Machine Learning Notes

This course: https://www.coursera.org/learn/machine-learning (w/ Prof Andrew Ng)

- Other resources: https://gist.github.com/off99555/b6190df237562aa6e8c922c485dc7ad0

Blog post: https://betterexplained.com/articles/adept-machine-learning-course/

1-sentence course summary

- “Create predictive models with Linear Algebra and improve them with Calculus.”
  - Linear Algebra efficiently describes complex operations using simple steps.
  - Calculus can optimize any model with derivatives (gradient). Minimize error.

1-sentence core concepts

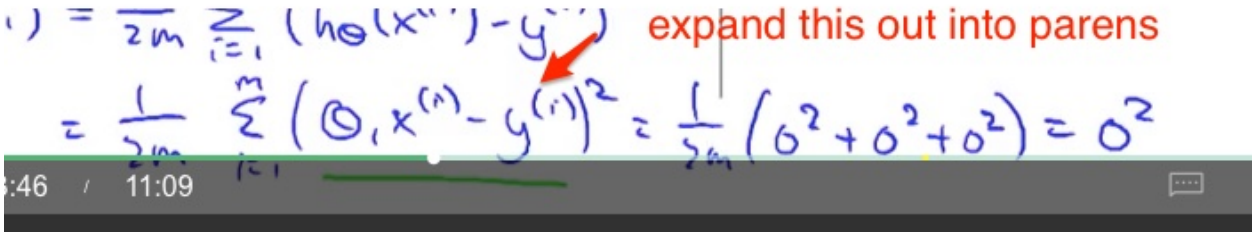
- Linear Algebra: spreadsheets for your equations. We "pour" data through various operations.
- Natural log: time needed to grow. Helps normalize widely varying numbers.
- e^x: models continuous growth, has a simple derivative.
- Gradient: direction of greatest change, helps optimize.
- Calculus -Art of breaking a system into steps. With the gradient, we can move in the best direction.

Project ideas

- Auto-classify / auto-relate articles on BE
- Correlation between words in title and popularity / comments / traffic

Week 1 - Linear Regression

Remember that you can expand out the summation… (0^2 + 0^2 + 0^2)



Q: Why are we using superscript notation and not subscript?

Aha: “Function of x” => translate to “Depends only on x”

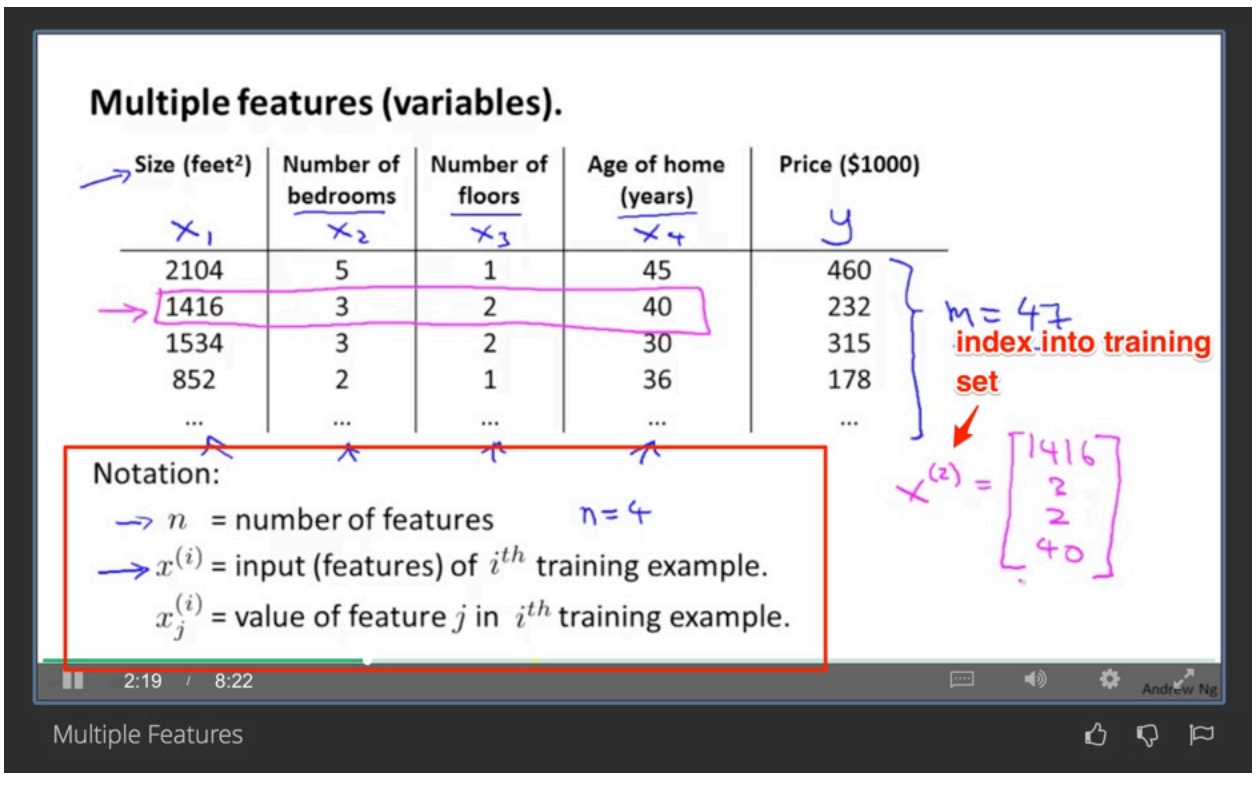
Gradient descent. (Have intuition on why gradient points in direction of greatest increase. In a linear coordinate system, reward each direction proportional to how much it helps. Because you can make any tradeoff you need, so can always trade into it.)

Gotcha: need simultaneous update. B/C you want to find the direction to move from all directions together. Not update 1, step, update 2, step.

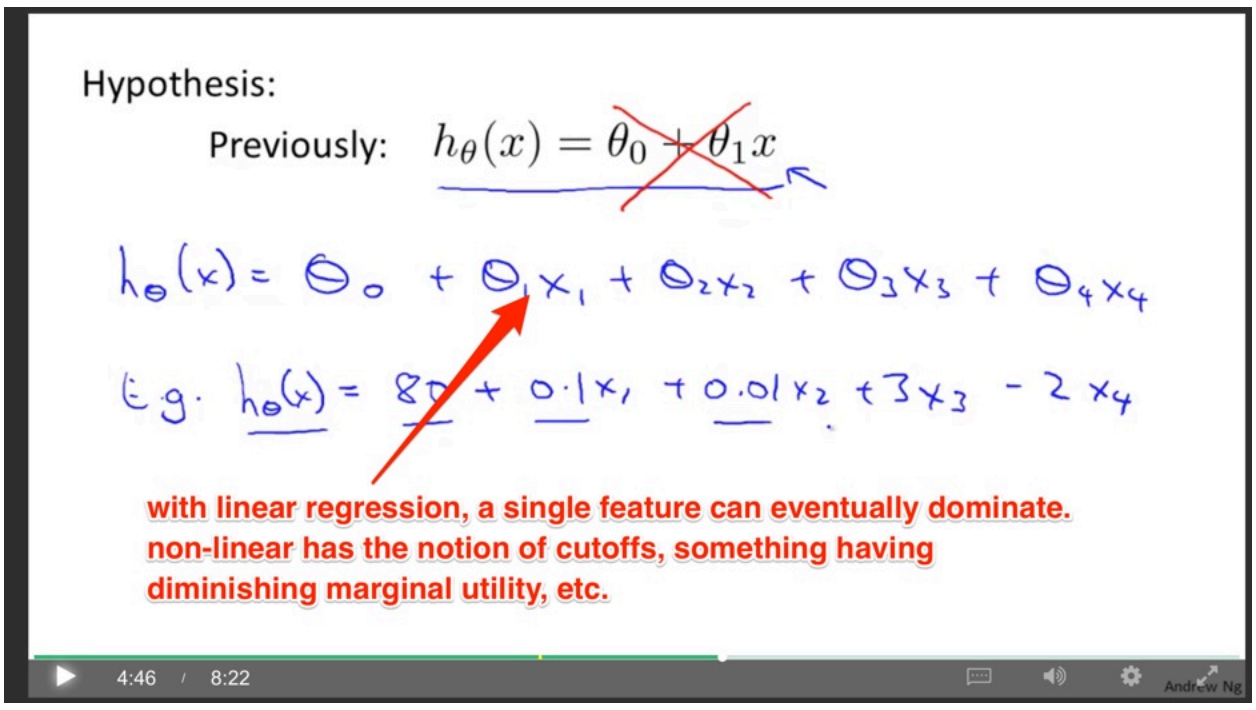
Aha: no need to change alpha (learning rate) since the derivative slows as we get closer to minimum.

Gradient Descent derivation: http://mccormickml.com/2014/03/04/gradient-descent-derivation/

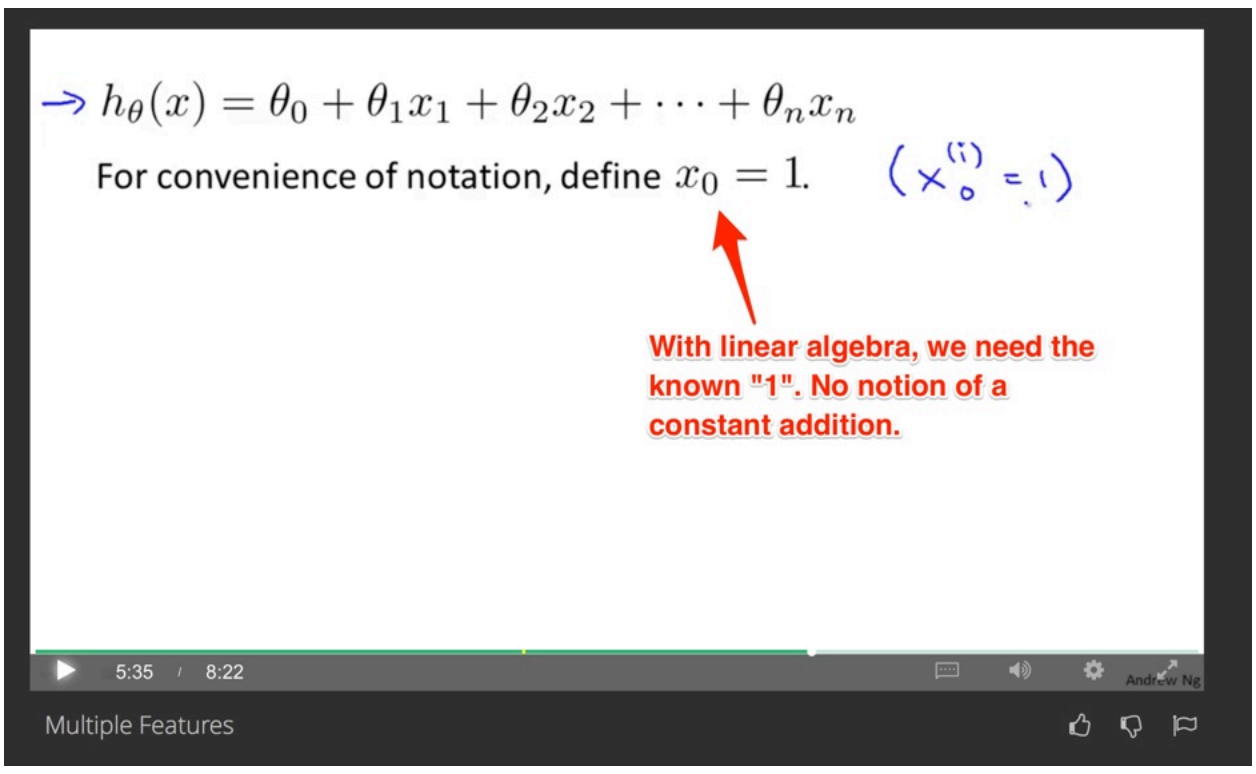
Week 2 - Linear Regression w/Multiple Variables



Q: I was confused about why in week 1 we only had superscript. Should have said super and subscripts are being used. Sub => more specific. Super => more general (the training row).

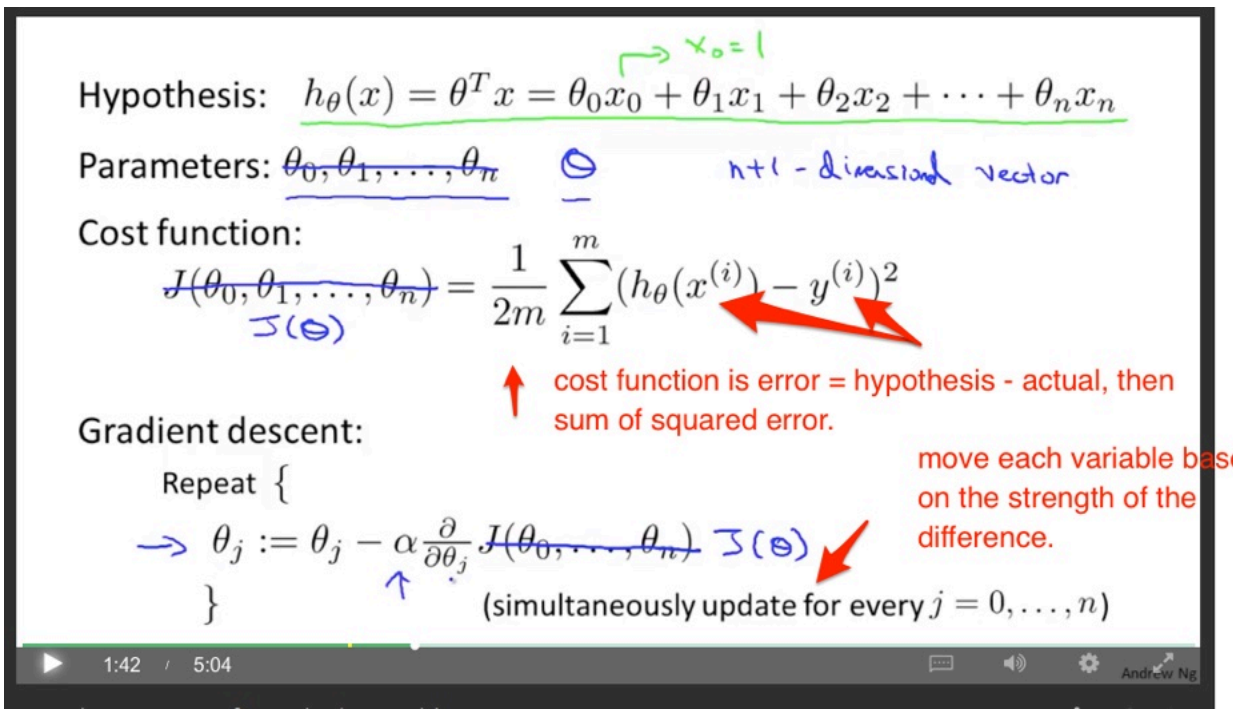


Aha: Can see the simplicity and drawbacks of linear regression. One feature can dominate. No idea of min/max value or a cutoff.



Dot product of feature vector with weight vector. Theta (transpose) x. Inner product.

Multivariate linear regression (multiple variables)



Now, remember the derivatives. d/dThetaj

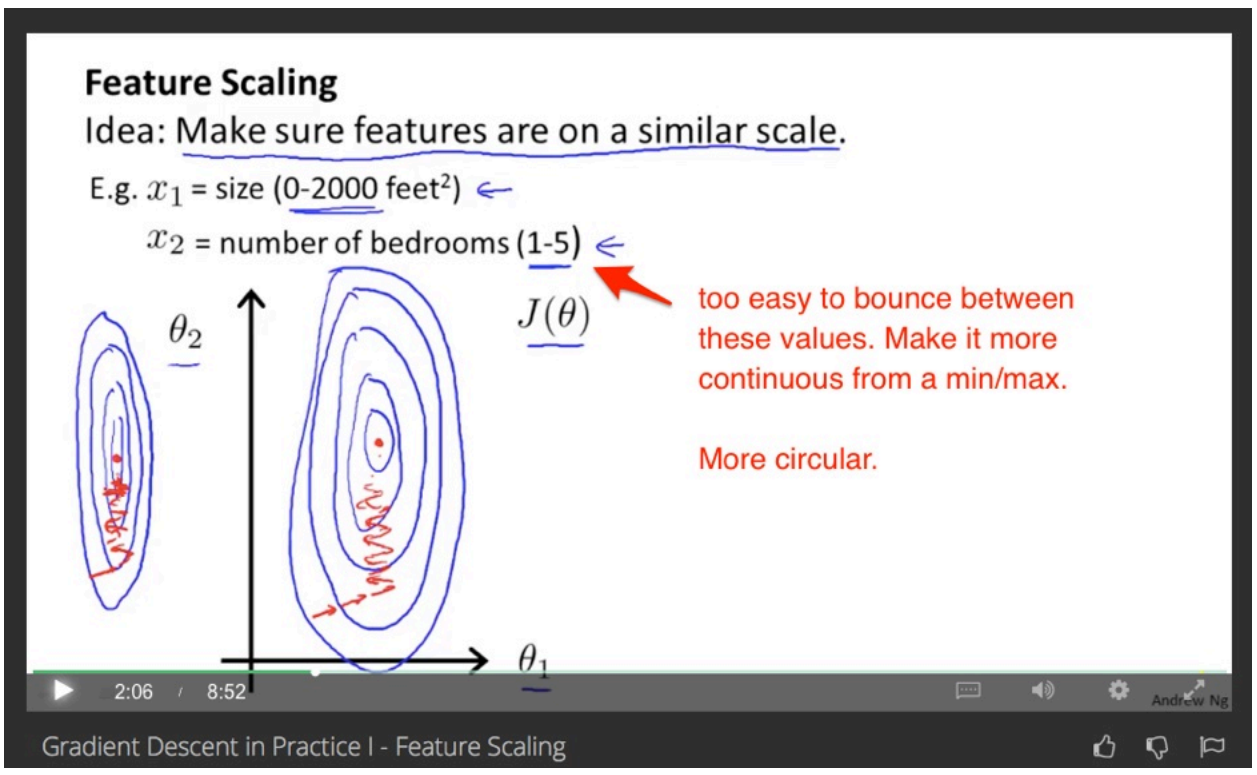
Derivative with respect to thetaJ. How much does error function change when we modify thetaJ?

For the constant value, it's 1. For the others, it's proportional to x_i, x_j, etc. If we change theta_j, we gain/lose cost proportional to x_j [because that's what's being multiplied].

Aha: Key intuition is that we are modifying the parameter based on how far off it is. If our error function gets better when the parameter changes, we change it in the appropriate direction.

- Should have a thought experiment. If you overshoot, then reverse it and change the angle. Imagine aiming at a target. If you are off (above) then you have to LOWER (negative above) your angle
- You lower proportional to the amount. If you're off by a lot, you lower by a lot. If you're off by a little, you change by a little.

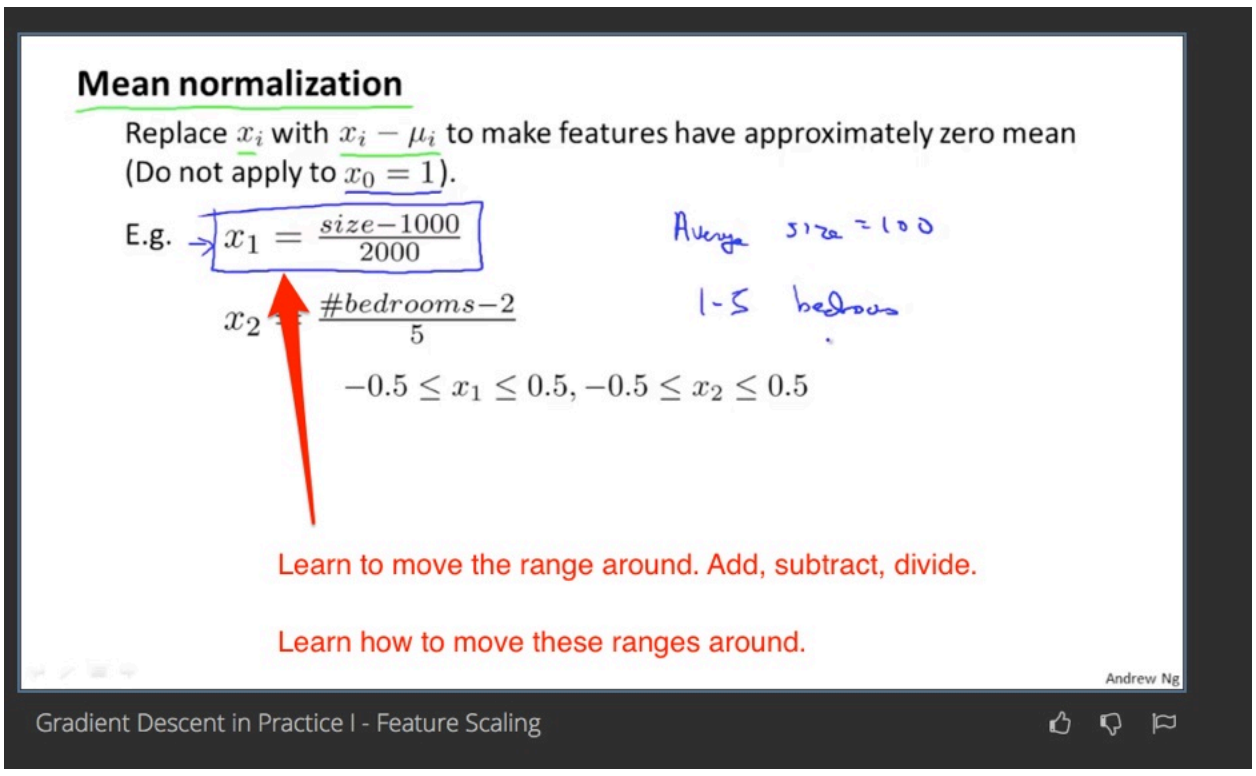
Feature scaling



Put it from a 0 to 1 scale. (normalize them to 100%).

Get every feature into a -1 to +1 range.

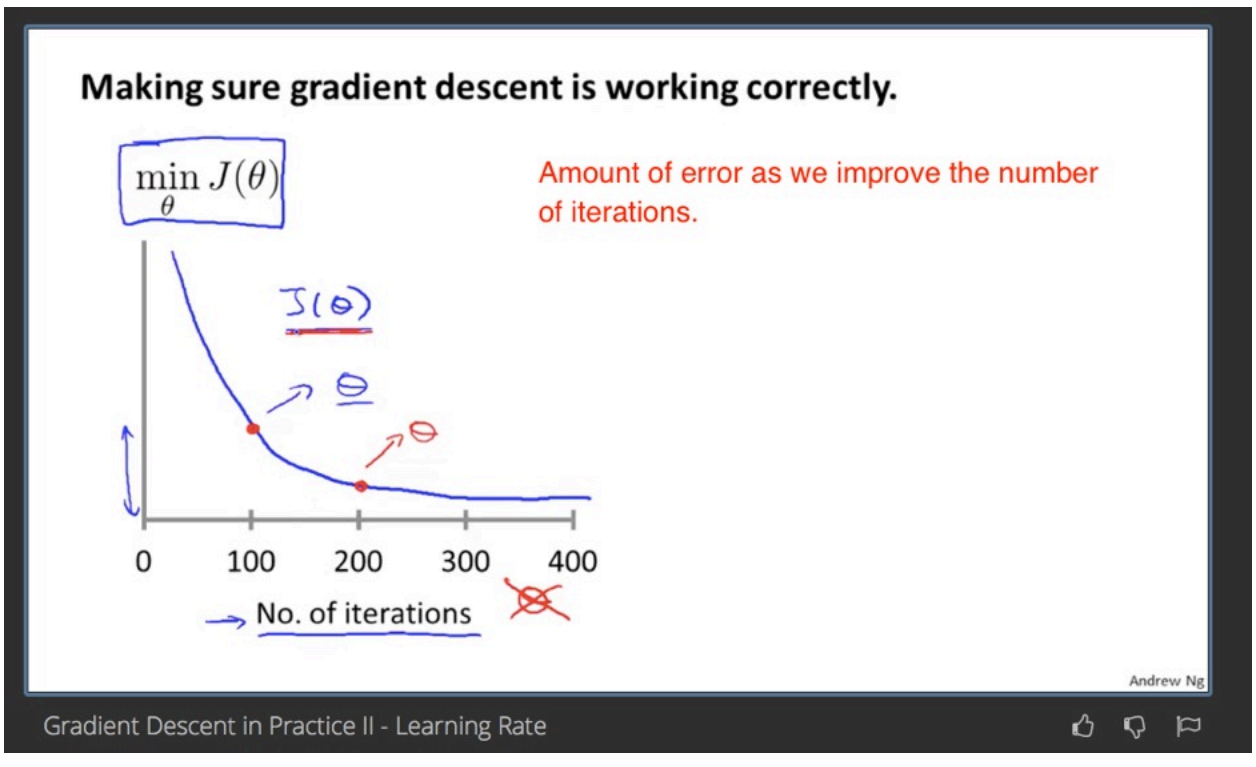
Mean normalization:



Remember the mental arithmetic? We can take the range (on the number line) and slide / scale it to fit. That's what's happening.

Normalized = value - average / range

Get the midpoint over 0. Get the spread as the range. (Or use the standard deviation, etc.)



Declare convergence if cost increases by less than 10^-3 in one iteration. (Should that be a percent? Less than 1/1000 part?)

If cost function as number of iterations is INCREASING then our alpha is too big.

- **Q: Can we dynamically adjust alpha / undo if it increases?** Optimistically try to adjust alpha as we go.
- .1, .3, 1, 3, 10, 30, 100… (by about 3x each time)

Idea: Create new features (such as area, being derived from length and width)

TODO: get derivation for how to minimize theta in one step

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

(x^T x)-1 x^T y => gives a list of theta values

Pros/cons

Giant matrices are slow to compute. O(n^3). Slow if number of features is huge. *n* features. n >= 10,000. Interesting… not related to number of number of samples (*m*)?

Gradient descent iterates (and you'll want to feature scale)

Normal equation (computing theta in a single computation) -- like normal vector? Find normal to the surface?

TODO: get intuition for meaning of “x^T x” … in plain English, what are we trying to do?

- DONE: Learn how to treat code as data, data as code. Vertical column… data, or a list of 3 functions to apply. Learn to switch between it. Horizontal row… function that takes 3 parameters, or 3 data elements (of 1 each)
- This is the dot product of all combinations. First row with itself, first dot second, second dot first, second dot second. Cool.
    - X X’ => dot product with itself (results in a 1x1)
    - X’ X => all dot products (results in an n x n)

Ah… determinant helping to solve simultaneous equations. I see! Remember how to solve a system of equations with regular linear algebra. That is basically it… we are solving the partial derivatives. Gotcha.

d/dTheta = Current Value * feature = 0

If linearly dependent features, redundant. (like redundant equations)

Too many features (m <= n)... we can overfit. Get rid of some features. Regularization.

.* is “element-wise product”. How do we do that in math? Name for this?

-> [Hadamard product (matrices) - Wikipedia, the free encyclopedia]

Vectorization example.

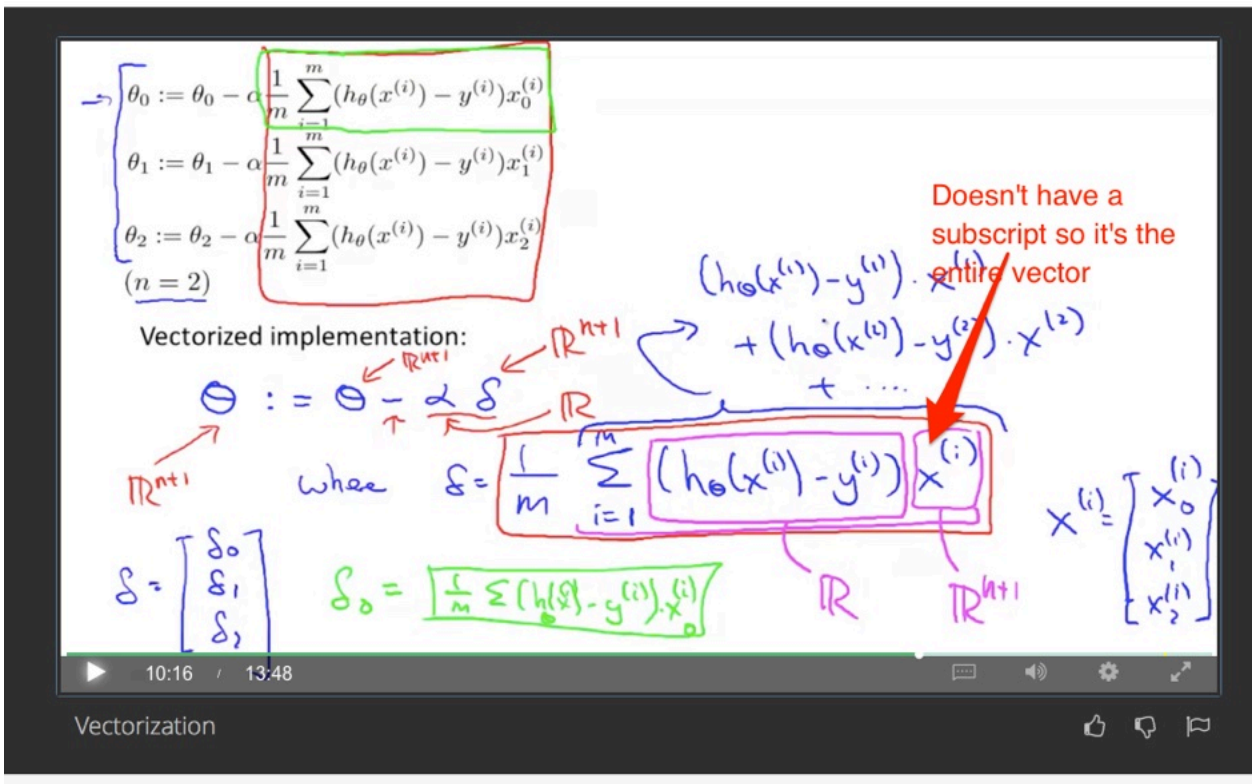
$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

$$= \theta^T x$$

Linear algebra gives us for loops in our math equations

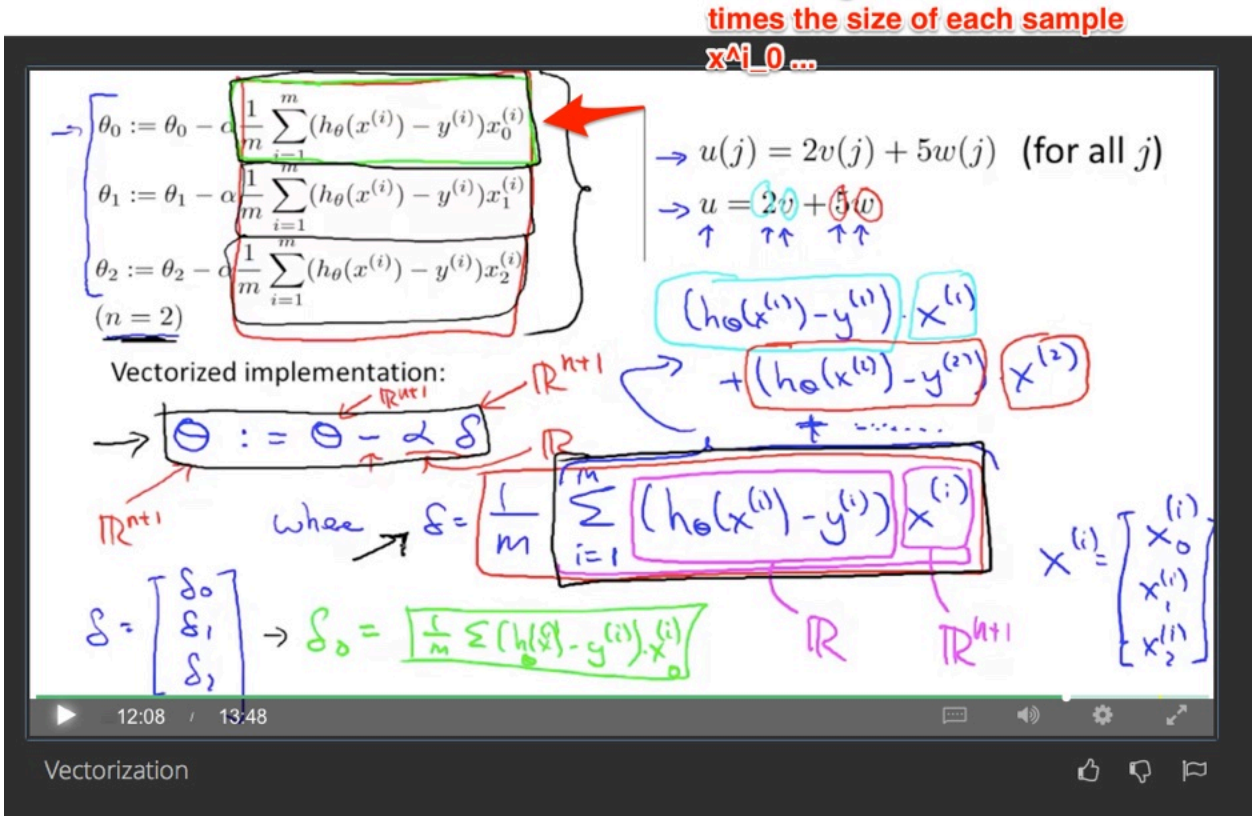
Unvectorized implementation

```
double prediction = 0.0;
for (int j = 0; j <= n; j++)
  prediction += theta[j] * x[j];
```



All the samples are fighting about
how to change Theta_0... we average the votes (whether to increase or decrease theta 0).

The change is the derivative times the size of each sample x^i_0 ...



That's the key intuition… how is EACH term voting for how to change theta. The derivative contains the value. The larger the current value, the more weight we get.

Homework notes

- Debug things step by step
- Talk through what is happening (each item is "voting" on where to pull theta)
- Debug with simple examples by hand / on command line

Normalizing:

- Convert the raw data into how many standard deviations it is from the mean
- Center & measure in "standard deviation units"

Dot product with itself (which is the squaring + sum)

Implementation Note: In the multivariate case, the cost function can also be written in the following vectorized form:

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

where

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}.$$

The vectorized version is efficient when you're working with numerical computing tools like Octave/MATLAB. If you are an expert with matrix operations, you can prove to yourself that the two forms are equivalent.

Vectorization shortcut

3.2.1 Optional (ungraded) exercise: Selecting learning rates

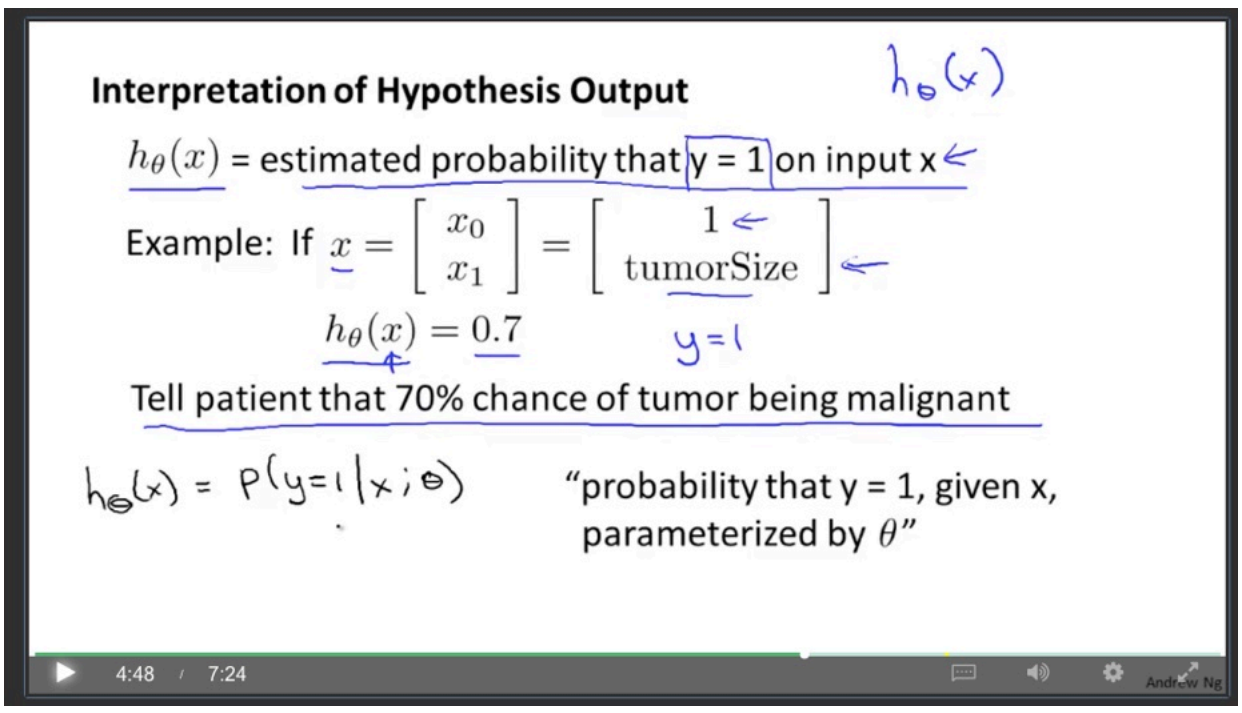
Make sure to normalize data before using theta (if needed)

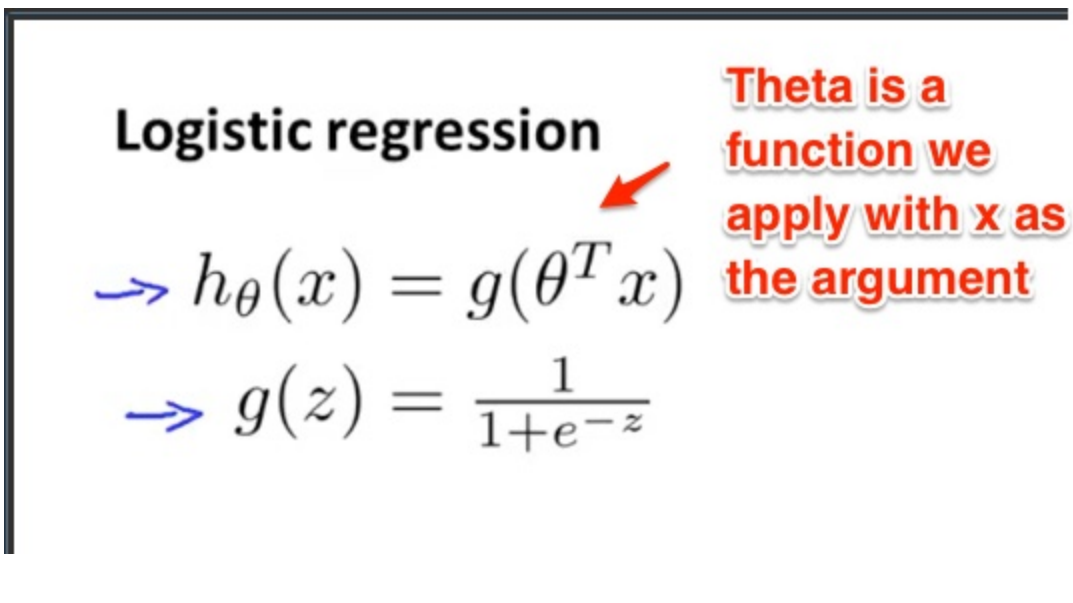
Week 3 - Logistic Regression / Regularization

Tradeoffs between regression and classifier.

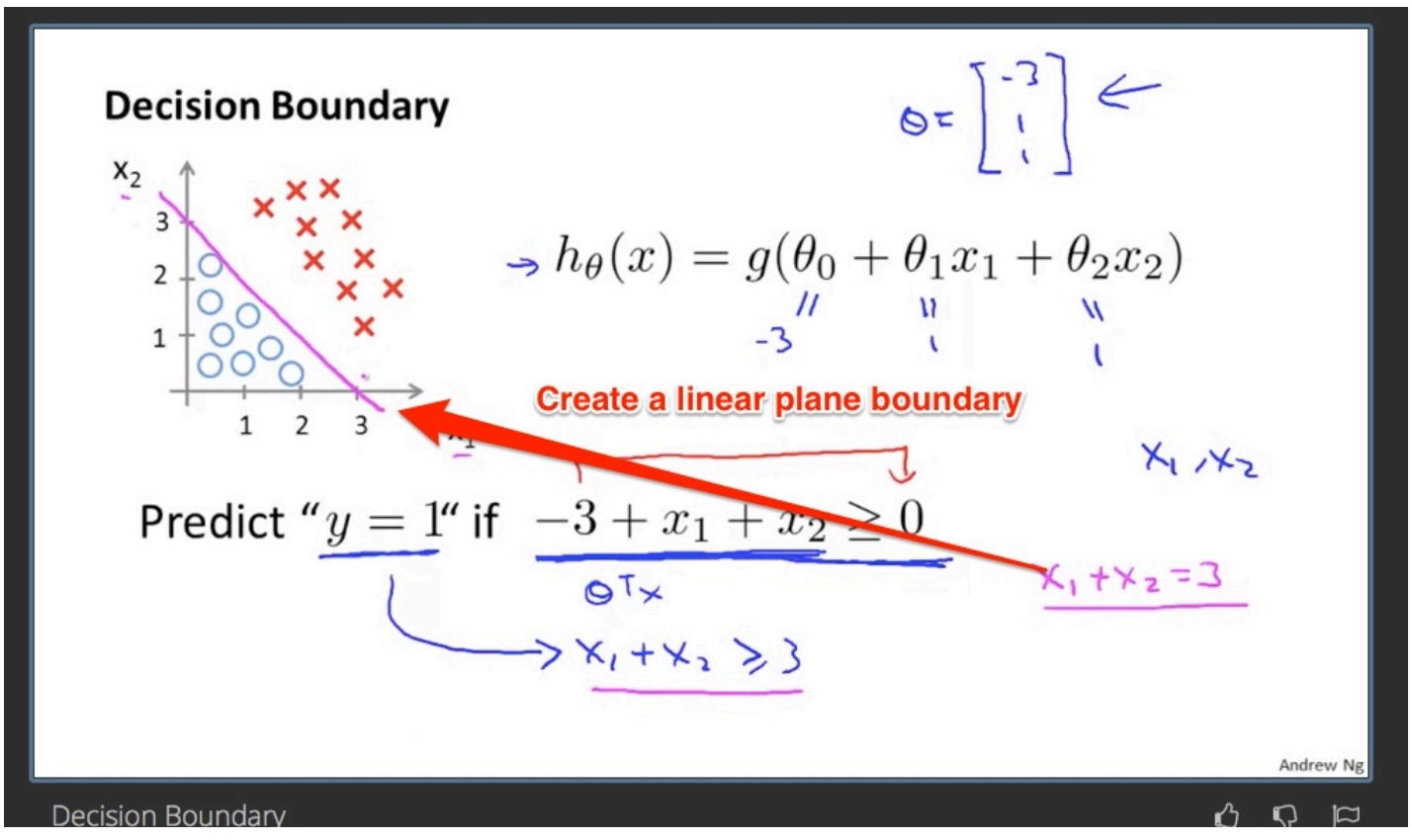
- Regression can be influenced by outliers
- Regression can predict values that don't exist in data set (0 <= y <= 1 (a binary classification), but regression predicts negative values and values > 1).
- Logistic regression 0 <= h(x) <= 1.

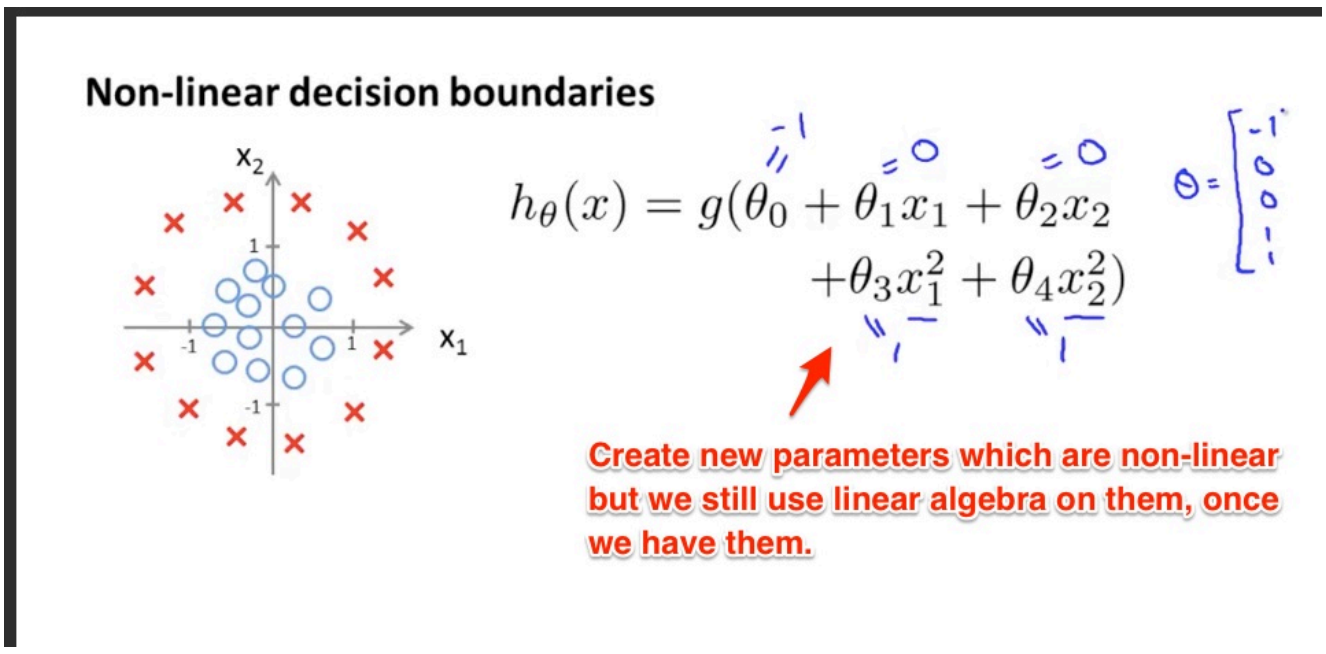
Use the sigmoid / logistic function, which is 1 / 1 + e^-z.

