ) with respect to θ
            - Set that derivative equal to 0
            - Allows you to solve for the value of θ which minimizes J(θ)
- In our more complex problems;
    - Here θ is an n+1 dimensional vector of real numbers
    - Cost function is a function of the vector value
        - How do we minimize this function
            - Take the partial derivative of J(θ) with respect θ_j and set to 0 for every j
            - Do that and solve for θ_0 to θ_n
            - This would give the values of θ which minimize J(θ)
    - If you work through the calculus and the solution, the derivation is pretty complex
        - Not going to go through here
        - Instead, what do you need to know to implement this process

Example of normal equation

Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178

- Here
    - m = 4
    - n = 4

- To implement the normal equation
  - Take examples
  - Add an extra column (x_0 feature)
  - Construct a matrix (X - **the design matrix**) which contains all the training data features in an [m x n+1] matrix
  - Do something similar for y
    - Construct a column vector y vector [m x 1] matrix
  - Using the following equation (X transpose * X) inverse times X transpose y

$$\theta = (X^T X)^{-1} X^T y$$

$$\left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2104 & 1416 & 1534 & 852 \\ 5 & 3 & 3 & 2 \\ 1 & 2 & 2 & 1 \\ 45 & 40 & 30 & 36 \end{bmatrix} \times \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \right)^{-1} \times \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2104 & 1416 & 1534 & 852 \\ 5 & 3 & 3 & 2 \\ 1 & 2 & 2 & 1 \\ 45 & 40 & 30 & 36 \end{bmatrix} \times \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

- If you compute this, you get the value of theta which minimize the cost function

General case

- Have m training examples and n features
  - The **design matrix** (X)
    - Each training example is a n+1 dimensional feature column vector
    - X is constructed by taking each training example, determining its transpose (i.e. column -> row) and using it for a row in the design A
    - This creates an [m x (n+1)] matrix

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1}$$

$$\begin{bmatrix} \\ \\ \\ \\ \end{bmatrix} \times = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}$$

(design matrix)

  - **Vector y**
    - Used by taking all the y values into a column vector

$$\theta = (X^T X)^{-1} X^T y$$

- What is this equation?!
  - $(X^T * X)^{-1}$
    - What is this --> the inverse of the matrix $(X^T * X)$
      - i.e. $A = X^T X$
      - $A^{-1} = (X^T X)^{-1}$
- In octave and MATLAB you could do;

```
pinv(X'*x)*x'*y
```

        - X' is the notation for X transpose
        - pinv is a function for the inverse of a matrix
- In a previous lecture discussed feature scaling
    - If you're using the normal equation then no need for feature scaling

When should you use gradient descent and when should you use feature scaling?

- ***Gradient descent***
    - Need to chose learning rate
    - Needs many iterations - could make it slower
    - Works well even when *n* is massive (millions)
        - Better suited to big data
        - What is a big *n* though
            - 100 or even a 1000 is still (relativity) small
            - If n is 10 000 then look at using gradient descent
- ***Normal equation***
    - No need to chose a learning rate
    - No need to iterate, check for convergence etc.
    - Normal equation needs to compute $(X^T X)^{-1}$
        - This is the inverse of an n x n matrix
        - With most implementations computing a matrix inverse grows by $O(n^3)$
            - So not great
    - Slow of *n* is large
        - Can be much slower

Normal equation and non-invertibility

- Advanced concept
    - Often asked about, but quite advanced, perhaps optional material
    - Phenomenon worth understanding, but not probably necessary
- When computing $(X^T X)^{-1} * X^T * y$)
    - What if $(X^T X)$ is non-invertible (singular/degenerate)
        - Only some matrices are invertible
        - This should be quite a rare problem
            - Octave can invert matrices using
                - pinv (pseudo inverse)
                    - This gets the right value even if $(X^T X)$ is non-invertible
                - inv (inverse)
    - What does it mean for $(X^T X)$ to be non-invertible
        - Normally two common causes
            - **Redundant features** in learning model
                - e.g.
                    - x_1 = size in feet
                    - x_2 = size in meters squared
            - **Too many features**
                - e.g. m <= n (m is much larger than n)
                    - m = 10
                    - n = 100
                - Trying to fit 101 parameters from 10 training examples
                - Sometimes work, but not always a good idea
                - Not enough data
                - Later look at *why* this may be too little data
                - To solve this we
                    - Delete features
                    - Use **regularization** (let's you use lots of features for a small training set)

- If you find ($X^T X$) to be non-invertible
  - Look at features --> are features linearly dependent?
    - So just delete one, will solve problem

05: Octave and MATLAB

Previous Next Index

The Octave Programming Language

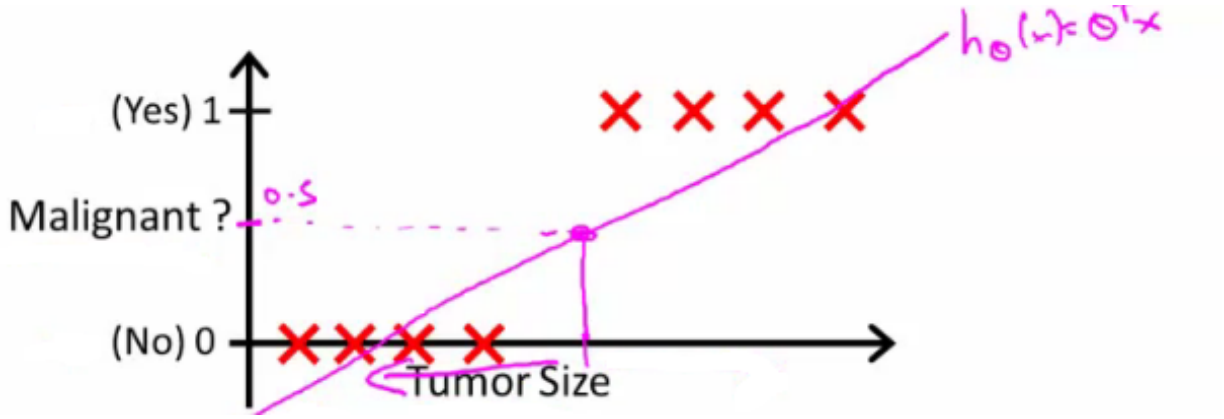
- These notes have not been done
  - Octave and MATLAB are very easy languages/frameworks to use so I recommend using a good online tutorial

06: Logistic Regression

Previous Next Index

Classification

- Where y is a discrete value
  - Develop the logistic regression algorithm to determine what class a new input should fall into
- Classification problems
  - Email -> spam/not spam?
  - Online transactions -> fraudulent?
  - Tumor -> Malignant/benign
- Variable in these problems is Y
  - Y is either 0 or 1
    - 0 = negative class (absence of something)
    - 1 = positive class (presence of something)
- Start with **binary class problems**
  - Later look at multiclass classification problem, although this is just an extension of binary classification
- How do we develop a classification algorithm?
  - Tumour size vs malignancy (0 or 1)
  - We *could* use linear regression
    - Then threshold the classifier output (i.e. anything over some value is yes, else no)
    - In our example below linear regression with thresholding seems to work