- Like the tangent, it caps the value between 0 and 1. Q: Could we use tangent or another "capping" function?
    - Thought: ln(x) has nice derivative of 1/x. That means later optimization is easier (setting derivative to zero).
- The bounds:
    - 1/(1 + 0) = 1 [as z increases positive]
    - 1/(1 + 1) = ½ [midpoint, z=0]
    - 1/(1 + infinity) = 0 [as z is very negative]
- h(x) is now a probability (between 0 and 1) that the input is classified as 1.
- 

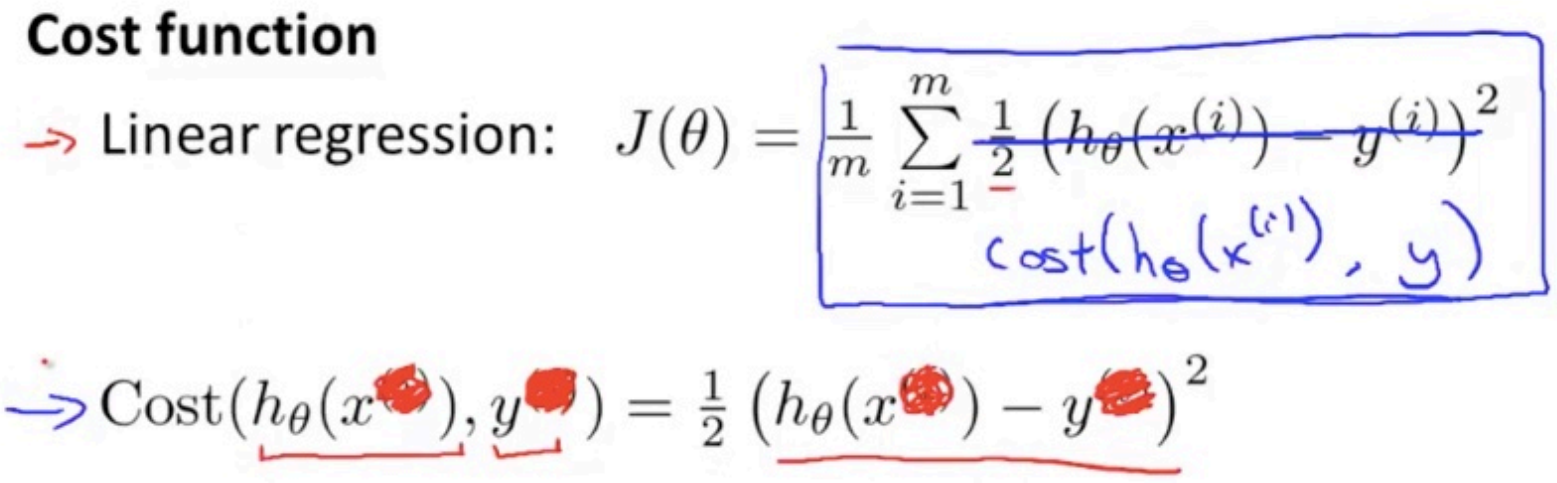


- Intuition: See theta transpose (column vector of data [by convention] turned into a function)
- If the function (theta transpose times x) is positive, we get g(z) >= .5 … because we have 1/(1 + smaller than 1)
    - We turn the broad spectrum of any positive z value into a range of 0.5 to 1.0
- Prediction: I guess we want a theta^t where [theta^t x] is positive for the values we want and negative for the others.
    - Use linear regression to make this happen? Gradient descent to find the best theta for this?

- 

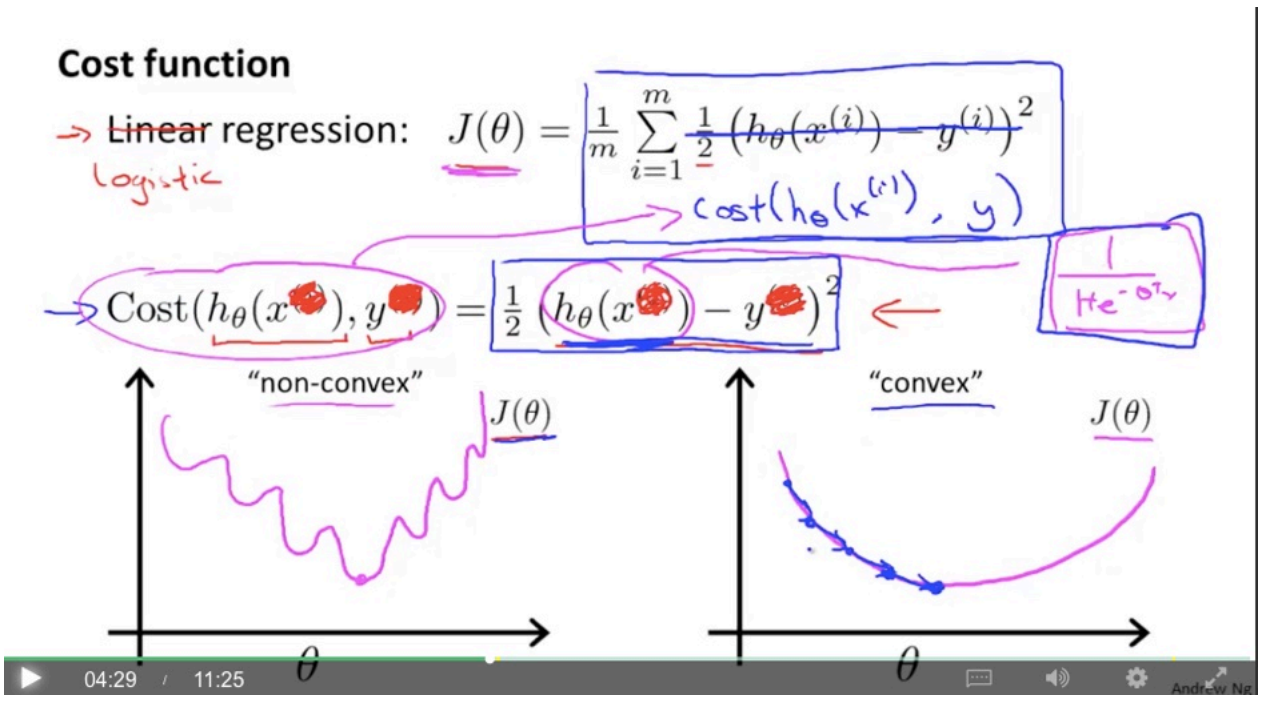


Named cost function, though I prefer "error function". Cost is what you pay for that choice of theta parameters. This is similar to the Lagrange optimization. To optimize something subject to a constraint, define an error function (difference from current value and constraint) and minimize that.

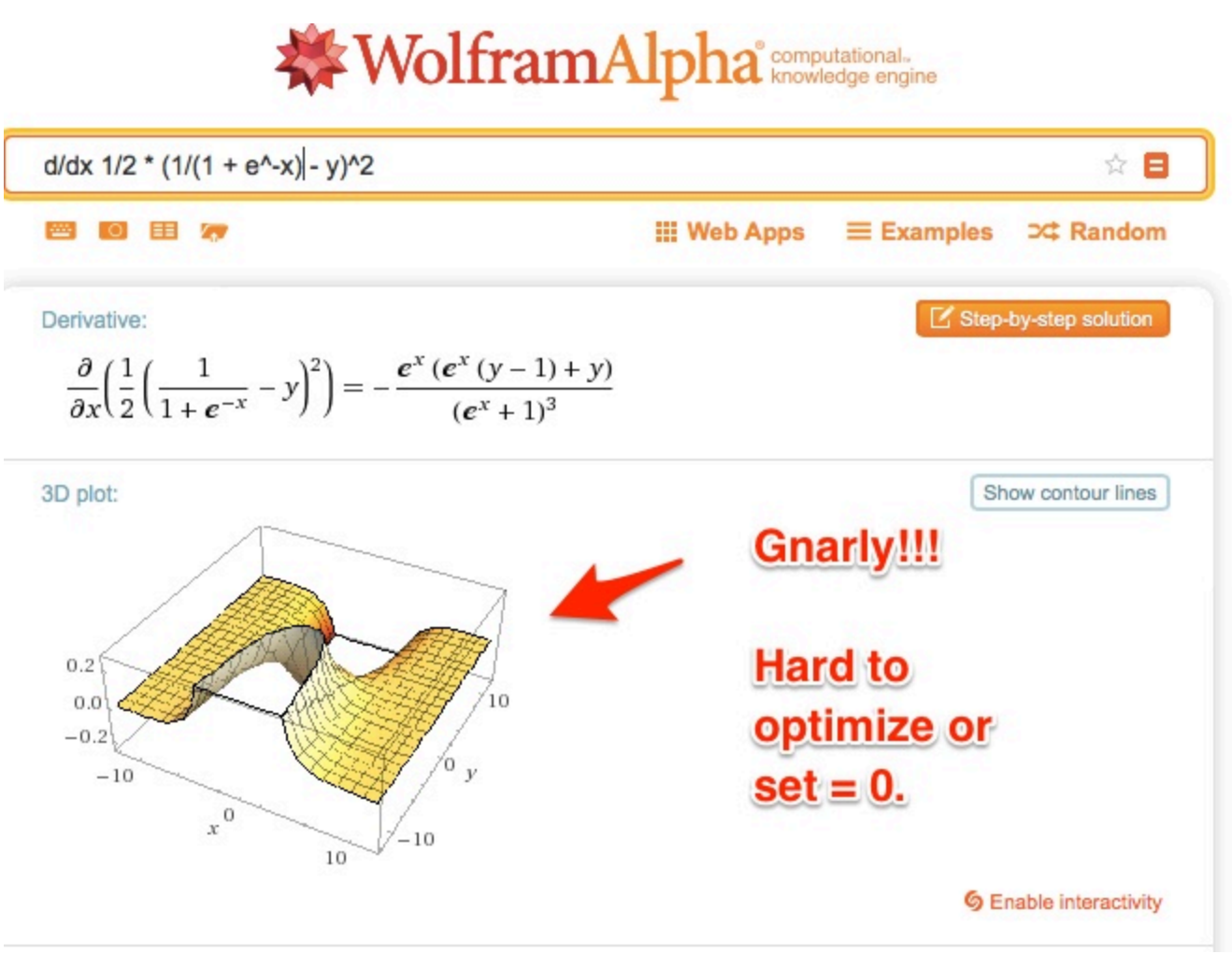


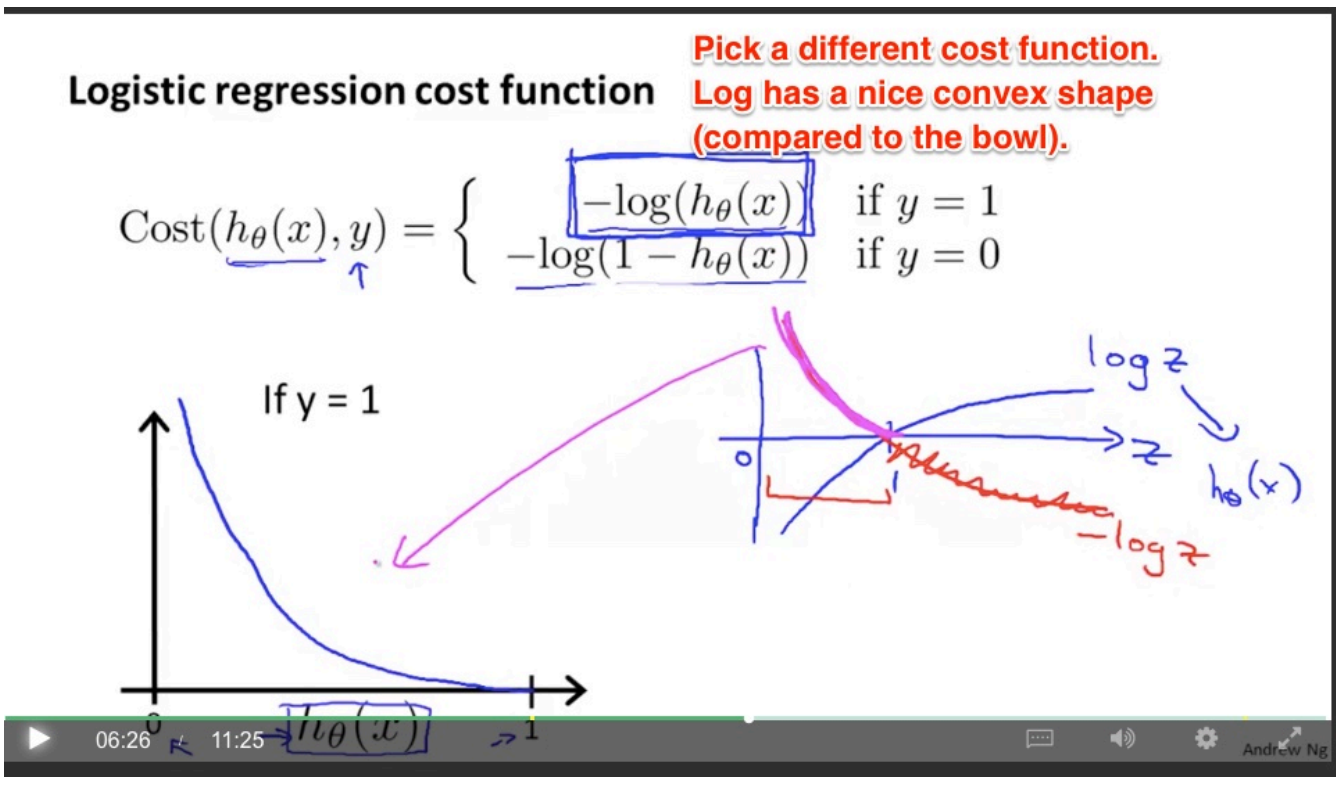
Prediction: minimize cost function, take derivative. Use chain rule. [h(x) - y] * derivative of logistic function. Use quotient rule on that…

Cost function can be non-convex… doesn't have a clean, single local min.



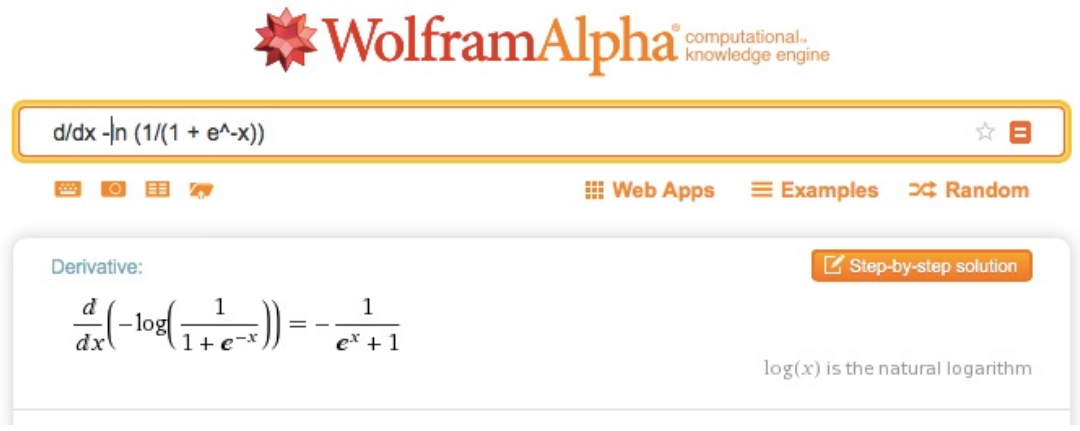
https://www.wolframalpha.com/input/?i=d%2Fdx+1%2F2+*+(1%2F(1+%2B+e%5E-x)+-+y)%5E2





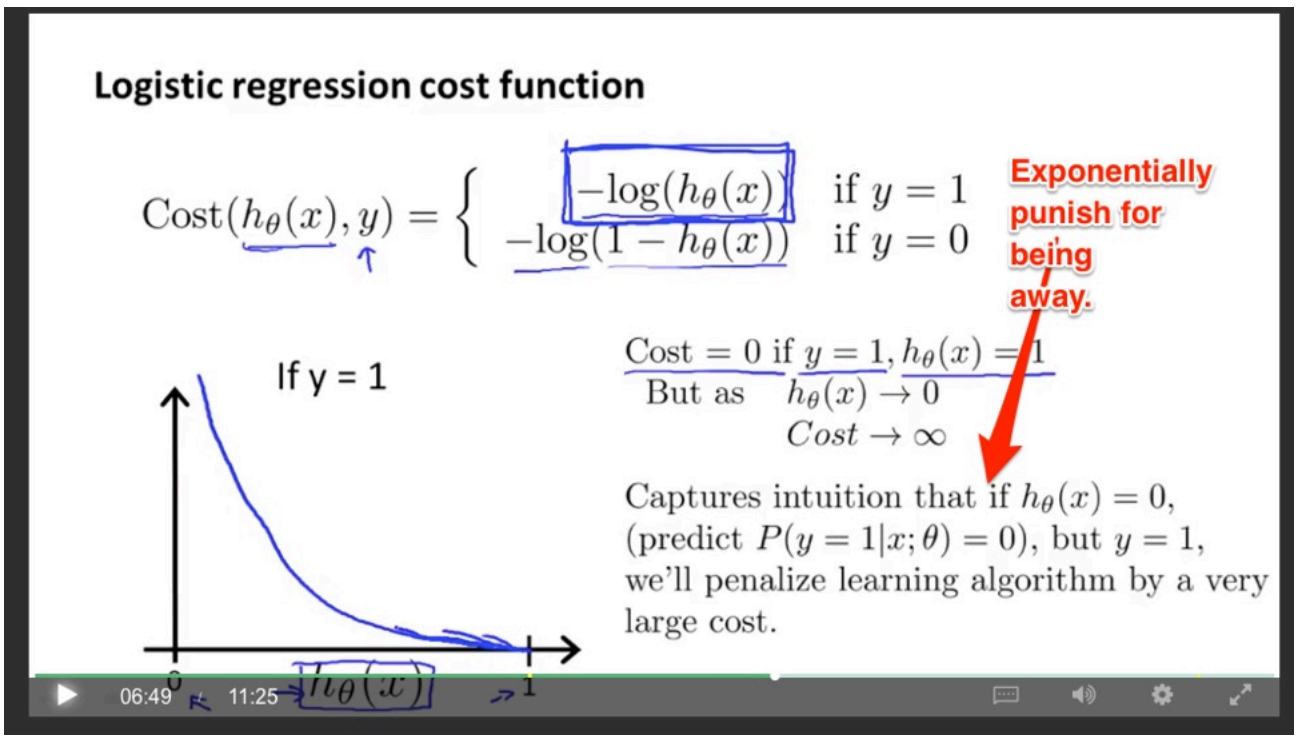
https://www.wolframalpha.com/input/?i=d%2Fdx+-ln+(1%2F(1+%2B+e%5E-x))

Q: But it will never reach 0?

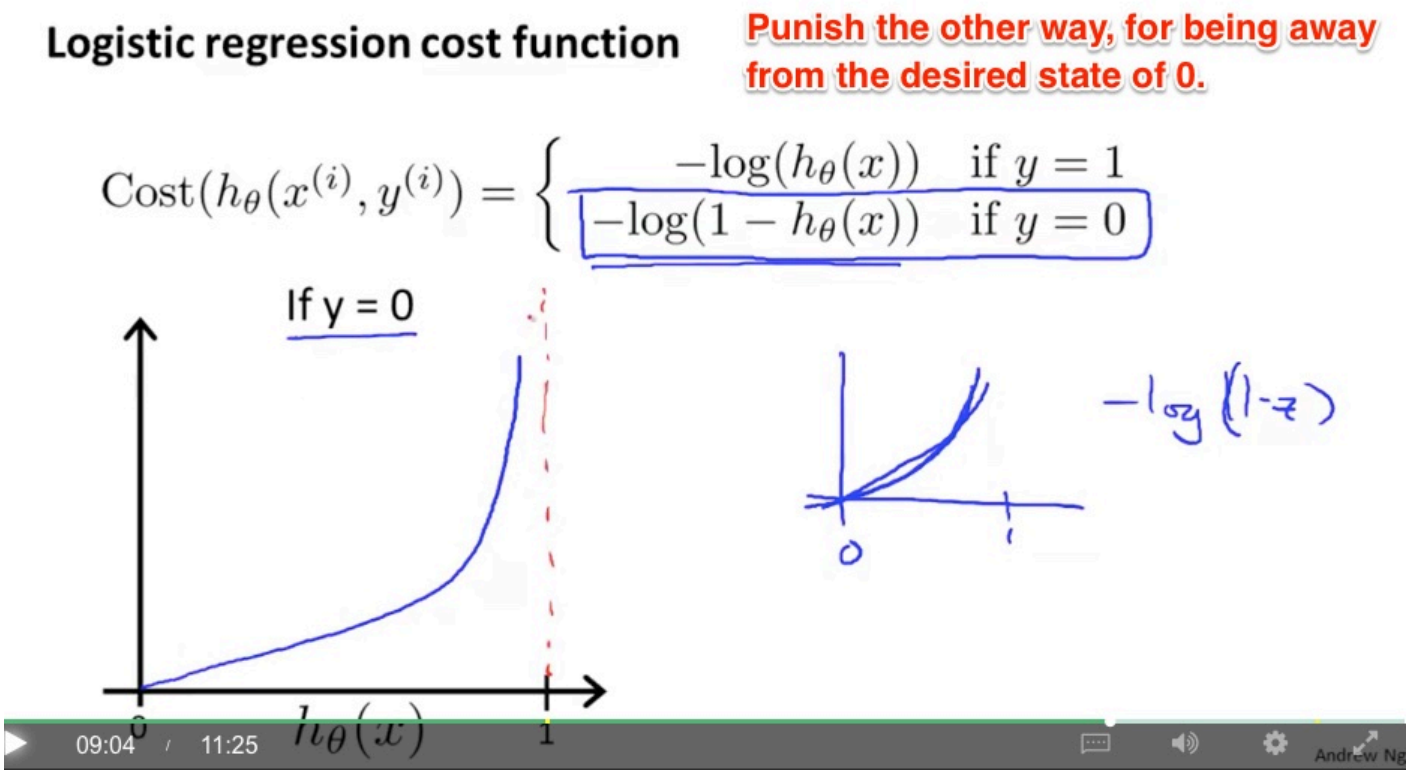


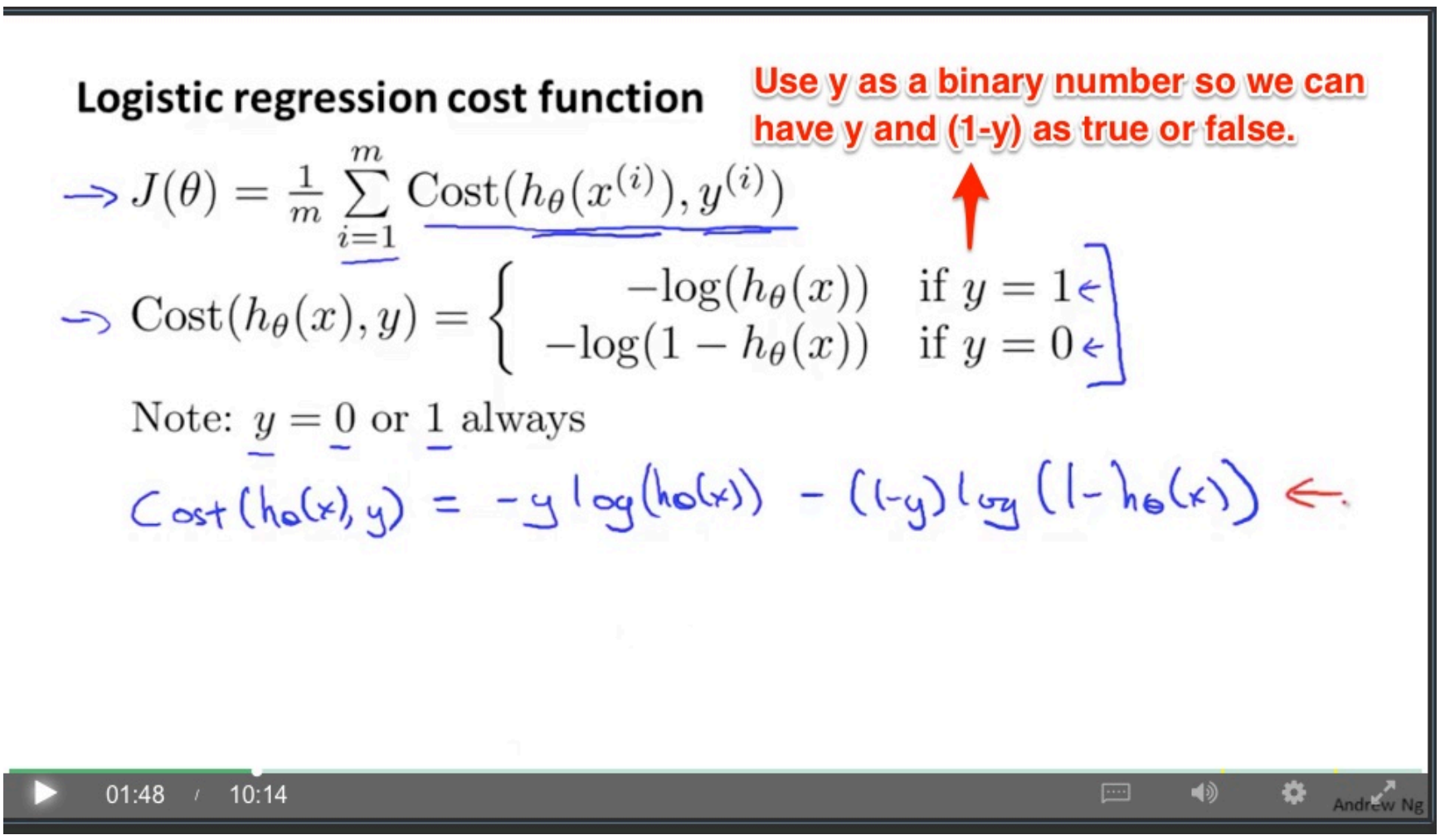
If y = 1, then h(x) should be 1 as well. log(1) = 0. (no change in time.)

Intuition: Make sure you understand how log is “time needed to grow”. No time needed to grow from 1 to 1.



We have two cost functions, designed to punish the case of y=1 and y=0 separately. Have mirror images, more or less, which punish for making the wrong prediction.

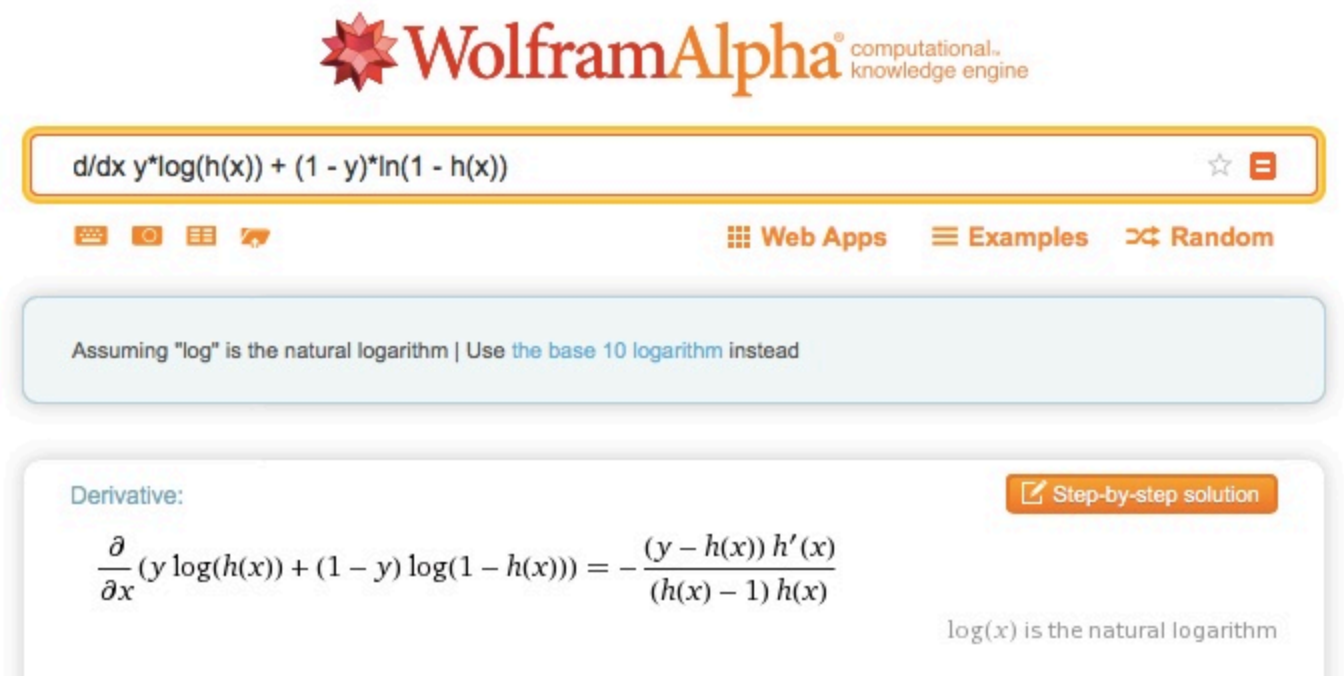




Aha: Cool technique! Convert an if statement into a SINGLE value. Just multiply it all out and you skip the if. The terms where y=1 appear and the other terms disappear as needed.

Tolearn: principle of maximum likelihood estimation

Time to minimize cost…

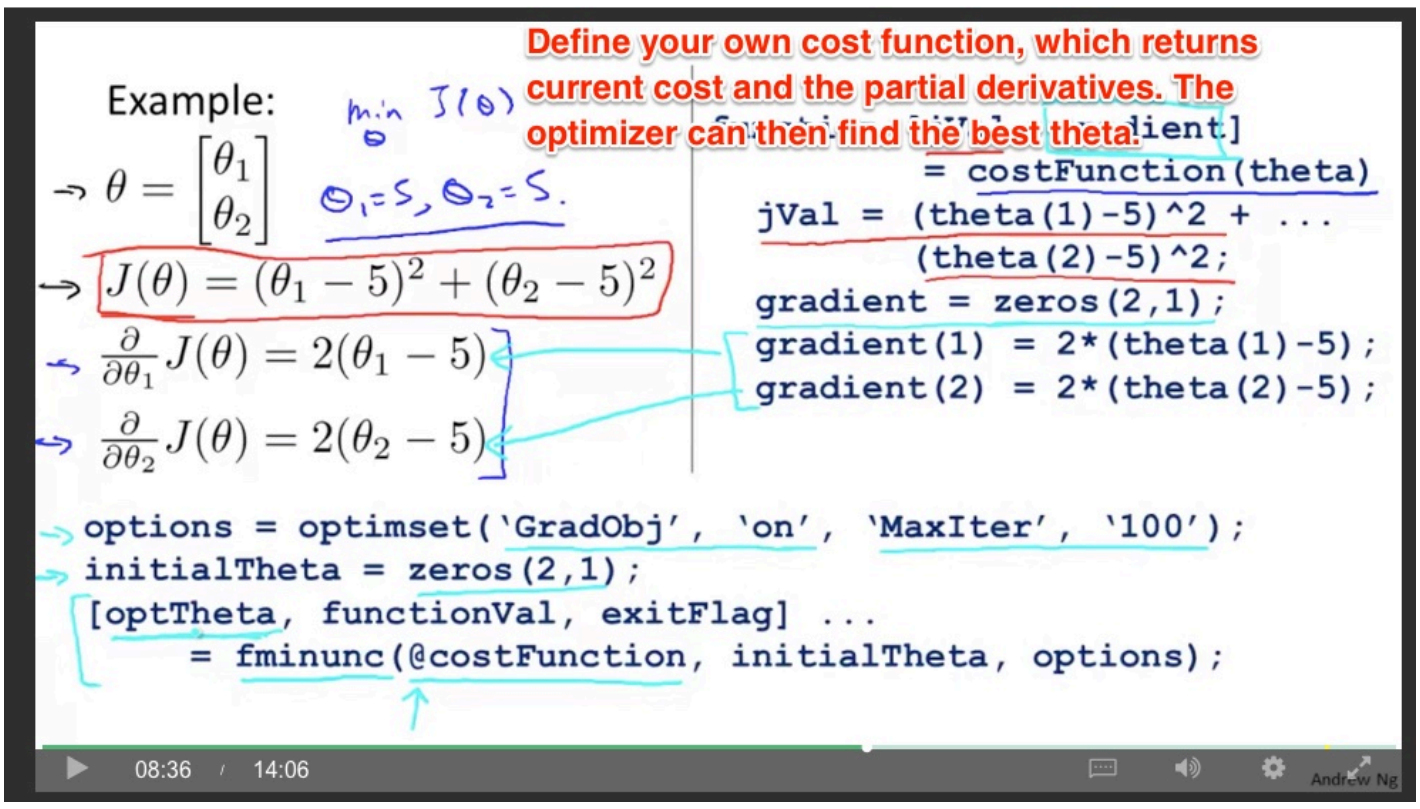


Which is basically (y - h(x)) * x [because h’(x) = x], all divided by the hypothesis term squared. Oh! Remember that we don’t have the same derivative, the hypothesis is different! TODO: do this by hand.

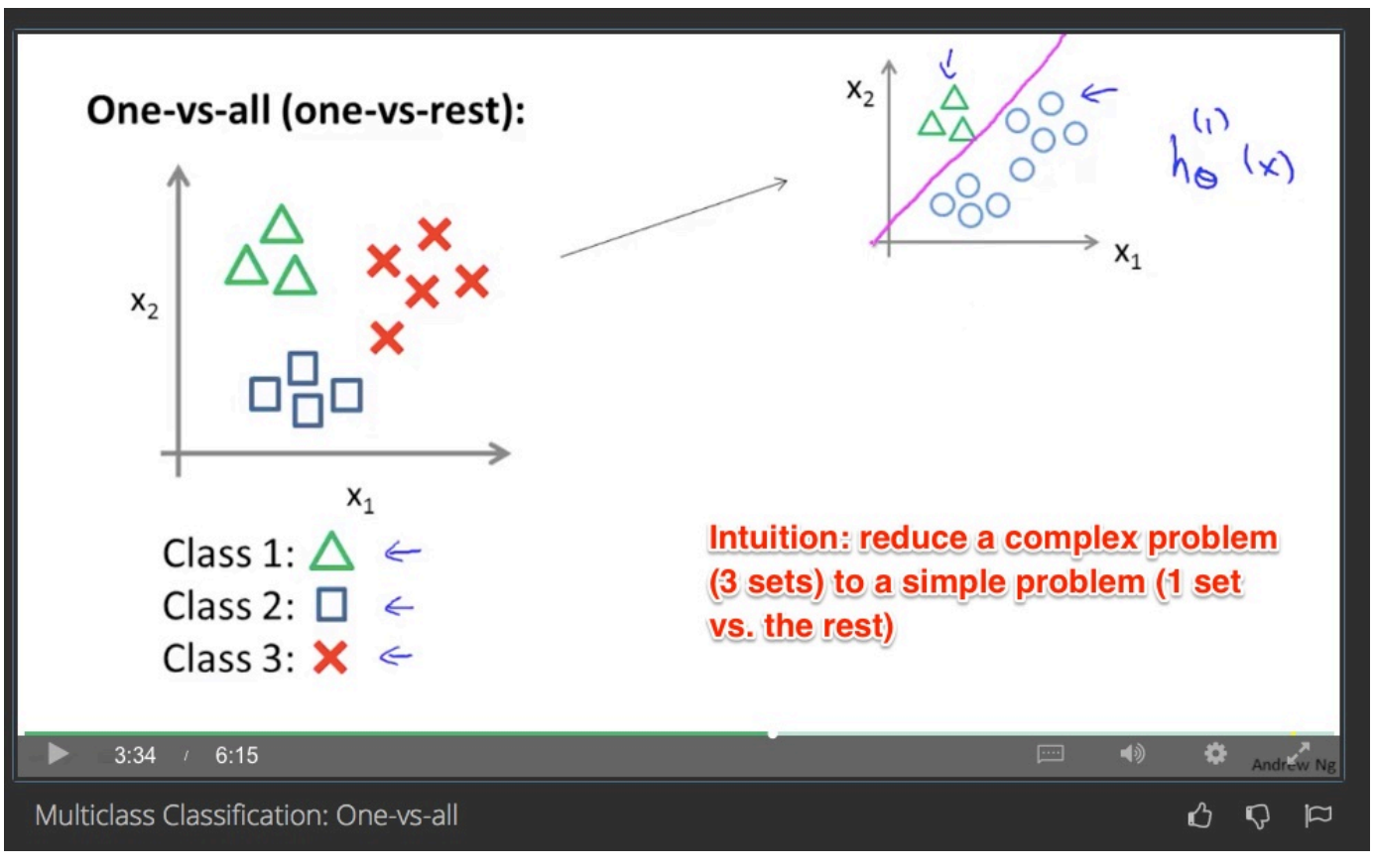
http://math.stackexchange.com/questions/477207/derivative-of-cost-function-for-logistic-regression

Insight: replace h with h(x). REMEMBER that the hypothesis is different. The format of the answer looks similar (for any convex function) but the actual cost function and hypothesis function are different.

Same update rule as linear regression. But hypothesis function is different.



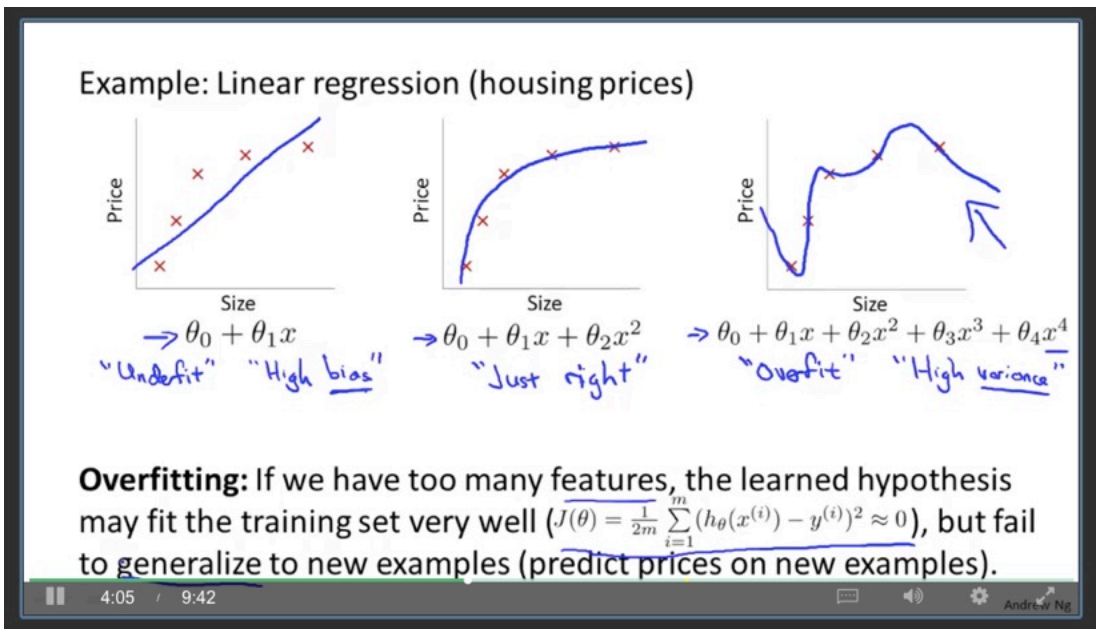
In a functional programming style, you can pass in functions (to define your cost given the current theta, and the derivatives) and it will optimize theta.

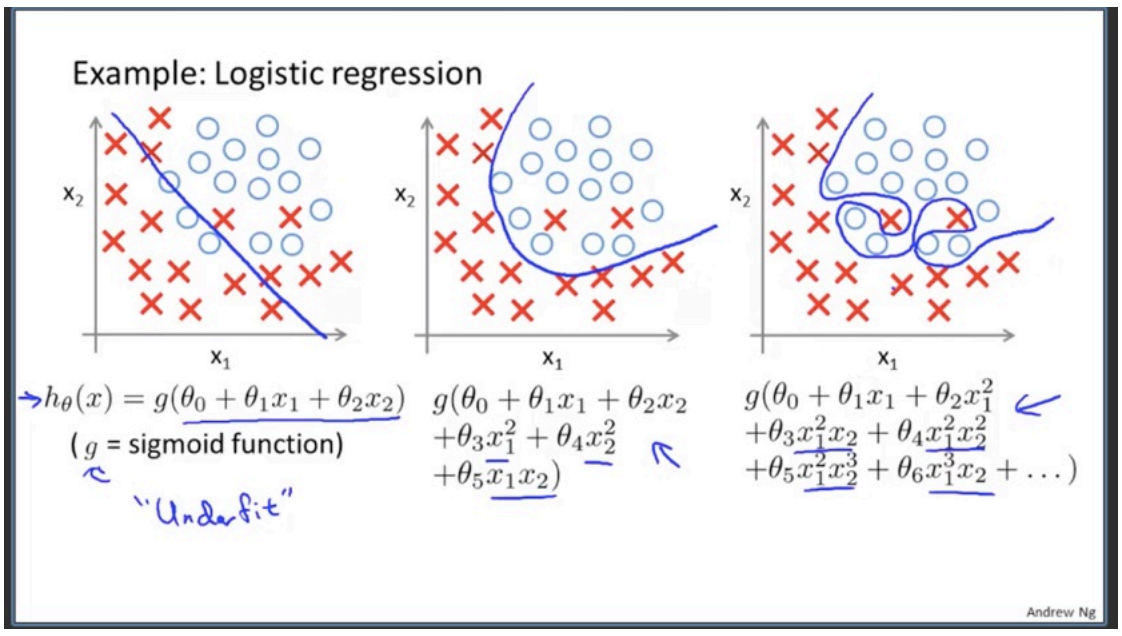


Then run all classifiers, and take the max value (highest probability).

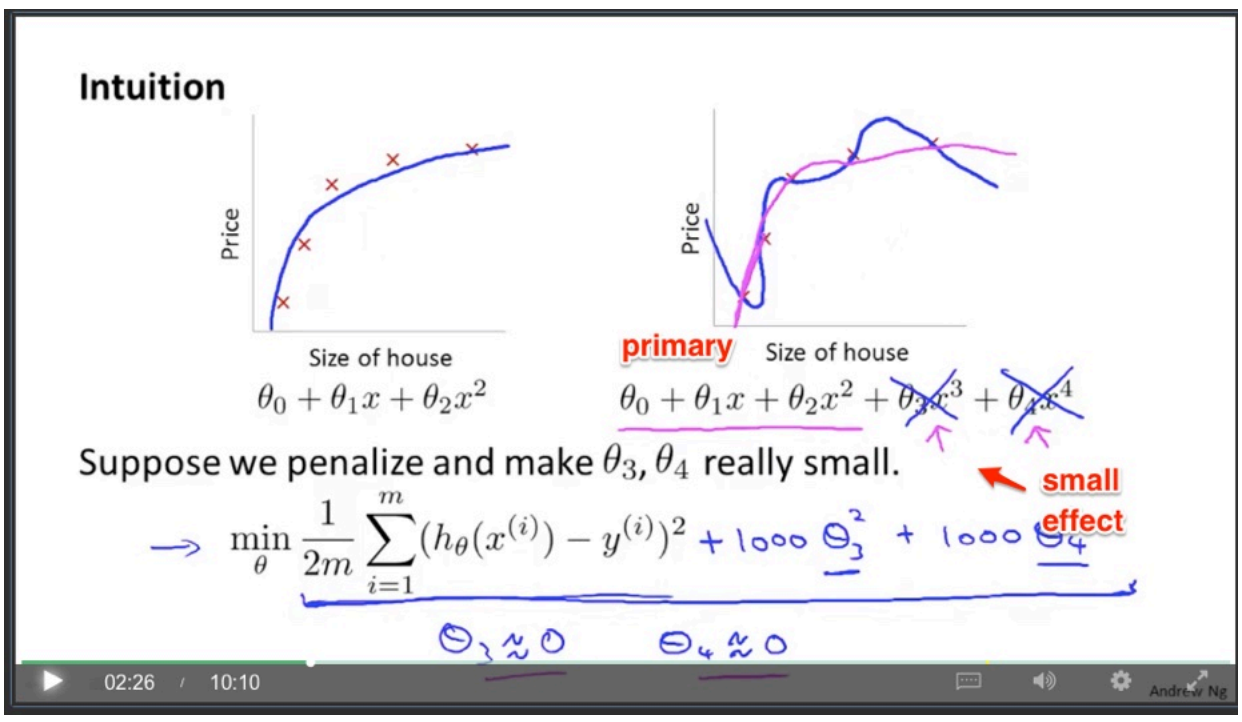
Over/underfitting

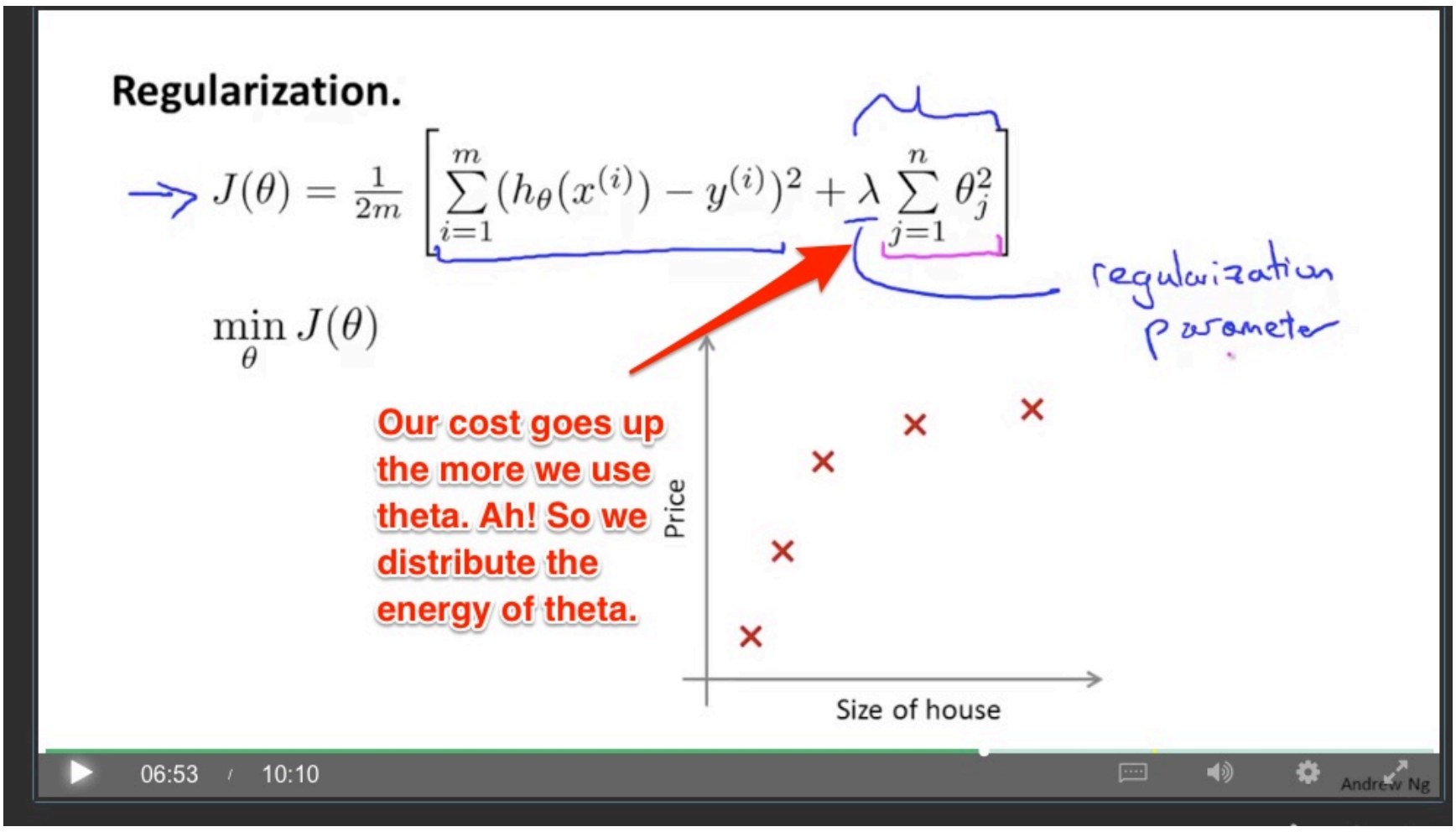
- Underfitting. “High bias” (has a preconception about what it is supposed to be like)
- Overfitting: “High variance” (too many possible hypotheses. Historical term)
- Fails to *generalize*, or cannot handle new samples well





One solution: regularization

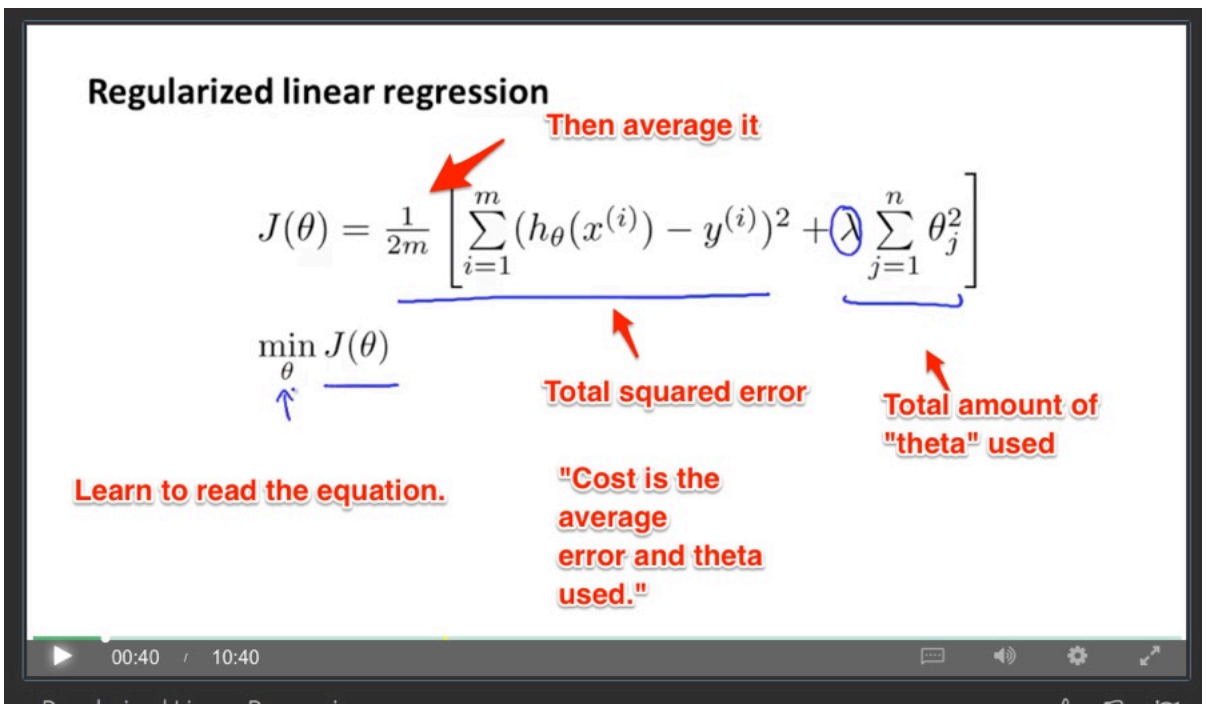
- Make the non-linear (curvy) terms very small. Therefore, they can't have much of an influence
- 



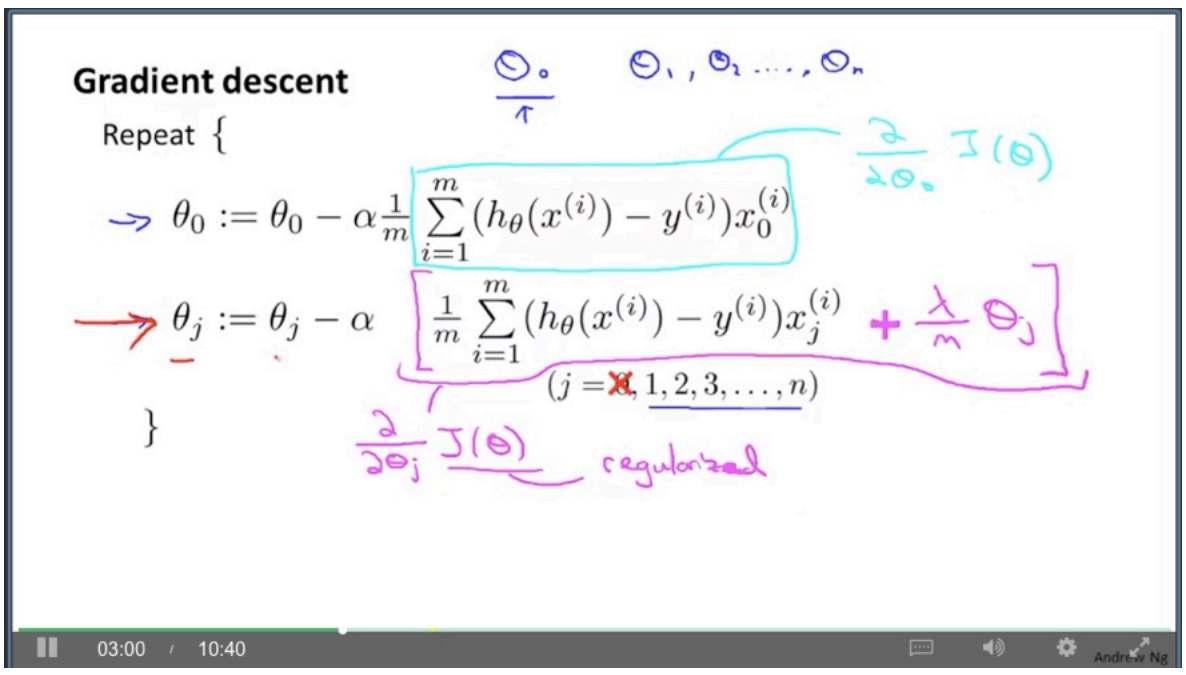
Aha: We have a certain amount of “theta energy” and we don’t want to be willy-nilly with it. Put theta where it helps the cost function the most. There is a cost to using theta itself. Neat.

- Use the smallest amount of theta possible. Best bang for the buck.
- We have a “regularization parameter” (lambda) which controls how much we weight the total theta.
    - Don’t punish theta0 (the constant term)
- Mnemonic: Think “Regulate the number of parameters used.”

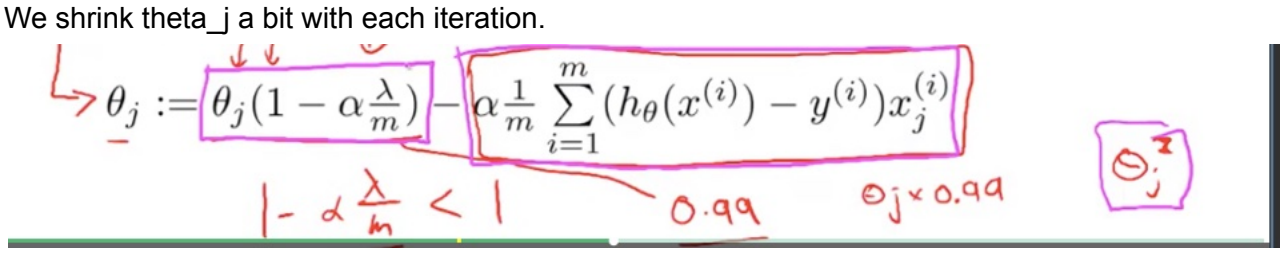
Learn to “read equations” / convert to plain english



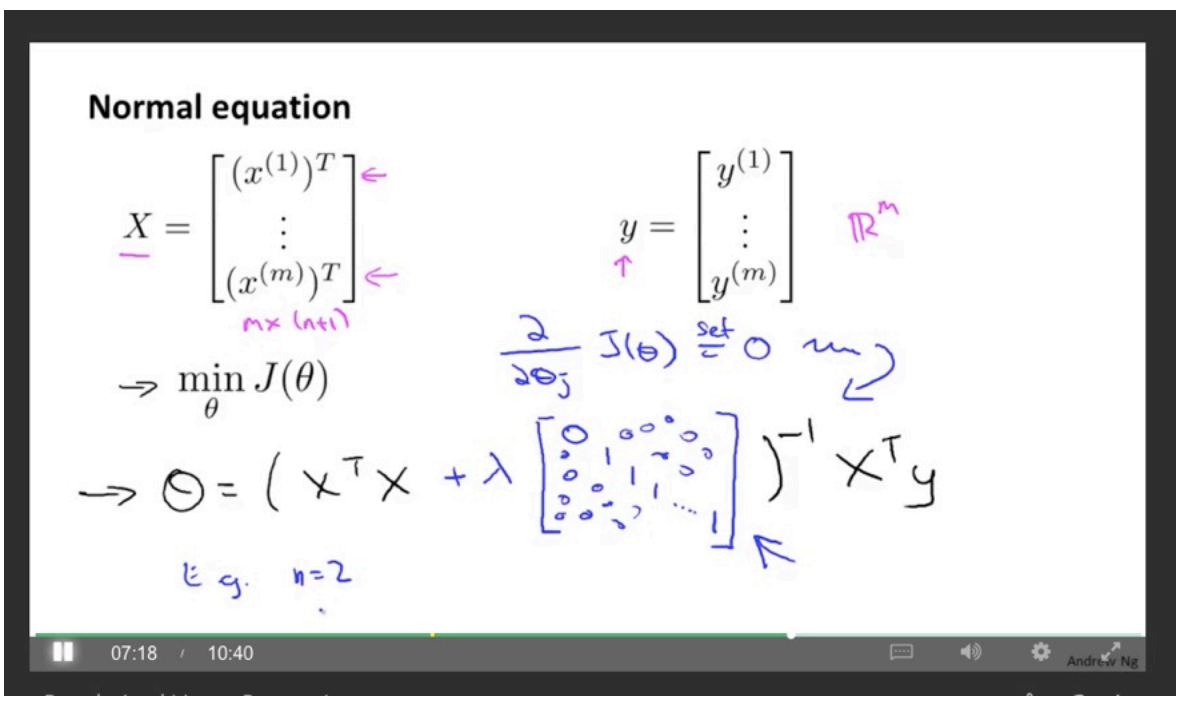
Plain English: “Cost is the average error plus theta used.”



- We shrink theta_j a bit with each iteration.

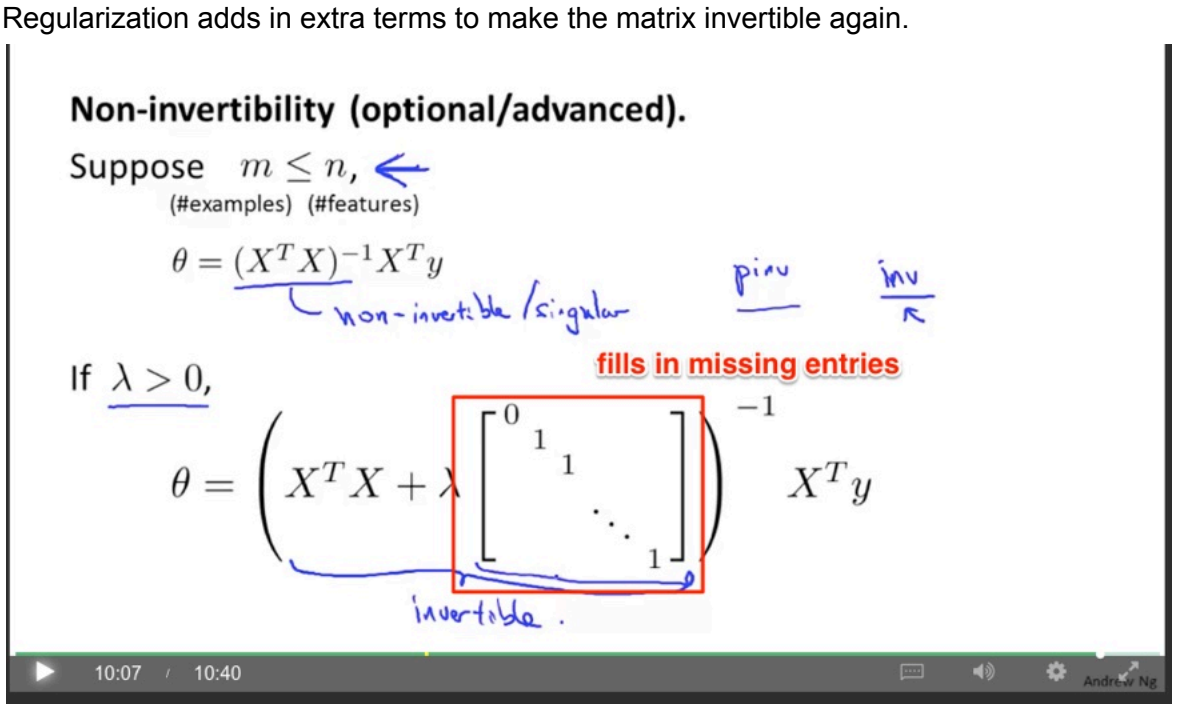


New normalization solution. You have all the partial derivatives set to zero, and you solve them simultaneously.



Non-invertible (singular) matrix: too few equations. Intuition: Missing information. m (samples) less than n (features).

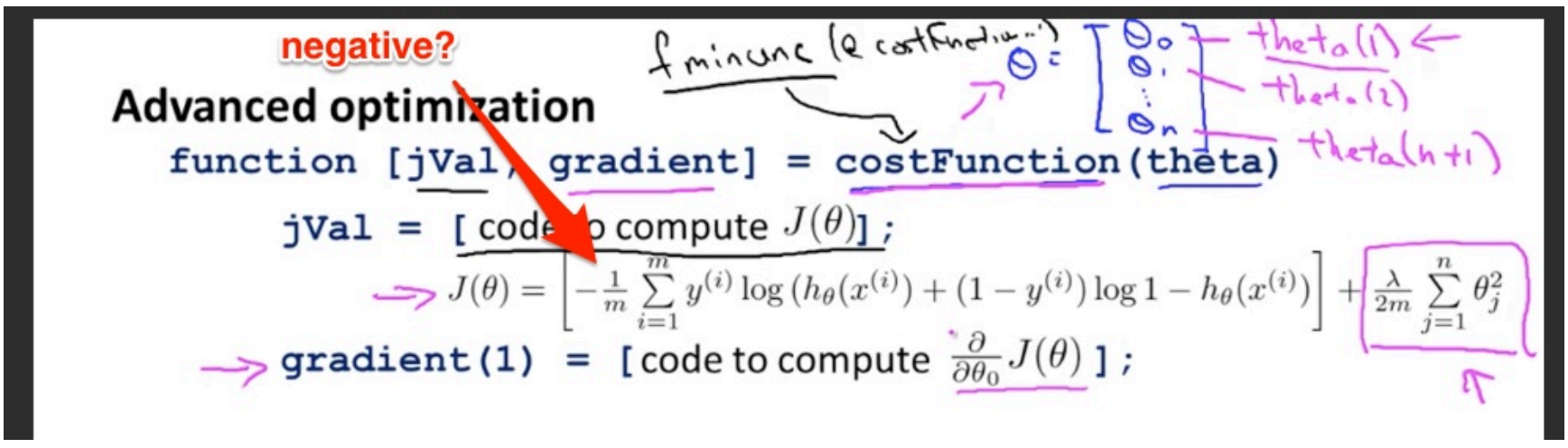
- Regularization adds in extra terms to make the matrix invertible again.



Remember: Regularization helps deal with overfitting.

Gotcha:

- Why is cost function negative in the first term? Did I forget the derivation?



- Aha: It's the way it's written. I'd prefer -log(...) + -log(…) vs. having the negative sign out front. Log of a value less than 1 is negative ("going backwards in time"), so this makes it positive again. Confusing to see a negative cost out front.

Homework

```
%  Set options for fminunc
options = optimset('GradObj', 'on', 'MaxIter', 400);

%  Run fminunc to obtain the optimal theta
%  This function will return theta and the cost
[theta, cost] = ...
    fminunc(@(t)(costFunction(t, X, y)), initial_theta, options);
```

In this code snippet, we first defined the options to be used with **fminunc**.

we are passed theta, hardcode in X and y

TODO: Why is this the boundary line?

```
hold on

if size(X, 2) <= 3
    % Only need 2 points to define a line, so choose two endpoints
    plot_x = [min(X(:,2))-2,  max(X(:,2))+2];

    % Calculate the decision boundary line
    plot_y = (-1./theta(3)).*(theta(2).*plot_x + theta(1));

    % Plot, and adjust axes for better viewing
    plot(plot_x, plot_y)

    % Legend, specific for the exercise
    legend('Admitted', 'Not admitted', 'Decision Boundary')
```

1.2.4 Evaluating logistic regression

After learning the parameters, you can use the model to predict whether a particular student will be admitted. For a student with an Exam 1 score of 45 and an Exam 2 score of 85, you should expect to see an admission probability of 0.776.

Checks out:

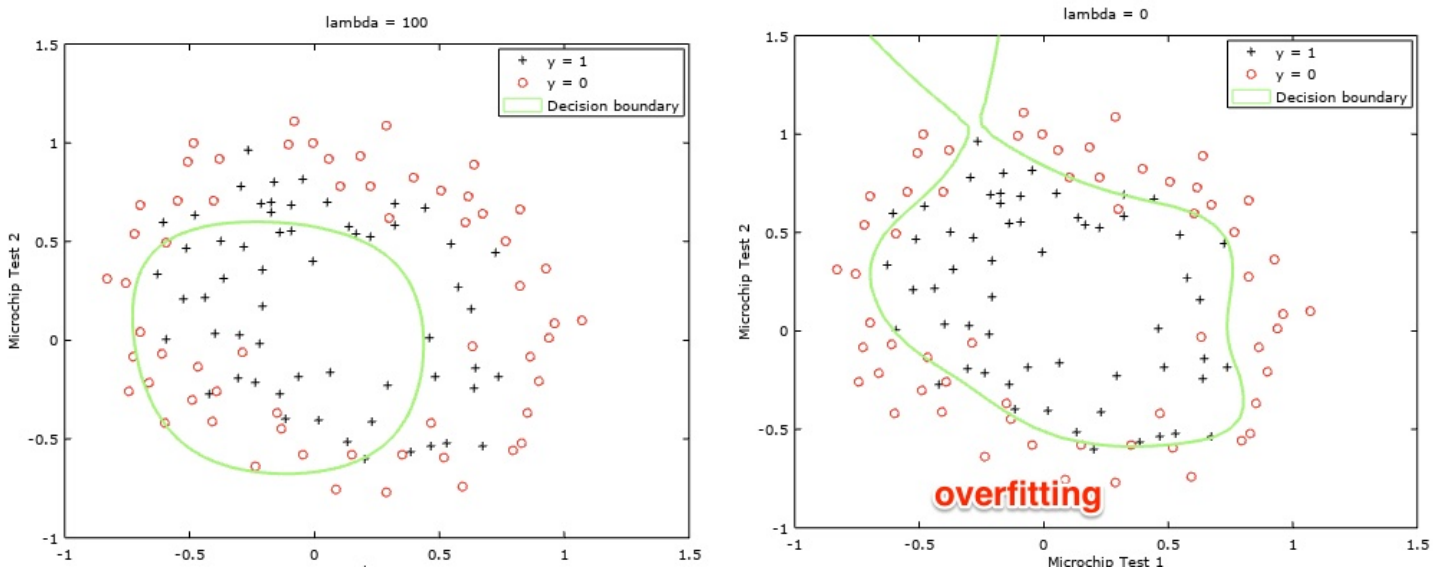
```
>> [1 45 85] * theta
ans =  1.2442
>> sigmoid(1.244)
ans =  0.77626
>>
```

Don't visualize 28 dimensions. Just think of a larger data set.
A function that takes more parameters (28).

$$\text{mapFeature}(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix}$$

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 28-dimensional vector. A logistic

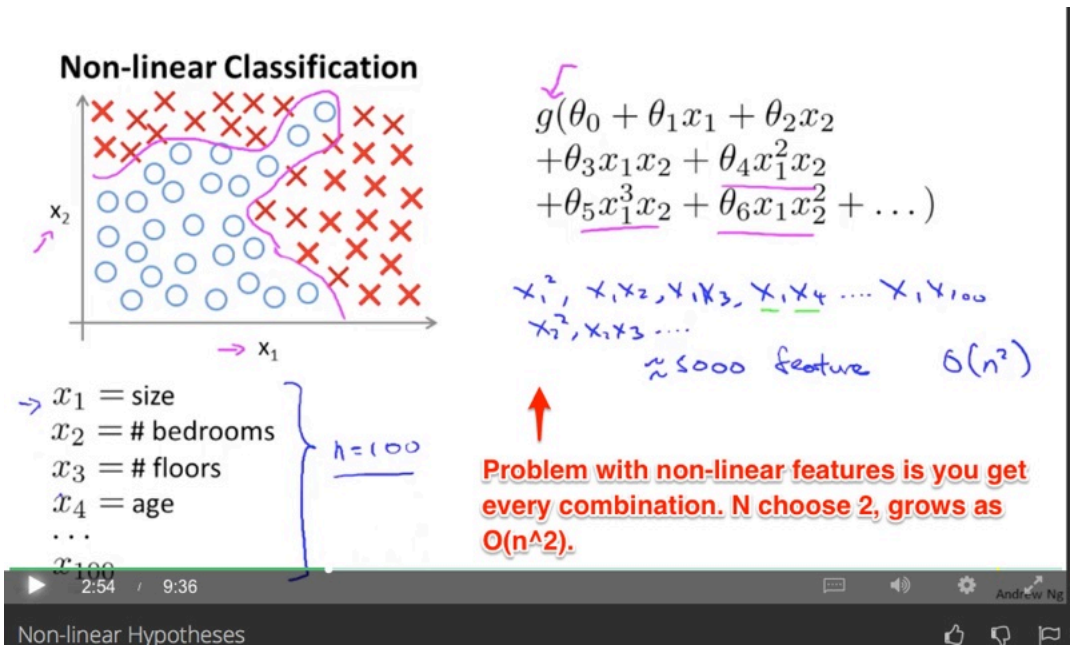
As you increase lambda (regularization factor), you penalize more complex shapes. So you can lose accuracy. Have to find the underfitting/overfitting middle ground.



Gotcha: dumb error, was doing lamba / 2 * m instead of (lambda) / (2 * m) in my equation. Be careful converting math equations to matlab/octave, get too comfortable.

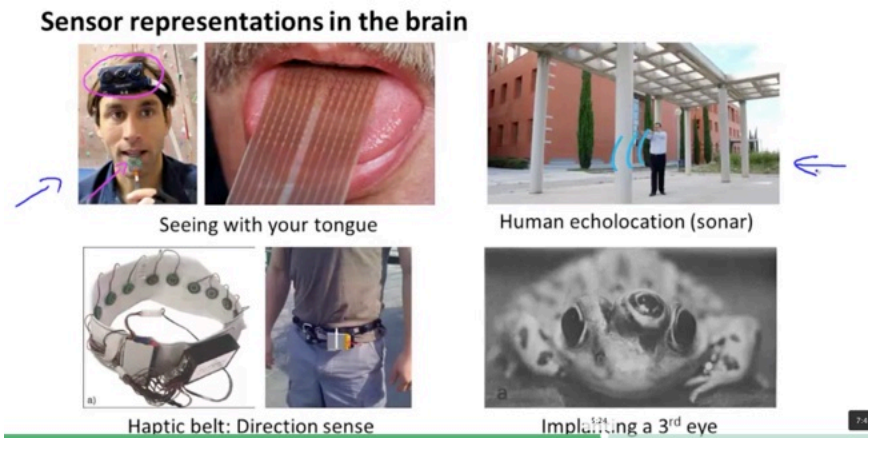
Week 4 - Neural Networks (Representation)

Issue: number of possible quadratic features grows as O(n^2). N choose 2. 100 features, ~5000 choices (100 * 99 / 2).

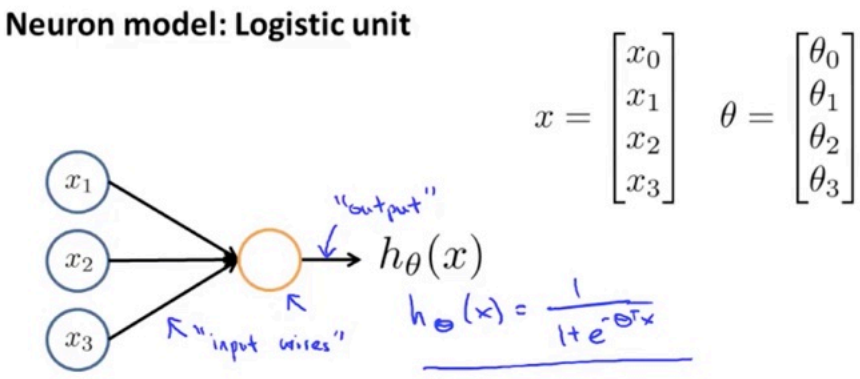


- Too many features leads to overfitting. Computationally expensive. Also, quadratic alone gives you oval-shaped features.
- Cubic features get more interesting, but again, tons of combinations. O(n^3)
    - Computer vision: each pixel is a possible feature (the intensity of that pixel in a 50x50 grid).

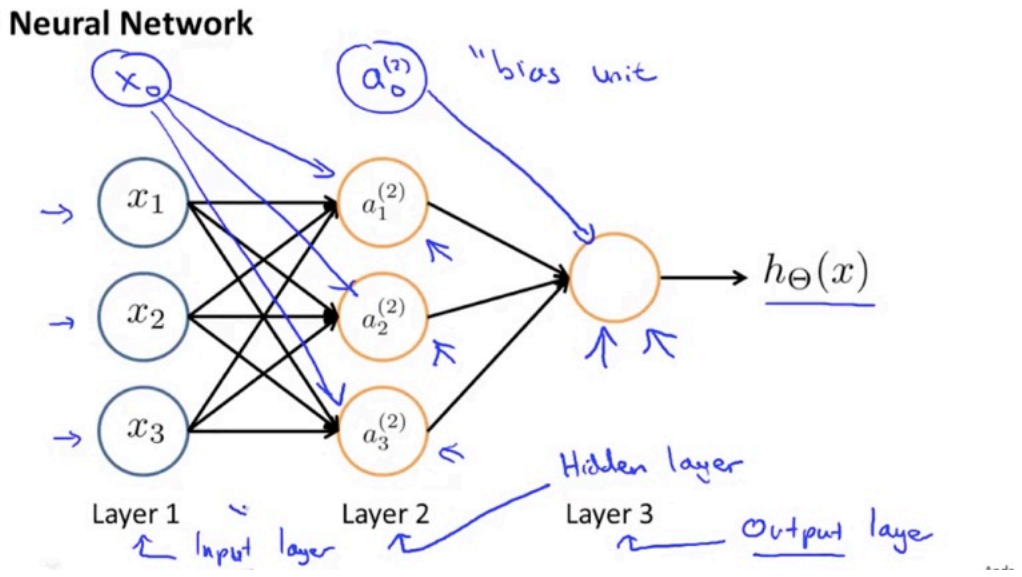
Prediction: Neural nets will help manage number of features by having layers to represent each variable (“100 * 99”), vs. the result (5000).

- “One learning algorithm” hypothesis. Same piece of brain tissue can process sound, touch, etc. (Adapts even after synapses cut. Learning to see with your tongue.)
- 

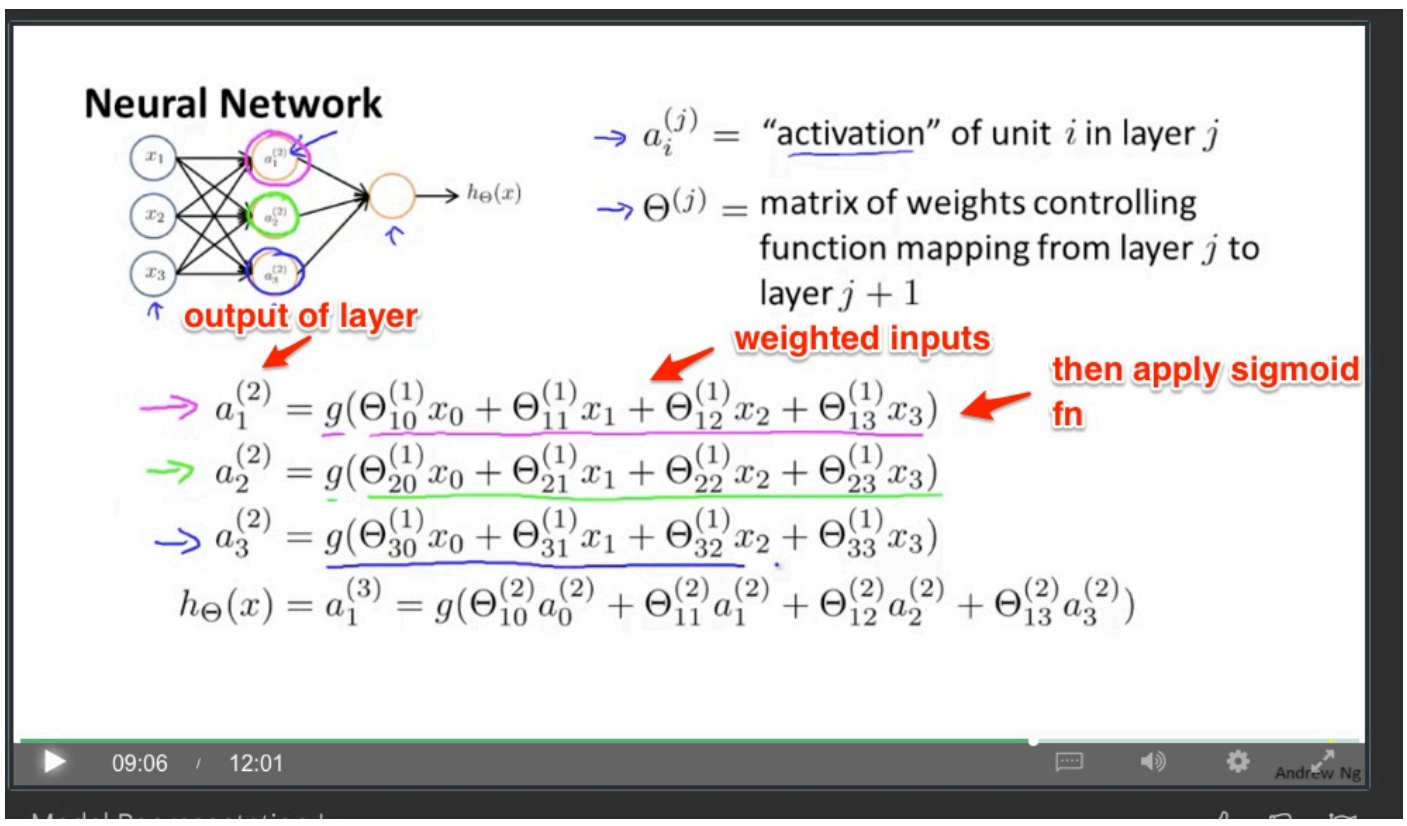
- Reverse engineer brain’s algorithm.
    - Neuron a set of input wires and output wires. Does computation in-between.
    - Aside: we are finding an analogy and using that to discover new things. Ultimately, a neural network is just another data structure.



-



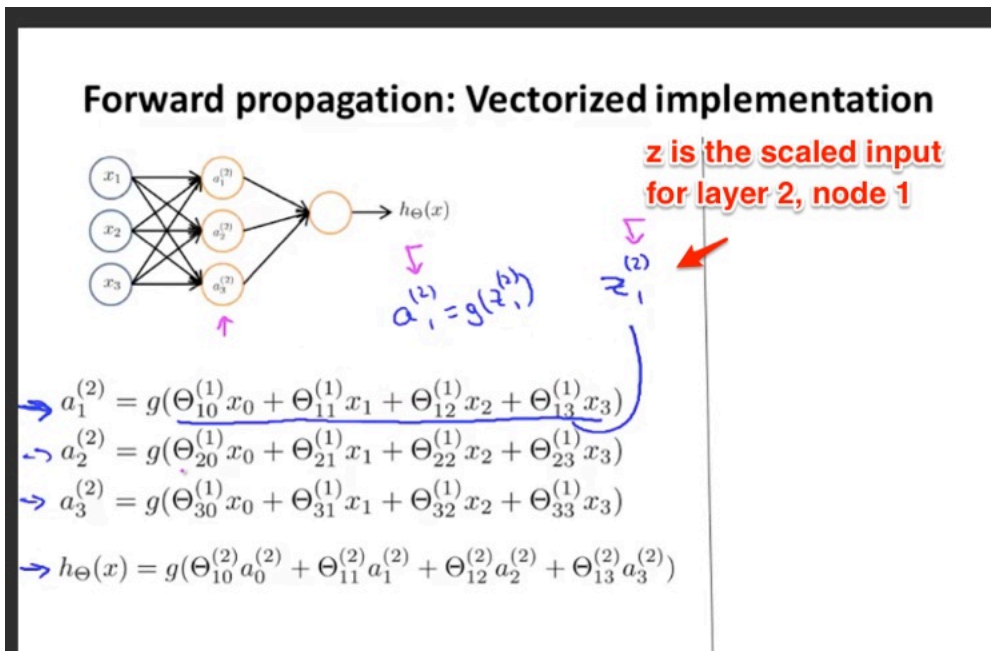
- Vocab
  - Activation function (what computation is happening in the neuron)
    - a_i^j
    - Remember, the superscript is more general (j), and the data put in is more specific (the subscript). Which is which seems to trip me up.
  - Weights (aka theta parameters that scale the input values)
  - Sometimes an implicit x_0 bias (default setting)
  - Input layer (seen), output layer (last one), intermediate hidden layers



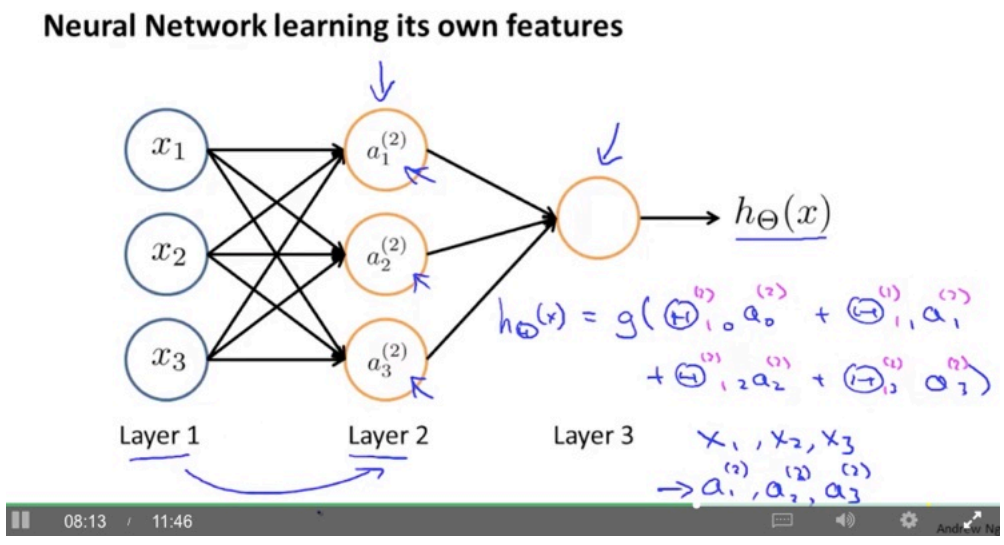
  - Gotcha: Each activation node can have its own weights. So Theta becomes a matrix (not a vector to scale the input).
    - In the case above, 3x4 matrix (3 activation layers, each takes 4 parameters).
    - And, of course, we have a hidden parameter (Theta 0). So theta matrix gets an additional entry.

Use subscripts / superscripts and abbreviations to simplify the ridiculous number of variables.

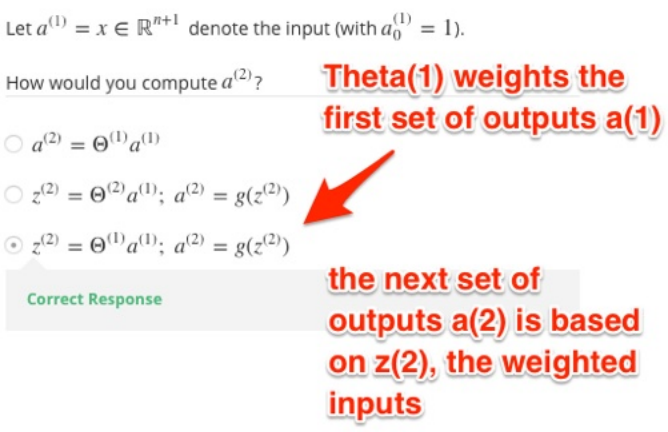
- Aha: this where linear algebra shines. How else do you handle dozens or hundreds of variables? Need arrays. “Regular math” limits you.



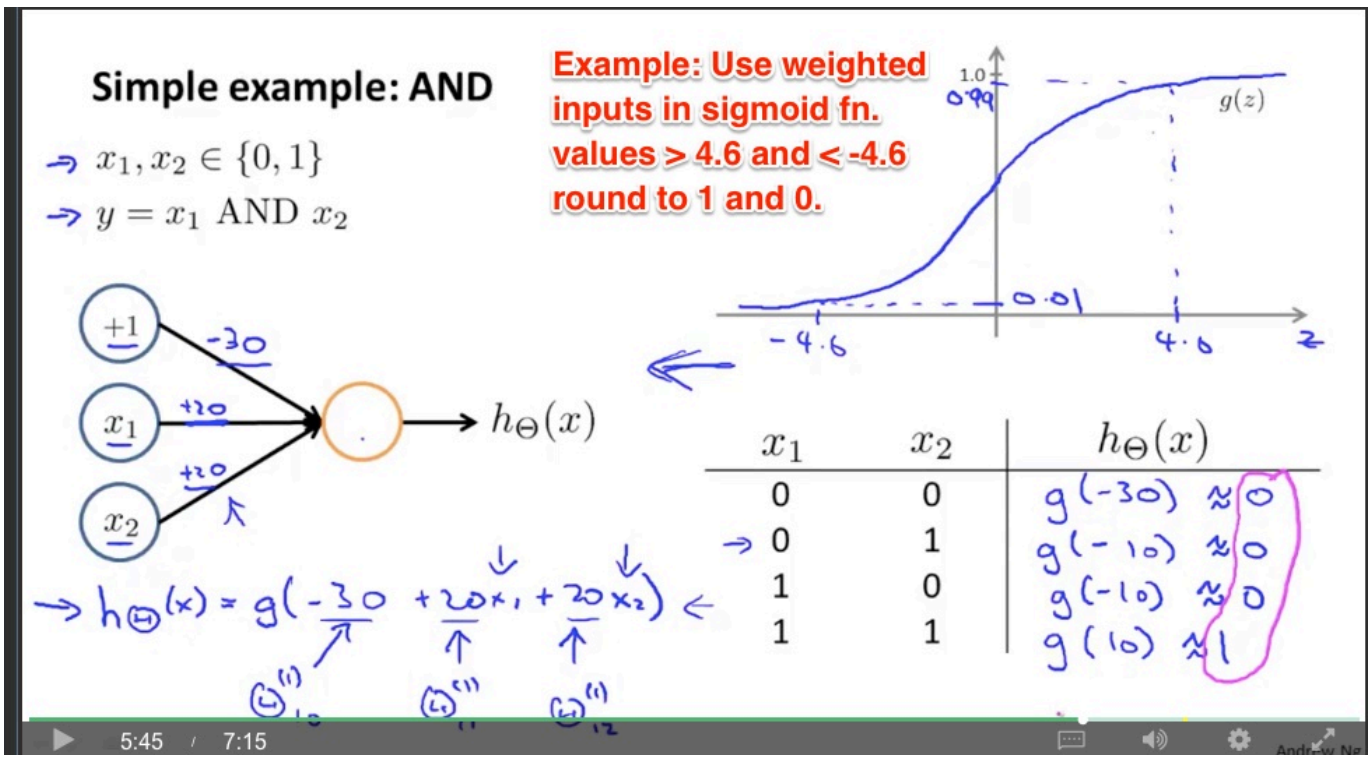
  - z-values are weighted linear combination of inputs
  - see the first layer as activations (just inputs). $a^{(1)}$
- Again, to avoid confusion, think of “inputs” and “outputs” in the matrix. Just need them to match up after each layer.
- Forward propagation - have inputs, weight them, push them through to the next layer.
- Can learn the features (similar to regression). Prediction: Describe function of entire network, take derivative, use gradient descent, modify weights



  - Again, watch out for the trickiness with all the superscripts, subscripts, etc.
- OK, we can do more complex features. Can we build x1 * x1 as a feature?
  - Level 1: x1 (becomes input)
  - Level 2: x1 is …
  - Aha: If we just have a linear activation function, we are just using matrices. http://stackoverflow.com/questions/9782071/why-must-a-nonlinear-activation-function-be-used-in-a-backpropagation-neural-net
  - Activation function must be non-linear in order to be interesting. So, the activation network itself could be square.
  - x1 comes in, (x1)^2 comes out. Or some other arrangement. Activation functions don’t have to
  - Aha: the goal is to use matrixes to represent sets of inputs, but have *non-linear* functions inside the matrix that do the work. Use matrices for what they are good at (sets of data and operations) and non-linear functions for what they are good at (complex behavior)
  - In a similar way, the activation function could just multiply its inputs (x1 * x2). Each node would represent a different set of multiplications.

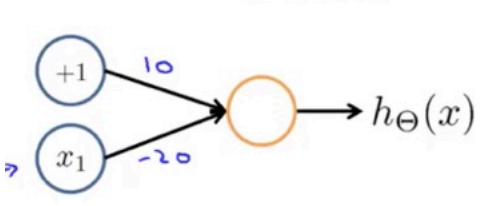


-
  - *a* - outputs. a(1), first set of outputs
  - theta(1) - first set of weights
    - These are combined to give z(2), the weighted inputs to layer 2
  - a(2) - outputs of layer 2, after activation function is run [called g].
    - Yep, so use “a” as the result of the activation function on the weighted inputs.