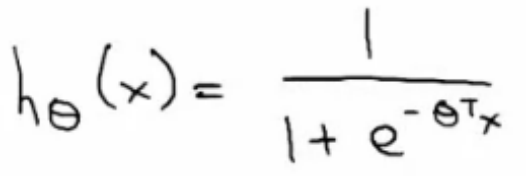


- We can see above this does a reasonable job of stratifying the data points into one of two classes
  - But what if we had a single Yes with a very small tumour
  - This would lead to classifying all the existing yeses as nos
- Another issues with linear regression
  - We know Y is 0 or 1
  - Hypothesis can give values large than 1 or less than 0

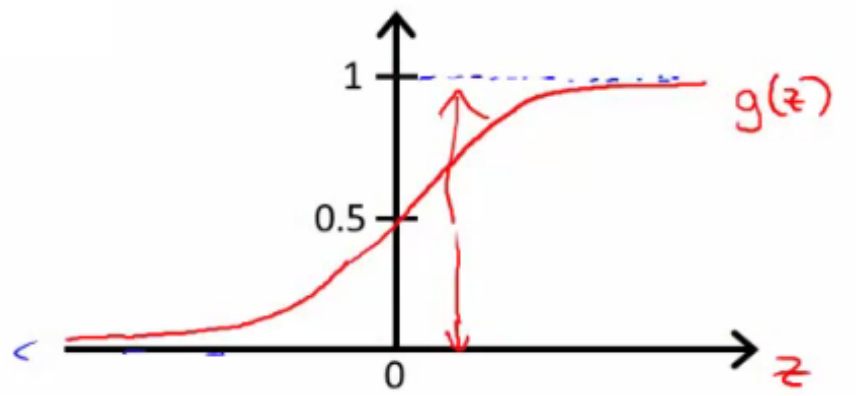
- So, logistic regression generates a value where is always either 0 or 1
  - Logistic regression is a **classification algorithm** - don't be confused

Hypothesis representation

- What function is used to represent our hypothesis in classification
- We want our classifier to output values between 0 and 1
  - When using linear regression we did $h_\theta(x) = (\theta^T x)$
  - For classification hypothesis representation we do $h_\theta(x) = g((\theta^T x))$
    - Where we define g(z)
      - z is a real number
    - $g(z) = 1/(1 + e^{-z})$
      - This is the **sigmoid function**, or the **logistic function**
    - If we combine these equations we can write out the hypothesis as



- What does the sigmoid function look like
- Crosses 0.5 at the origin, then flattens out]
  - Asymptotes at 0 and 1



- Given this we need to fit θ to our data

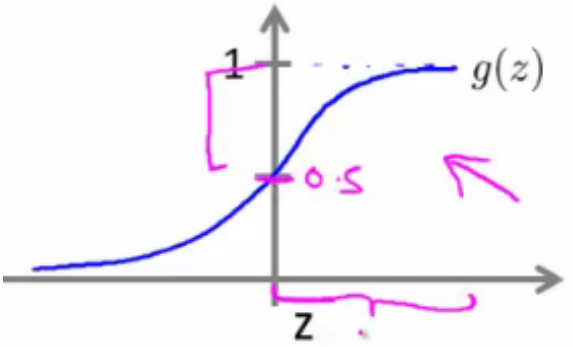
Interpreting hypothesis output

- When our hypothesis ($h_\theta(x)$) outputs a number, we treat that value as the estimated probability that y=1 on input x
  - Example
    - If X is a feature vector with $x_0 = 1$ (as always) and x_1 = tumourSize
    - $h_\theta(x) = 0.7$
      - Tells a patient they have a 70% chance of a tumor being malignant
  - We can write this using the following notation

        - $h_\theta(x) = P(y=1|x ; \theta)$
    - What does this mean?
        - Probability that y=1, given x, parameterized by θ
- Since this is a binary classification task we know y = 0 or 1
    - So the following must be true
        - $P(y=1|x ; \theta) + P(y=0|x ; \theta) = 1$
        - $P(y=0|x ; \theta) = 1 - P(y=1|x ; \theta)$

Decision boundary

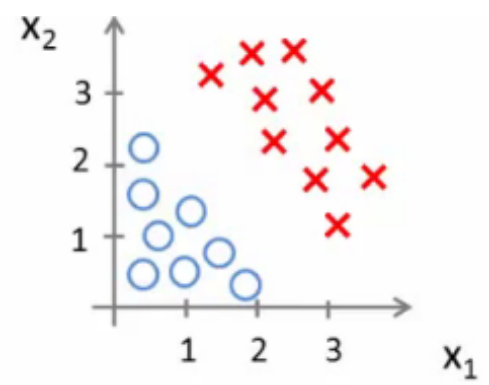
- Gives a better sense of what the hypothesis function is computing
- Better understand of what the hypothesis function looks like
    - One way of using the sigmoid function is;
        - When the probability of y being 1 is greater than 0.5 then we can predict y = 1
        - Else we predict y = 0
    - When is it exactly that $h_\theta(x)$ is greater than 0.5?
        - Look at sigmoid function
            - g(z) is greater than or equal to 0.5 when z is greater than or equal to 0



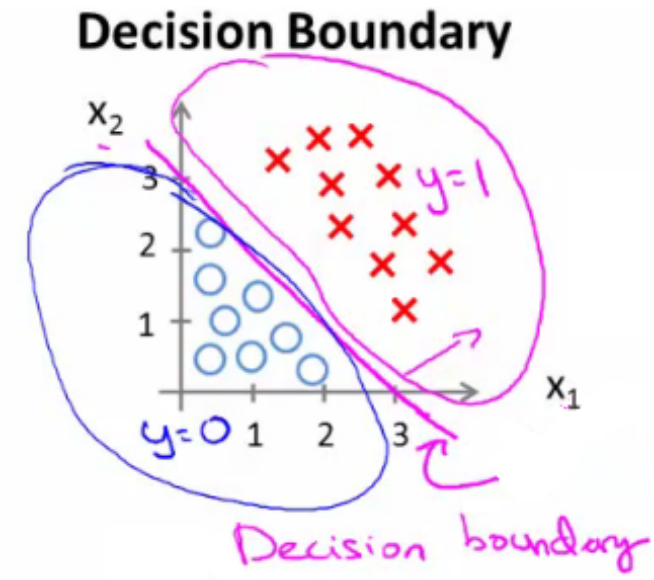
        - So if z is positive, g(z) is greater than 0.5
            - $z = (\theta^T x)$
        - So when
            - $\theta^T x >= 0$
        - Then $h_\theta >= 0.5$
- So what we've shown is that the hypothesis predicts y = 1 when $\theta^T x >= 0$
    - The corollary of that when $\theta^T x <= 0$ then the hypothesis predicts y = 0
    - Let's use this to better understand how the hypothesis makes its predictions