- Intuition for the above: only when x1 and x2 are present will it overcome the bias of -30.
- Can build the other logical operations (AND, OR, NOT) using sigmoid

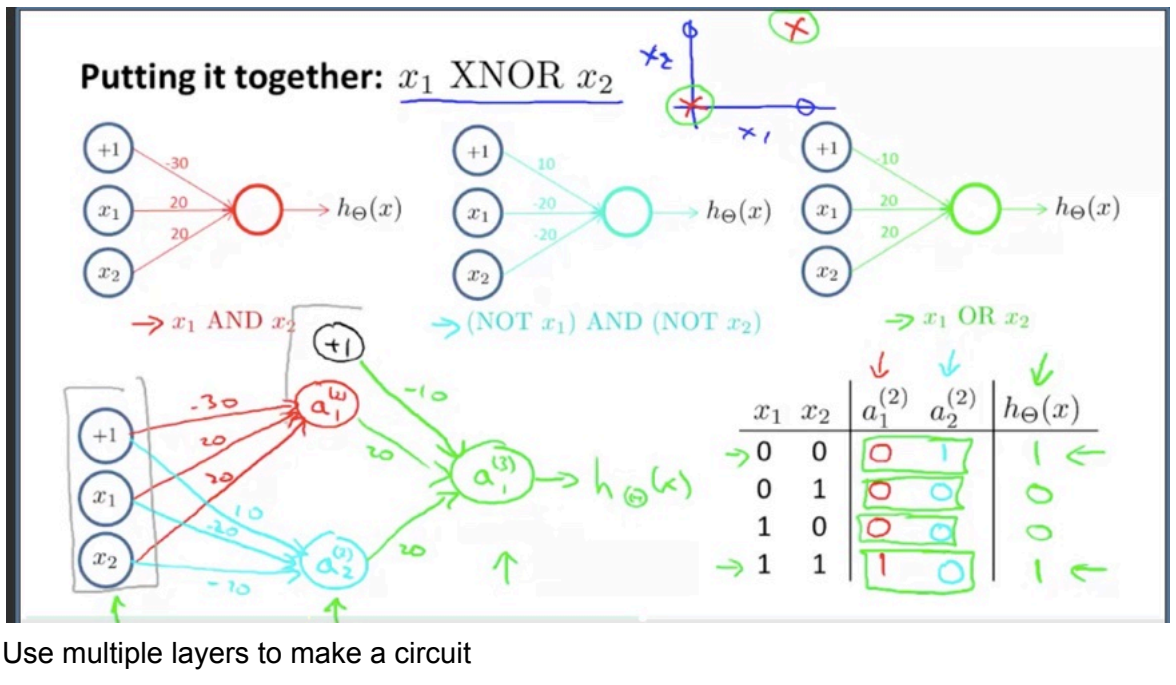
Negation:

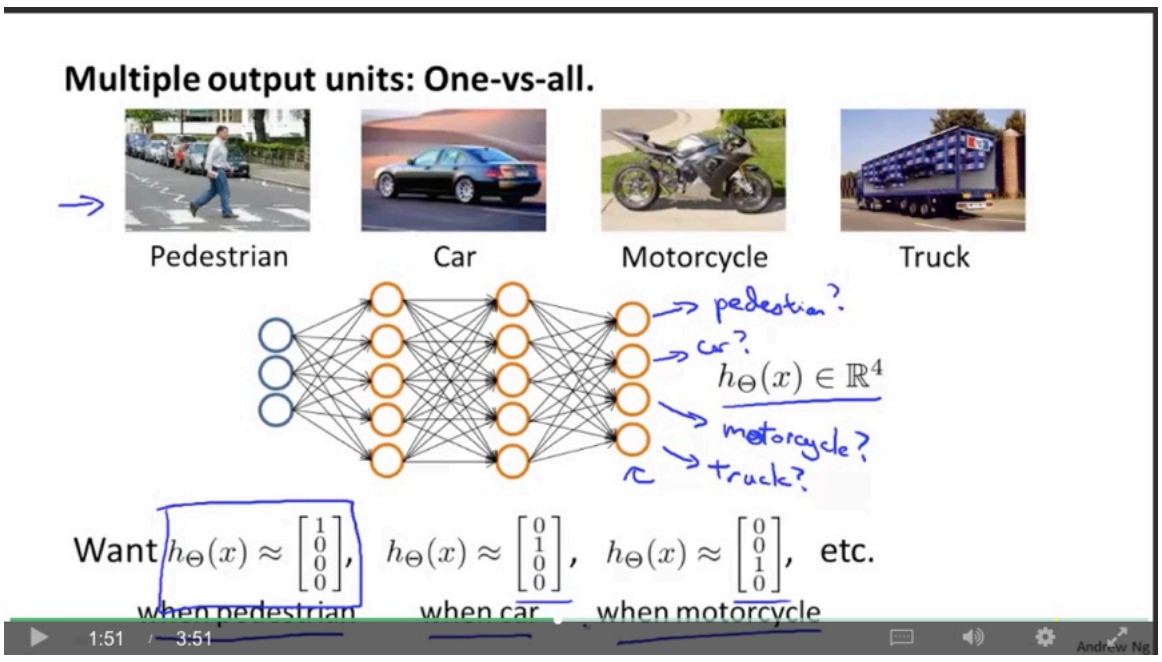


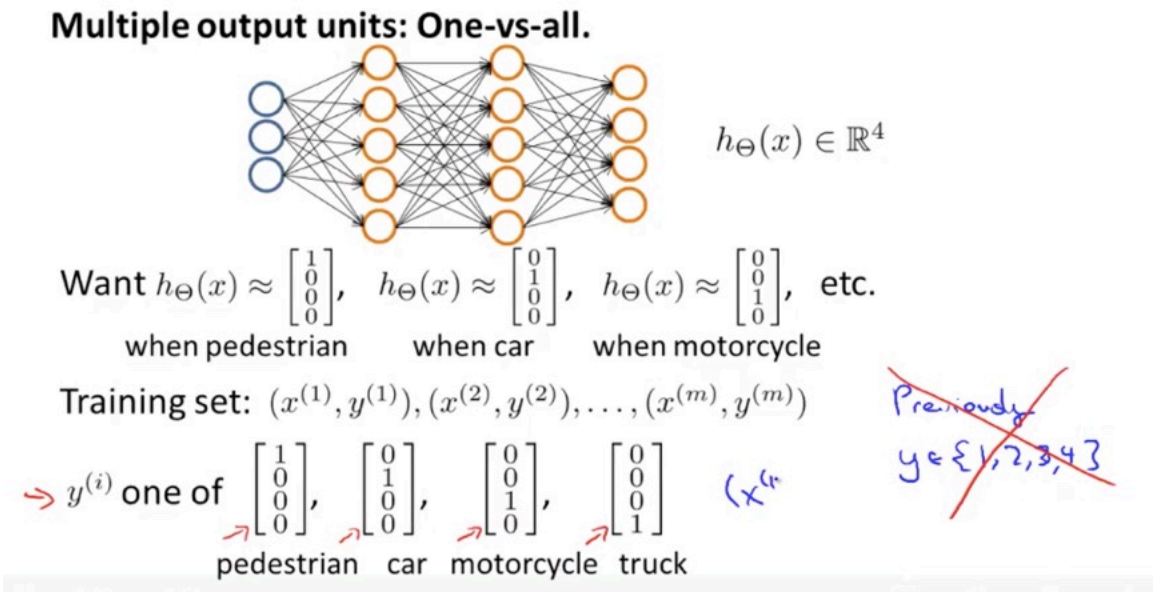
x_1	$h_\Theta(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

$h_\Theta(x) = g(10 - 20x_1)$

- Use multiple layers to make a circuit



- 



Use multiple outputs to store each type of classification. Now the training example (y) is a vector instead of a single {0, 1} value.

Homework

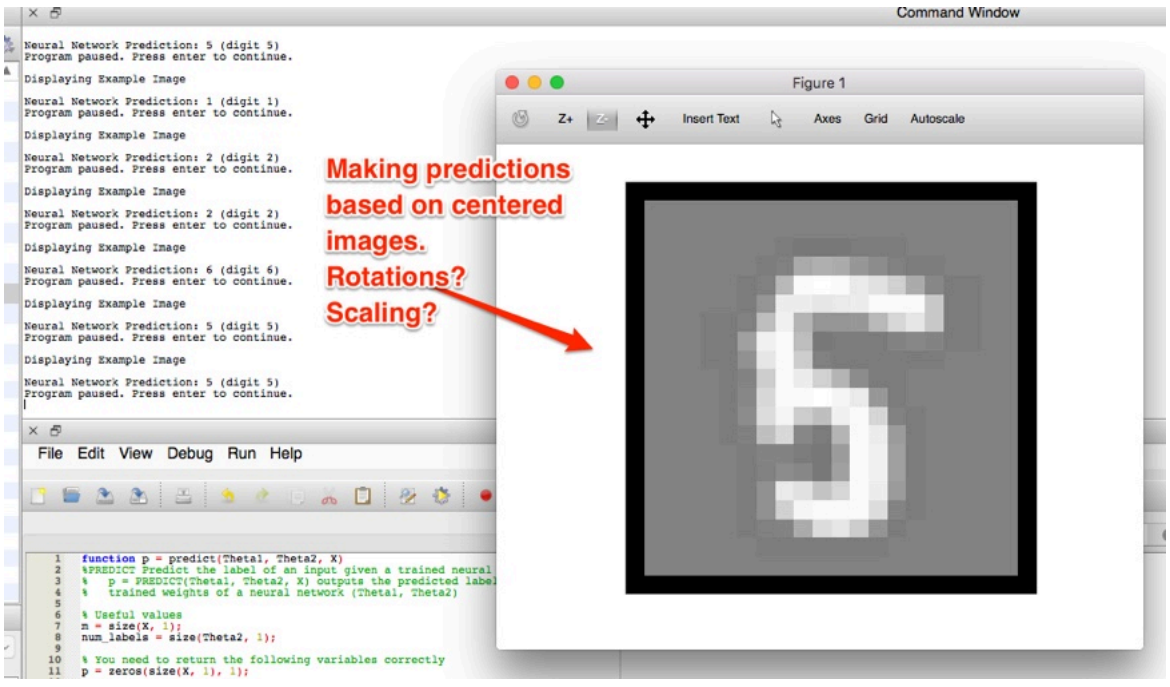
TODO: intuition for a^t = b^t a if a and b are vectors. (Doing dot product.)

Also, make sure to internalize the intuition between n x m (the rows and columns), along with the input/output flow. This helps keep everything together.

the training data will be loaded into the variables X and y.

You have been provided with a set of network parameters $(\Theta^{(1)}, \Theta^{(2)})$ already trained by us. These are stored in `ex3weights.mat` and will be loaded by `ex3_nn.m` into `Theta1` and `Theta2` The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

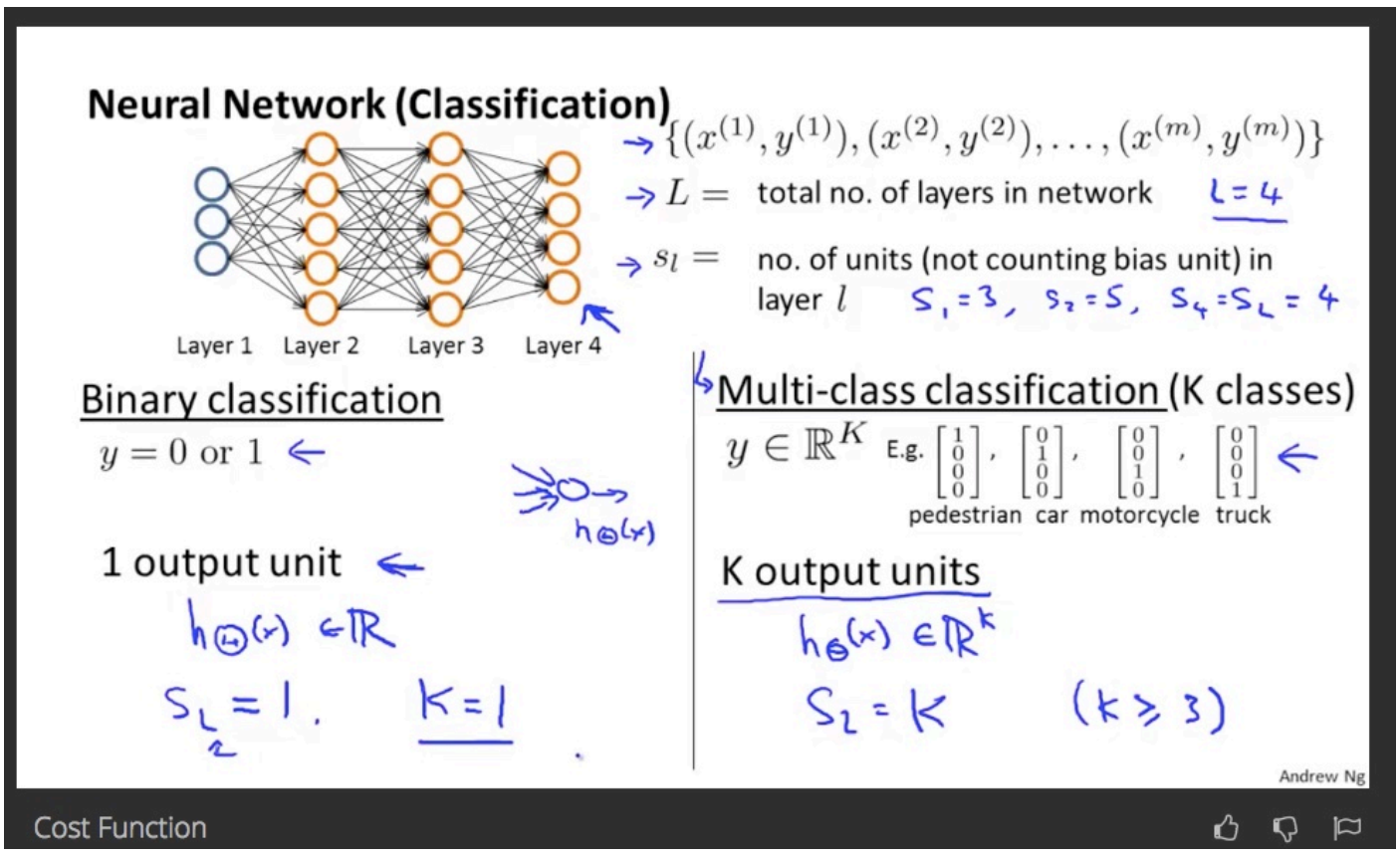
```
% Load saved matrices from file
load('ex3weights.mat');

% The matrices Theta1 and Theta2 will now be in your Octave
% environment
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```



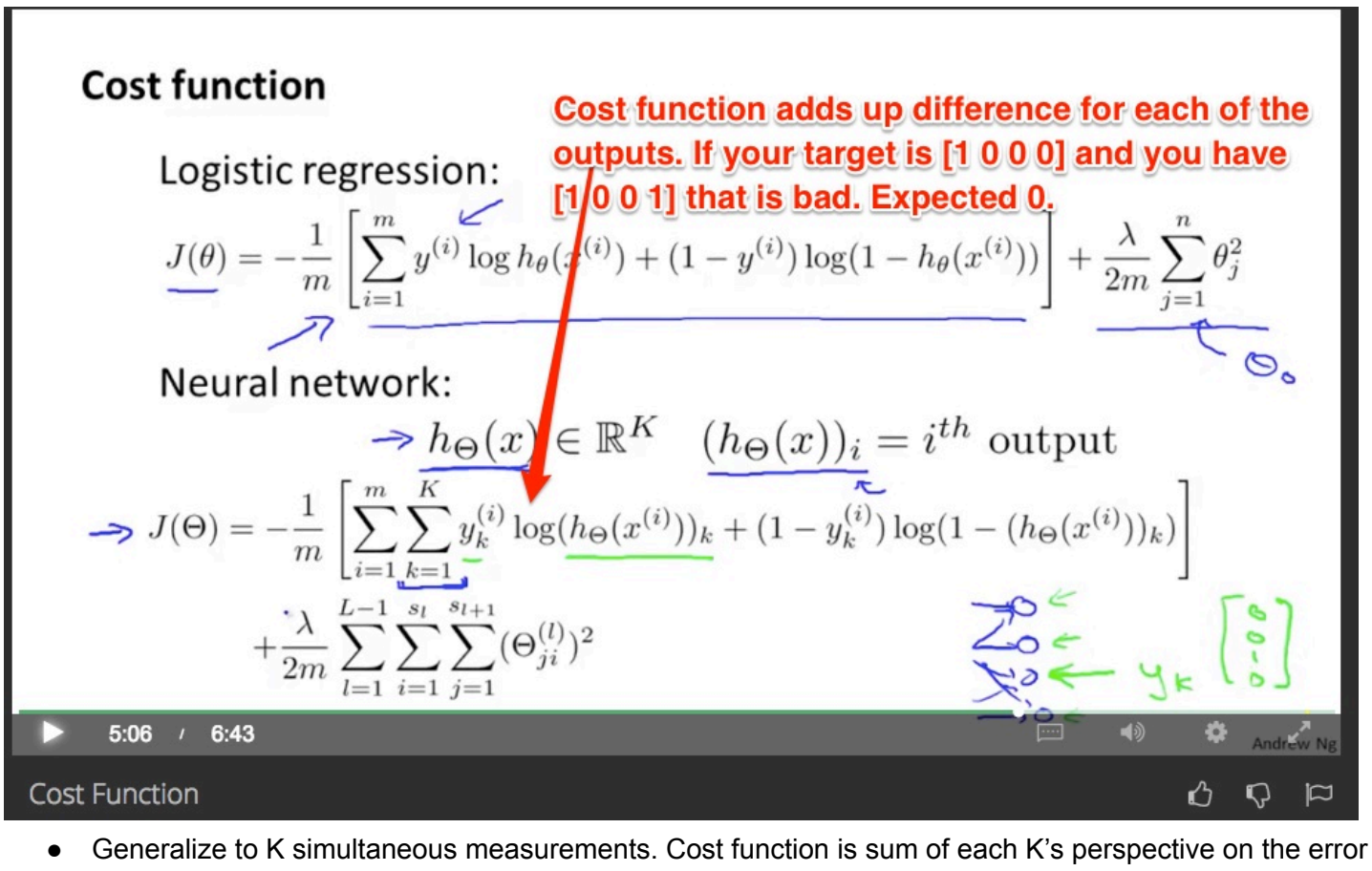
- Neural net is working, but it's looking at single pixel values and the presence of various sets of pixels. Can it handle scaling, rotation, size up/down, flipping, etc.? (Likely not with this set…)

Week 5 - Neural Networks (Learning)

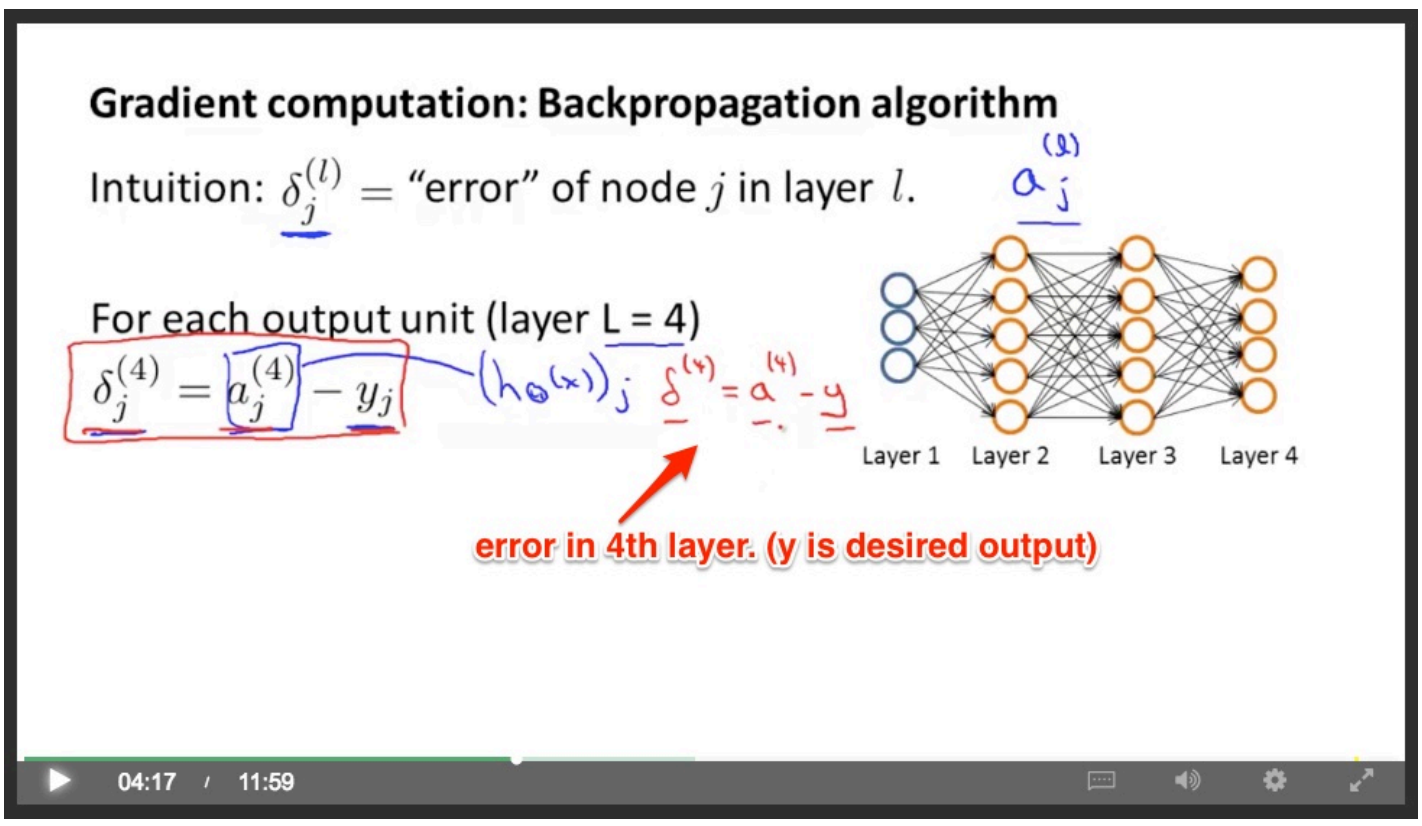
Prediction: We'll learn to take the derivative of the NN equation, gradient descent, improve the weights.



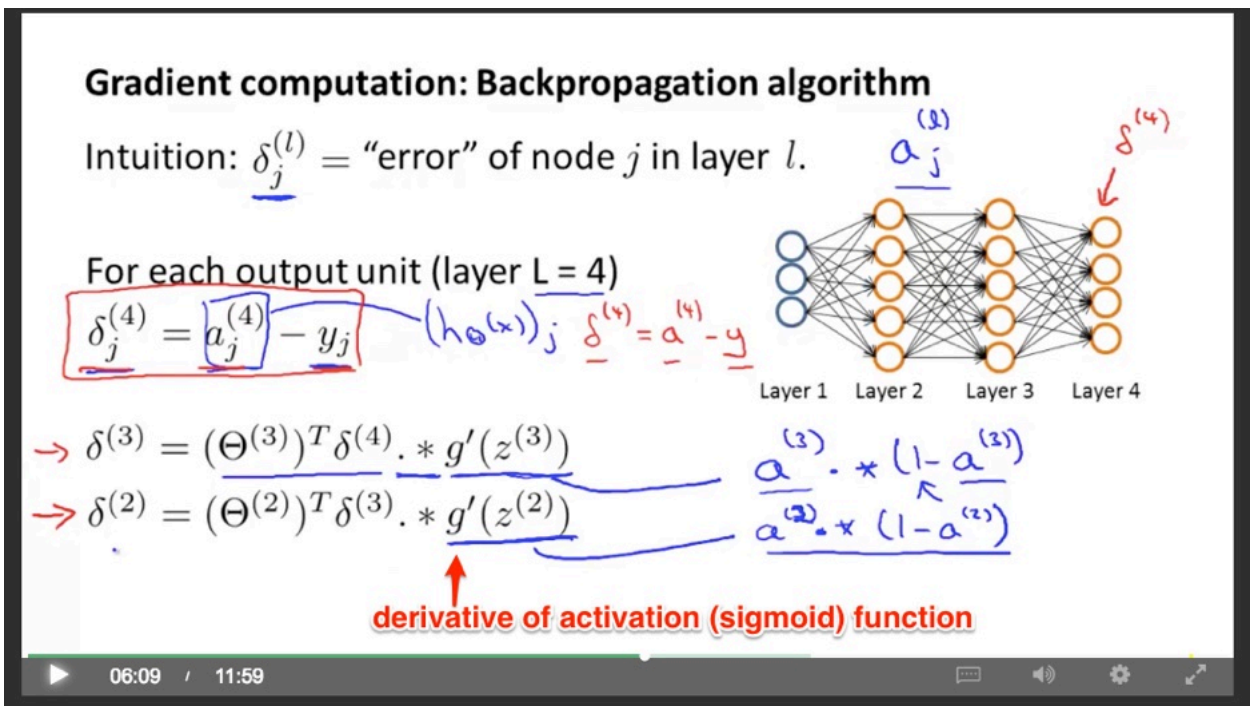
- We have a separate output node for each class. Only 1 active at a time. Easier vs. having a binary number? (00, 01, 10, 11)

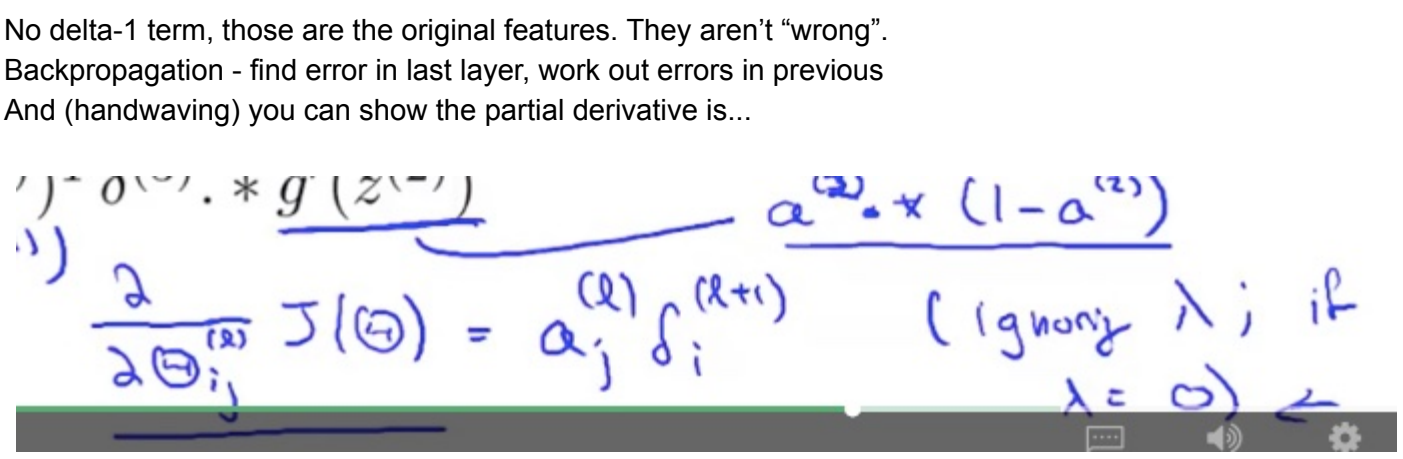


- Generalize to K simultaneous measurements. Cost function is sum of each K's perspective on the error
- The regularization looks hairy, but you are adding up the weights (except for theta0) along each of the paths. Punish for using more weights.
- Remember, to optimize the cost function we need 1) the cost function at a certain theta and 2) the gradient at that theta
    - With only the gradient, we know the direction to move, but we might overshoot. Need the cost function to tell us if we've gotten worse.



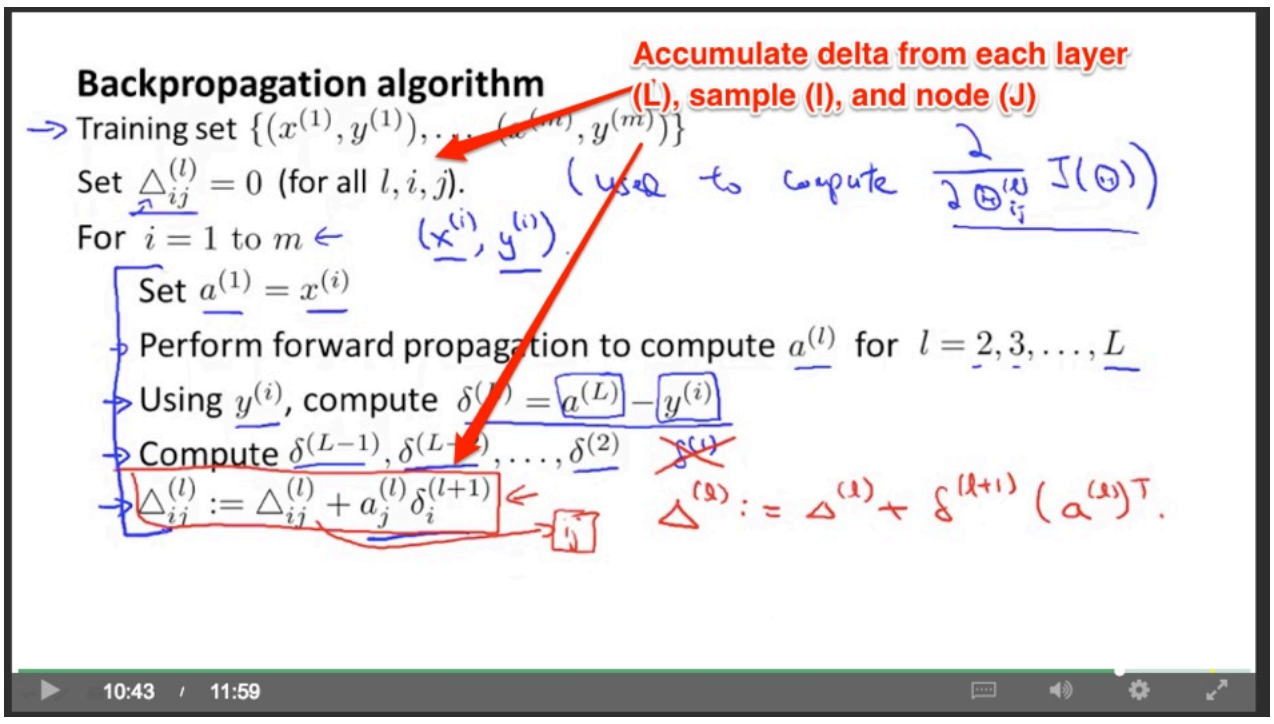
- Prediction: Now, we know $a^{(4)}$ is based on $a^{(3)}$ and so on… so can we plug in.
  - Hrm… we are working backwards to see that the error in the previous step is built by walking the level 4 error backwards to see what it originally would have been? Huh. Need intuition here.
    - This can really use a real example with numbers. There is an "intuition" video coming right after. Let's see.



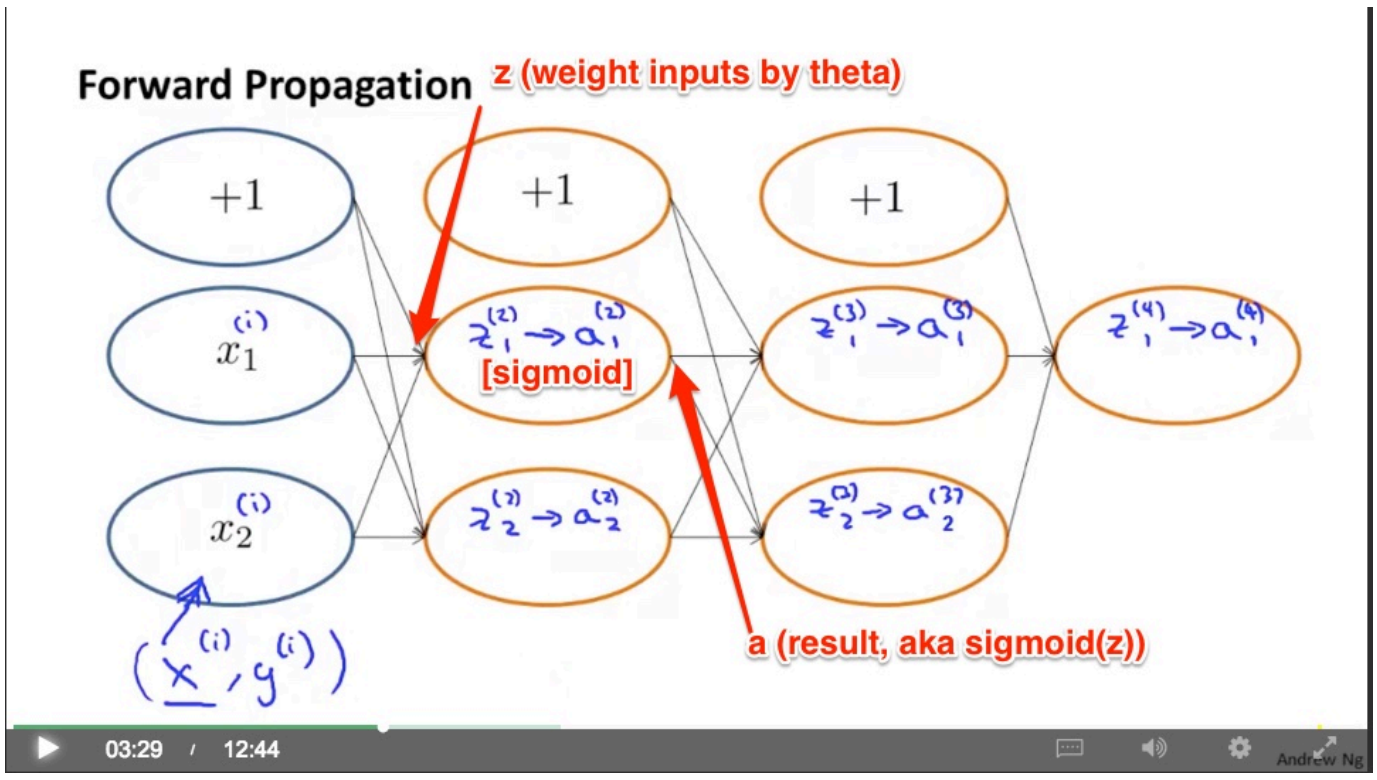


- No delta-1 term, those are the original features. They aren't "wrong".
- Backpropagation - find error in last layer, work out errors in previous
- And (handwaving) you can show the partial derivative is...

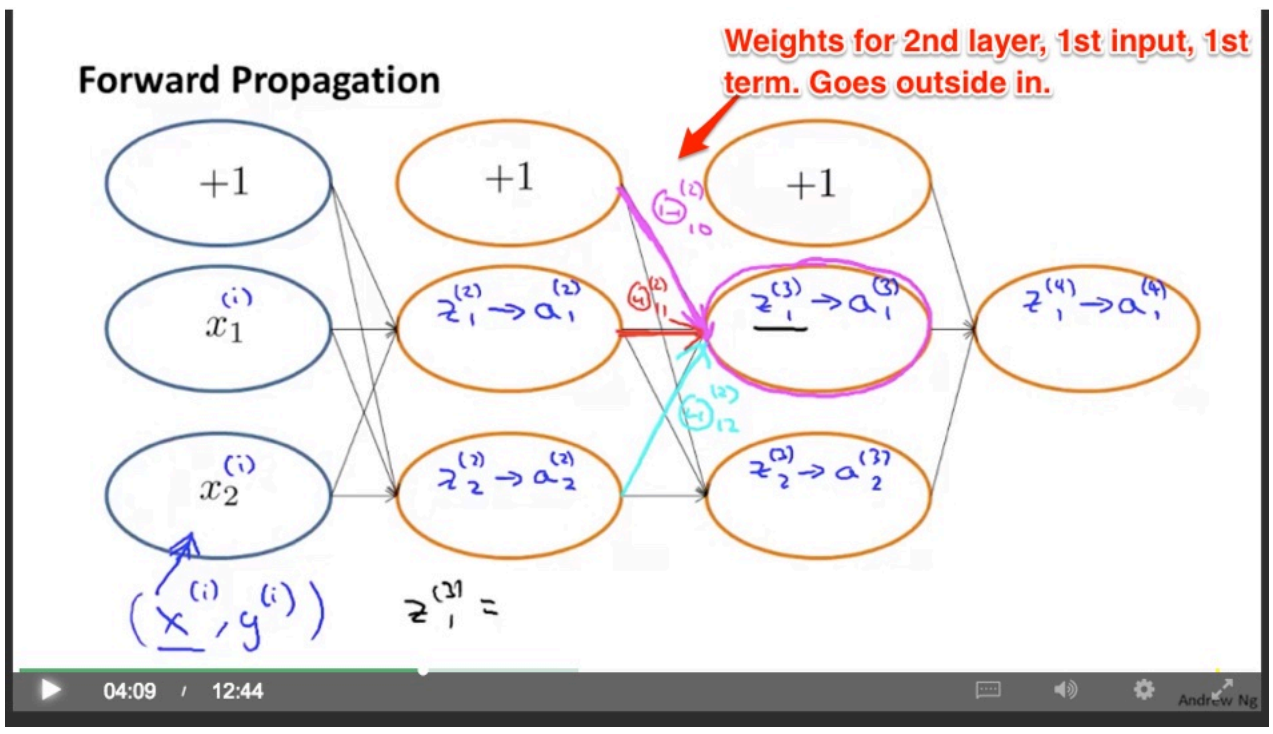
This could make sense… the change in the cost (e.g., performance) is proportional to the current error and the activation value.



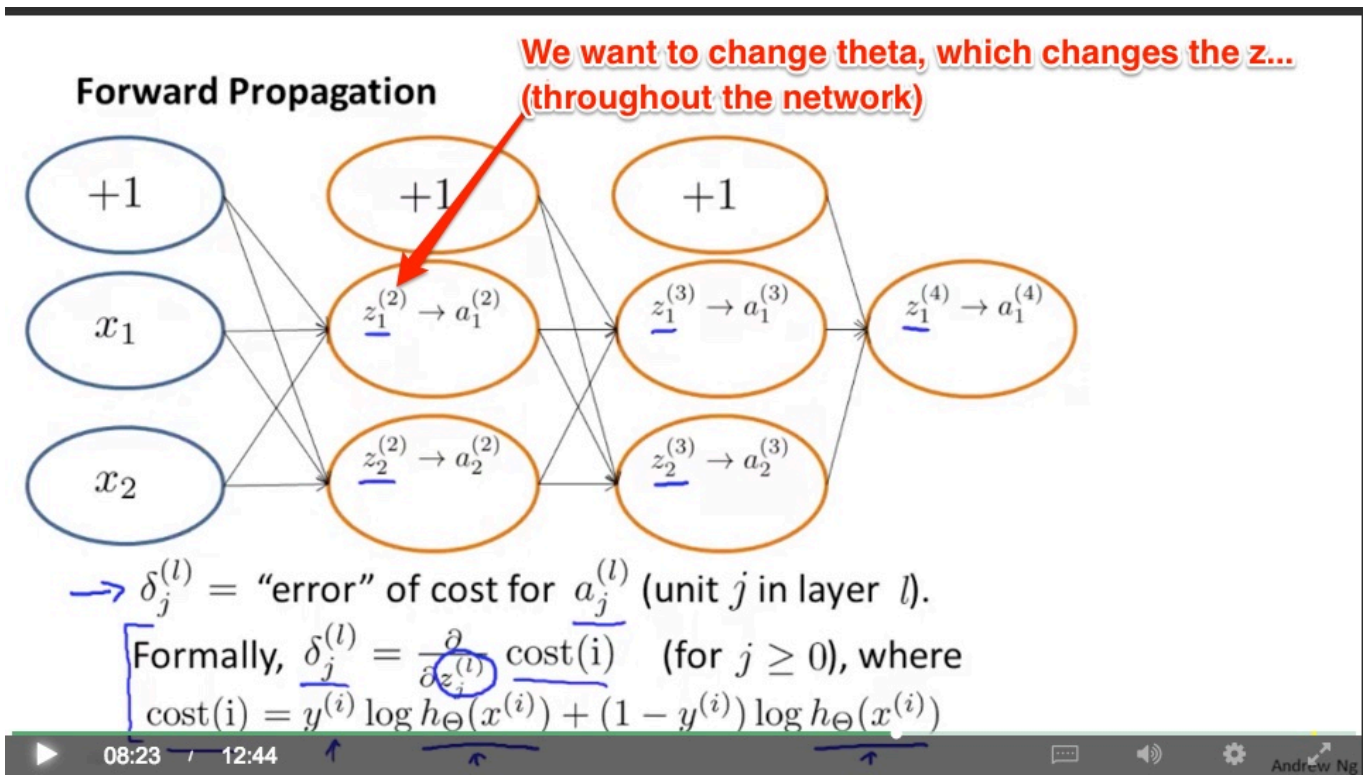
- We accumulate how much each sample (i) influences the node (j) at layer (l). The error for each is accounted for.



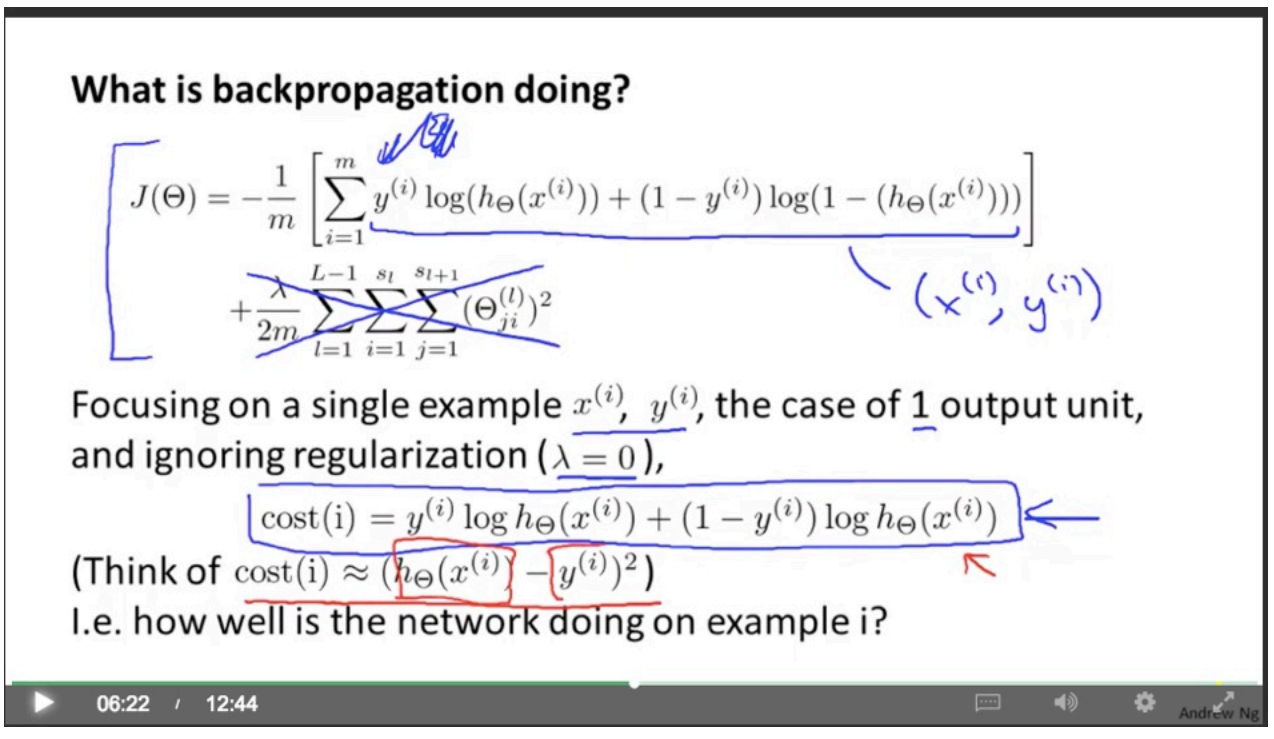
- Have a clear mental model of z (weight inputs by theta, summed), the sigmoid in the center, then a, the result.



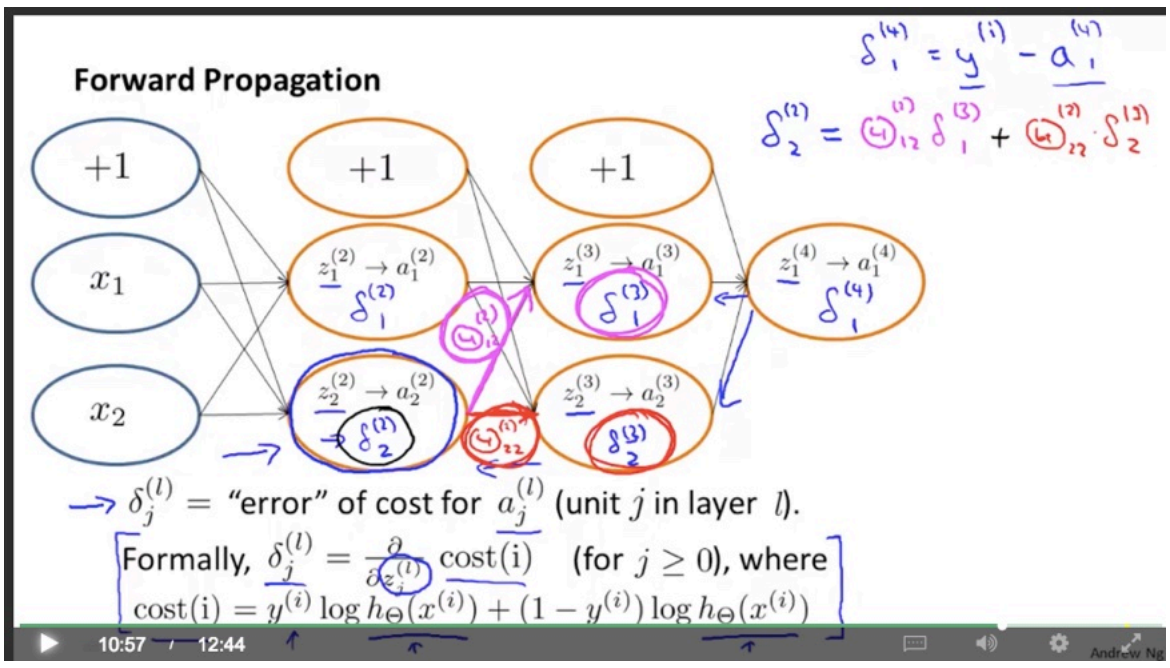
- Aha: Think about how each term goes from general to specific. The layer (2), the input (1), the subterm of that input (0).
- Half the challenge is internalizing the notation.



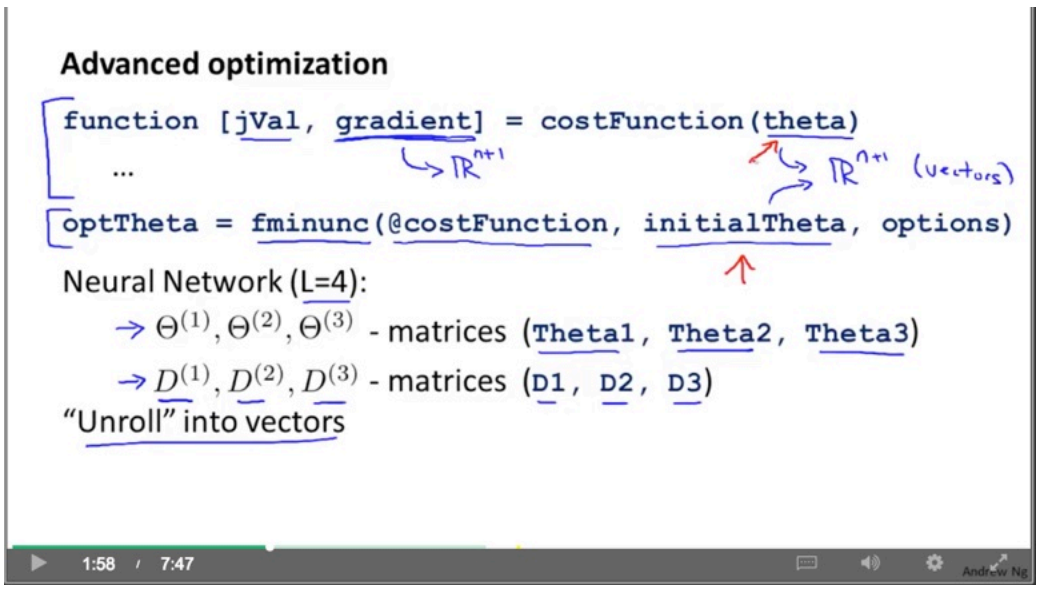
- TODO: I want to see the full derivation here. Even for a simple 2-item network. Why not start with a simpler example and use actual numbers?
  - https://www.quora.com/What-are-the-good-sources-to-understand-the-mathematical-understanding-of-the-Backpropagation-algorithm
  - http://neuralnetworksanddeeplearning.com/chap2.html
- OK… we have this cost method here:



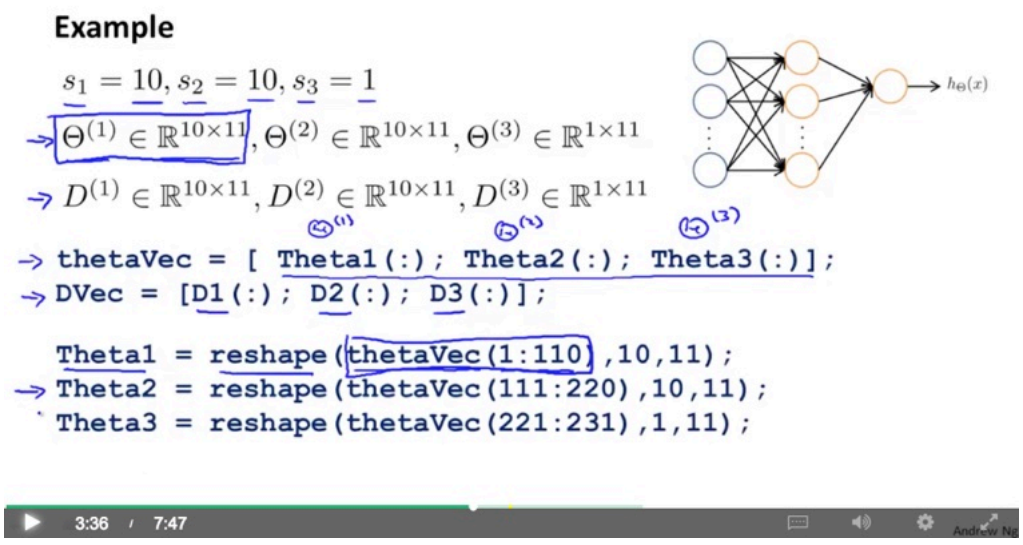
But the "hypothesis" is a very complex multi-layered function (with all the layers). Wrapping it up as h_theta is disingenuous IMO. You hide all the layers. We then need to compute the partial derivatives *at each layer* and sum them.



- Can we solve for the value of d2 that makes the logistic function return d3 which then shows up in d4? How is it so simple and just a multiplication by theta without a ln(x) term or inverse logistic or somesuch.
- Project idea
  - Create neural net to model simple functions (x^2, e^x, etc.). See how well it can do.



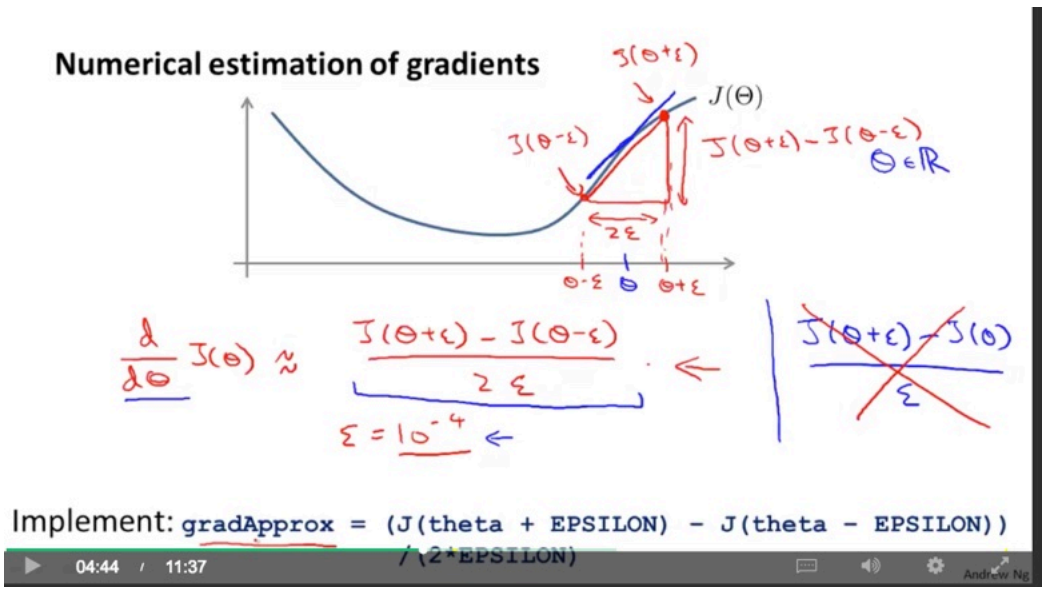
Optimizing neural nets: remember we have a matrix of parameters (for each layer, weight for node i input j), and a matrix for the gradient (for each layer, how much changing the weight of node i input j changes the cost).



- You can roll up the individual layer matrices into a single matrix. Similarly, you can explode the rolled-up matrix into individual parts.
    - The elements are serialized into a single column vector (500 x 1, etc.). Serialize, deserialize.
- You pull out a range of elements and "reshape" them into a square matrix (a 10x11 matrix for example)
    - Watch out for fencepost problem on the range: 2:3 is 2 elements (2, 3).
- Aha: Ultimately, we can treat the neural net as a giant function to optimize. We have all the parameters rolled up into a single giant theta vector. (And what's the difference anyway? The optimizer doesn't care what internal arrangement you have.)

Gradient checking

- Easy to have subtle bugs in your backprop implementations. Where the cost function looks like it's decreasing, etc., but you get worse performance (how?).
- Numerically estimate the derivative at the point (using J(theta))
    - One-sided difference (theta) to (theta + epsilon), take slope vs. two-sided (theta - epsilon) to (theta + epsilon), take slope



Parameter vector θ

- $\theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)
- $\theta = [\theta_1, \theta_2, \theta_3, \ldots, \theta_n]$
- $\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1+\epsilon, \theta_2, \theta_3, \ldots, \theta_n) - J(\theta_1-\epsilon, \theta_2, \theta_3, \ldots, \theta_n)}{2\epsilon}$
- $\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2+\epsilon, \theta_3, \ldots, \theta_n) - J(\theta_1, \theta_2-\epsilon, \theta_3, \ldots, \theta_n)}{2\epsilon}$

$$\vdots$$

$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n+\epsilon) - J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n-\epsilon)}{2\epsilon}$$

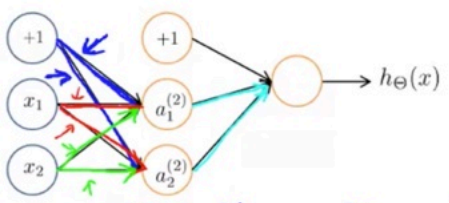
```
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) = thetaPlus(i) + EPSILON;
  thetaMinus = theta;
  thetaMinus(i) = thetaMinus(i) - EPSILON;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                  /(2*EPSILON);
end;
```

- Numerically approximate the partial derivatives for each parameter (1 to n)
  - Then, verify the numerically computed derivative is approx. equal to the backprop answer. Then we know backprop was correct.
  - Turn off gradient checking after you are using it for learning. Just a "debug-time" check when you are testing your implementation. (Otherwise you are re-computing the neural network for each iteration…)

Initializing a neural network

- Initializing all weights to zero? That means all your hidden layers compute the same value (since they have the same activation function… you are passing z = 0 to each of them).
- The partial derivatives are equal to each other (since the neurons are equal). That means the activation functions are moving in concert.

Zero initialization



$\Theta_{ij}^{(l)} = 0$ for all i, j, l.

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

  - You lose the advantage of having separate neurons. Super-redundant.
- Fix: random initialization

Random initialization: Symmetry breaking

→ Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

E.g.

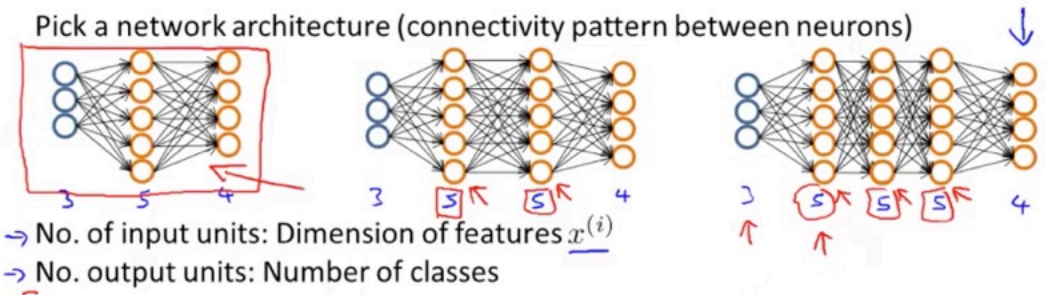
```
Theta1 = rand(10,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;

Theta2 = rand(1,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;
```

-
- Aha: “Break the symmetry” (nice summarized phrase of what you are trying to do). “If two people agree on decision, one of them is not necessary.”

Choosing NN architecture

Training a neural network

Pick a network architecture (connectivity pattern between neurons)



→ No. of input units: Dimension of features $x^{(i)}$
→ No. output units: Number of classes
Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

$y \in \{1, 2, 3, \ldots, 10\}$

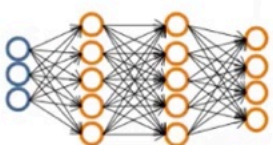
$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$

- # of inputs (determined by # of features)
- # of outputs (determined by # of output classes)
- Hidden layers: more, better. (Have more neurons than outputs?)

Training a neural network

→ 1. Randomly initialize weights
→ 2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
→ 3. Implement code to compute cost function $J(\Theta)$
→ 4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

→ `for i = 1:m`
Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$
(Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \ldots, L$).

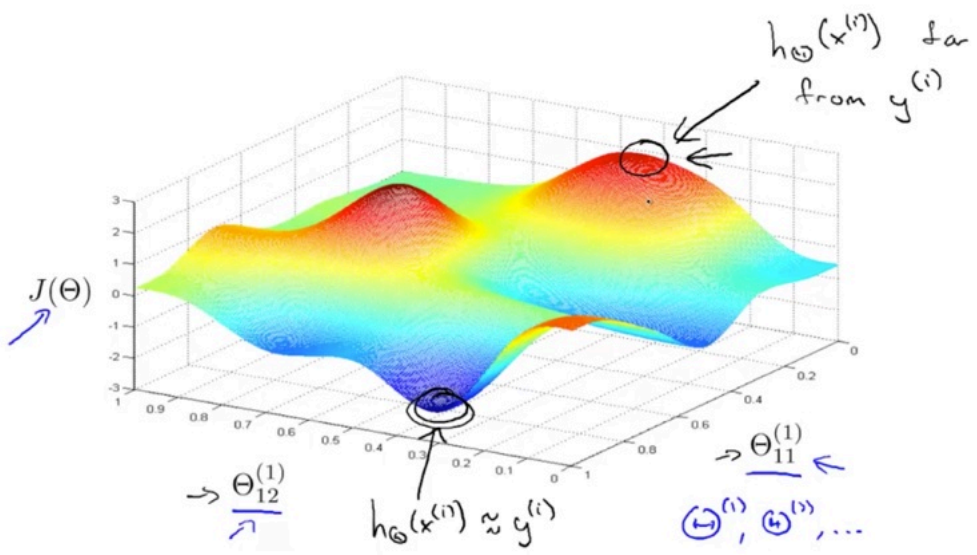


Training a neural network

→ 5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$.
Then disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

Result: A NN that has a set of weights that can classify inputs with minimized error.

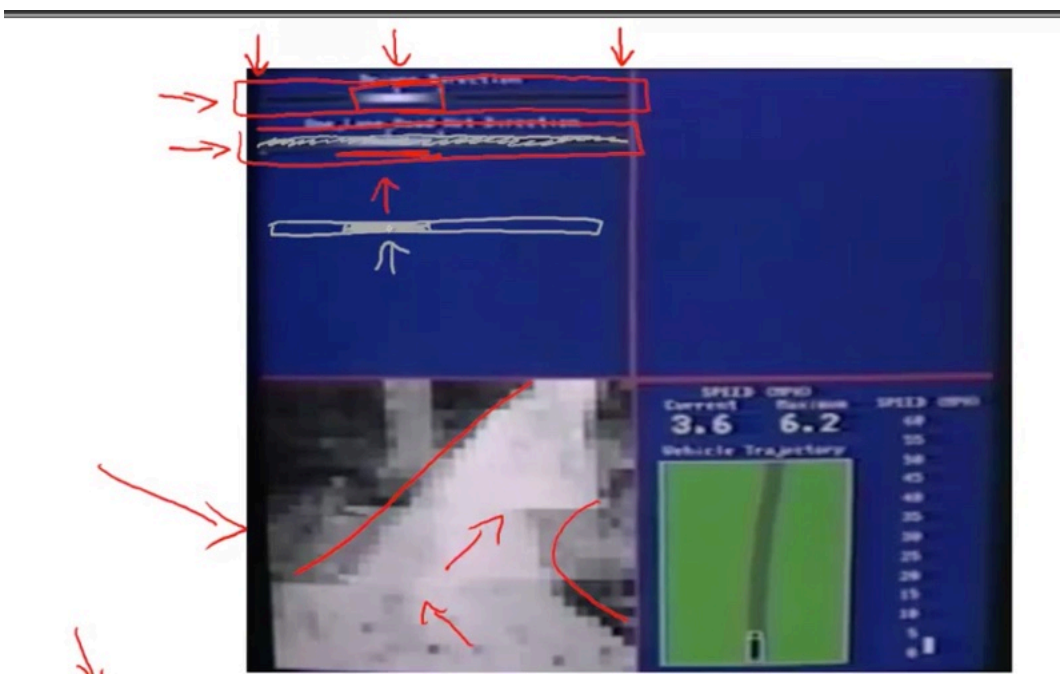
- Ultimately, just another mathematical function with n inputs that returns y, and has been optimized.



- Start randomly, walk downhill, get a good value. Q: try a few times so you don't get stuck in local min?
    - Huh: there is still some hand-waviness on the math of backprop. Would be better to walk through on our own and learn to predict by hand what values certain things should take. A simple 2x2 network, etc.

Autonomous driving

- Given pixels, make a steering direction. The human driver is the reference (pixel input, output is the left/right steering direction).



- Learning algorithm is to output the correct steering direction given the pixel input. That's pretty sweet.

Homework

Recall that the cost function for the neural network (without regularization) is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right],$$

- In plain english: The cost function is the error for each classification (K total), for each sample (m total). Then take the average error per sample.
- Hrm, there seems to be some strange precision thing going on. The vectorized implementation from last week returns results different from the loop. Just going to use vectorized. (D'oh -- used m, not i, as the index…)

```
% make prediction for all samples (from last week)
X_bias = [ones(size(X, 1), 1) X];
A1 = sigmoid(Theta1 * X_bias');
A1_bias = [ones(1, size(A1, 2)); A1];
A2 = sigmoid(Theta2 * A1_bias);

% total cost for all predictions
for i=1:m

  z1 = [1; X(m, :)'];  %% ARGH, stupid typo… should be i, not m
  a1 = sigmoid(Theta1 * z1);
  z2 = [1; a1];
  a2 = sigmoid(Theta2 * z2);
  % something weird is happening above...

  y_vec = zeros(num_labels, 1);
```

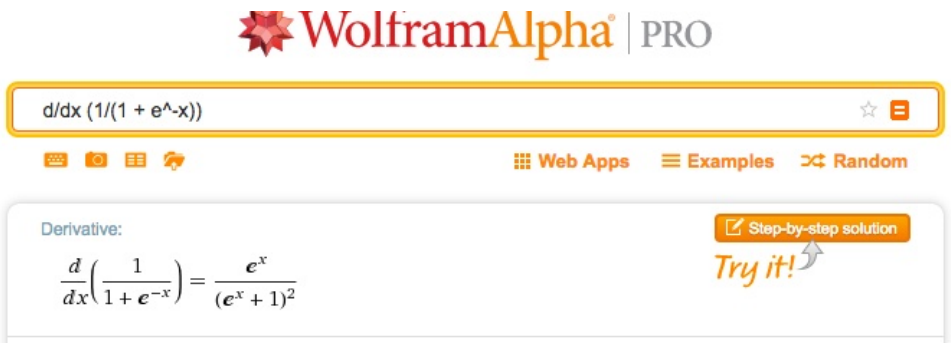
```
  y_vec(y(i)) = 1;

  total_cost = -y_vec.* log(a2) - (1 - y_vec).*log(1 - a2);
  J += sum(total_cost) / m;
end
```

Derivative of the sigmoid is nicely behaved (another nice benefit of using e^x)

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z))$$

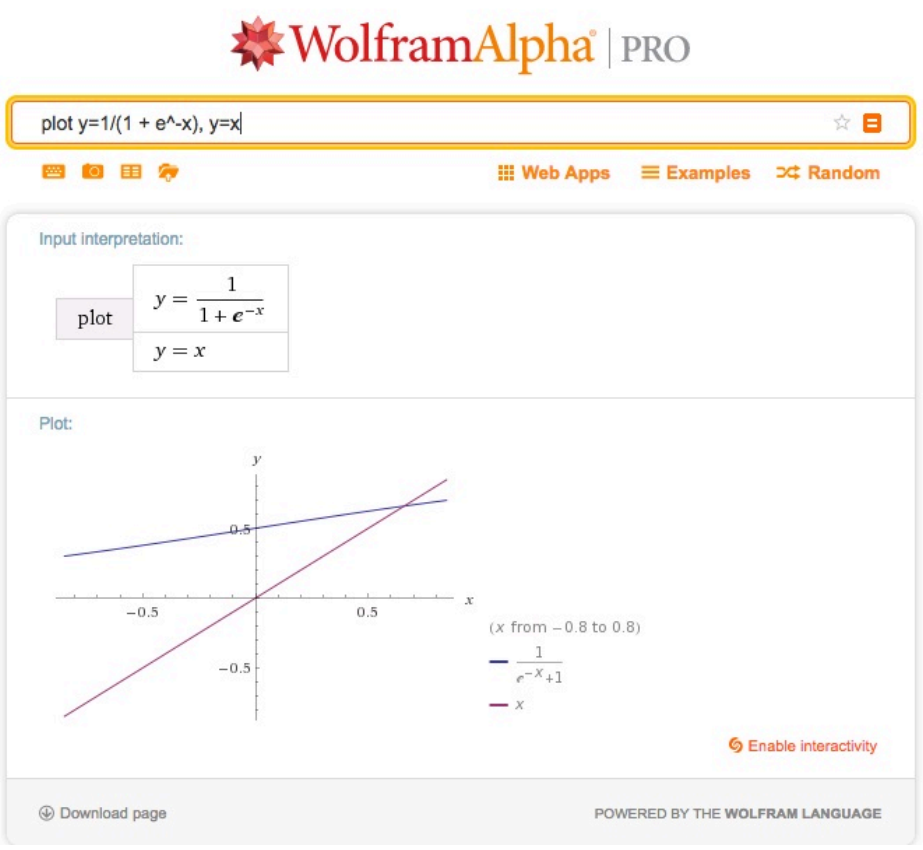
$$\text{sigmoid}(z) = g(z) = \frac{1}{1 + e^{-z}}.$$

http://www.wolframalpha.com/input/?i=d%2Fdx+(1%2F(1+%2B+e%5E-x))



Aha: instead of the official derivative above, refactor to g(z)(1 - g(z)), which lets you re-use previous computations. Faster.

For mental sanity check, plot the sigmoid (vs line y=x) and see how it slopes slower:
http://www.wolframalpha.com/input/?i=plot+y%3D1%2F(1+%2B+e%5E-x),+y%3Dx



In the hw, you see that the gradient at z=0 is .25 (a slow ramp up, not changing that fast compared to e^x or y=x).

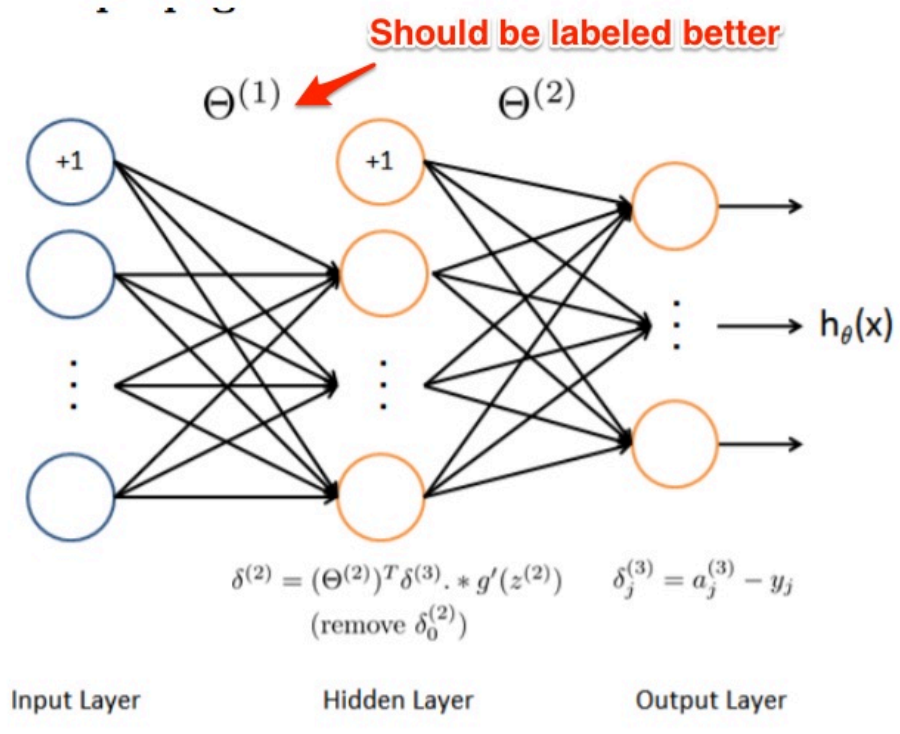
Randomly initialize weights based on number of units/neurons in the entire network. Keep weights small (like .12)

Error term

- delta$^{(l)}_j$ - how much layer l, node j was responsible for the error in our output.

Backpropagation

- Yep, this course needs better labeling of what the various inputs/outputs are



- I'm using my own notation for a1, z1, etc. a1=z2 (outputs of first layer are inputs of second…, since the first layer just passes them along.)
- Huh: Half the difficulty is keeping the various indices straight

Suppose you have a function $f_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i} J(\theta)$; you'd like to check if f_i is outputting correct derivative values.

$$\text{Let} \quad \theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \quad \text{and} \quad \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ϵ. Similarly, $\theta^{(i-)}$ is the corresponding vector with the i-th element decreased by ϵ. You can now numerically verify $f_i(\theta)$'s correctness by checking, for each i, that:

$$f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}.$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\epsilon = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

We have implemented the function to compute the numerical gradient for you in `computeNumericalGradient.m`. While you are not required to modify the file, we highly encourage you to take a look at the code to understand how it works.

```
function numgrad = computeNumericalGradient(J, theta)
%COMPUTENUMERICALGRADIENT Computes the gradient using "finite differences"
%and gives us a numerical estimate of the gradient.
%   numgrad = COMPUTENUMERICALGRADIENT(J, theta) computes the numerical
%   gradient of the function J around theta. Calling y = J(theta) should
%   return the function value at theta.

% Notes: The following code implements numerical gradient checking, and
%        returns the numerical gradient.It sets numgrad(i) to (a numerical
%        approximation of) the partial derivative of J with respect to the
%        i-th input argument, evaluated at theta. (i.e., numgrad(i) should
%        be the (approximately) the partial derivative of J with respect
%        to theta(i).)
%

numgrad = zeros(size(theta));
perturb = zeros(size(theta));
e = 1e-4;
for p = 1:numel(theta)
    % Set perturbation vector
    perturb(p) = e;
    loss1 = J(theta - perturb);
    loss2 = J(theta + perturb);
    % Compute Numerical Gradient
    numgrad(p) = (loss2 - loss1) / (2*e);
    perturb(p) = 0;
end

end
```

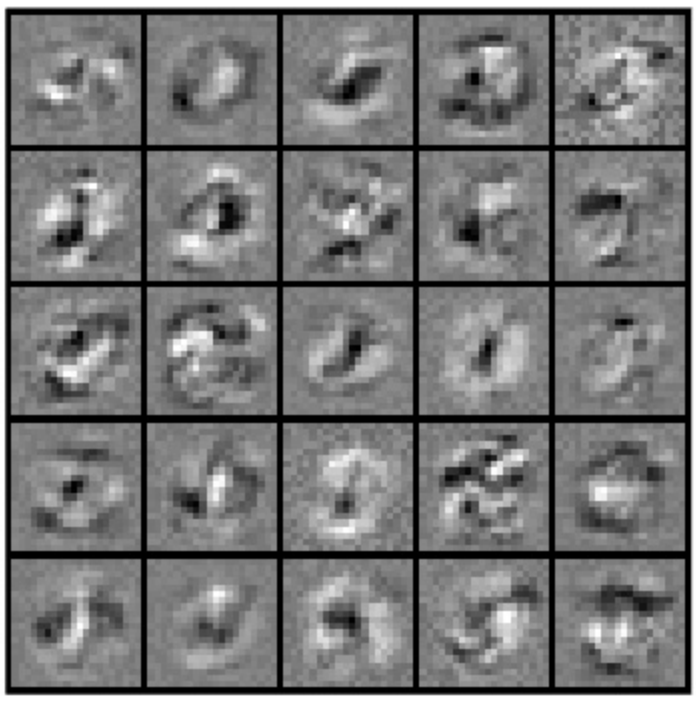
change every parameter of theta and compute numerical gradient there

- Yep, there is confusion about who should set the bias term. Should specify: add a(0) = 1 in the activation layer. Then z() can work on those inputs.

```
% compute gradients
for t=1:m
  a1 = [1; X(t, :)'];          % activation of Level 1 is bias and the raw input
  z2 = Theta1 * a1;            % weight by theta
  a2 = [1; sigmoid(z2)];       % activate
  z3 = Theta2 * a2;
  a3 = sigmoid(z3);

  y_vec = zeros(num_labels, 1);
  y_vec(y(t)) = 1;
  d3 = (a3 - y_vec);
  d2 = ((Theta2'(2:end,:) * d3) .* sigmoidGradient(z2));