Decision boundary

- $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$



- So, for example
  - $\theta_0 = -3$
  - $\theta_1 = 1$
  - $\theta_2 = 1$
- So our parameter vector is a column vector with the above values
  - So, θ^T is a row vector = [-3,1,1]
- What does this mean?
  - The z here becomes $\theta^T x$
  - We predict "y = 1" if
    - $-3x_0 + 1x_1 + 1x_2 >= 0$
    - $-3 + x_1 + x_2 >= 0$
- We can also re-write this as
  - If ($x_1 + x_2 >= 3$) then we predict y = 1
  - If we plot
    - $x_1 + x_2 = 3$ we graphically plot our **decision boundary**



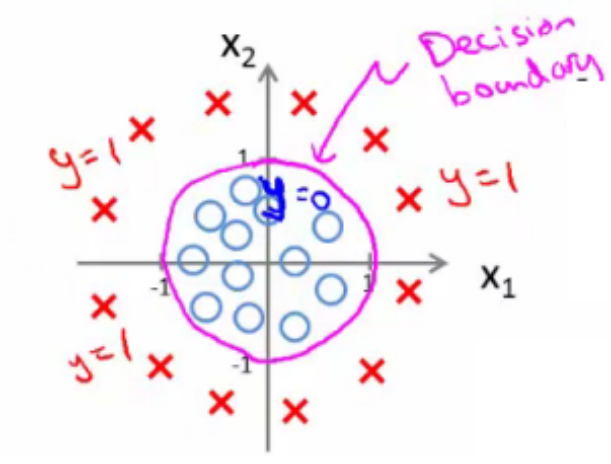
- Means we have these two regions on the graph
  - Blue = false

- Magenta = true
- Line = decision boundary
  - Concretely, the straight line is the set of points where $h_\theta(x) = 0.5$ exactly
- The decision boundary is a property of the hypothesis
  - Means we can create the boundary with the hypothesis and parameters without any data
    - Later, we use the data to determine the parameter values
  - i.e. y = 1 if
    - $5 - x_1 > 0$
    - $5 > x_1$

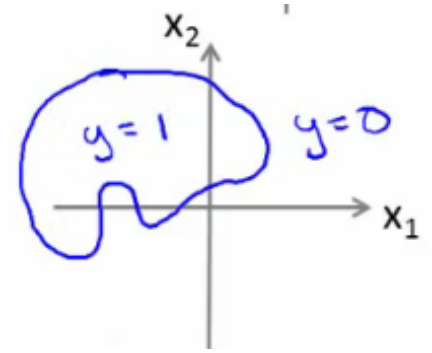
Non-linear decision boundaries

- Get logistic regression to fit a complex non-linear data set
  - Like polynomial regress add higher order terms
  - So say we have
    - $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_3 x_1^2 + \theta_4 x_2^2)$
    - We take the transpose of the θ vector times the input vector
      - Say θ^T was [-1,0,0,1,1] then we say;
      - Predict that "y = 1" *if*
        - $-1 + x_1^2 + x_2^2 >= 0$

          or
        - $x_1^2 + x_2^2 >= 1$
      - If we plot $x_1^2 + x_2^2 = 1$
        - This gives us a circle with a radius of 1 around 0



- Mean we can build more complex decision boundaries by fitting complex parameters to this (relatively) simple hypothesis
- More complex decision boundaries?
  - By using higher order polynomial terms, we can get even more complex decision boundaries



Cost function for logistic regression

- Fit θ parameters
- Define the optimization object for the cost function we use the fit the parameters
  - Training set of *m* training examples
    - Each example has is n+1 length column vector

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

m examples $x \in \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix}$ $x_0 = 1, y \in \{0, 1\}$

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

- This is the situation
  - Set of m training examples
  - Each example is a feature vector which is n+1 dimensional
  - $x_0 = 1$
  - $y \in \{0,1\}$
  - Hypothesis is based on parameters (θ)
    - Given the training set how to we chose/fit θ?
- Linear regression uses the following function to determine θ

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

- Instead of writing the squared error term, we can write

    - If we define "cost()" as;
        - $\text{cost}(h_\theta(x^i), y) = 1/2(h_\theta(x^i) - y^i)^2$
        - Which evaluates to the cost for an individual example using the same measure as used in linear regression
    - We can **redefine J(θ) as**

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

        - Which, appropriately, is the sum of all the individual costs over the training data (i.e. the same as linear regression)
- To further simplify it we can get rid of the superscripts
    - So

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

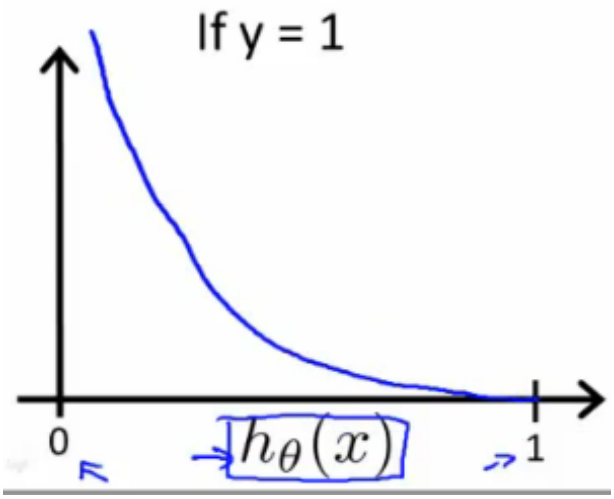
- What does this actually mean?
    - This is the cost you want the learning algorithm to pay if the outcome is $h_\theta(x)$ and the actual outcome is y
    - If we use this function for logistic regression this is a **non-convex function** for parameter optimization
        - Could work....
- What do we mean by non convex?
    - We have some function - J(θ) - for determining the parameters
    - Our hypothesis function has a non-linearity (sigmoid function of $h_\theta(x)$)
        - This is a complicated non-linear function
    - If you take $h_\theta(x)$ and plug it into the Cost() function, and them plug the Cost() function into J(θ) and plot J(θ) we find many local optimum -> *non convex function*
    - Why is this a problem
        - Lots of local minima mean gradient descent may not find the global optimum - may get stuck in a global minimum
    - We would like a convex function so if you run gradient descent you converge to a global minimum

A convex logistic regression cost function

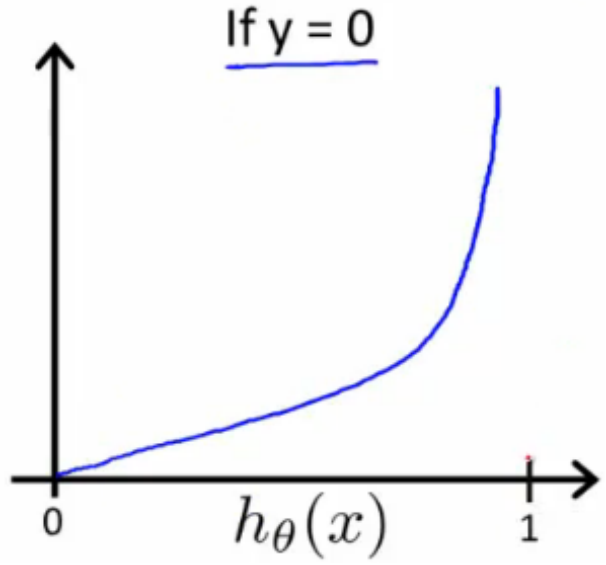
- To get around this we need a different, convex Cost() function which means we can apply gradient descent

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

- **This is our logistic regression cost function**
  - This is the penalty the algorithm pays
  - Plot the function
- Plot y = 1
  - So $h_\theta(x)$ evaluates as -log($h_\theta(x)$)



- So when we're right, cost function is 0
  - Else it slowly increases cost function as we become "more" wrong
  - X axis is what we predict
  - Y axis is the cost associated with that prediction
- This cost functions has some interesting properties
  - If y = 1 and $h_\theta(x)$ = 1
    - If hypothesis predicts exactly 1 and thats exactly correct then that corresponds to 0 (exactly, not nearly 0)
  - As $h_\theta(x)$ goes to 0
    - Cost goes to infinity
    - This captures the intuition that if $h_\theta(x)$ = 0 (predict P (y=1|x; θ) = 0) but y = 1 this will penalize the learning algorithm with a massive cost
- What about if y = 0
- then cost is evaluated as -log(1- h_θ(x))
  - Just get inverse of the other function



- Now it goes to plus infinity as $h_\theta(x)$ goes to 1
- With our particular cost functions J(θ) is going to be convex and avoid local minimum

Simplified cost function and gradient descent

- Define a simpler way to write the cost function and apply gradient descent to the logistic regression
  - By the end should be able to implement a fully functional logistic regression function
- Logistic regression cost function is as follows

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

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Note: $y = 0$ or 1 always

- This is the cost for a single example
  - For binary classification problems y is always 0 or 1
    - Because of this, we can have a simpler way to write the cost function
      - Rather than writing cost function on two lines/two cases
      - Can compress them into one equation - more efficient
  - Can write cost function is
    - **$\text{cost}(h_\theta(x), y) = -y\log(h_\theta(x)) - (1-y)\log(1 - h_\theta(x))$**

        - This equation is a more compact of the two cases above
    - We know that there are only two possible cases
        - y = 1
            - Then our equation simplifies to
                - $-\log(h_\theta(x)) - (0)\log(1 - h_\theta(x))$
                    - $-\log(h_\theta(x))$
                    - Which is what we had before when y = 1
        - y = 0
            - Then our equation simplifies to
                - $-(0)\log(h_\theta(x)) - (1)\log(1 - h_\theta(x))$
                - $= -\log(1- h_\theta(x))$
                - Which is what we had before when y = 0
        - Clever!
- So, in summary, our cost function for the θ parameters can be defined as

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

- Why do we chose this function when other cost functions exist?
    - This cost function can be derived from statistics using the principle of **maximum likelihood estimation**
        - Note this does mean there's an underlying Gaussian assumption relating to the distribution of features
    - Also has the nice property that it's convex
- To fit parameters θ:
    - Find parameters θ which minimize J(θ)
    - This means we have a set of parameters to use in our model for future predictions
- Then, if we're given some new example with set of features x, we can take the θ which we generated, and output our prediction using

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

    - This result is
        - p(y=1 | x ; θ)
            - Probability y = 1, given x, parameterized by θ

How to minimize the logistic regression cost function

- Now we need to figure out how to minimize J(θ)
    - Use gradient descent as before
    - Repeatedly update each parameter using a learning rate

Repeat {

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

(simultaneously update all θ_j)

}

- If you had *n* features, you would have an n+1 column vector for θ
- This equation is the same as the linear regression rule
  - The only difference is that our definition for the hypothesis has changed
- Previously, we spoke about how to monitor gradient descent to check it's working
  - Can do the same thing here for logistic regression
- When implementing logistic regression with gradient descent, we have to update all the θ values (θ_0 to θ_n) simultaneously
  - Could use a for loop
  - Better would be a vectorized implementation
- Feature scaling for gradient descent for logistic regression also applies here

Advanced optimization

- Previously we looked at gradient descent for minimizing the cost function
- Here look at advanced concepts for minimizing the cost function for logistic regression
  - Good for large machine learning problems (e.g. huge feature set)
- *What is gradient descent actually doing?*
  - We have some cost function J(θ), and we want to minimize it
  - We need to write code which can take θ as input and compute the following
    - J(θ)
    - Partial derivative if J(θ) with respect to j (where j=0 to j = n)

$$J(\theta)$$

$$\frac{\partial}{\partial \theta_j} J(\theta) \quad \text{(for } j = 0, 1, \ldots, n \text{)}$$

- Given code that can do these two things
  - Gradient descent repeatedly does the following update

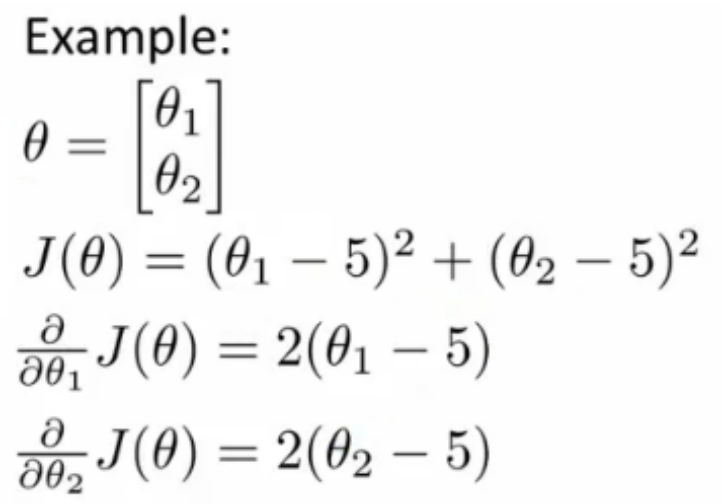
Repeat $\{ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \}$

- So update each j in θ sequentially
- So, we must;

    - Supply code to compute J(θ) and the derivatives
    - Then plug these values into gradient descent
- Alternatively, instead of gradient descent to minimize the cost function we could use
    - **Conjugate gradient**
    - **BFGS** (Broyden-Fletcher-Goldfarb-Shanno)
    - **L-BFGS** (Limited memory - BFGS)
- These are more optimized algorithms which take that same input and minimize the cost function
- These are *very* complicated algorithms
- Some properties
    - **Advantages**
        - No need to manually pick alpha (learning rate)
            - Have a clever inner loop (line search algorithm) which tries a bunch of alpha values and picks a good one
        - Often faster than gradient descent
            - Do more than just pick a good learning rate
        - Can be used successfully without understanding their complexity
    - **Disadvantages**
        - Could make debugging more difficult
        - Should not be implemented themselves
        - Different libraries may use different implementations - may hit performance