  Theta2_grad += (d3 * a2');
  Theta1_grad += (d2 * a1');
end

Theta2_grad /= m;
Theta1_grad /= m;
```

add bias to activation

- a(2) is the output at layer 2, which is the bias term and the sigmoid of the inputs to that layer (z2).



We can visualize the features the NN is detecting. Show the weights for each of the pixels (0 -- no weight -- to highly weighted).

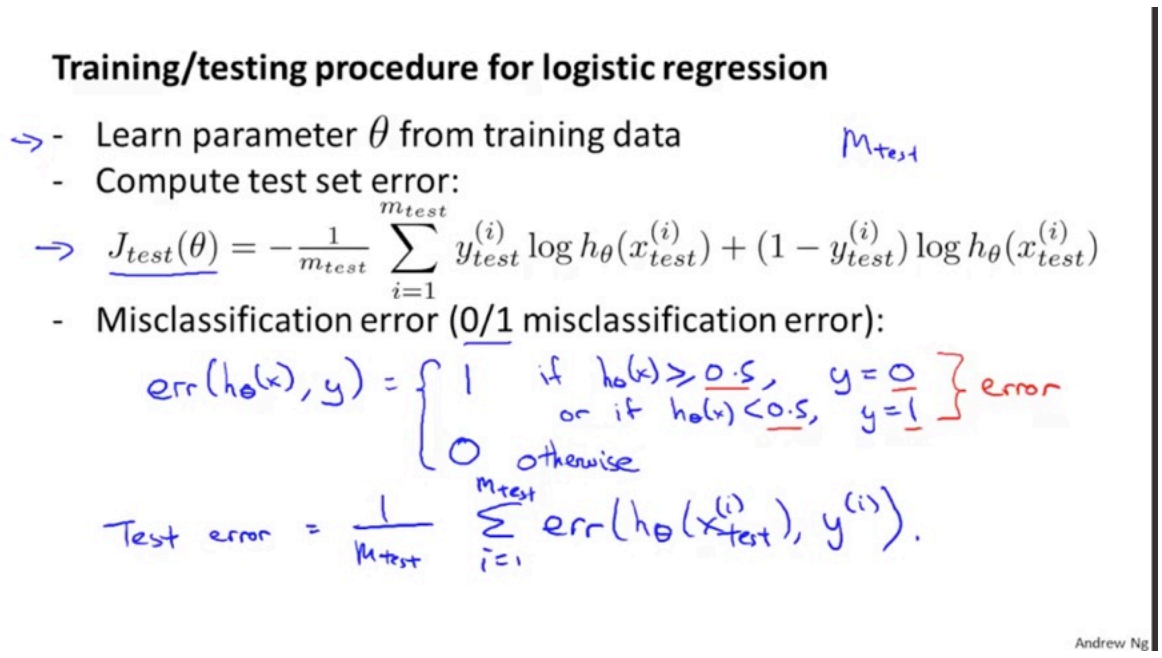
Week 6 - Advice for Machine Learning

Suppose your model isn't performing well -- what to do?

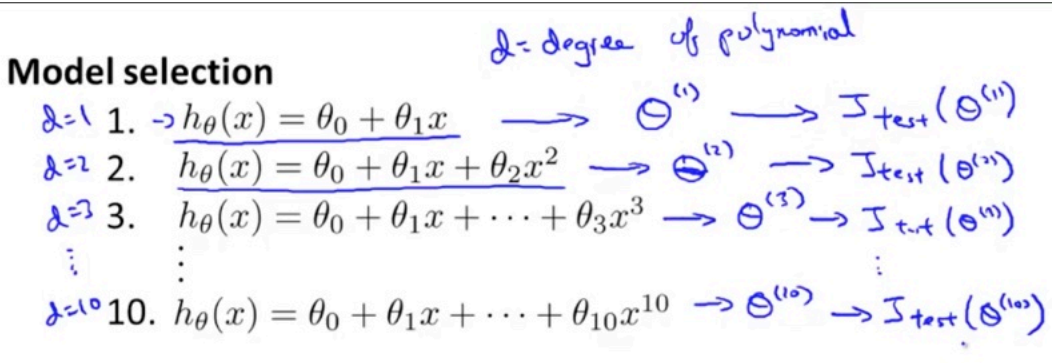
- More examples, more features, smaller sets of features, polynomial features ((x1)^2, (x2)^2, etc.), tweaking lambda (regularization)
- Can waste lots of time making random adjustments. Better: diagnostics

How do you know if you're overfitting?

- Hard to plot when there's many features
- Split data into training set, test set (70%/30%). m training examples, m_{test} for test examples. If data is ordered, pick randomly (or… just randomly pick no matter what).
- Procedure: Learn Theta for training (minimize J(theta)), compute test set error (J_test (Theta), or the total error on the training set)
- Misclassification error (0/1). See how many items have an error (100% to 0%). Aha: it's easier to understand an error % vs an absolute error rate. Easier to compare as well between runs (how much better is it getting relative to how good it could be).



Model selection problem: what polynomial to use? What regularization factor? Try various models. Then see which has lowest test set error.



- But… you are fitting the degree of polynomial (d) to the test set itself! So you are still overfitting on the model, you won't predict error reliably.
- Fix: Split into training set, cross validation set (CV), test set. 60/20/20. Track error for each

Train/validation/test error

Training error:

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

Cross Validation error:

$$J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

Test error:

$$J_{test}(\theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

Model selection

1. $h_\theta(x) = \theta_0 + \theta_1 x$ → $\min_\theta J(\theta) \rightarrow \theta^{(1)} \rightarrow J_{cv}(\theta^{(1)})$
2. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$ → $\theta^{(2)} \rightarrow J_{cv}(\theta^{(2)})$
3. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_3 x^3$ → $\theta^{(3)}$

$\vdots$ $\quad$ $J_{cv}(\theta^{(4)})$

10. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}$ → $\theta^{(10)} \rightarrow J_{cv}(\theta^{(10)})$

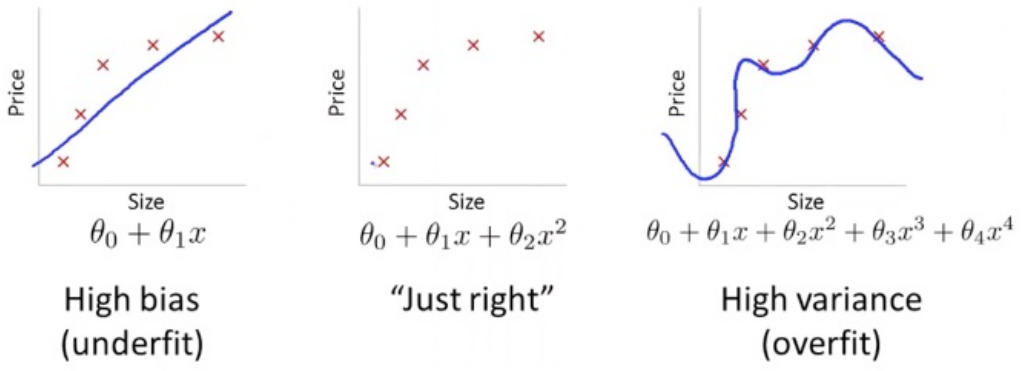
$d = 4$

Pick $\theta_0 + \theta_1 x_1 + \cdots + \theta_4 x^4$ ←

Estimate generalization error for test set $J_{test}(\theta^{(4)})$ ←

- Pick the best model on cross validation set (let's say d=4), then compute error on the test set with d=4. That gives an accurate estimate of the likely error.
  - Aha: In other words, use each test set to figure out ONE question at a time (like an equation in multiple variables)
    - Q1: How many dimensions should I use?
    - Q2: What is my rate of error?

Bias and variance (underfitting / overfitting)



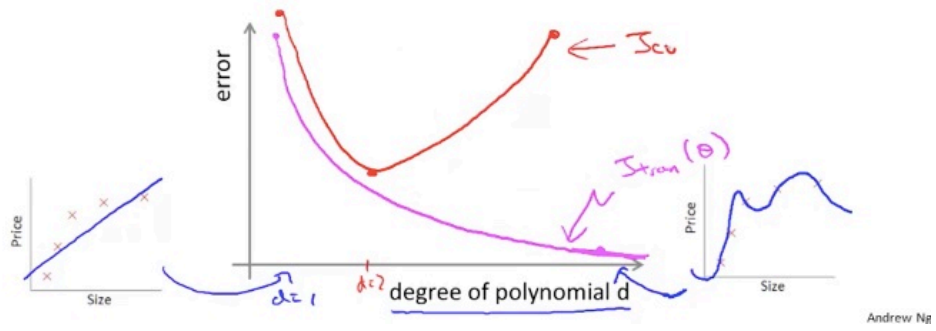
- Bias - your model has too much of a preconceived notion
- Variance - model changes around too much

Training error decreases as we increase degree of polynomial (d)

Bias/variance

Training error: $J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$

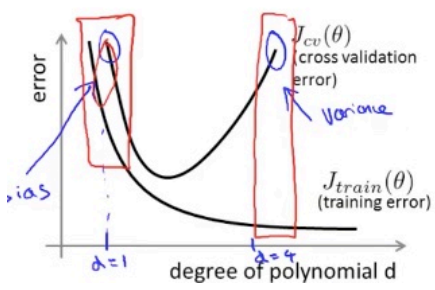
Cross validation error: $J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$ (or $J_{test}(\theta)$)



- Cross-validation or test error is high as we add more degrees
- Bias and cross-validation high: (underfit)
- Variance (overfit) when bias is low (you follow closely) but cross validation error high
    - Either way, your test set has error

Diagnosing bias vs. variance

Suppose your learning algorithm is performing less well than you were hoping. ($J_{cv}(\theta)$ or $J_{test}(\theta)$ is high.) Is it a bias problem or a variance problem?



Bias (underfit):
$J_{train}(\theta)$ will be high
$J_{cv}(\theta) \approx J_{train}(\theta)$

Variance (overfit):
$J_{train}(\theta)$ will be low
$J_{cv}(\theta) \gg J_{train}(\theta)$

Regularization

- Idea is to avoid overfitting -- have cost for too many parameters

Linear regression with regularization

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

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

Large λ
High bias (underfit)
$\lambda = 10000.\ \theta_1 \approx 0, \theta_2 \approx 0, \ldots$
$h_\theta(x) \approx \theta_0$

Intermediate λ
"Just right"

Small λ
High variance (overfit)
$\lambda = 0$

- Goal: automatically find regularization setting
    - Idea: find lambda parameter that minimizes training error
    - Yep. That's what's happens.

Choosing the regularization parameter λ

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

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

1. Try $\lambda = 0$ → $\min_\theta J(\theta)$ → $\theta^{(1)}$ → $J_{cv}(\theta^{(1)})$
2. Try $\lambda = 0.01$ → $\min_\theta J(\theta)$ → $\theta^{(2)}$ → $J_{cv}(\theta^{(2)})$
3. Try $\lambda = 0.02$ → $\theta^{(3)}$ → $J_{cv}(\theta^{(3)})$
4. Try $\lambda = 0.04$
5. Try $\lambda = 0.08$ → $\theta^{(5)}$ → $J_{cv}(\theta^{(5)})$

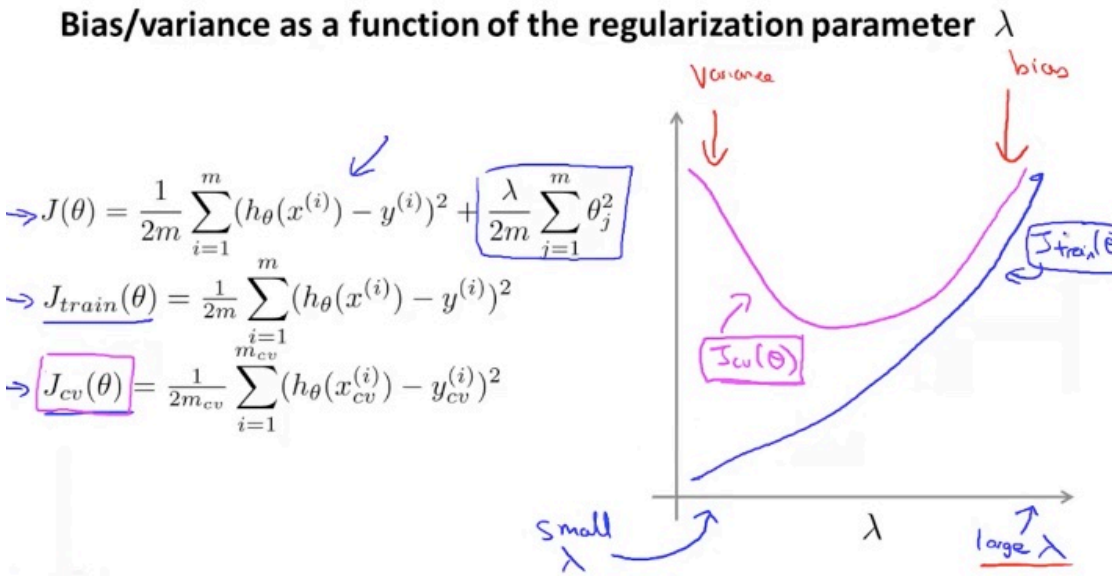
⋮

12. Try $\lambda = 10$ (10.24) → $\theta^{(12)}$ → $J_{cv}(\theta^{(12)})$

Pick (say) $\theta^{(5)}$. Test error: $J_{test}(\theta^{(5)})$

Andrew Ng

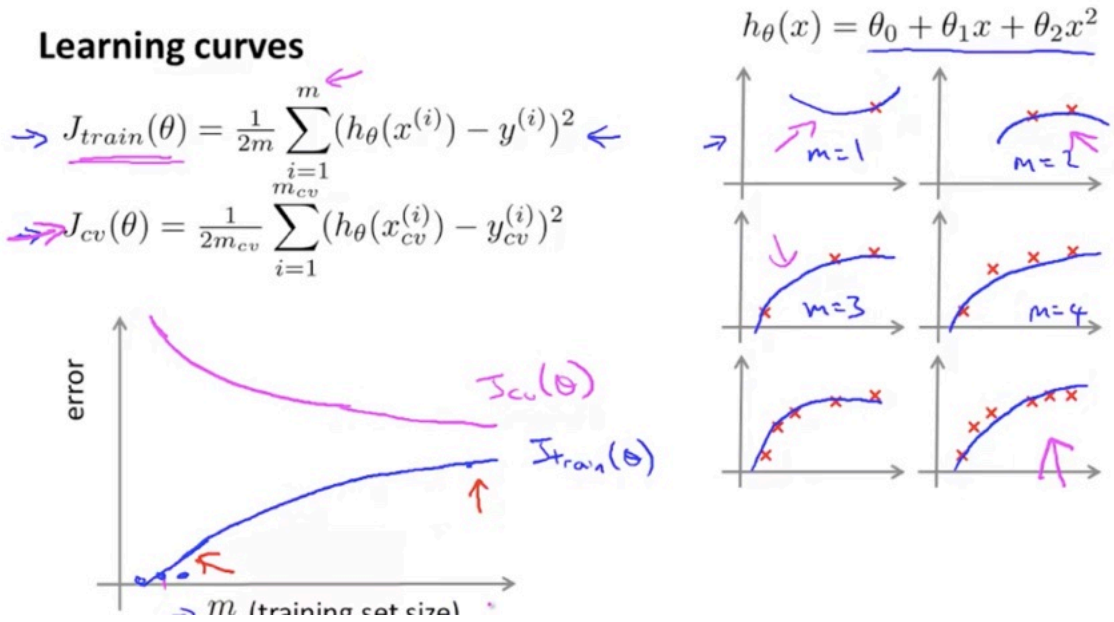
- Still need to find empirical error rate via the test set. (Because we fit theta to the cross validation set)

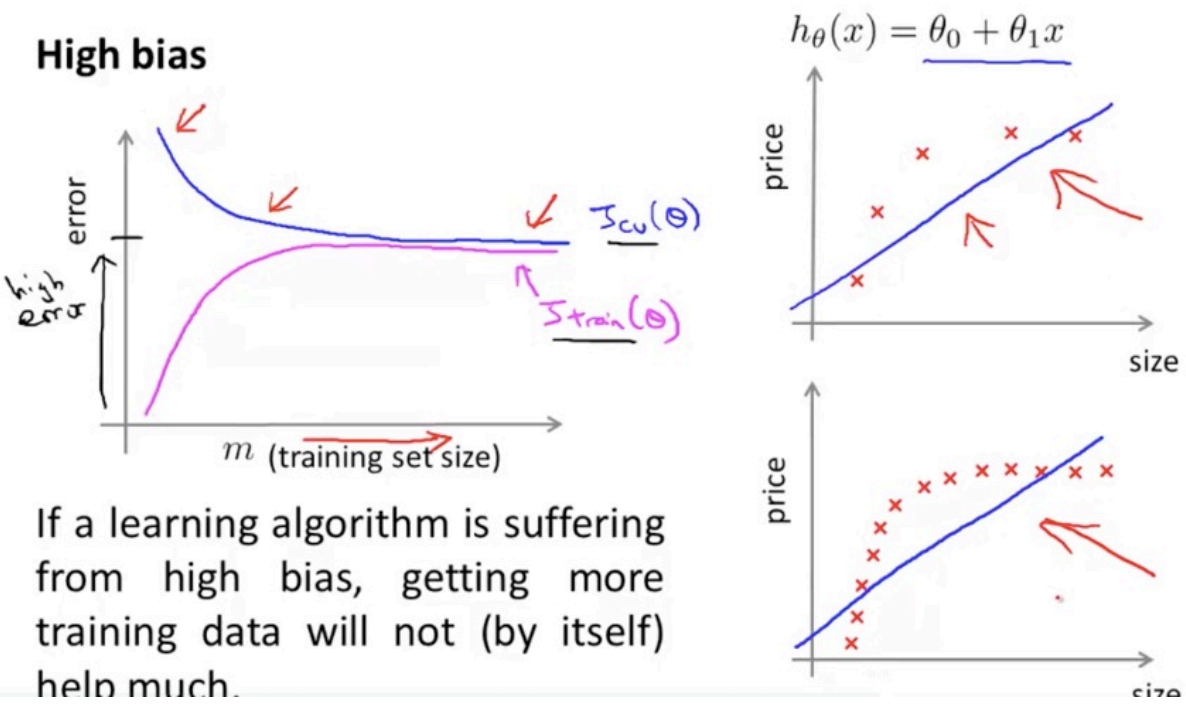


- If lambda is too small, variance is high (overfit). If lambda large, bias high (underfit).

Learning curves

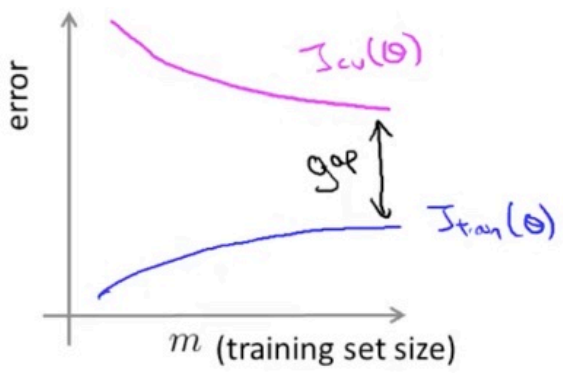
- See how much error is there (avg training error for the samples you've tried)
- When m=small, you can fit every example. As m grows, will start to miss some.
- As you get to larger sets, error increases… but you can improve cross-validation error.





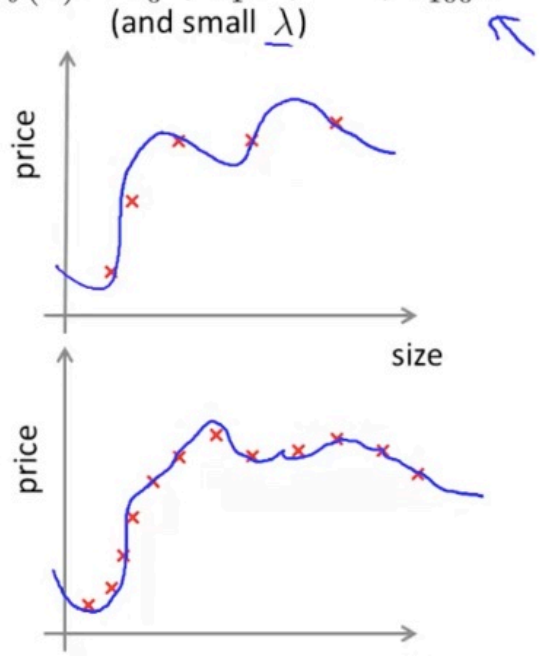
- With high bias, more data doesn't help. You're still underfitting.

High variance



If a learning algorithm is suffering from high variance, getting more training data is likely to help.

$h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{100} x^{100}$
(and small λ)



- There is a gap between cv error and training error. Means you can get more data and reduce cv error.

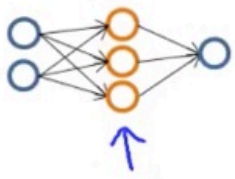
High bias - too simple
HIgh variance - too complex

Debugging a learning algorithm:
Suppose you have implemented regularized linear regression to predict housing prices. However, when you test your hypothesis in a new set of houses, you find that it makes unacceptably large errors in its prediction. What should you try next?

- Get more training examples → fixes high variance
- Try smaller sets of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features $(x_1^2, x_2^2, x_1 x_2, \text{etc})$ → fixes high bias.
- Try decreasing λ → fixes high bias
- Try increasing λ → fixes high var

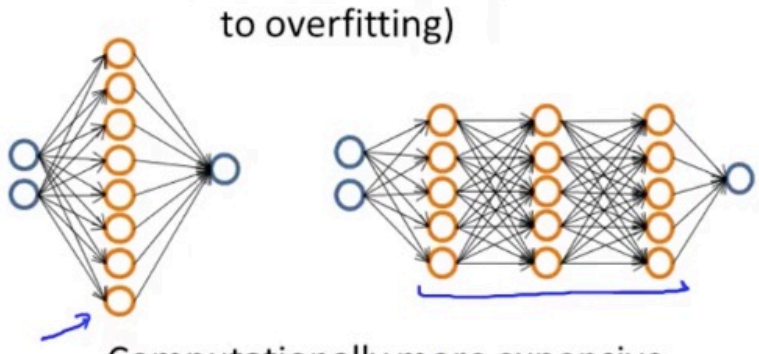
Neural networks and overfitting

→ "Small" neural network (fewer parameters; more prone to underfitting)



Computationally cheaper

→ "Large" neural network (more parameters; more prone to overfitting)



Computationally more expensive.

Use regularization (λ) to address overfitting.

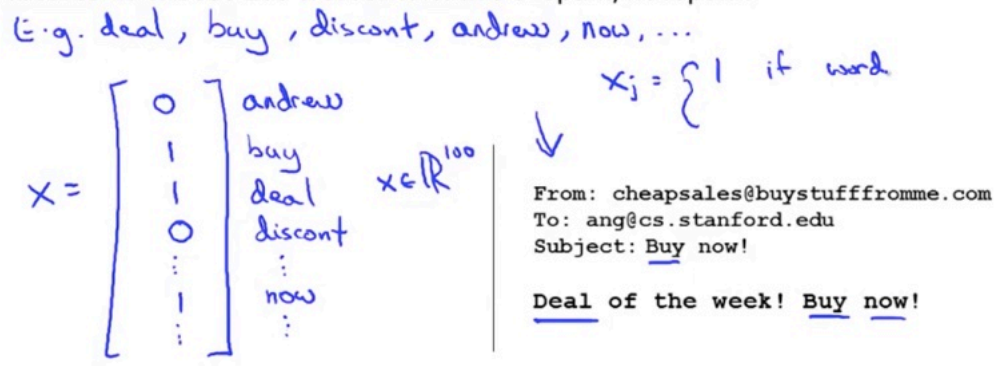
- NN has underfitting / overfitting issues too

Spam classifier

Building a spam classifier

Supervised learning. $x =$ features of email. $y =$ spam (1) or not spam (0).

Features x: Choose 100 words indicative of spam/not spam.



- Make a vector based on words. Top 50k words, etc.
  - Remember: Classifier (yes/no with a percentage) vs regression (house price).
  - Just getting more data may not help. Smarter features may help.
- Goal is to have various options and then pick between them (not going with random hunch)

Recommended approach

- Start with a simple algorithm that you can implement quickly. Implement it and test it on your cross-validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Error analysis: Manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

Error Analysis

$m_{CV} =$ 500 examples in cross validation set

Algorithm misclassifies 100 emails.

Manually examine the 100 errors, and categorize them based on:

(i) What type of email it is — pharma, replica, steal passwords, ...

(ii) What cues (features) you think would have helped the algorithm classify them correctly.

Pharma: 12

Replica/fake: 4

Steal passwords: 53

Other: 31

Deliberate misspellings:
(m0rgage, med1cine, etc.)

Unusual email routing:

Unusual (spamming) punctuation:

- Look at failed cases, see if any features are in common. (manually)
- Quick and dirty implementation can tease out these errors / issues
- Also: have an accuracy / error number you can work to improve
  - Should we use stemming software? Try it, see if it works. Measure results. (Cross validation error)

The importance of numerical evaluation

Should discount/discounts/discounted/discounting be treated as the same word?

Can use "stemming" software (E.g. "Porter stemmer")

universe/university.

Error analysis may not be helpful for deciding if this is likely to improve performance. Only solution is to try it and see if it works.

Need numerical evaluation (e.g., cross validation error) of algorithm's performance with and without stemming.

Without stemming: 5% error With stemming: 3% error

Distinguish upper vs. lower case (Mom/mom): 3.2%

-

Skewed data -- when many more examples from one class than the other

Cancer classification example

Train logistic regression model $h_\theta(x)$. ($y = 1$ if cancer, $y = 0$ otherwise)

Find that you got 1% error on test set.

(99% correct diagnoses)

Only 0.50% of patients have cancer.

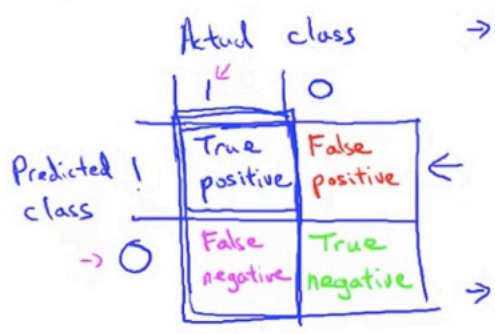
```
function y = predictCancer(x)
  y = 0; %ignore x!
return
```

0.5% error

-
- Can simply guess “no cancer” and be very accurate.
- Precision / Recall.
  - Precision: What fraction of

Precision/Recall

$y = 1$ in presence of rare class that we want to detect



→ **Precision**
(Of all patients where we predicted $y = 1$, what fraction actually has cancer?)

$$\frac{\text{True positives}}{\#\text{predicted positive}} = \frac{\text{True positive}}{\text{True pos + False pos}}$$

→ **Recall**
(Of all patients that actually have cancer, what fraction did we correctly detect as having cancer?)

$$\frac{\text{True positives}}{\#\text{actual positives}} = \frac{\text{True positives}}{\text{True pos + FN}}$$

  -
  - Precision: How accurate are your diagnoses?
  - Recall: How many cancer people get detected?
    - If you always guess y=0, then you have recall=0. Don’t detect anyone who does.
  - Aha: think about the fraction the true positives are of the row or column.
- Trading precision and recall
  - Can raise the bar from 50% to 70% confidence. But lower recall, predicting y=1 for a smaller number of people.
  - If you want to avoid false negatives, lower the bar. Be more noisy.

Trading off precision and recall

→ Logistic regression: $0 \le h_\theta(x) \le 1$

Predict 1 if $h_\theta(x) \ge 0.5$ 0.7 0.9 0.3

Predict 0 if $h_\theta(x) < 0.5$ 0.7 0.9 0.3

→ Suppose we want to predict $y = 1$ (cancer) only if very confident.

→ Higher precision, lower recall.

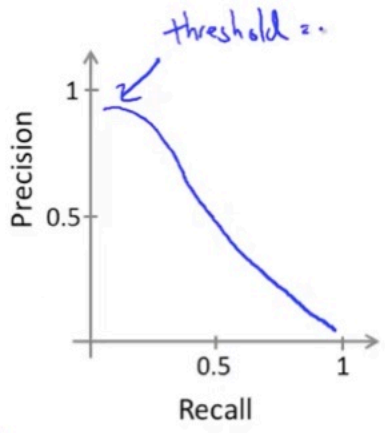
→ Suppose we want to avoid missing too many cases of cancer (avoid false negatives).

→ Higher recall, lower precision.

More generally: Predict 1 if $h_\theta(x) \ge$ threshold.

→ precision = $\frac{\text{true positives}}{\text{no. of predicted positive}}$

→ recall = $\frac{\text{true positives}}{\text{no. of actual positive}}$



-
- Ah… question is always, can we pick the threshold automatically?
  - Went from a single error metric to 2 numbers (precision/recall). Makes decisions slow.
    - Averaging the two isn’t good. Because y=0 automatic can give you high precision.
    - Highest -- not good either.

F_1 Score (F score)

How to compare precision/recall numbers?

	Precision(P)	Recall (R)	Average	F_1 Score
Algorithm 1	0.5	0.4	0.45	0.444
Algorithm 2	0.7	0.1	0.4	0.175
Algorithm 3	0.02	1.0	0.51	0.0392

Predict y=1 all the time

Average: $\frac{P+R}{2}$

F_1 Score: $2\frac{PR}{P+R}$

- Combines precision and recall, F-score does well. If one element is low, brings down the F-score. Perfect score is 2*(1*1)/(1 + 1) = 1.0

Training data

- Seems like all algos improve with more data.

Designing a high accuracy learning system

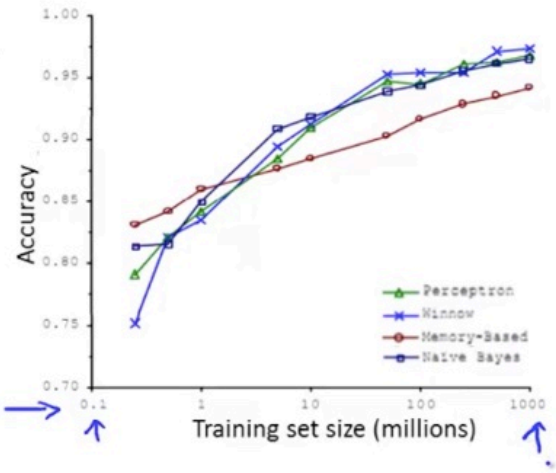
E.g. Classify between confusable words.

{to, two, too} {then, than}

For breakfast I ate two eggs.

Algorithms

- Perceptron (Logistic regression)
- Winnow
- Memory-based
- Naïve Bayes



- When true?
- Test: Could a human make a prediction with the same x inputs?
- Idea: having low-bias algo, with TONS of training data, can't fit to all of that.

Large data rationale

Use a learning algorithm with many parameters (e.g. logistic regression/linear regression with many features; neural network with many hidden units). low bias algorithms.

$J_{train}(\theta)$ will be small.

Use a very large training set (unlikely to overfit) low variance

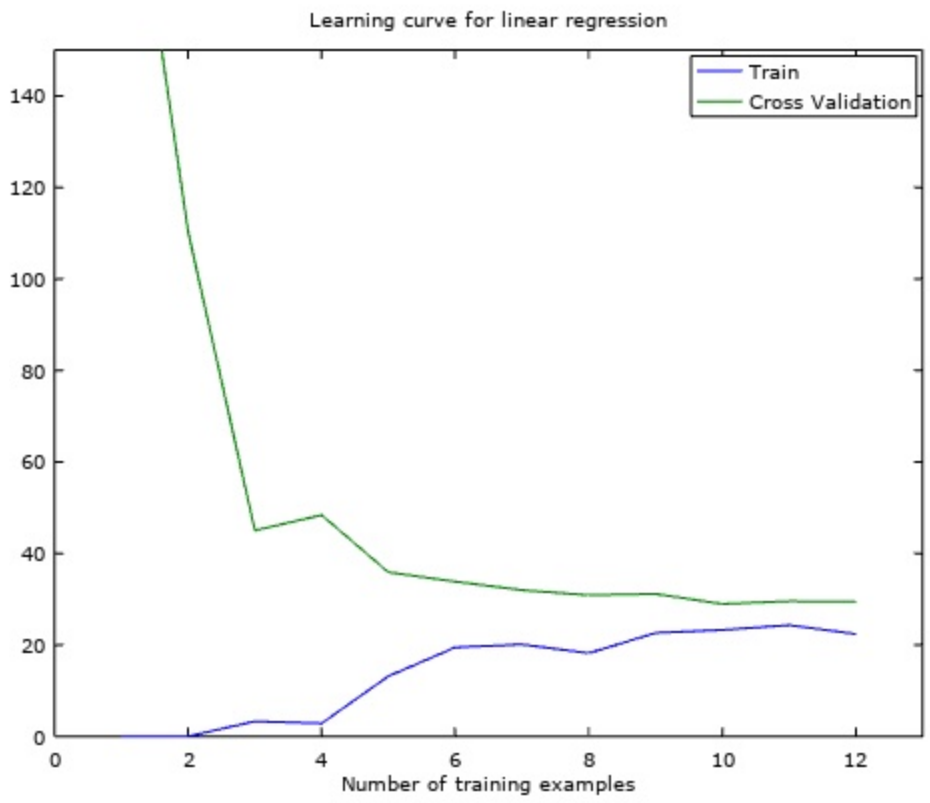
$J_{train}(\theta) \approx J_{test}(\theta)$

$J_{test}(\theta)$ will be small

- The various properties fight it out. More data fights more variance.

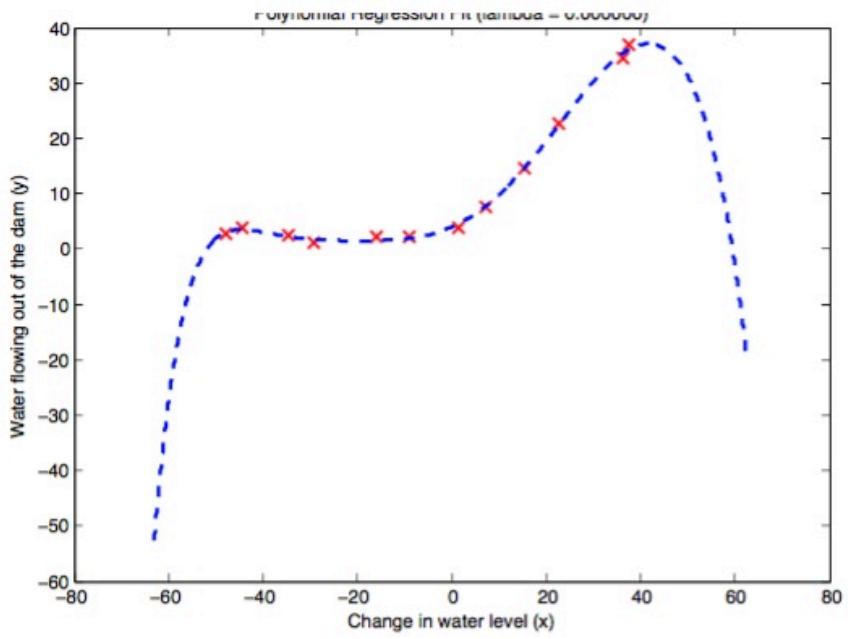
Homework

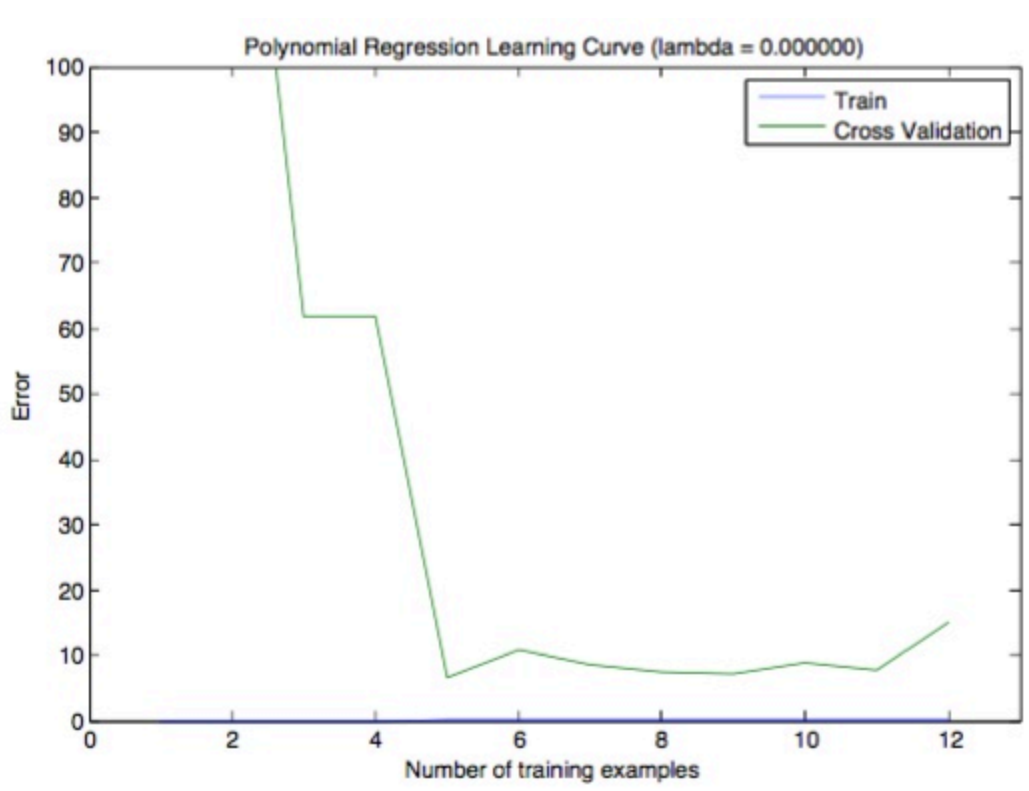
- Gotchas around the row/column ordering, whether the bias term (1) is already included, etc.. Sometimes when summing things get collapsed in weird ways. Have to inspect and double check.
- Summary: An important concept in machine learning is the bias-variance tradeoff. Models with high bias are not complex enough for the data and tend to underfit, while models with high variance overfit to the training data.
- Yep, more gotchas on the size of various arrays. LinearRegCostFunction returns J (the cost) as an array.



As we give it more training data, the cross validation error improves. But it stays high, so the model is too simple. (Again, want an intuition for these graphs…)

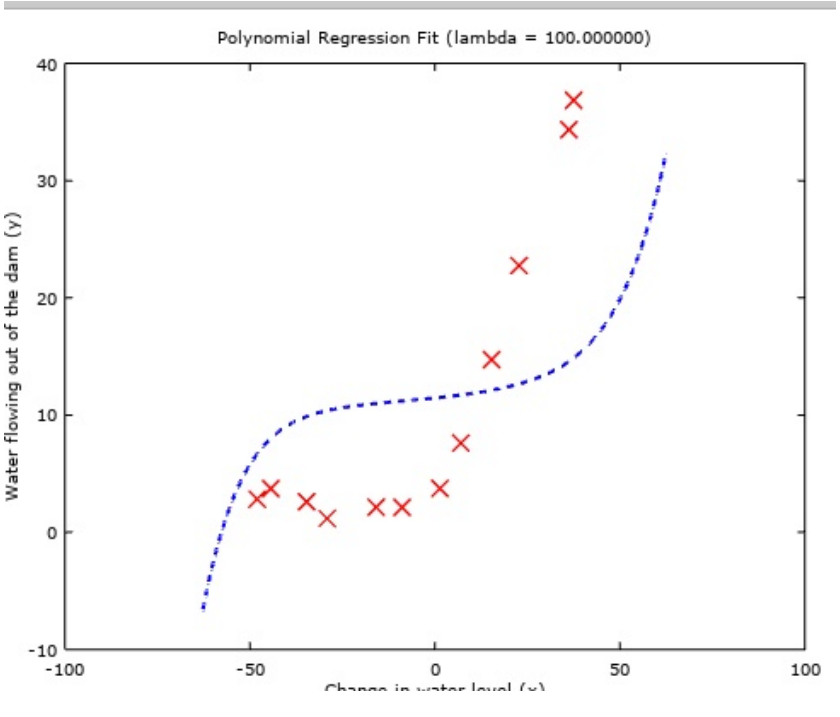
- Aha: Again, we can “cheat” by making new features (waterLevel)^2, (waterLevel)^3, etc. and being “linear” in these.
    - Remember to normalize these values because they can get enormous (taking waterLevel to the 8th power…). Normalizing is plotting it with a mean of 0 & standard deviation
- Polynomial can overfit (even dropping at extremes)





Ah! Look at the **gap/difference** between training error and CV error. It should be similar. The difference means we have overfit on the training (little error) but cross validation error is large. We can’t generalize.

We can vary lambda (regularization) to correct, but too much punishment (lambda = 100) means we underfit again. A moderate lambda (1) works well.



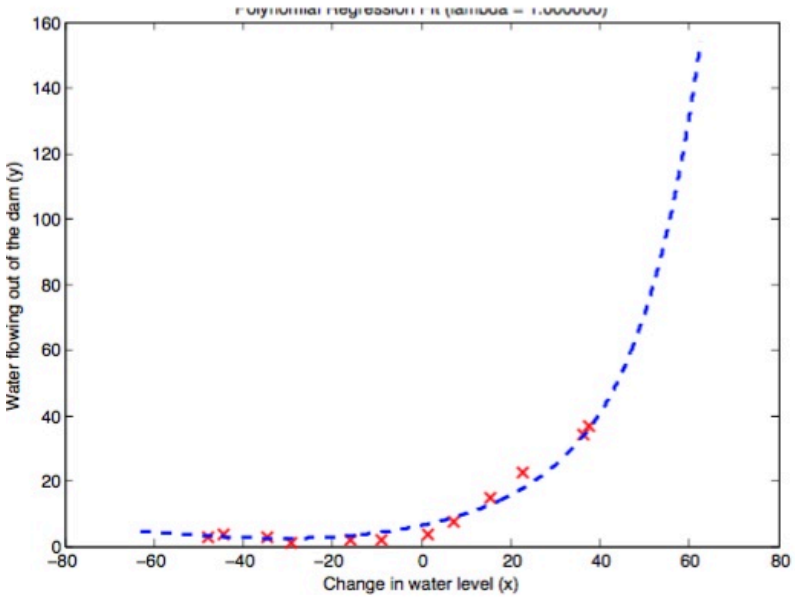
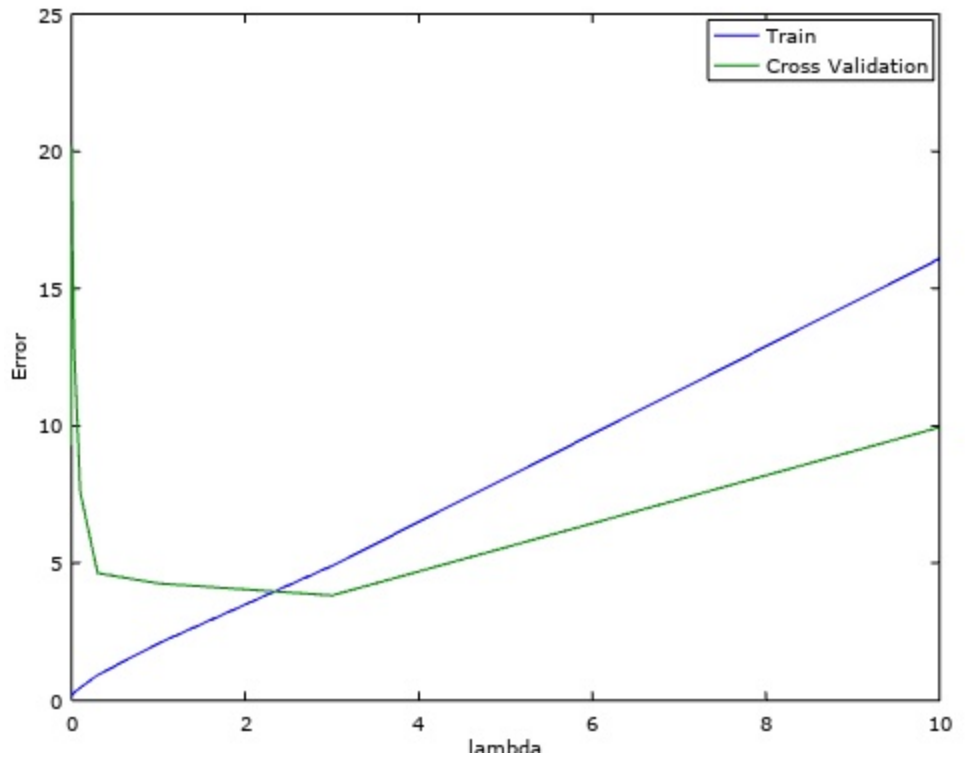


Figure 6: Polynomial fit, $\lambda = 1$

We can find the right lambda by using the cross validation set (in essence, we are fitting lambda to the CV set).



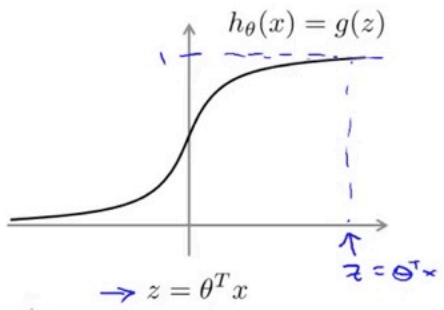
- Intuition: Each new set of data (test set, validation set) can answer ONE question about what parameters to use. (which theta, which lambda).
- Also: you should randomize the data used for training and validation sets. Run 50 trials, take average error (for example).

Week 7 - Support Vector Machines

Review: Logistic regression helps get a number from 0 to 1. To get y=1, we need z (weighted input) to be way to the right

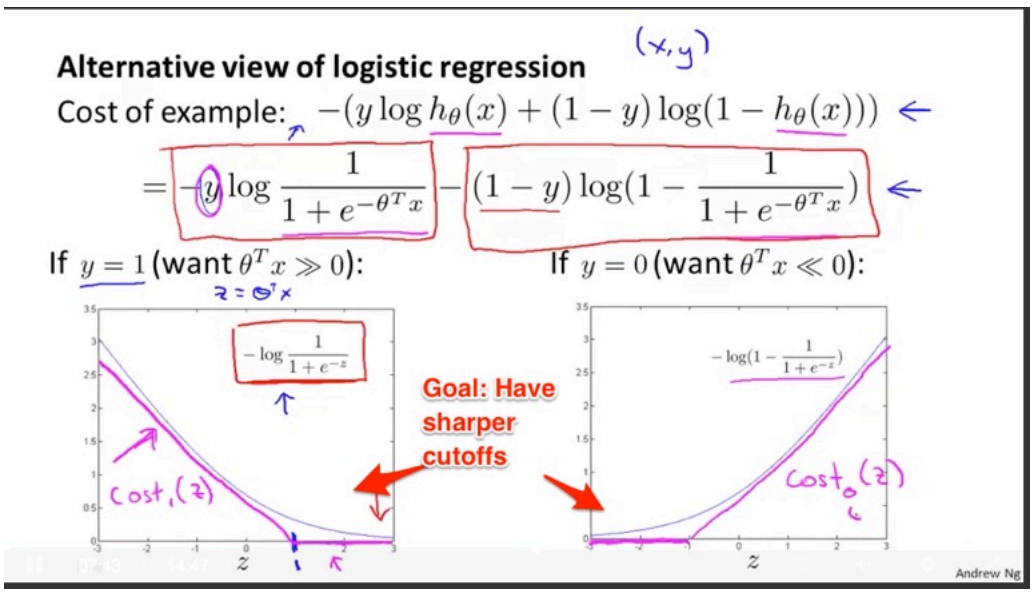
Alternative view of logistic regression

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

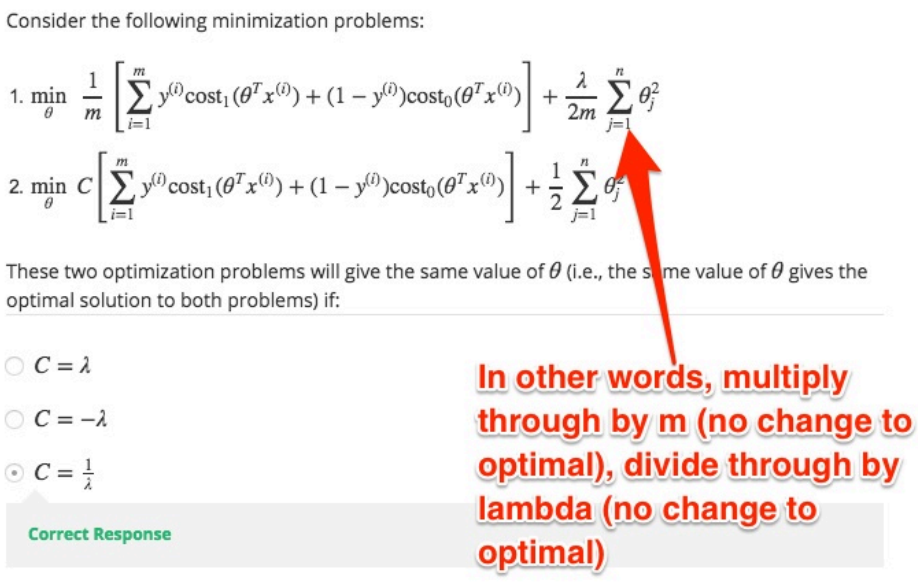


If $y = 1$, we want $h_\theta(x) \approx 1$, $\theta^T x \gg 0$
If $y = 0$, we want $h_\theta(x) \approx 0$, $\theta^T x \ll 0$

- Each training example contributes to cost
- When y = 1, we want the cost function to be 0. Have a cutoff (once z=1) you are perfect. In logistic regression you really can't get zero cost easily.



- Use a new cost function cost_1(z) and cost_0(z) which have a sharper cutoff (are these functions differentiable?). Swap these out in the logistic regression cost.
- Another difference: scale the cost function (relative to B, the total number of parameters used). Just a different choice on how you want to run the scaling.



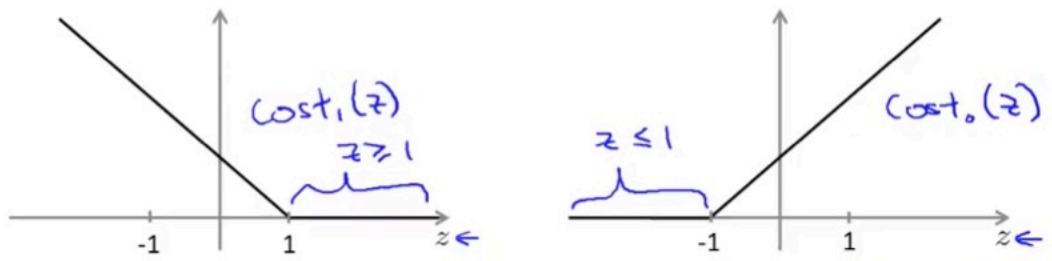
-
- Difference: Support Vector Machine outputs a hypothesis (1 or 0), not a prediction probability.

SVM described as "Large Margin Classifiers"

- You need to be much bigger than 0 to push a classification. (>= 1, not just >= 0) to nudge this.
- **Gotcha: A few days later, I've already forgotten why it's called a "Support Vector Machine". What gives?**
  - "A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane."
    - Yes! That's what it is. A hyperplane separation. Why machine?
    - Ah: The closest data points are the "support vectors" that help "support" the boundary of the hyperplane. You could throw away the further data points and end up with the same hyperline. The "machine" is for machine learning (I guess). It's a cruddy name though.
    - https://onionesquereality.wordpress.com/2009/03/22/why-are-support-vectors-machines-called-so/

Support Vector Machine

$$\min_\theta C\sum_{i=1}^{m}\left[y^{(i)}cost_1(\theta^T x^{(i)}) + (1-y^{(i)})cost_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$



→ If $y = 1$, we want $\theta^T x \geq 1$ (not just ≥ 0)

→ If $y = 0$, we want $\theta^T x \leq -1$ (not just < 0)

SVM Decision Boundary

$$\min_\theta C\left[\sum_{i=1}^{m}\left[y^{(i)}cost_1(\theta^T x^{(i)}) + (1-y^{(i)})cost_0(\theta^T x^{(i)})\right]\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

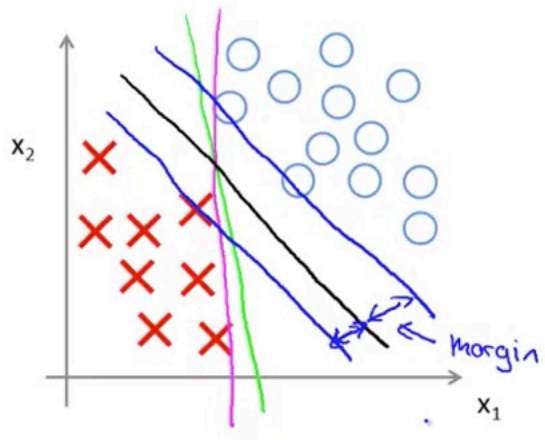
Whenever $y^{(i)} = 1$:

$\theta^T x^{(i)} \geq 1$

Whenever $y^{(i)} = 0$:

$\theta^T x^{(i)} \leq -1$

SVM Decision Boundary: Linearly separable case

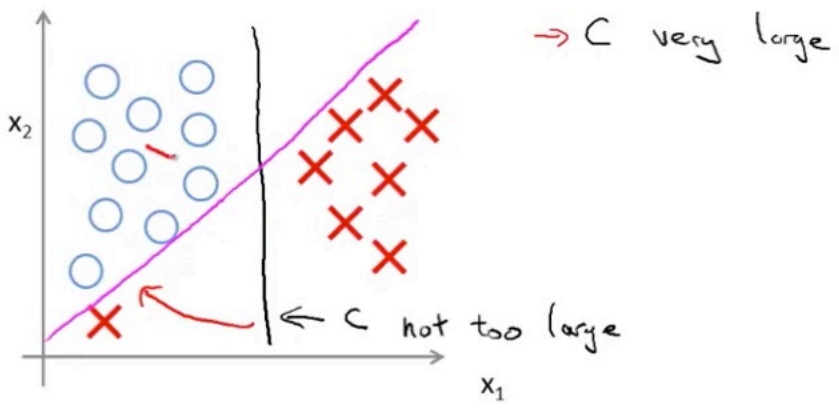


Large margin classifier

SVM will create the black boundary that separates -- larger distance from training examples.

- Why does this happen? Hrm.
- Ah! Ok. Your distance from the margin determines how you are classified. But only get a 1 or 0 if your distance is >= 1. Therefore you find a boundary that “leaves room” for the samples to be separated nicely.
- However, outliers can change the decision boundary

Large margin classifier in presence of outliers

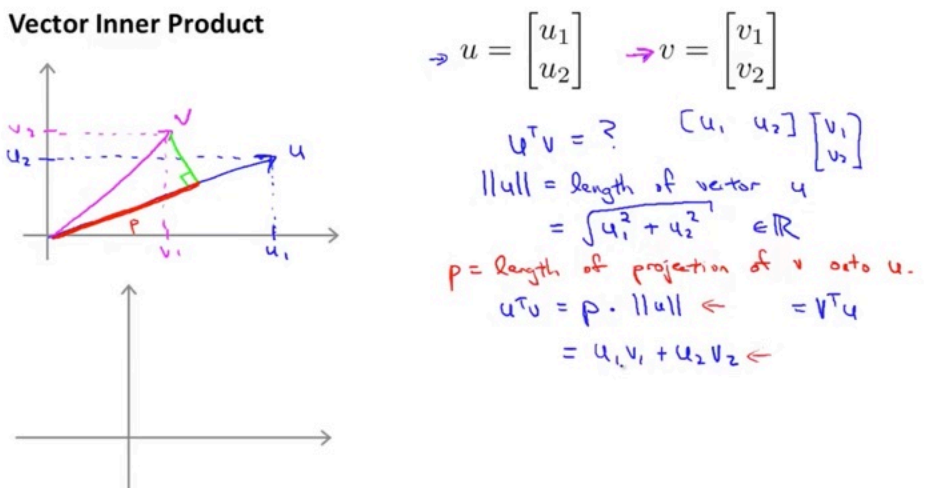


- If C is too large, then it gets sensitive to outliers. C is like 1/lambda. Lambda punishes you for parameters, C punishes you for the cost being off.

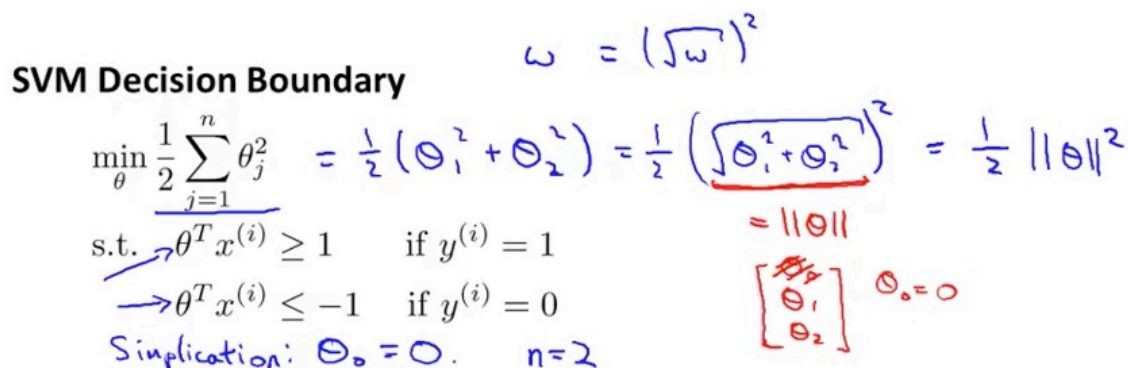
Math behind SVMs

Vector inner product: in your head, think “similarity percentage”.

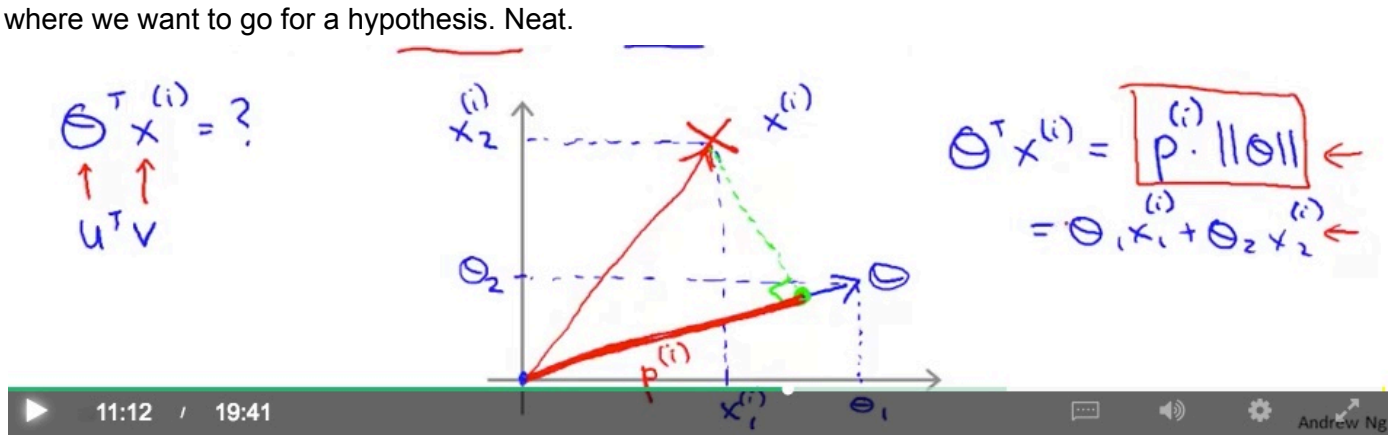
- Remember that u^t u = p * |u|. Remember that u^t u is “running the operation on the same data”



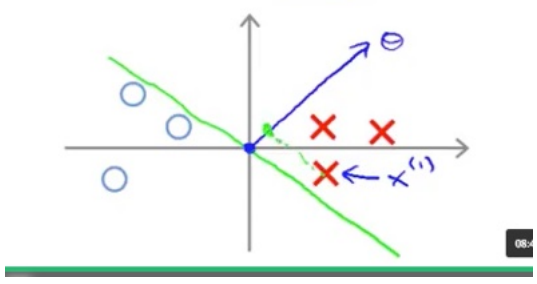
- Assume only 2 params. The regularization term (theta1^2 + theta2^2) can be reached by squaring the length of the theta parameter vector.



- So now… what is theta^t * x_i ?
  - We are projecting the vector on our theta parameter vector
    - To have the result > 1, means theta and x_i are pointing in the same direction! So theta is really pointing to where we want to go for a hypothesis. Neat.



    - Prediction: So, the intuition is theta determines a line/plane, points perpendicular to it. You get rewarded/punished based on how you point in this way too.
    - Yep!!! Theta is a vector perpendicular to the boundary.



    - The projections will be small. But the goal is to have large values (>=1) to help separate.

SVM Decision Boundary

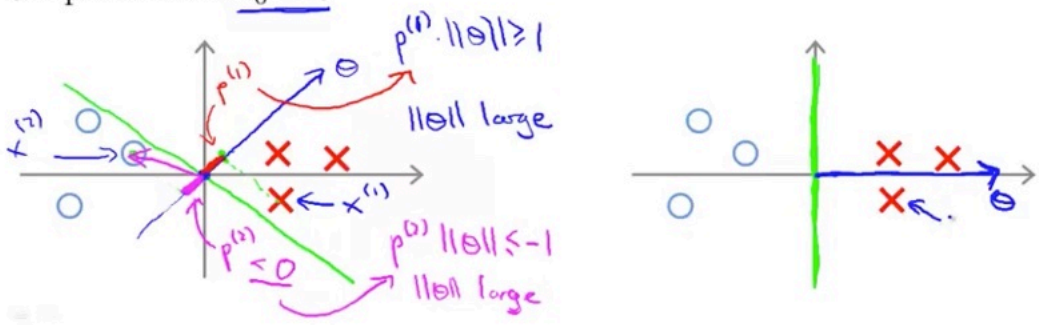
$$\min_\theta \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \quad = \frac{1}{2}\|\theta\|^2$$

s.t. $p^{(i)} \cdot \|\theta\| \geq 1$ if $y^{(i)} = 1$

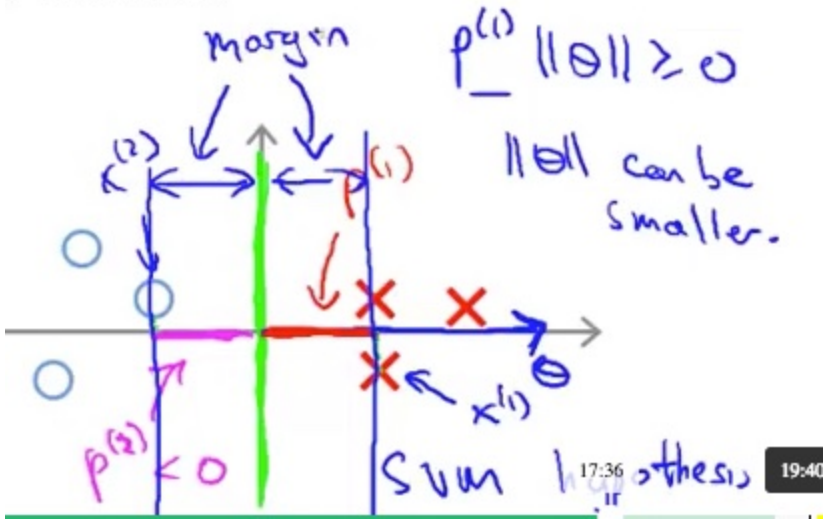
$p^{(i)} \cdot \|\theta\| \leq -1$ if $y^{(i)} = 1$

where $p^{(i)}$ is the projection of $x^{(i)}$ onto the vector θ.

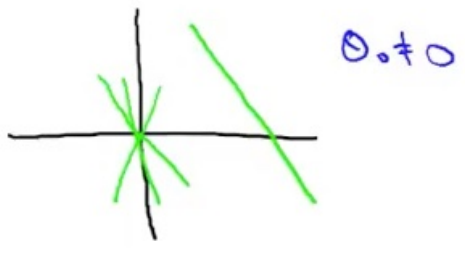
Simplification: $\theta_0 = 0$



- In other words, theta parameters can be smaller because the projections are larger. Easier to have p (the projection) be larger instead of theta (which costs us)



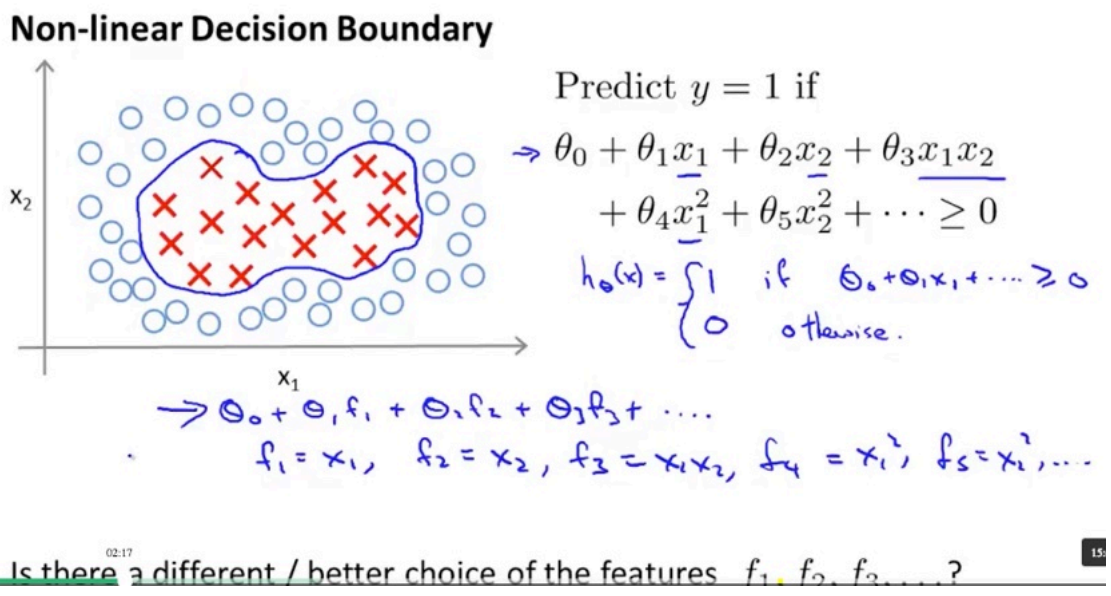
- If we simplify and pick theta_0 = 0, then we only have decision boundaries through origin (no translation of the line)



- All the intuitions about lines, planes, offsets, scaling, etc. come back. Want to have a solid understanding here.

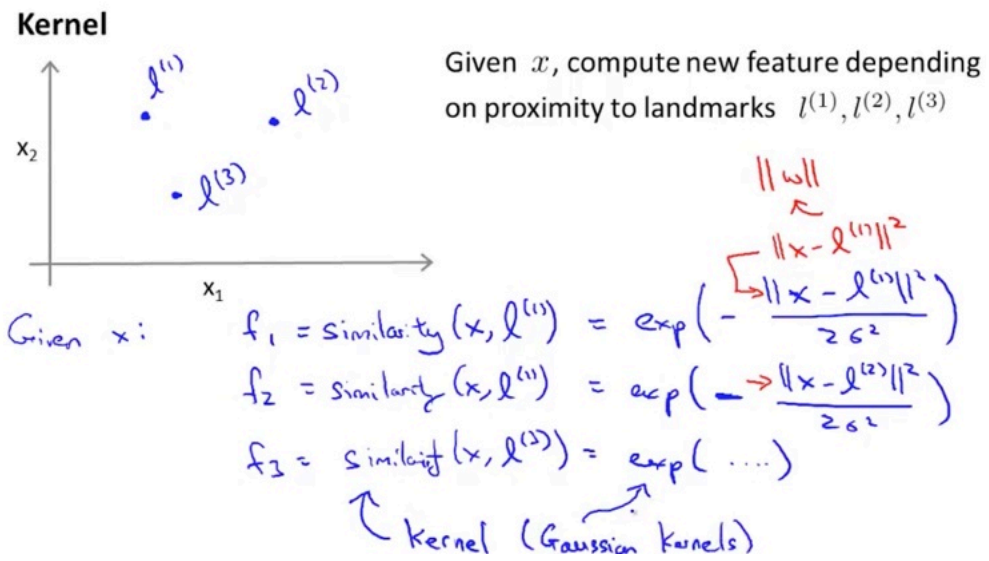
Kernels

Complex decision boundary: use many polynomial features (features f1, f2, f3… which are the combinatorial terms for x1, x2, x1*x2, and so on)

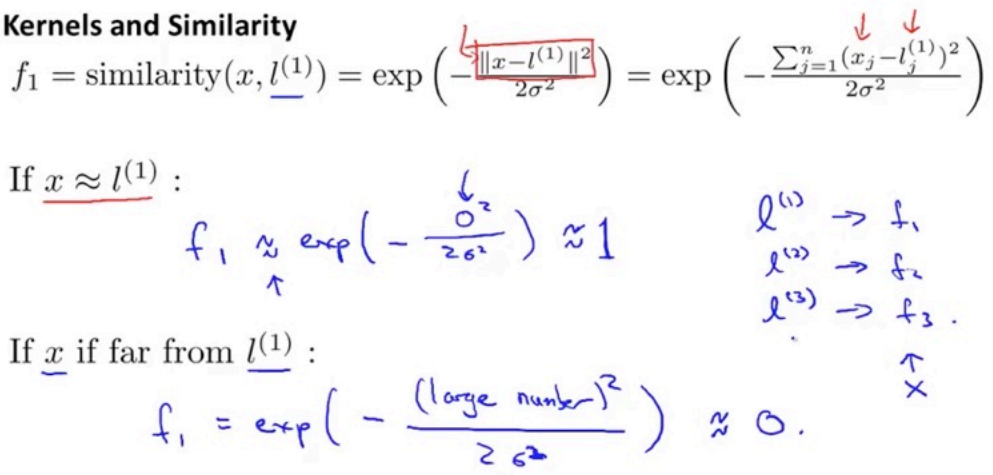


- Can we pick different features?

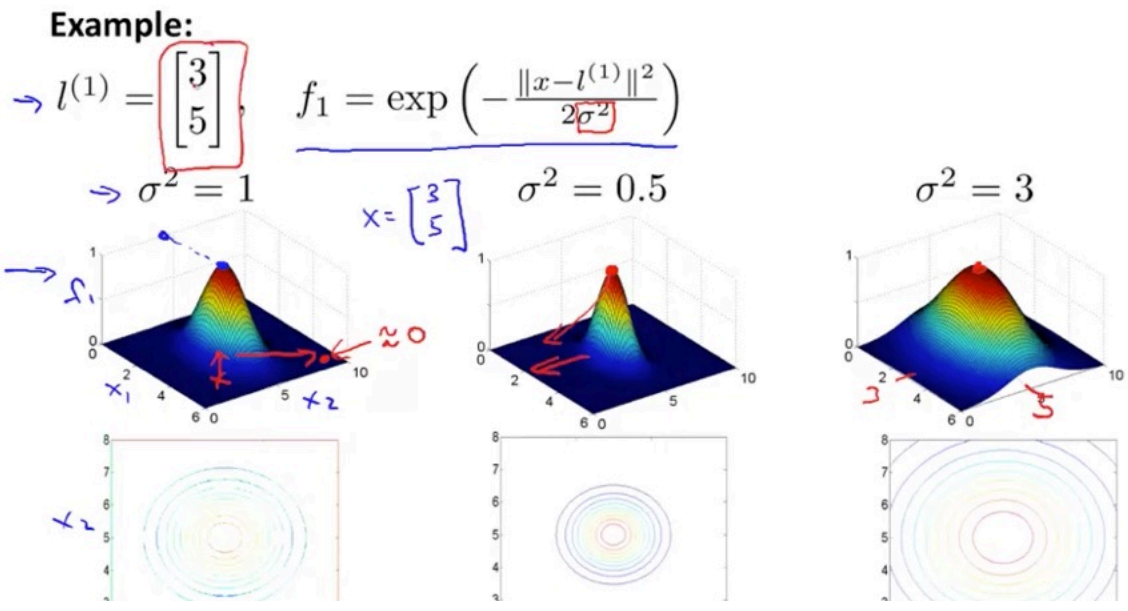
- Idea: have distance to various landmarks (l1, l2, l3…).



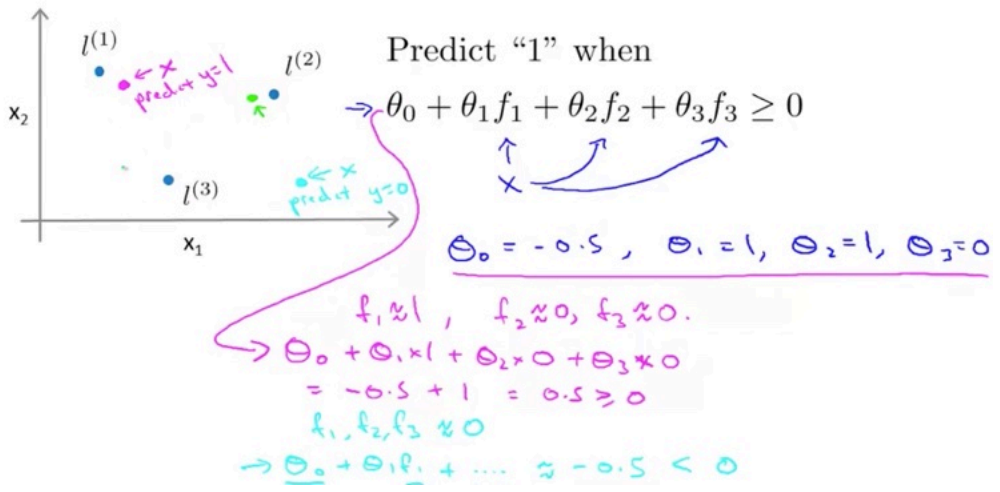
- Kernel ~ similarity function. What are we using to compute the feature. Here, a Gaussian (normal) distribution… (0).
  - If close to landmark, then we get 1.
  - If far from landmark, we get 0.



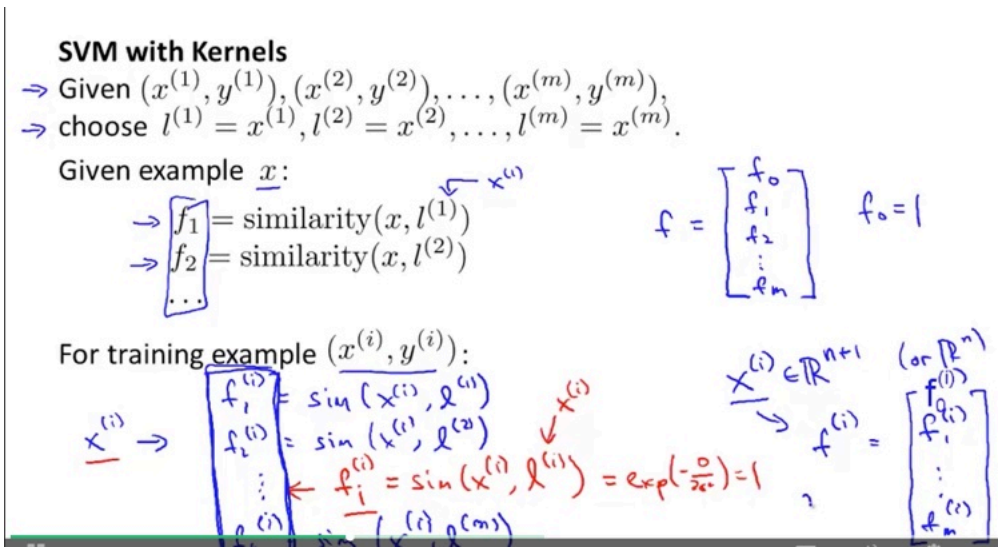
- The feature is a “spike” as you get closer to the feature. There is a sigma squared parameter to decide how sharp/steep the cutoff is.



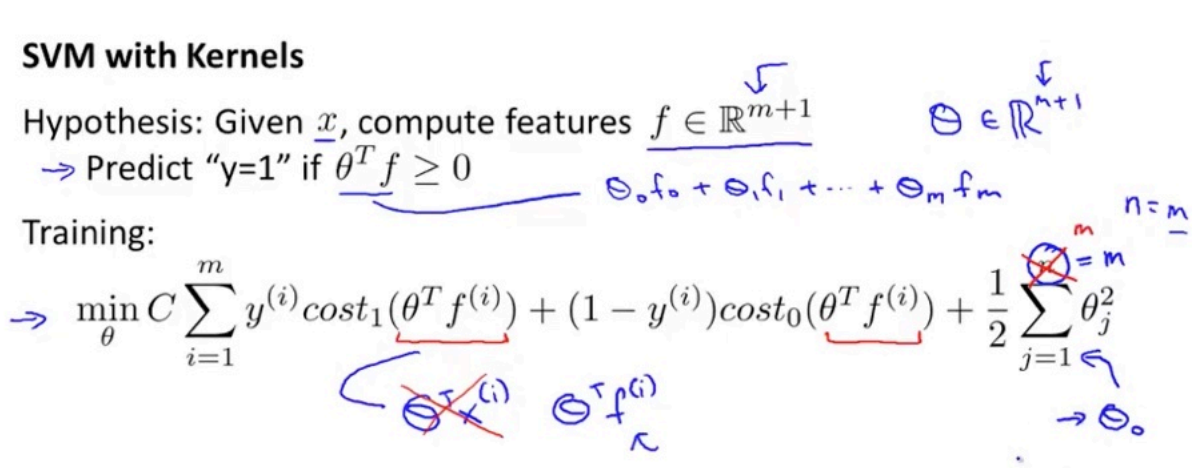
- Weight the distance to each landmark. (theta1, theta2, theta3).
  - If far, get a 0
  - So the closer you get to the various features, get 1 for f1, f2, f3… then you get >= 0 for the total.

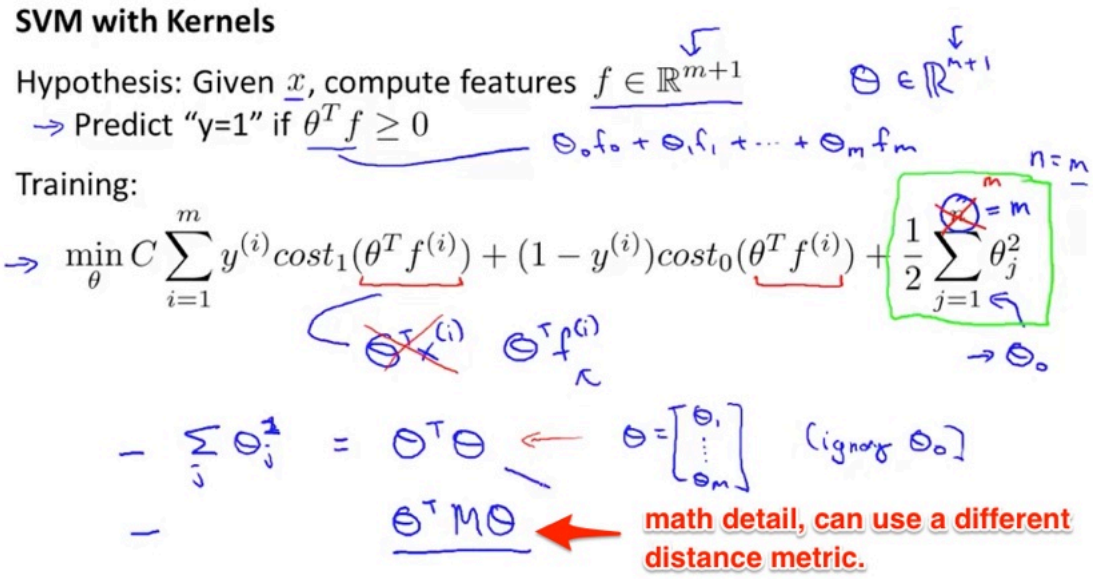


- So… how to find landmarks? (they define the boundary).
  - Ah! We set landmarks for the existing training examples. Neat. Assume a “blob”. The data sets the boundaries. Be close to the others. Measure how close we are to the existing training examples.
  - Get the similarity to each other sample. Wow. so m=5000, then we have 5000 features.

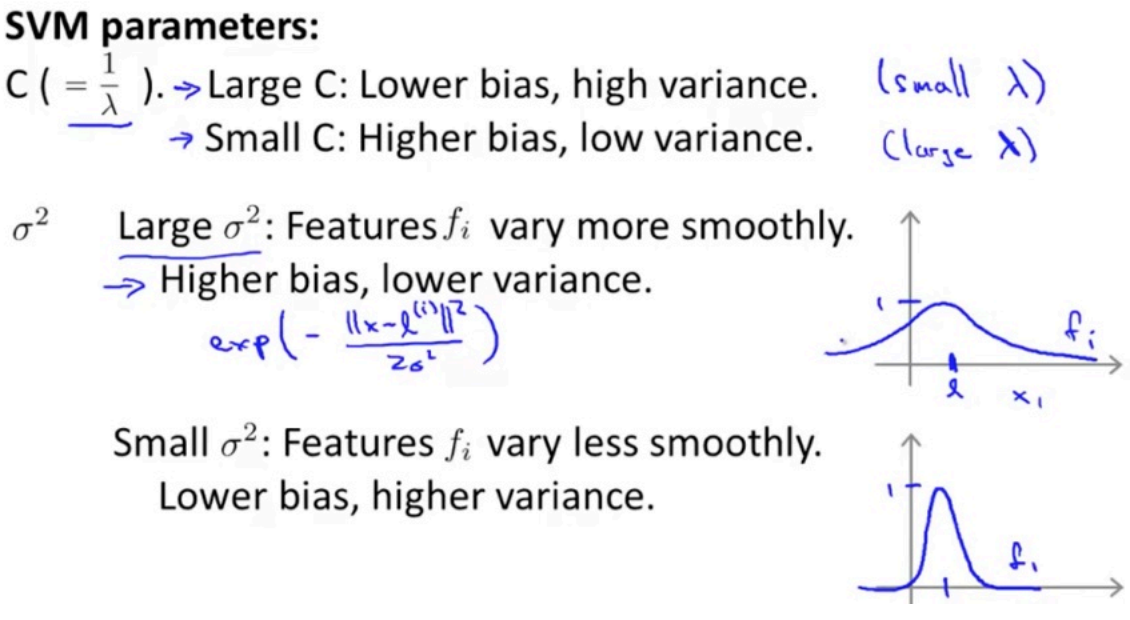
- 

- And have f_0 = 1 (everything is 100% similar to the “null point”)
- Theta is m+1 dimensional as well.
  - And we compute theta using the SVM algorithms. Instead of theta on the original features, we have theta on the f1...fn features.
  - The number of features matches the number of samples. Huge!

- 

- 

- Using eigenvector/value to find theta^t * theta to make the computation simpler (otherwise it’s huge for M=10,000).

- 

- If you overfit… punish cost a bit less (smaller C). Smooth it out a bit (larger sigma sq). Sigma for “smear” or “smooth” (less spiky).

Kernel choice:

- No kernel (linear kernel). You are just using features directly. Linear classifier.
    - If many features (n), low samples (m) -- just use the features you have. Don't want to overfit on training samples.
    - If many samples, not many features (n), use the samples to help shape what you want to use.
- Choosing the right sigma^2.

Use SVM software package (e.g. liblinear, libsvm, …) to solve for parameters θ.

Need to specify:

→ Choice of parameter C.

Choice of kernel (similarity function):

E.g. No kernel ("linear kernel")

Predict "y = 1" if $\theta^T x \geq 0$

$\theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n \geq 0$

→ n large, m small

$x \in \mathbb{R}^{n+1}$

Gaussian kernel:

$f_i = \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$, where $l^{(i)} = x^{(i)}$.

Need to choose σ^2.

$x \in \mathbb{R}^n$, n small and/or m large

- Sometimes you provide the Gaussian kernel yourself. Want to do feature scaling before using the kernel. Don't let features dominate just based on their value (bedrooms vs. sq feet).

Kernel (similarity) functions:

```
function f = kernel(x1,x2)
    f = exp( -|| x1 - x2 ||^2 / (2σ^2) )
return
```

$x^{(i)}$, $l^{(j)} = x^{(j)}$

$x \to \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}$

→ Note: Do perform feature scaling before using the Gaussian kernel.

$||x - l||^2$

$v = x - l$

$||v||^2 = v_1^2 + v_2^2 + \cdots + v_n^2$

$= (x_1 - l_1)^2 + (x_2 - l_2)^2 + \cdots + (x_n - l_n)^2$

1000 feet^2 vs. 1-5 bedrooms

$x \in \mathbb{R}^n$

- Kernels must meet conditions ("Mercer's Theorem") for numerical computation reasons.

Other choices of kernel

Note: Not all similarity functions $\text{similarity}(x, l)$ make valid kernels.

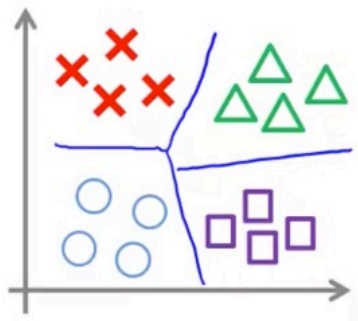
→ (Need to satisfy technical condition called "Mercer's Theorem" to make sure SVM packages' optimizations run correctly, and do not diverge).

Many off-the-shelf kernels available:

- Polynomial kernel: $k(x, l) = (x^T l)^2$, $(x^T l)^3$, $(x^T l + 1)^3$, $(x^T l + 5)^4$, $(x^T l + \text{constant})^{\text{degree}}$
- More esoteric: String kernel, chi-square kernel, histogram intersection kernel, …

- Pick your parameters (C, sigma, etc.) based on what has the best in cross-validation.
- Multi-class -- can do one vs. all.

Multi-class classification



$y \in \{1, 2, 3, \ldots, K\}$

Many SVM packages already have built-in multi-class classification functionality.

→ Otherwise, use one-vs.-all method. (Train K SVMs, one to distinguish $y = i$ from the rest, for $i = 1, 2, \ldots, K$), get $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(K)}$

Pick class i with largest $(\theta^{(i)})^T x$

- Logistic regression vs. SVM. Depends on features and data.
  - More features, logistic regression (linear function is fine)
  - If up to 1000 features, more training examples (10,000): SVM with Gaussian kernel.
  - If n is small, m is large (m >= 50k)
    - Add features, use logistic regression

Logistic regression vs. SVMs

n = number of features ($x \in \mathbb{R}^{n+1}$), m = number of training examples

→ If n is large (relative to m): (e.g. $n \geq m$, $n = 10{,}000$, $m = 10 \cdots 1000$)

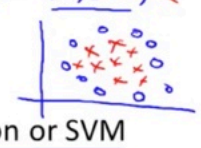
→ Use logistic regression, or SVM without a kernel ("linear kernel")

→ If n is small, m is intermediate: ($n = 1\text{-}1000$, $m = 10\text{-}10{,}000$)

→ Use SVM with Gaussian kernel

If n is small, m is large: ($n = 1\text{-}1000$, $m = 50{,}000+$)

→ Create/add more features, then use logistic regression or SVM without a kernel



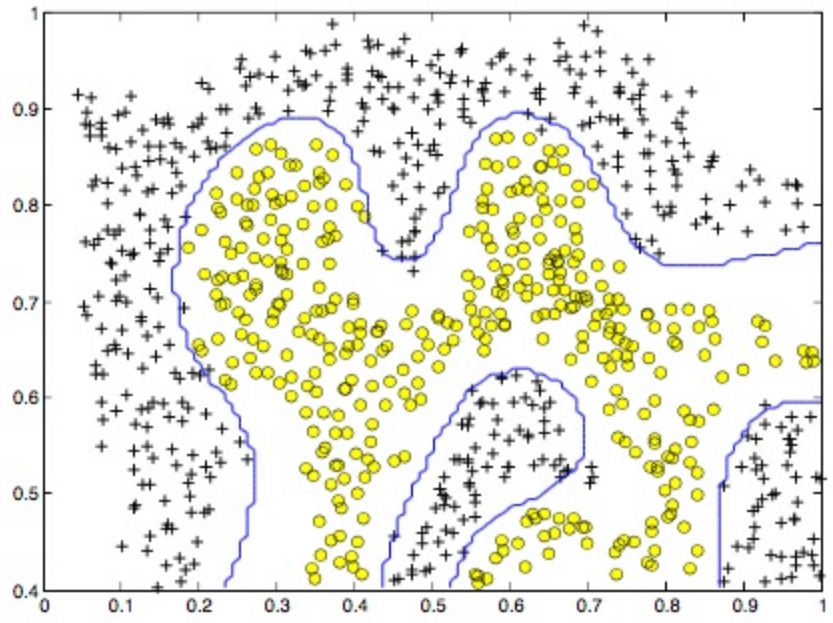
  - Neural network likely to work well for most of these settings, but may be
  - Hrm.. but isn't SVM just a superset? SVM without a kernel, maybe you can swap out various kernels.

Homework

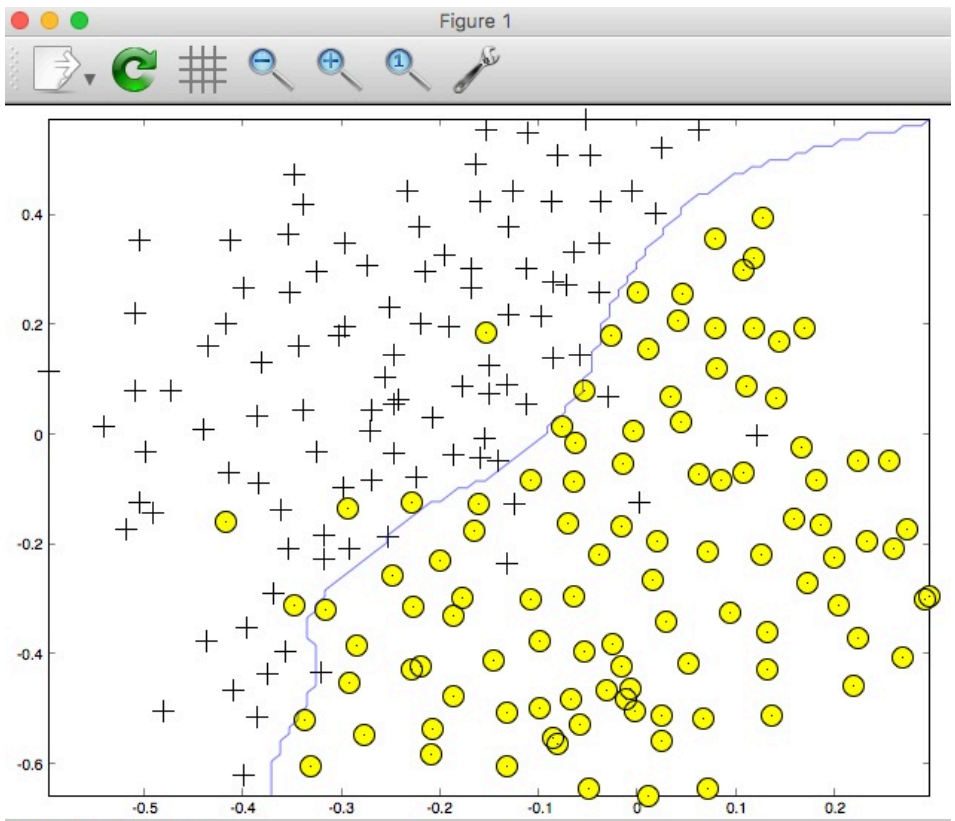
- C: intuitive definition, it's the penalty for misclassifying samples. (1/lambda, intuitively)
- Data "not linearly separable"
- Sigma (in a Gaussian kernel) seen as a "bandwidth" parameter. How close do samples need to be.

$$K_{gaussian}(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{k=1}^{n}(x_k^{(i)} - x_k^{(j)})^2}{2\sigma^2}\right)$$

  - Get the total difference and plug into the normal distribution… how many std deviations away are they?
    - If the difference is 0, then we get a e^0 = 1.



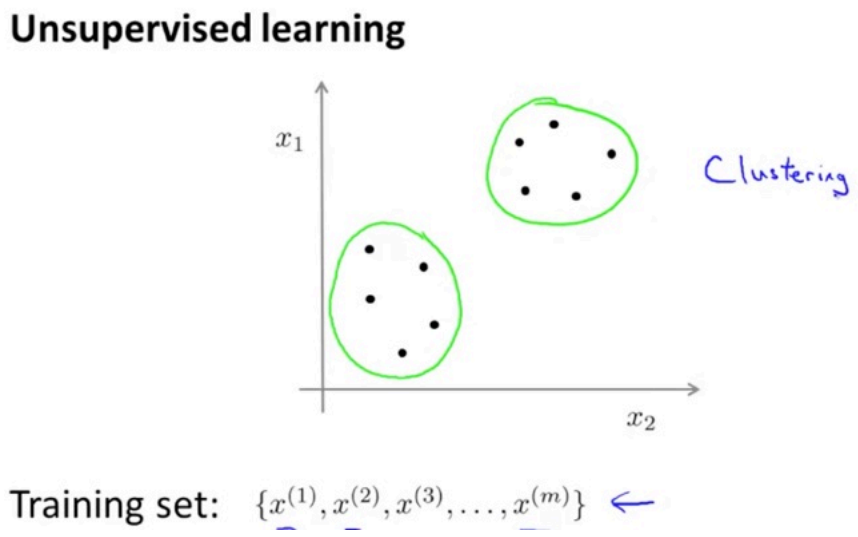
- Yay, I had a program that tried to fit C and sigma to the cross-validation set… (loop over [.01, .03… 1 3… 10, 30] for each). Here's the boundary I got:



- For email, have a word list. An email is a vector of possible words (0 if not there, 1 if there).
  - $$x = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^n.$$
- "you should see that the classifier gets a training accuracy of about 99.8% and a test accuracy of about 98.5%."
  - Remember the difference between the errors in the model and real-world performance.

Week 8 - Clustering

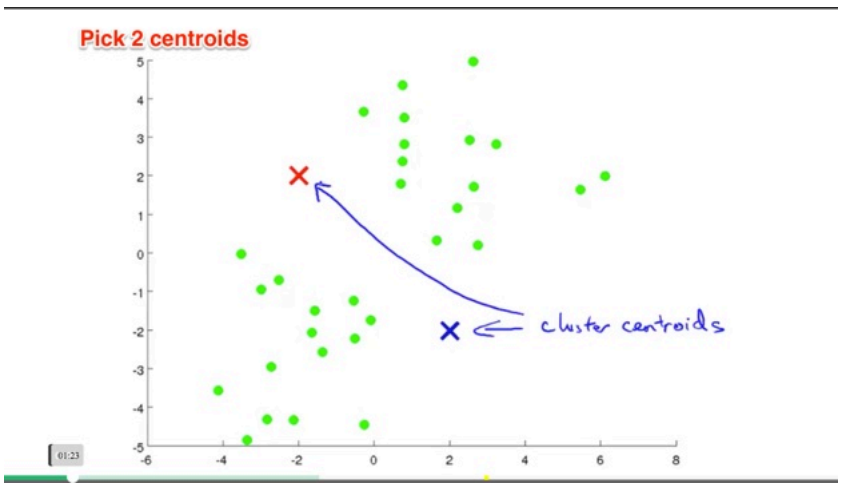
Unsupervised -- just find similar items. x(i) without a y(i) label. Goal to find structure inside.

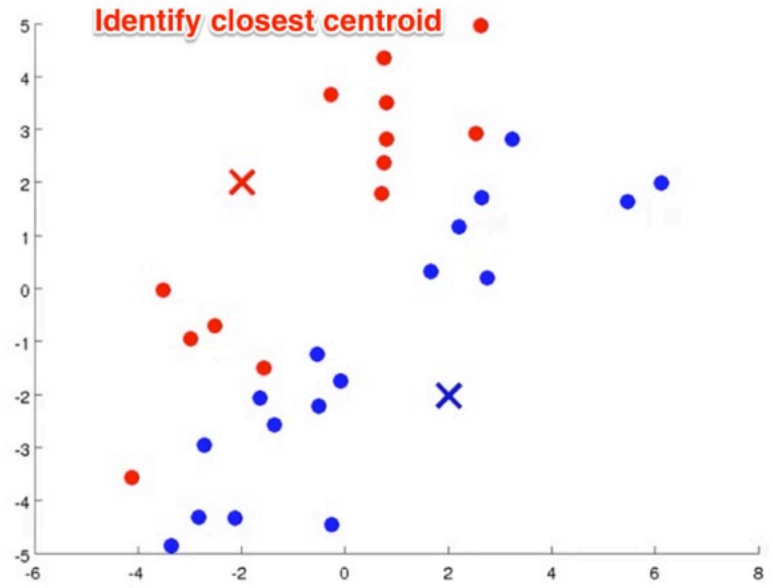


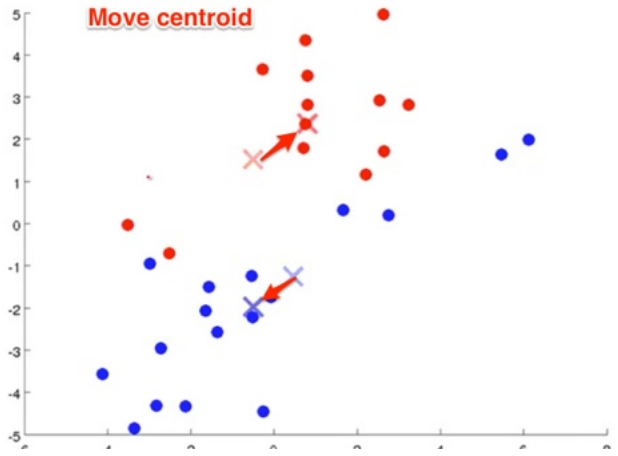
- Find structure in groups. Market segmentation, social network analysis (cliques), organize computing clusters

K-means

- Randomly initialize 2 cluster centroids
  - Cluster assignment: Find which cluster we're closest to. (Color red/blue)
  - Move centroid: Then, move to the average of the groups.







K-means algorithm

Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$

Repeat {

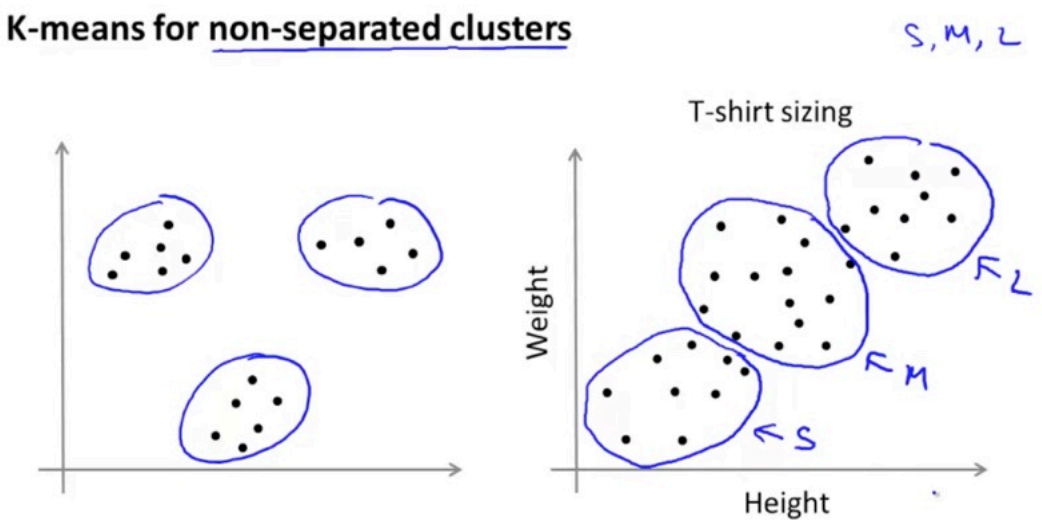
Cluster assignment step:
for i = 1 to m
$c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$

Move centroid:
for k = 1 to K
μ_k := average (mean) of points assigned to cluster k

}

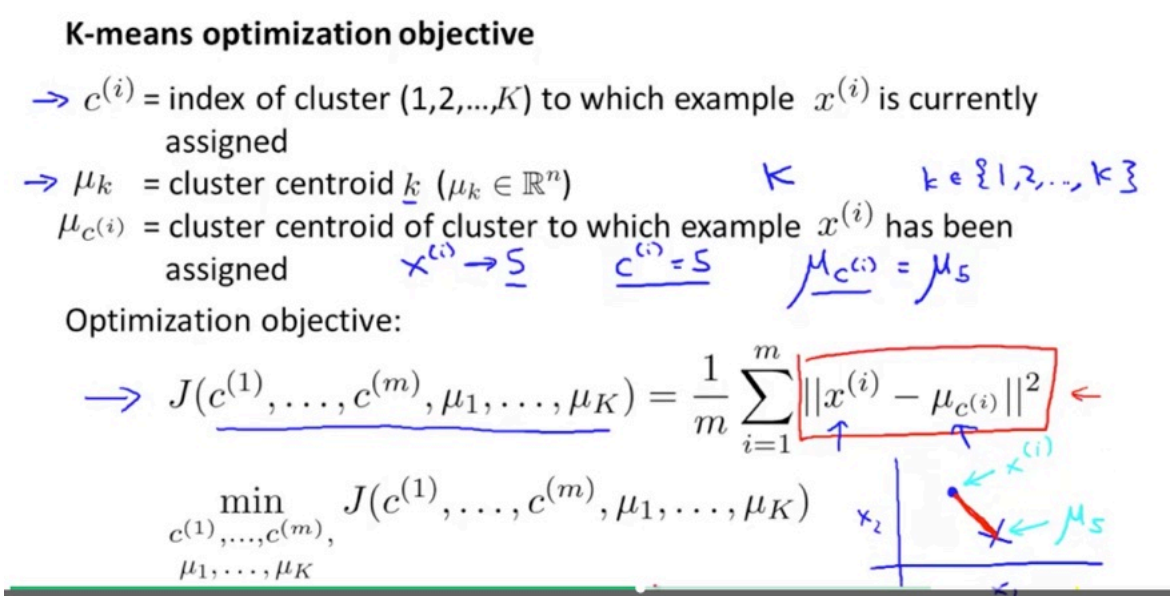
- Parameters: k (number of clusters - later, how to identify #)
- Drop x_0 = 1 (Aha: no “bias term” here, no concept of a default cluster, makes sense.)

For non-separated clusters - how to figure out where the small, medium, large size should be for your t-shirt sizes?

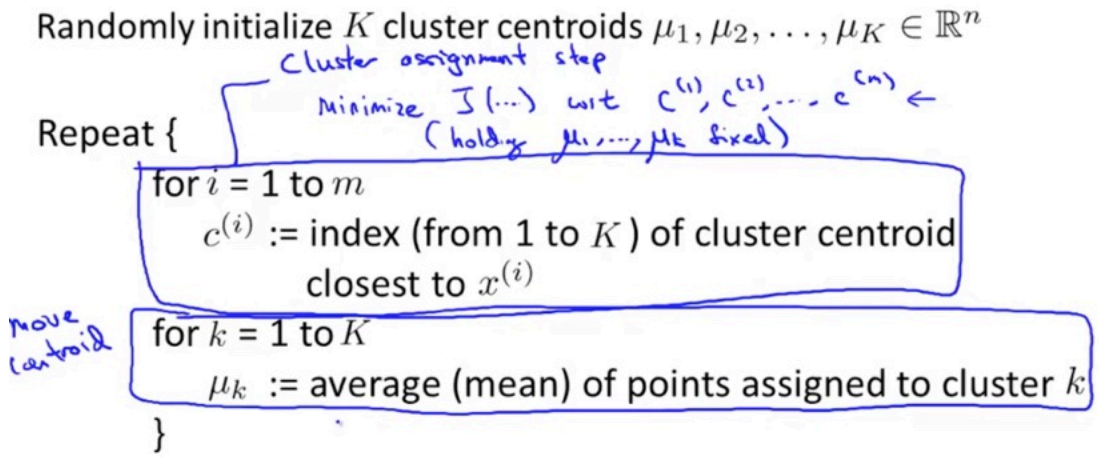


- Application: Market segmentation

Optimization Objective for K-means

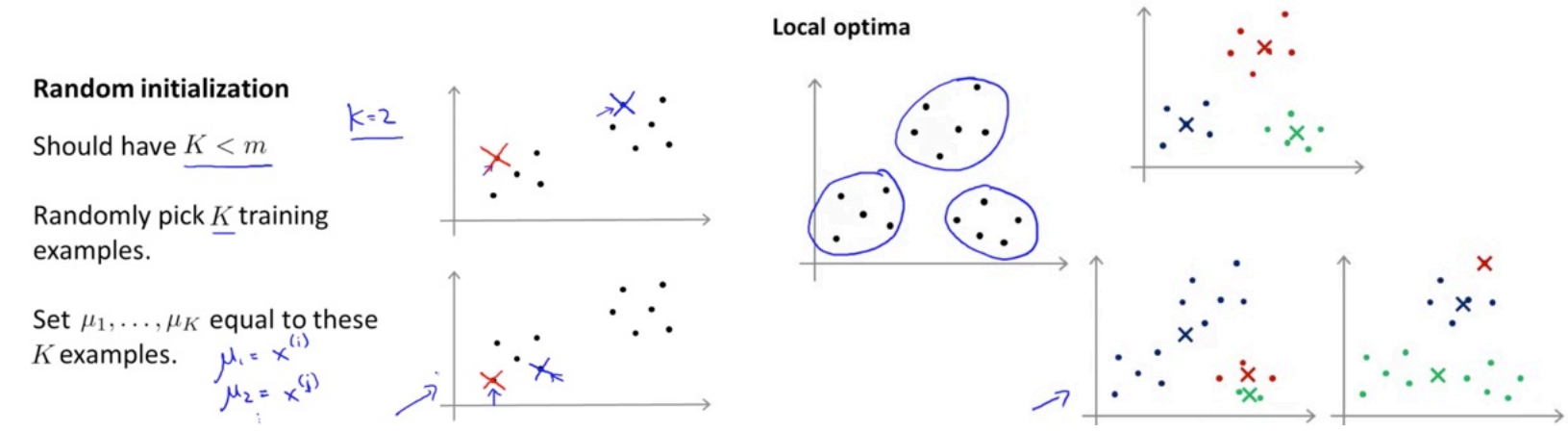


- Goal: minimize total distance from each sample from its assigned cluster. Cost function sometimes called “distortion”



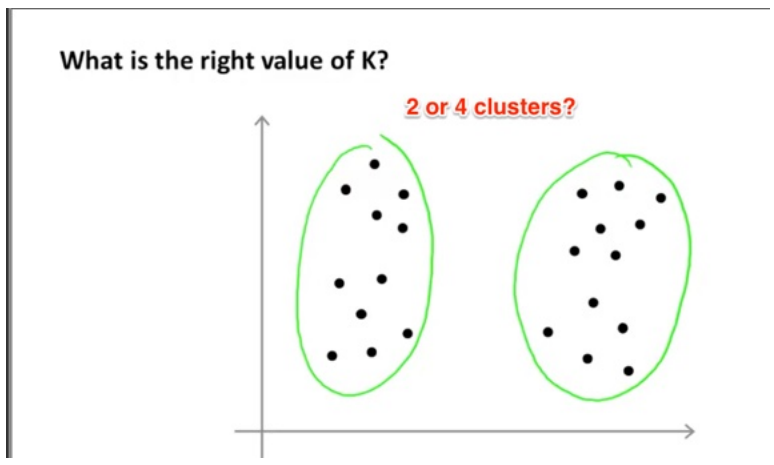
Random initialization:

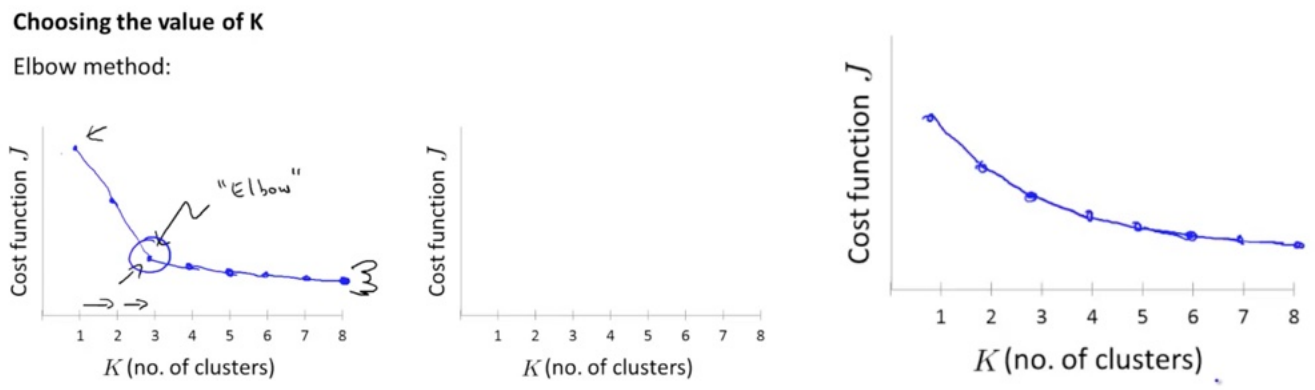
- First, K < m (need fewer clusters than samples of course)
- Pick centroids as K random samples (neat idea… use the data, which already has clusters, to identify what clusters you need).
    - What if you get 2 neighboring examples?
    - Run it several times and see if clusters are similar between runs? Yep, k-means can end up with different solutions.



Run K-means 100 times. Get the clusters. Get the one with the lowest cost. For small # of clusters (K = 2..10), good to randomly initialize.

Choosing number of clusters

- Often chosen manually / by hand
- Often ambiguous
    - 

- Elbow method
    - Compute cost function as we vary number of clusters. (Clearly, with having K=m we have no error)
    - Aha: “Elbow” in the curve (derivative changes sharply). This means we get the best bang for the buck at that point.
    - 

    - Diminishing marginal returns. But in practice, it’s not that clear-cut, can vary slowly.

- Example: how many t-shirt sizes to make? 3, 5, etc. Think about the external constraints (how many sizes can you actually make?) vs. just having the data speak.

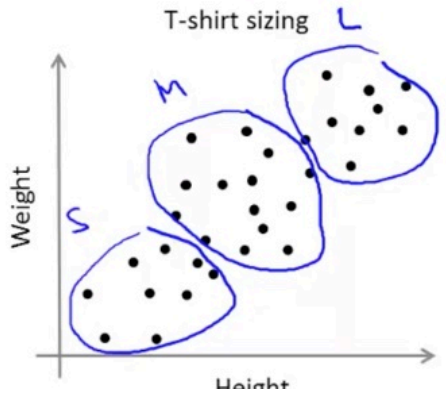
Choosing the value of K

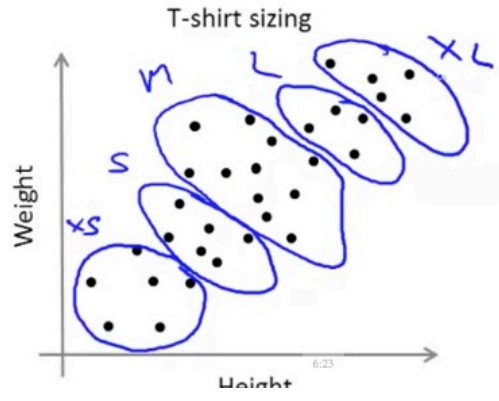
Sometimes, you're running K-means to get clusters to use for some later/downstream purpose. Evaluate K-means based on a metric for how well it performs for that later purpose.

K=3 S, M, L

K=5 XS, S, M, L, XL

E.g.

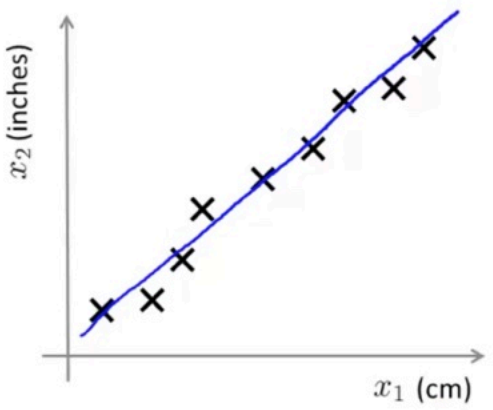




Week 8 - Dimensionality Reduction

Application: Data compression

Data Compression

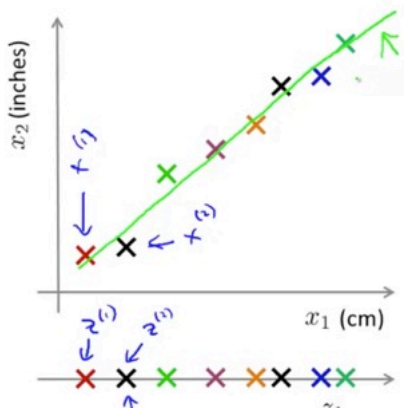


Reduce data from 2D to 1D

- Find structure within the data which simplifies the information needed to store. Why have x, y coordinates when we can just have a line and say how far we are along it?
- In other words, if the data is already correlated, this represents "redundant" information.

Project data onto a new feature (which is the line)

Data Compression



Reduce data from 2D to 1D

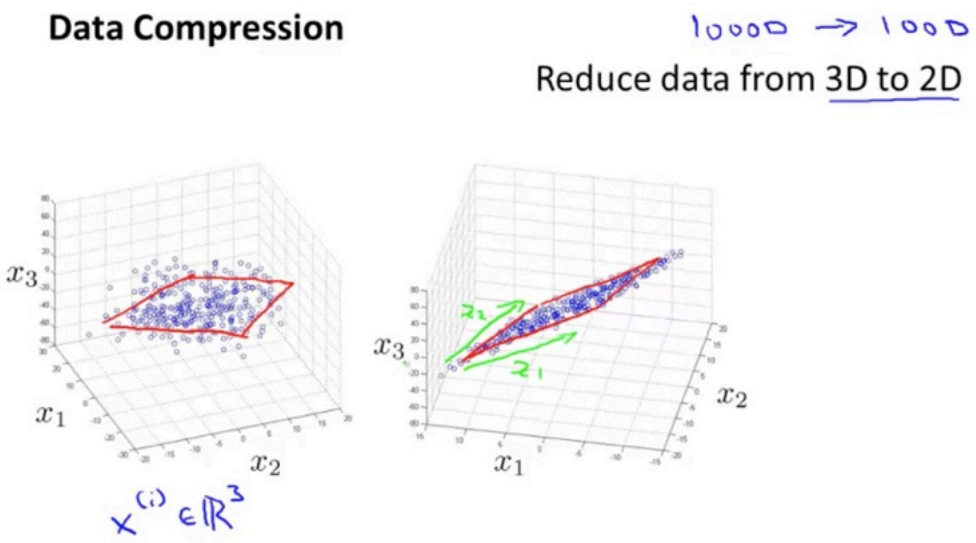
$x^{(1)} \in \mathbb{R}^2 \rightarrow z^{(1)} \in \mathbb{R}$

$x^{(2)} \in \mathbb{R}^2 \rightarrow z^{(2)} \in \mathbb{R}$

$\vdots$

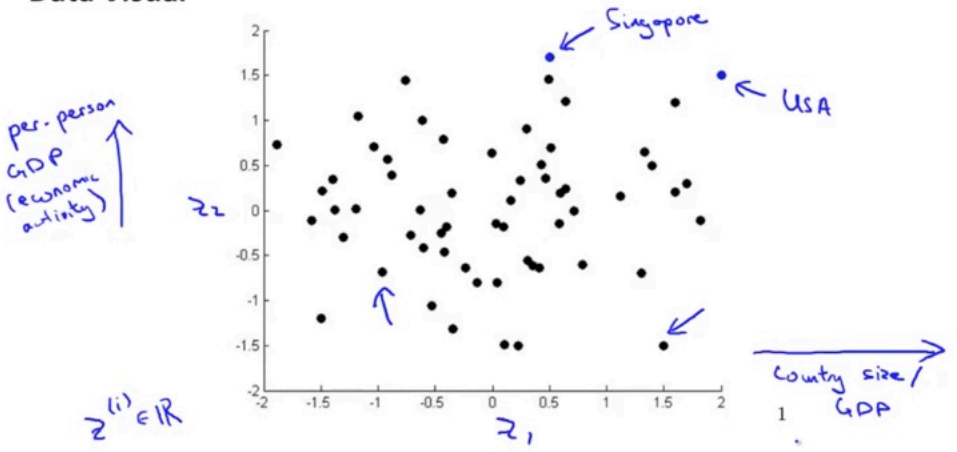
$x^{(m)} \rightarrow z^{(m)}$

- We went from 2 numbers to 1. Can do 3d to 2d, etc. Getting the “shadow” of the original on a single plane.



Point on plane needs 2 numbers (x, y coordinates relative to planar axes). Idea: when should data compress this way? See how close the data points are to the plane, how much distance is there between the projection and the original. If not much difference, then go for it.

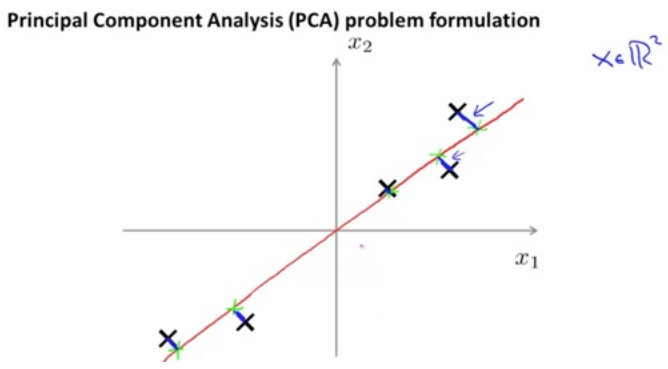
Application: Visualization

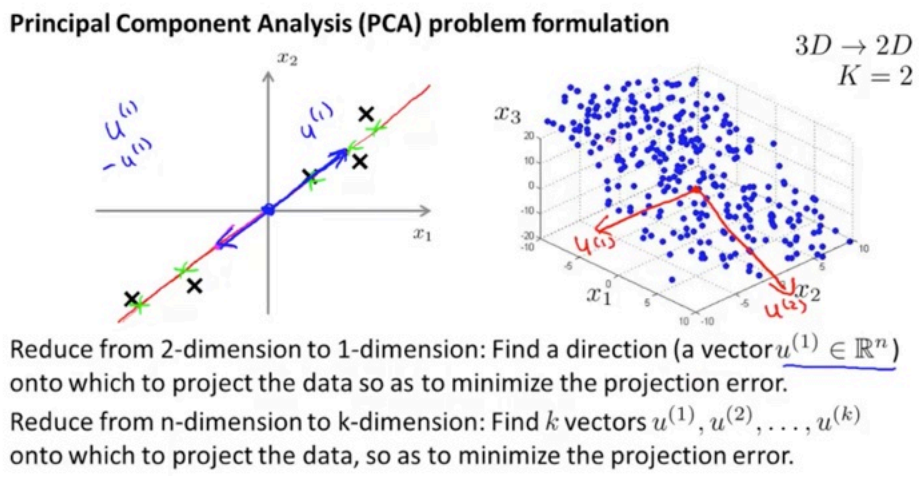


- Suppose we have a bunch of information about countries, and it gets reduced to a few dimension (GDP per country, GDP per capita)
- Typically we reduce to 2 or 3 dimensions so we (humans) can plot/analyze it. (But in essence, we are hoping our visual system notices a correlation that exists… right? Seems like we could apply machine-learning to this again too)

Principal Component Analysis

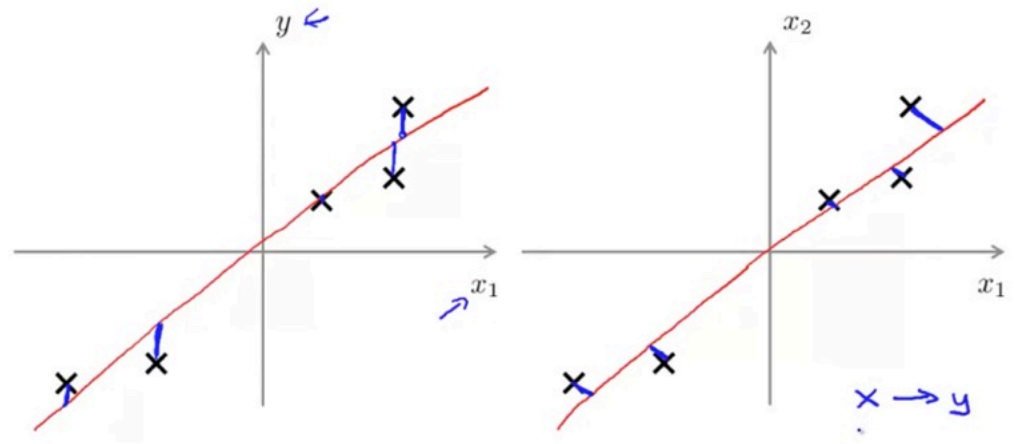
- Ah! Yes. PCA minimizes the error. Yay! (Made the projection). Minimize projection error. Nice!!!





- We can pick k vectors onto which to project. 1 vector is a line, 2 define a plane, and so on. Neat.

PCA is not linear regression: linear regression minimizes the VERTICAL error.
PCA minimizes the perpendicular error.

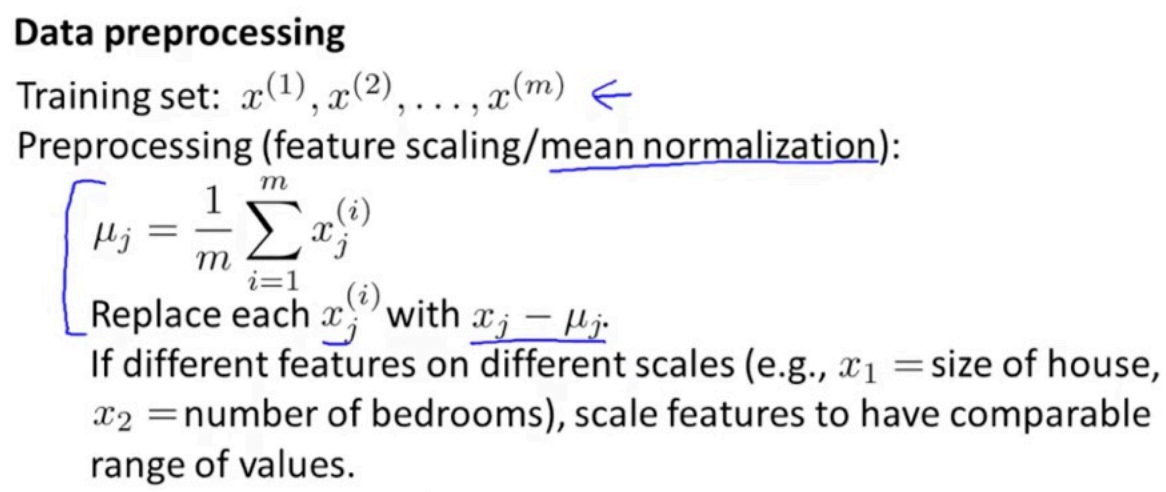


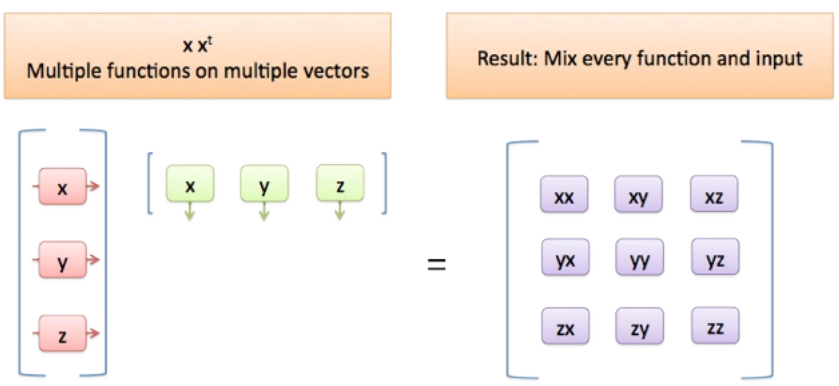
- Also… PCA just finds structure in the data. Linear regression is trying to make a prediction about a value “y” (and how far that prediction is off… we assume there’s a linear function that can predict y). PCA is finding the best “skeleton” inside the data.

PCA plain english: find a simpler model hiding inside the data.
PCA tech description: “Find a low-dimensional surface onto which to project data.”

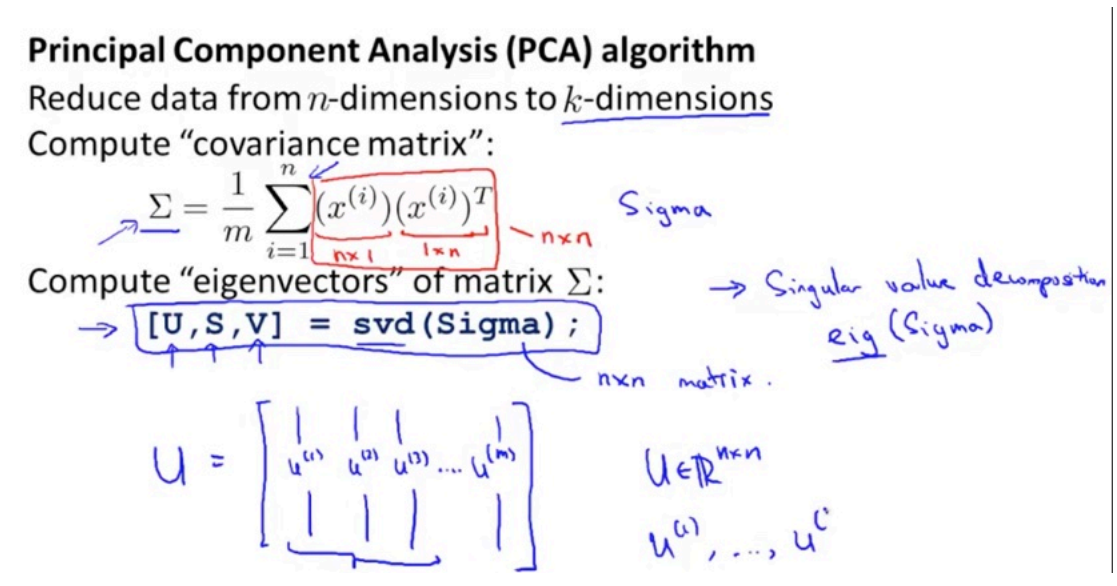
Implementing PCA:

- Preprocess: feature scale / mean normalization. Replace each with the difference from the mean, so 0 mean. Then divide by s_j, which is the range you want to normalize to.



-
- Can find distance from line to point:
  - Call P (px, py) the projected point. Dot product of P to X and the line is zero. Can solve for this. https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line
- Compute covariance matrix, which is x x^T => notice how this is different from the dot product, x^t x.
  - 

- Compute eigenvectors of this matrix. Ah. We are finding the “axes” of this matrix. Neat!
  - TODO: Getting a derivation of this would be good. But I can (squinting) see how that could be true. Covariance matrix is the average of all the interactions, then look for structure inside (eigenvectors).
- Sigma is also used as a variable, the covariance matrix.
  - SVD (singular value decomposition)

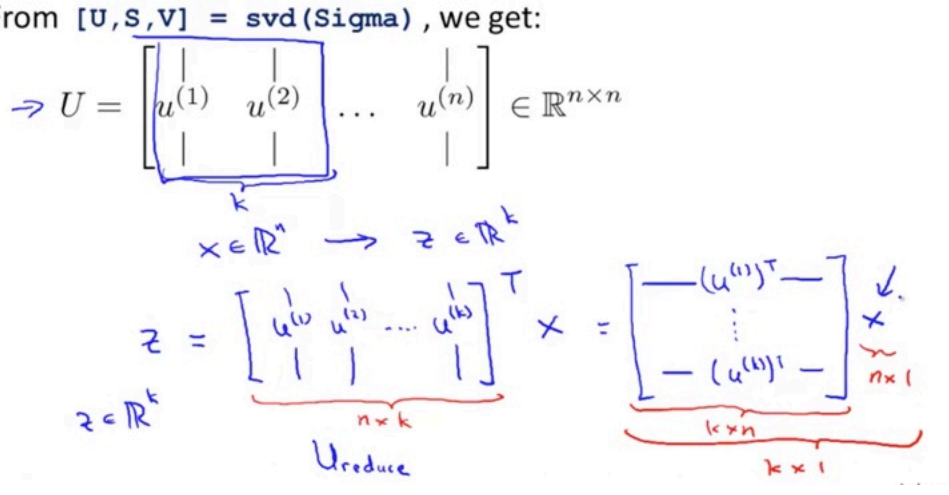


-

- Take first k vectors for the first k PCA vectors.

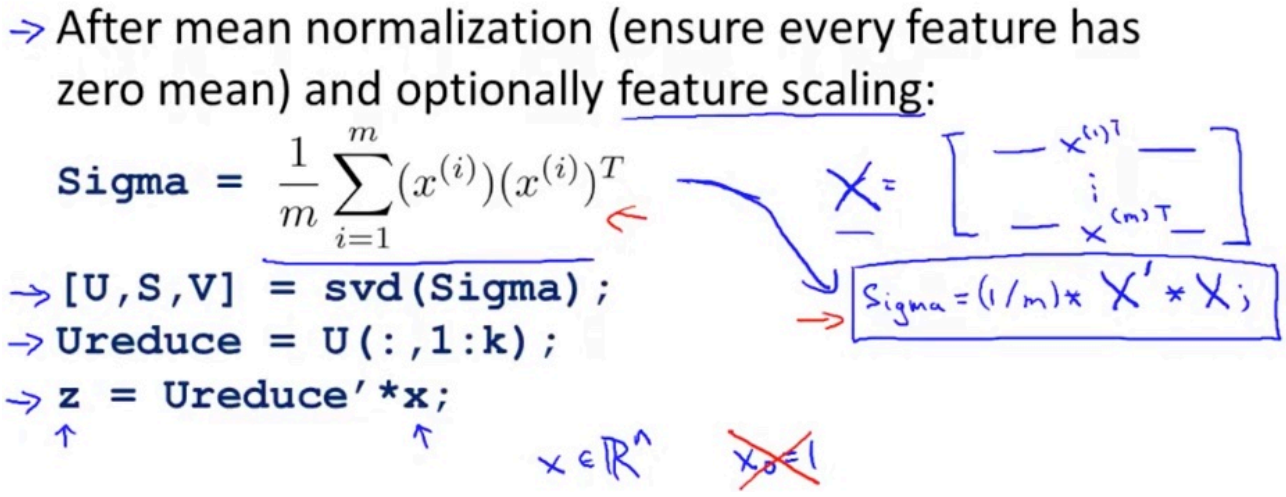
Principal Component Analysis (PCA) algorithm

From `[U,S,V] = svd(Sigma)`, we get:



- Keep track of “n x k”, etc. is important. Rows and columns. How many inputs, amount of data in each. (10 operations of 3 parameters). Then pass in 3 parameters, 25 times. Then you have 10 operations, 25 times. [10 x 3] * [3 x 25] = [10 x 25]

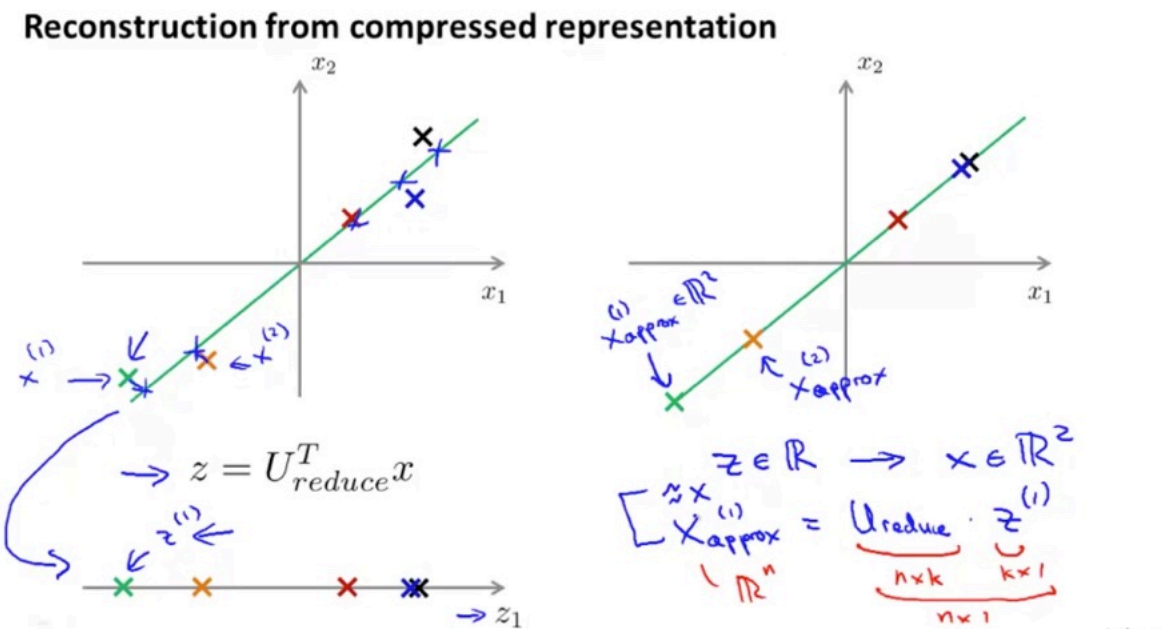
Principal Component Analysis (PCA) algorithm summary



- z (the projections of the PCA vectors), take the first k vectors, transpose, multiply by x (original data set)

Applying PCA

- Can “uncompress” the reduced dimensions by putting them back onto the axis.



- Reconstruct x from the compressed representation.
  - Lossy - “percent of variance retained”

Choosing number of principal components (k)

- Goal: minimize average squared projection error. (Idea: look for the “elbow” in the curve)

- 99% of variance is retained. (the error compared to the total size of the data set is 1%)

Choosing k (number of principal components)

Average squared projection error: $\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)} - x_{approx}^{(i)}\|^2$

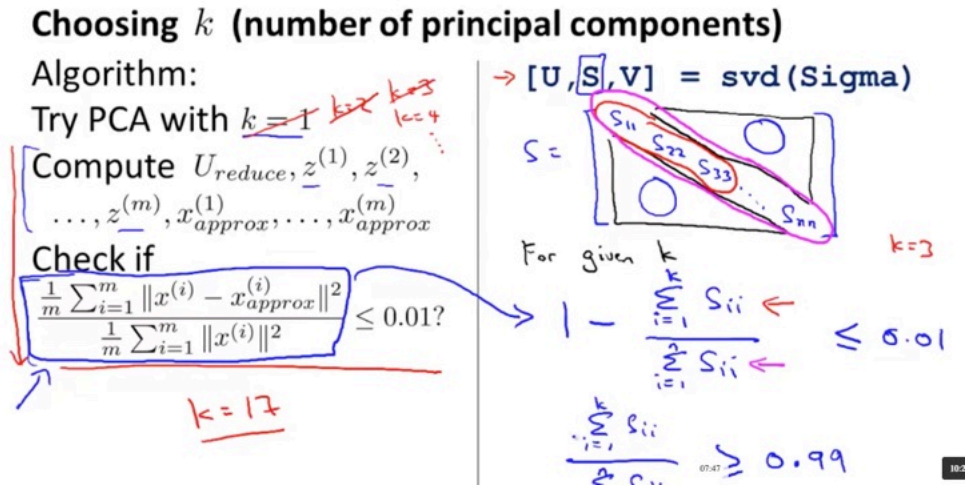
Total variation in the data: $\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2$

Typically, choose k to be smallest value so that

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} \leq 0.01 \qquad \text{(1\%)}$$

"99% of variance is retained"

- Keep adding components (k) until we are 99% accurate. Neat. ("I chose k so 99% of variance was retained."). 1%, 5%, etc.
    - Many features tend to be highly correlated
- Simple algo: just keep increasing k.
    - Instead… do a binary search in the sorted list! Because we know variance only increases as we increase k.
- The library returns S (diagonal matrix). We can add up the diagonals we want compared to ALL entries.

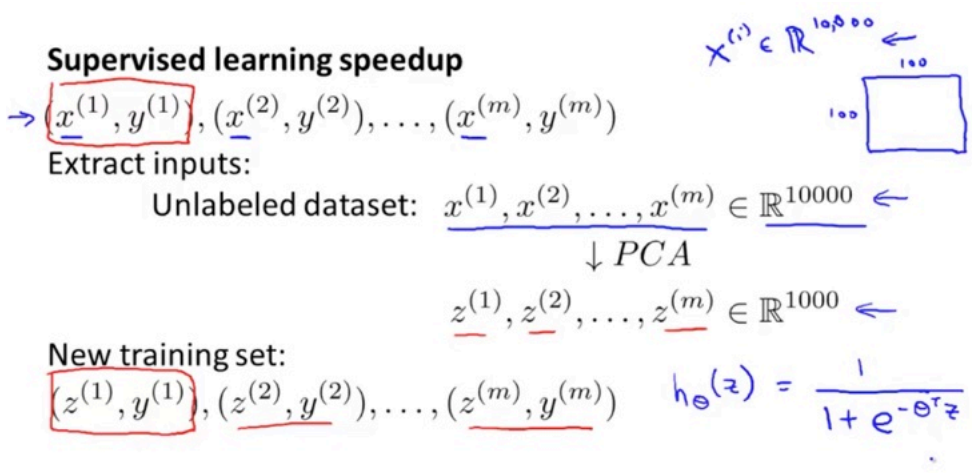


- Once you have the S-matrix, can just increase value of k and find the level you need.

PCA Advice

Supervised learning speedup: (Predict… how can this be done? Well, we are supervised (we have the classifications (x => y)) so we can do PCA, keep 99% of variance, then do z => y. We have a reduced-dimensionality data set which can be much faster to work with (for a linear regression, NN, etc.)

- Imagine your image is 100x100 (10,000 feature vectors -- each pixel)
- Extract x. Apply PCA, get z (1000 feature vectors, let's say).
- Then new training set is simpler. Yep! Nice, got prediction.



- PCA maps from x to z. So for a new data point, you map to z, then you make the prediction on the class.
    - Mapping applied to x_cv and x_test (cross validation, training set)
- For many problems, can reduce feature size by 5-10x. Neat.

"Percent of variance retained" => "percent of accuracy".

Bad use of PCA: To prevent overfitting

→ Use $z^{(i)}$ instead of $x^{(i)}$ to reduce the number of features to $k < n$. — 1000 < 10000

Thus, fewer features, less likely to overfit.

Bad!

This might work OK, but isn't a good way to address overfitting. Use regularization instead.

→ $\min_\theta \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ ←

09:10

If concerned about overfitting… use regularization, other techniques.

- Intuition: use the technique that was designed to minimize overfitting (punish parameters) instead of accidentally stumbling into it.

PCA is sometimes used where it shouldn't be

Design of ML system:

- → Get training set $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$
- → Run PCA to reduce $x^{(i)}$ in dimension to get $z^{(i)}$
- → Train logistic regression on $\{(z^{(1)}, y^{(1)}), \ldots, (z^{(m)}, y^{(m)})\}$
- → Test on test set: Map $x_{test}^{(i)}$ to $z_{test}^{(i)}$. Run $h_\theta(z)$ on $\{(z_{test}^{(1)}, y_{test}^{(1)}), \ldots, (z_{test}^{(m)}, y_{test}^{(m)})\}$

→ How about doing the whole thing without using PCA?

→ Before implementing PCA, first try running whatever you want to do with the original/raw data $x^{(i)}$. Only if that doesn't do what you want, then implement PCA and consider using $z^{(i)}$.

- Use PCA when necessary. Try regular logistic regression, etc. first. It's an optimization / visualization technique that throws away data.

Homework

- Can see the K-means moving

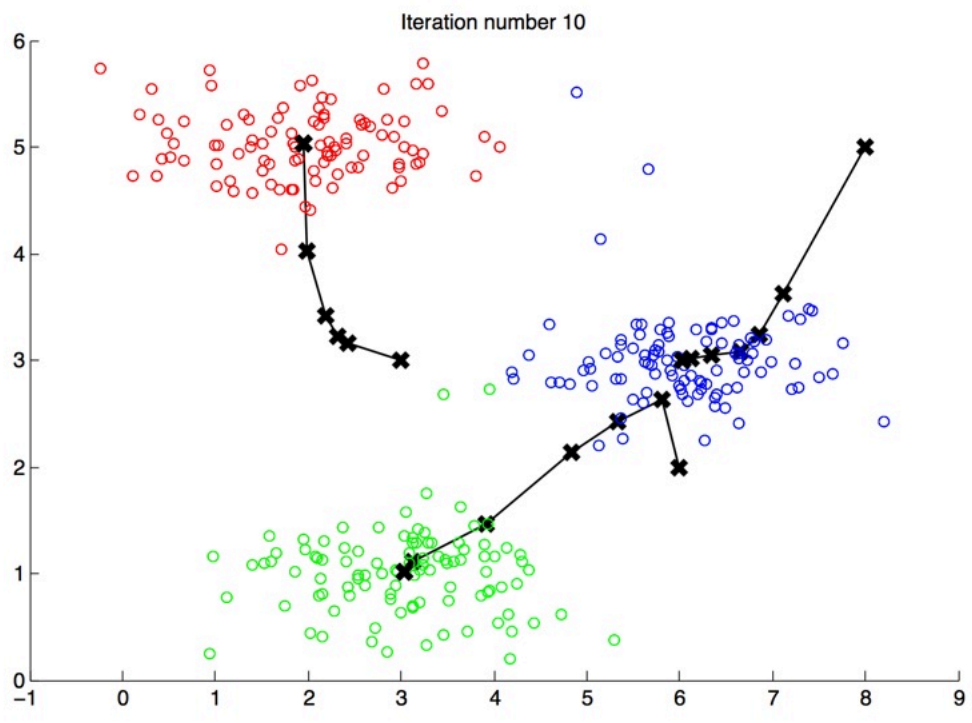


Figure 1: The expected output.

- Pick N random items as the centroid. Avoid picking the same items twice (randomly reorder the indexes, pick first N).
- Color reduction is a cool example: Pick 16 colors that are closest to the majority of the samples (reduce the distance to them on average). Find the best clusters.

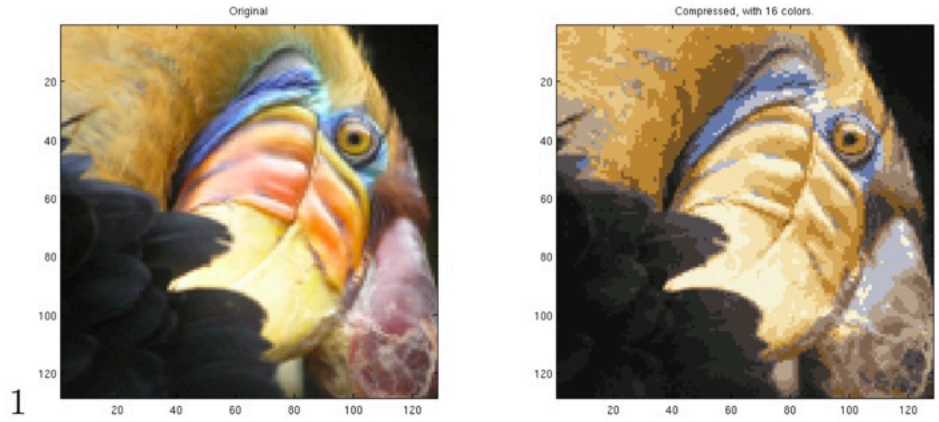
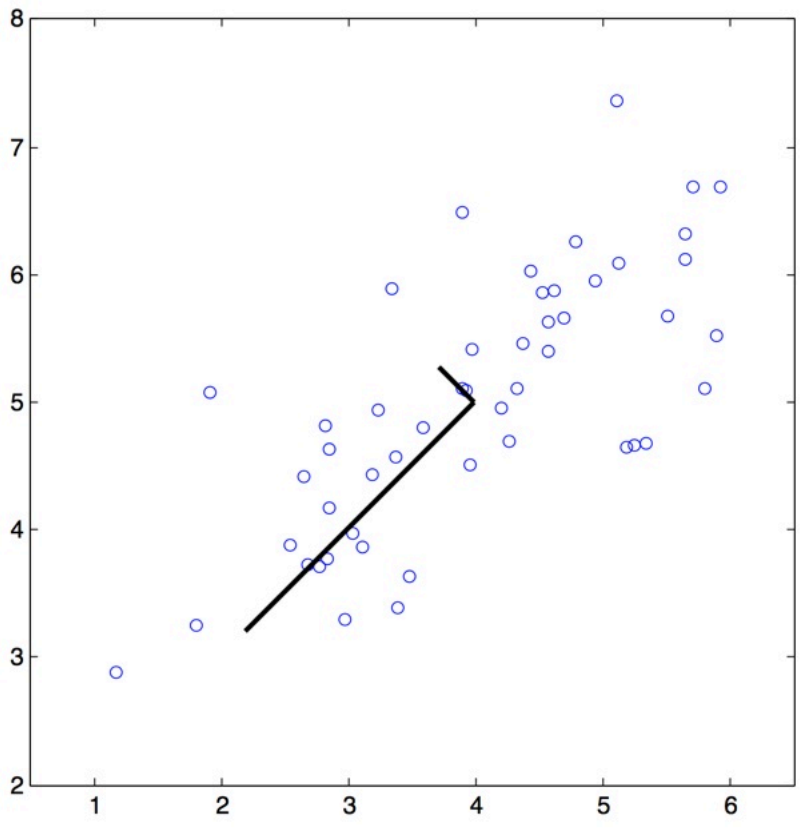


Figure 3: Original and reconstructed image (when using K-means to compress the image).

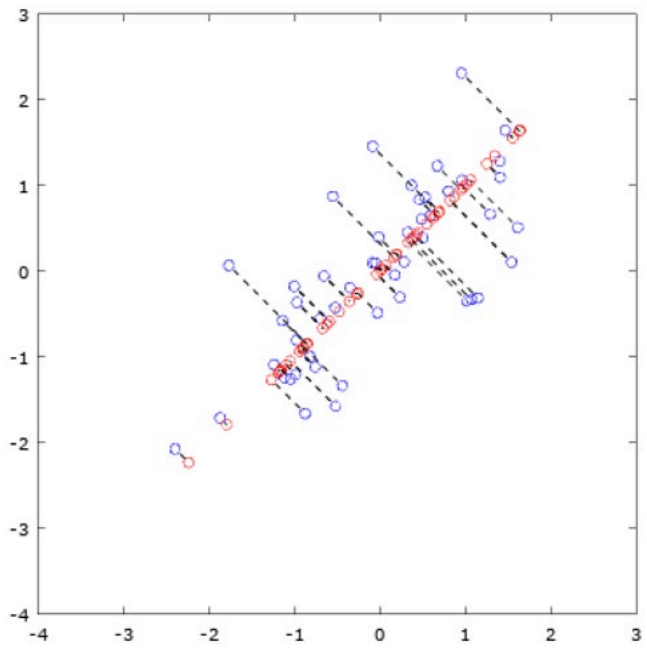
- For PCA, intuition that we are projecting onto a different basis vector. (And reducing dimension from 2d to 1d, saving data).



- Computed principal components. Note they could point the other other way.
    - Again, we project the data onto each PCA vector. Take the top N.
- Gotcha: for PCA recovery, we project *each* dimension back using the first K eigenvectors available. TODO: Run through the examples more clearly to not get caught on the numbers of dimension for the source, eigenvectors, K, and recovery.
    - X: original data (m=300 samples, n=20 dimensions, let's say)
    - U: covariance matrix (n x n, has eigenvectors. 20x20)
    - Z: projections of X onto first K vectors in covariance matrix. m=300 samples, n = 5 (only using 5 dimensions). Have a score for how much of each of first 5 eigenvectors appears in data.
    - X_rec: recover original data set. For each Z entry (projection), figure out contribution to each dimension from each eigenvector. We can sum them with `v' * U(j, 1:K)`. Here, we took the dot product to get the sum.