Using advanced cost minimization algorithms

- How to use algorithms
    - Say we have the following example



- Example above
    - θ_1 and θ_2 (two parameters)
    - Cost function here is $J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$
    - The derivatives of the $J(\theta)$ with respect to either θ_1 and θ_2 turns out to be the $2(\theta_i - 5)$

- First we need to define our cost function, which should have the following signature

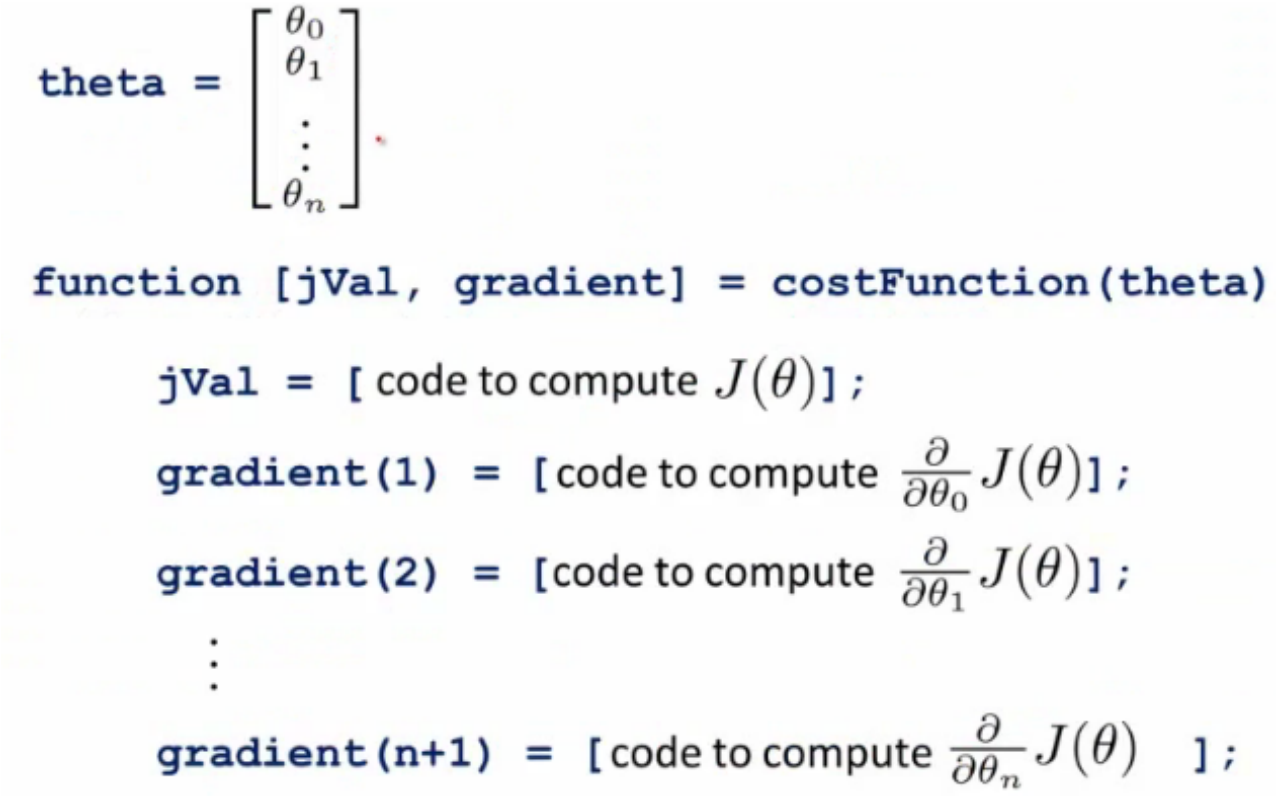
function [jval, gradent] = costFunction(THETA)

- Input for the cost function is **THETA**, which is a vector of the θ parameters
- Two return values from **costFunction** are
  - **jval**
    - How we compute the cost function θ (the underived cost function)
      - In this case = $(\theta_1 - 5)^2 + (\theta_2 - 5)^2$
  - **gradient**
    - 2 by 1 vector
    - 2 elements are the two partial derivative terms
    - i.e. this is an n-dimensional vector
      - Each indexed value gives the partial derivatives for the partial derivative of J(θ) with respect to θ_i
      - Where i is the index position in the **gradient** vector
- With the cost function implemented, we can call the advanced algorithm using

```
options= optimset('GradObj', 'on', 'MaxIter', '100'); % define the options data structure
initialTheta= zeros(2,1); # set the initial dimensions for theta % initialize the theta values
[optTheta, funtionVal, exitFlag]= fminunc(@costFunction, initialTheta, options); % run the algorithm
```

- Here
  - **options** is a data structure giving options for the algorithm
  - **fminunc**
    - function minimize the cost function (**f**ind **min**imum of **unc**onstrained multivariable function)
  - **@costFunction** is a pointer to the costFunction function to be used
- For the octave implementation
  - **initialTheta** must be a matrix of at least two dimensions

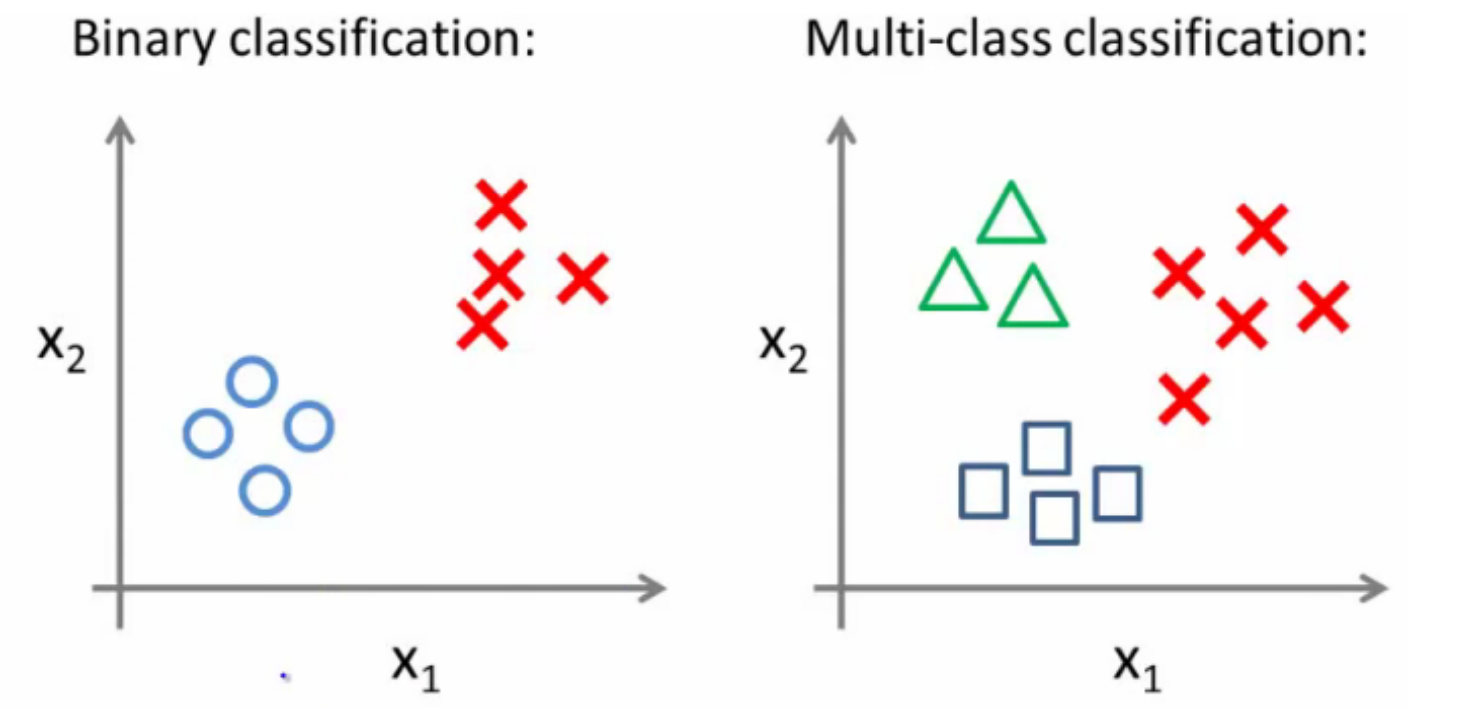
- How do we apply this to logistic regression?
  - Here we have a vector



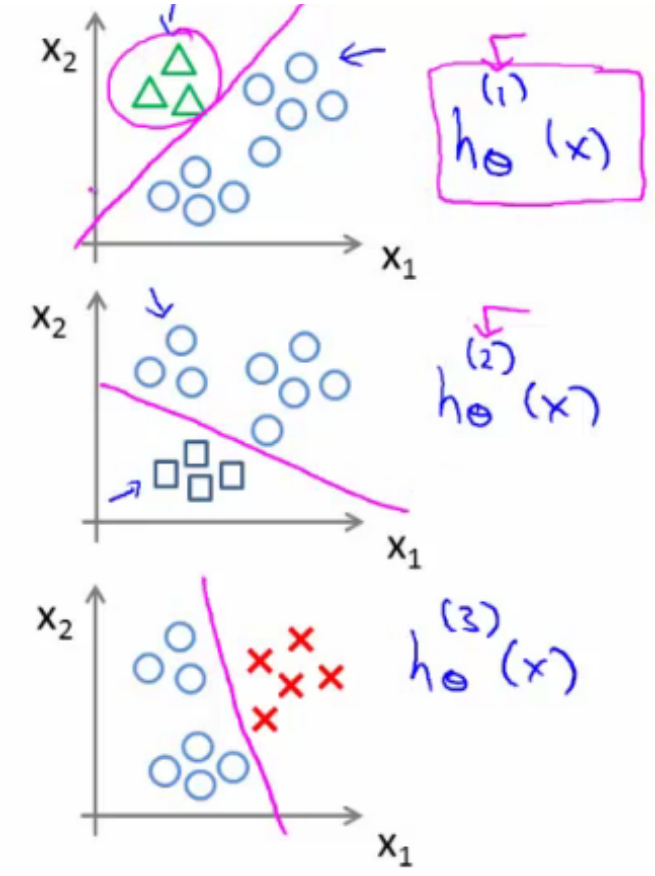
- Here
  - theta is a n+1 dimensional column vector
  - Octave indexes from 1, not 0
- Write a cost function which captures the cost function for logistic regression

Multiclass classification problems

- Getting logistic regression for multiclass classification using **one vs. all**
- Multiclass - more than yes or no (1 or 0)
  - Classification with multiple classes for assignment



- Given a dataset with three classes, how do we get a learning algorithm to work?
  - Use one vs. all classification make binary classification work for multiclass classification
- **One vs. all classification**
  - Split the training set into three separate binary classification problems
    - i.e. create a new fake training set
      - Triangle (1) vs crosses and squares (0) $h_\theta^1(x)$
        - $P(y=1 \mid x_1; \theta)$
      - Crosses (1) vs triangle and square (0) $h_\theta^2(x)$
        - $P(y=1 \mid x_2; \theta)$
      - Square (1) vs crosses and square (0) $h_\theta^3(x)$
        - $P(y=1 \mid x_3; \theta)$



- **Overall**
  - Train a logistic regression classifier $h_\theta^{(i)}(x)$ for each class i to predict the probability that y = i
  - On a new input, *x* to make a prediction, pick the class *i* that maximizes the probability that $h_\theta^{(i)}(x) = 1$

07: Regularization

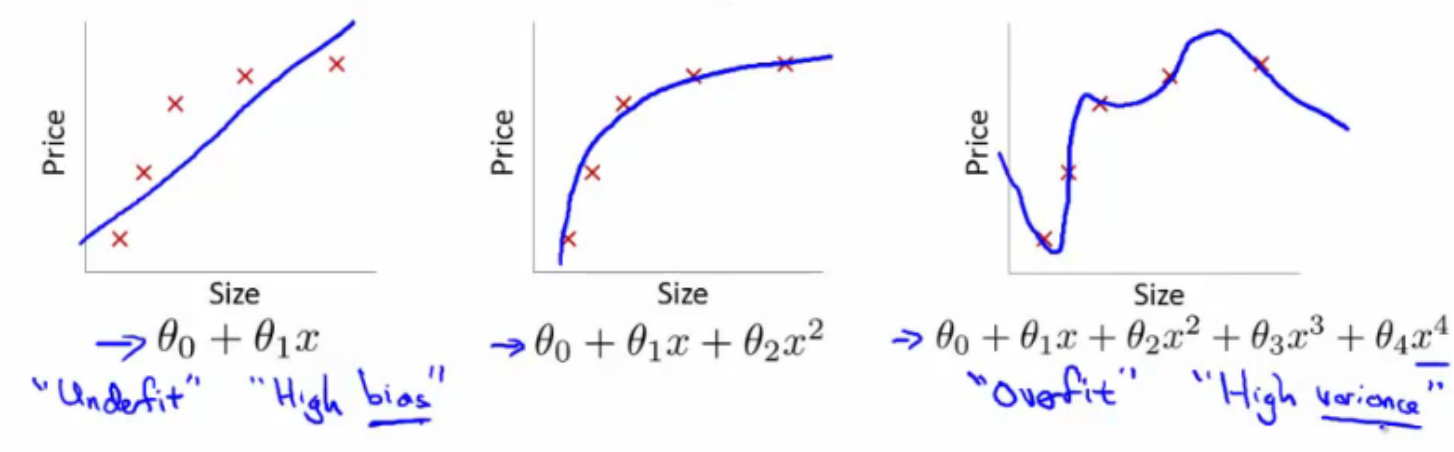
Previous Next Index

The problem of overfitting

- So far we've seen a few algorithms - work well for many applications, but can suffer from the problem of overfitting
- What is overfitting?
- What is regularization and how does it help

Overfitting with linear regression

- Using our house pricing example again
  - Fit a linear function to the data - not a great model
    - This is **underfitting** - also known as **high bias**
    - Bias is a historic/technical one - if we're fitting a straight line to the data we have a strong preconception that there should be a linear fit
      - In this case, this is not correct, but a straight line can't help being straight!
  - Fit a quadratic function
    - Works well
  - Fit a 4th order polynomial
    - Now curve fit's through all five examples
      - Seems to do a good job fitting the training set
      - But, despite fitting the data we've provided very well, this is actually not such a good model
    - This is **overfitting** - also known as **high variance**
  - Algorithm has high variance
    - High variance - if fitting high order polynomial then the hypothesis can basically fit any data
    - Space of hypothesis is too large

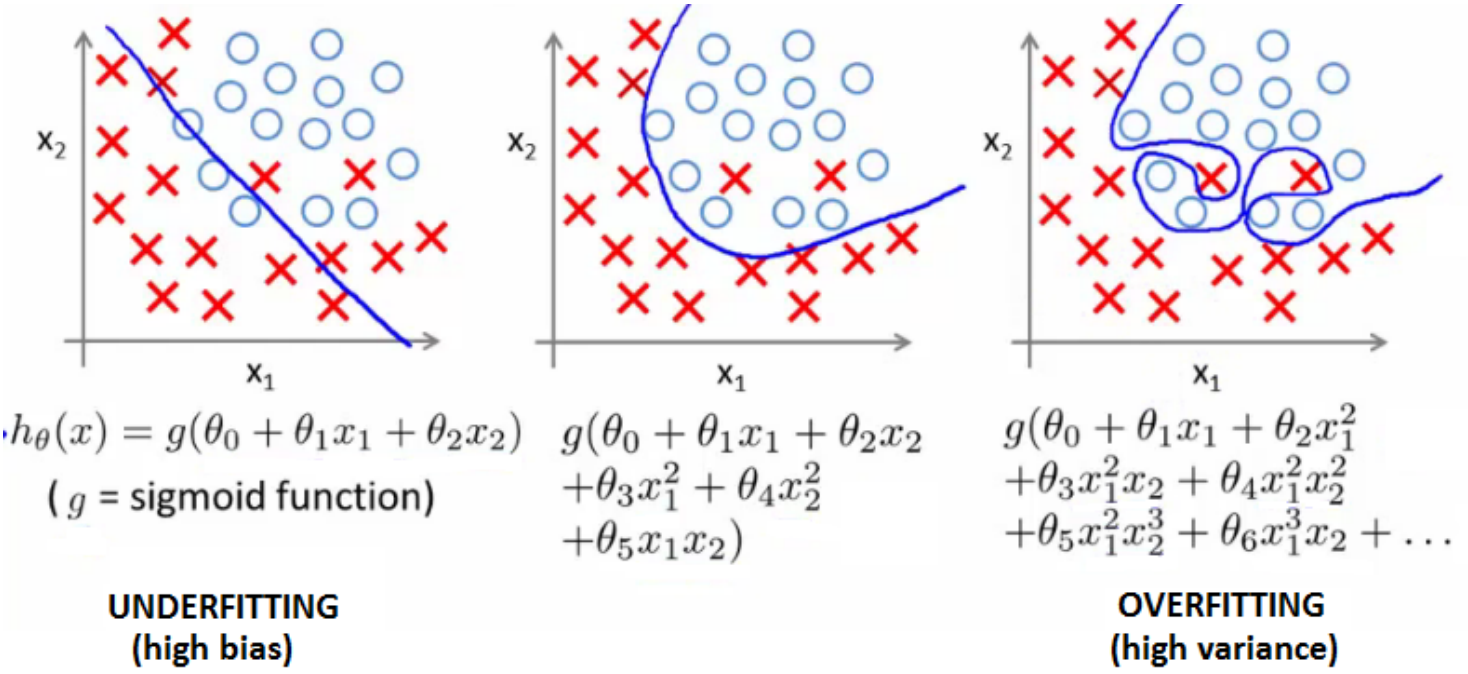


- To recap, if we have too many features then the learned hypothesis may give a cost function of exactly zero
  - But this tries too hard to fit the training set

- Fails to provide a *general* solution - **unable to generalize** (apply to new examples)

Overfitting with logistic regression

- Same thing can happen to logistic regression
  - Sigmoidal function is an underfit
  - But a high order polynomial gives and overfitting (high variance hypothesis)



Addressing overfitting

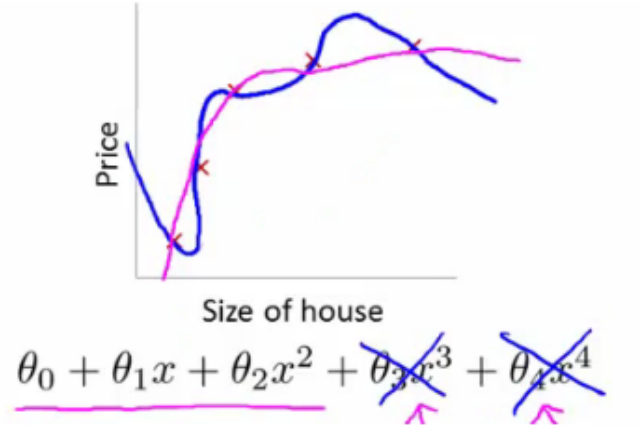
- Later we'll look at identifying when overfitting and underfitting is occurring
- Earlier we just plotted a higher order function - saw that it looks "too curvy"
  - Plotting hypothesis is one way to decide, but doesn't always work
  - Often have lots of a features - here it's not just a case of selecting a degree polynomial, but also harder to plot the data and visualize to decide what features to keep and which to drop
  - If you have lots of features and little data - overfitting can be a problem
- How do we deal with this?
  - 1) **Reduce number of features**
    - Manually select which features to keep
    - Model selection algorithms are discussed later (good for reducing number of features)
    - But, in reducing the number of features we lose some information
      - Ideally select those features which minimize data loss, but even so, some info is lost
  - 2) **Regularization**
    - Keep all features, but reduce magnitude of parameters θ
    - Works well when we have a lot of features, each of which contributes a bit to predicting y

Cost function optimization for regularization

- Penalize and make some of the θ parameters really small
  - e.g. here θ_3 and θ_4

$$\min_\theta \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\,\theta_3^2 + 1000\,\theta_4^2$$

- The addition in blue is a modification of our cost function to help penalize θ_3 and θ_4
  - So here we end up with θ_3 and θ_4 being close to zero (because the constants are massive)
  - So we're basically left with a quadratic function



- In this example, we penalized two of the parameter values
  - More generally, regularization is as follows

- Regularization
  - Small values for parameters corresponds to a simpler hypothesis (you effectively get rid of some of the terms)
  - A simpler hypothesis is less prone to overfitting
- Another example
  - Have 100 features x_1, x_2, ..., x_{100}
  - Unlike the polynomial example, we don't know what are the high order terms
    - How do we pick the ones to pick to shrink?
  - With regularization, take cost function and modify it to shrink all the parameters
    - Add a term at the end
      - This regularization term shrinks every parameter
      - By convention you don't penalize θ_0 - minimization is from θ_1 onwards

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

$\theta_1, \theta_2, \theta_3, \ldots, \theta_{100}$

- In practice, if you include θ_0 has little impact
- **λ** is the **regularization parameter**
  - Controls a trade off between our two goals
    - 1) Want to fit the training set well
    - 2) Want to keep parameters small
- With our example, using the **regularized objective** (i.e. the cost function with the regularization term) you get a much smoother curve which fits the data and gives a much better hypothesis
  - If **λ** is very large we end up penalizing ALL the parameters (θ_1, θ_2 etc.) so all the parameters end up being close to zero
    - If this happens, it's like we got rid of all the terms in the hypothesis
      - This results here is then underfitting
    - So this hypothesis is too biased because of the absence of any parameters (effectively)
- So, **λ** should be chosen carefully - not too big...
  - We look at some automatic ways to select **λ** later in the course

Regularized linear regression

- Previously, we looked at two algorithms for linear regression
  - Gradient descent
  - Normal equation
- Our linear regression with regularization is shown below

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

$$\min_\theta J(\theta)$$

- Previously, gradient descent would repeatedly update the parameters θ_j, where j = 0,1,2...n simultaneously
  - Shown below

Repeat $\{$

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

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

$$(j = 1, 2, 3, \ldots, n)\ \}$$

- We've got the θ_0 update here shown explicitly

    - This is because for regularization we don't penalize θ_0 so treat it slightly differently
- How do we regularize these two rules?
    - Take the term and add $\lambda/m * \theta_j$
        - Sum for every θ (i.e. j = 0 to n)
    - This gives regularization for gradient descent
- We can show using calculus that the equation given below is the partial derivative of the regularized J(θ)

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

$$(j = 0, 1, 2, 3, \ldots, n)$$

$\frac{\partial}{\partial \theta_j} J(\theta)$ — regularized

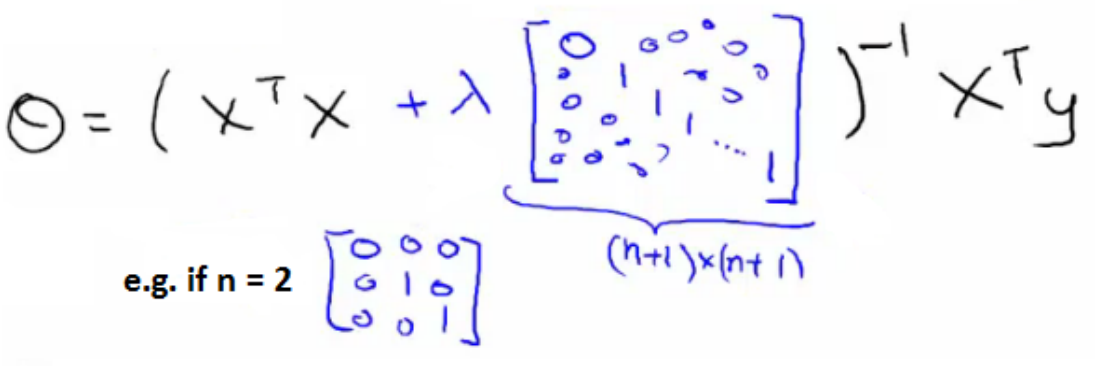
- The update for θ_j
    - θ_j gets updated to
        - θ_j - α * [a big term which also depends on θ_j]
- So if you group the θ_j terms together

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

- The term
  $$(1 - \alpha \frac{\lambda}{m})$$
    - Is going to be a number less than 1 usually
    - Usually learning rate is small and m is large
        - So this typically evaluates to (1 - a small number)
        - So the term is often around 0.99 to 0.95
- This in effect means θ_j gets multiplied by 0.99
    - Means the squared norm of θ_j a little smaller
    - The second term is exactly the same as the original gradient descent

Regularization with the normal equation

- Normal equation is the other linear regression model
    - Minimize the J(θ) using the normal equation
    - To use regularization we add a term (+ λ [n+1 x n+1]) to the equation
        - [n+1 x n+1] is the n+1 identity matrix



Regularization for logistic regression

- We saw earlier that logistic regression can be prone to overfitting with lots of features
- Logistic regression cost function is as follows;

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

- To modify it we have to add an extra term

$$+ \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

- This has the effect of penalizing the parameters θ_1, θ_2 up to θ_n
  - Means, like with linear regression, we can get what appears to be a better fitting lower order hypothesis
- How do we implement this?
  - Original logistic regression with gradient descent function was as follows

$$\theta_j := \theta_j - \alpha \quad \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$
$$(j = 0, 1, 2, 3, \ldots, n)$$

- Again, to modify the algorithm we simply need to modify the update rule for θ_1, onwards
  - Looks cosmetically the same as linear regression, except obviously the hypothesis is very different

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

Advanced optimization of regularized linear regression

- As before, define a costFunction which takes a θ parameter and gives jVal and gradient back

```
function [jVal, gradient] = costFunction(theta)
    jVal = [ code to compute J(θ)] ;
    gradient(1) = [code to compute ∂/∂θ0 J(θ) ] ;
    gradient(2) = [code to compute ∂/∂θ1 J(θ) ] ;
    gradient(3) = [code to compute ∂/∂θ2 J(θ) ] ;
      :
    gradient(n+1) = [code to compute ∂/∂θn J(θ) ] ;
```

- use **fminunc**
  - Pass it an **@costfunction** argument
  - Minimizes in an optimized manner using the cost function
- **jVal**
  - Need code to compute J(θ)
    - Need to include regularization term
- Gradient
  - Needs to be the partial derivative of J(θ) with respect to θ_i
  - Adding the appropriate term here is also necessary

```
function [jVal, gradient] = costFunction(theta)
    jVal = [ code to compute J(θ)] ;
           J(θ) = [ -1/m Σ_{i=1}^{m} y^(i) log(h_θ(x^(i)) + (1 - y^(i)) log 1 - h_θ(x^(i)) ] + λ/2m Σ_{j=1}^{n} θ_j^2
    gradient(1) = [code to compute ∂/∂θ0 J(θ) ] ;
           1/m Σ_{i=1}^{m} (h_θ(x^(i)) - y^(i)) x_0^(i)
    gradient(2) = [code to compute ∂/∂θ1 J(θ) ] ;
           1/m Σ_{i=1}^{m} (h_θ(x^(i)) - y^(i)) x_1^(i) + λ/m θ1
    gradient(3) = [code to compute ∂/∂θ2 J(θ) ] ;
      :    1/m Σ_{i=1}^{m} (h_θ(x^(i)) - y^(i)) x_2^(i) + λ/m θ2
    gradient(n+1) = [code to compute ∂/∂θn J(θ) ] ;
```

- Ensure summation doesn't extend to to the lambda term!
  - It doesn't, but, you know, don't be daft!