```
for i = 1:size(Z,1)
    for j = 1:size(U,1)
        v = Z(i, :)';                       % Z has the projections onto the eigenvectors
        recovered_j = v' * U(j, 1:K)';  % sum the contributions of each eigenvector to this dimension
        X_rec(i, j) = recovered_j;
    end
end
```

    - Keeping track of all the dimensions is tricky at first. Data projected onto the line which is then warped back into our space (notice the loss of a dimension)



- Face example is cool. Going from 1024 to 100 dimension. Essentially blurring the faces. But reduce the data needs by 10x (for face recognition, for example).

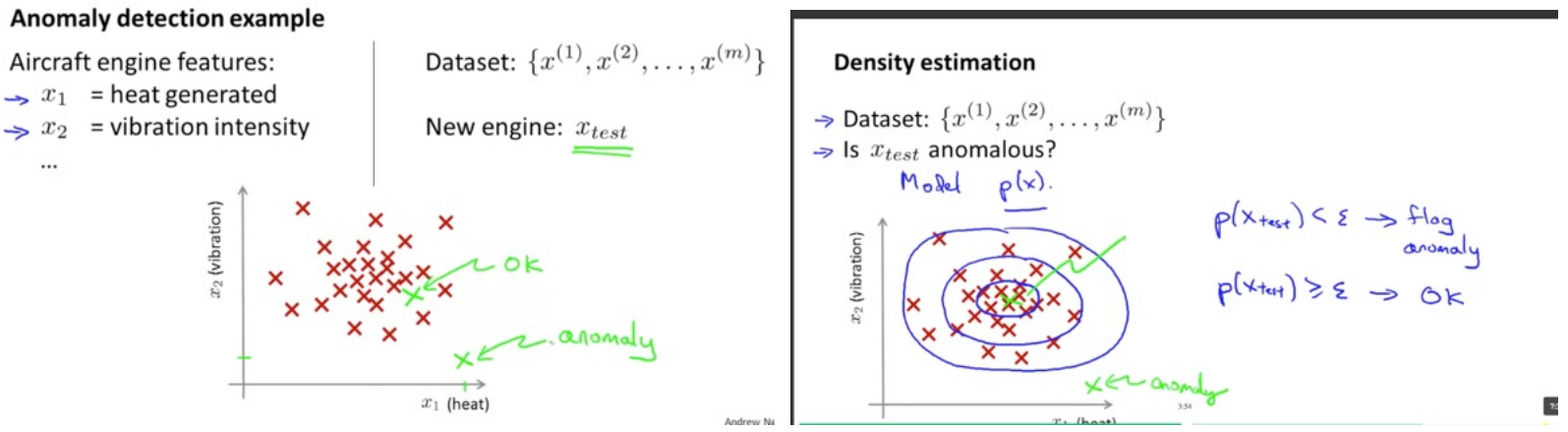
Figure 8: Principal components on the face dataset

Figure 9: Original images of faces and ones reconstructed from only the top 100 principal components.

Week 9 - Density Estimation

Anomaly detection

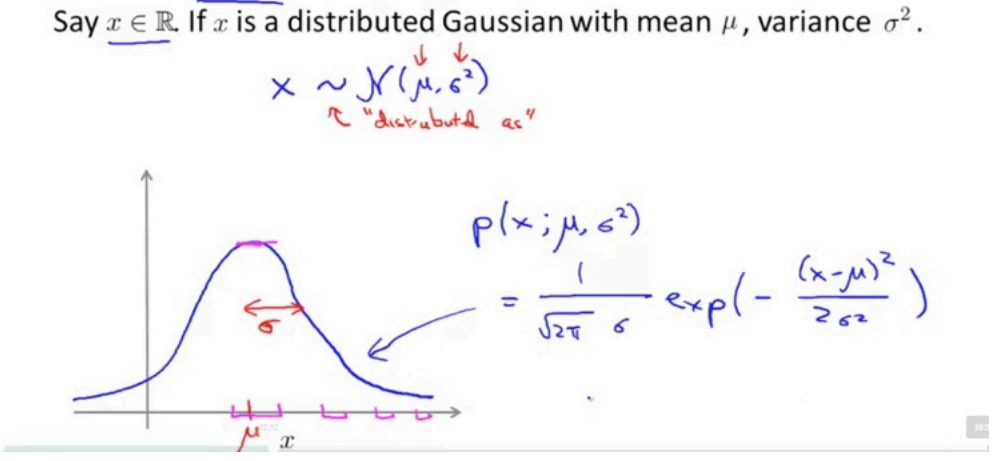
- Prediction: get a linear regression. If a training sample is far from the prediction we predict an anomaly (i.e., we can call it unlike the other samples in the set.)
- Get probability that our model would return that data point x. If < epsilon, flag it.



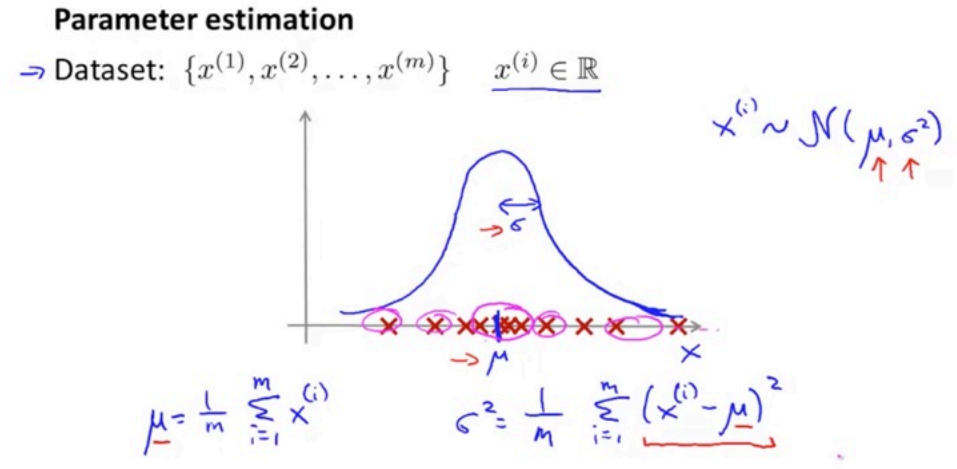
- A lot of this seems like “just” regular statistics. How many sigmas outside of normal.
    - Fraud detection
    - Manufacturing
    - Monitoring computers in data center (CPU load, memory use, disk access, etc.). Does it look like it’s having trouble.

Gaussian distribution (Normal distribution)

- Can think of sigma as the “width” of the curve. mu is the center point. Nice intuition.



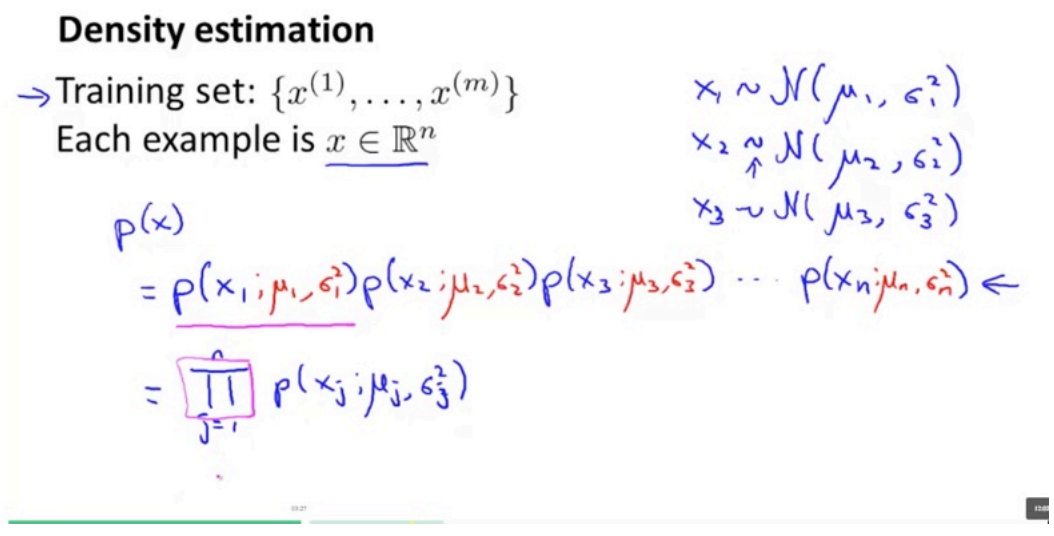
- TODO: do the derivation of the normal distribution as an article (there was the PDF paper I found a while back).
- Parameter estimation: have some dataset, think it’s Guassian -- then find mu (mean) and sigma (std dev)
    - Mean (mu) is just avg
    - Variance (squared): avg sum of squared differences from mean



    - 1/m vs 1/(m-1) -- ah, because of degrees of freedom. In machine learning they just take the 1/m. Hah.

Algorithm:

- Assume each feature (x_1, x_2, etc…) are all Gaussian.
- Probability of x is product of each feature probability. (assumes independence between each feature).



- PI as Product (vs. sigma for Sum)

Anomaly detection algorithm

1. Choose features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

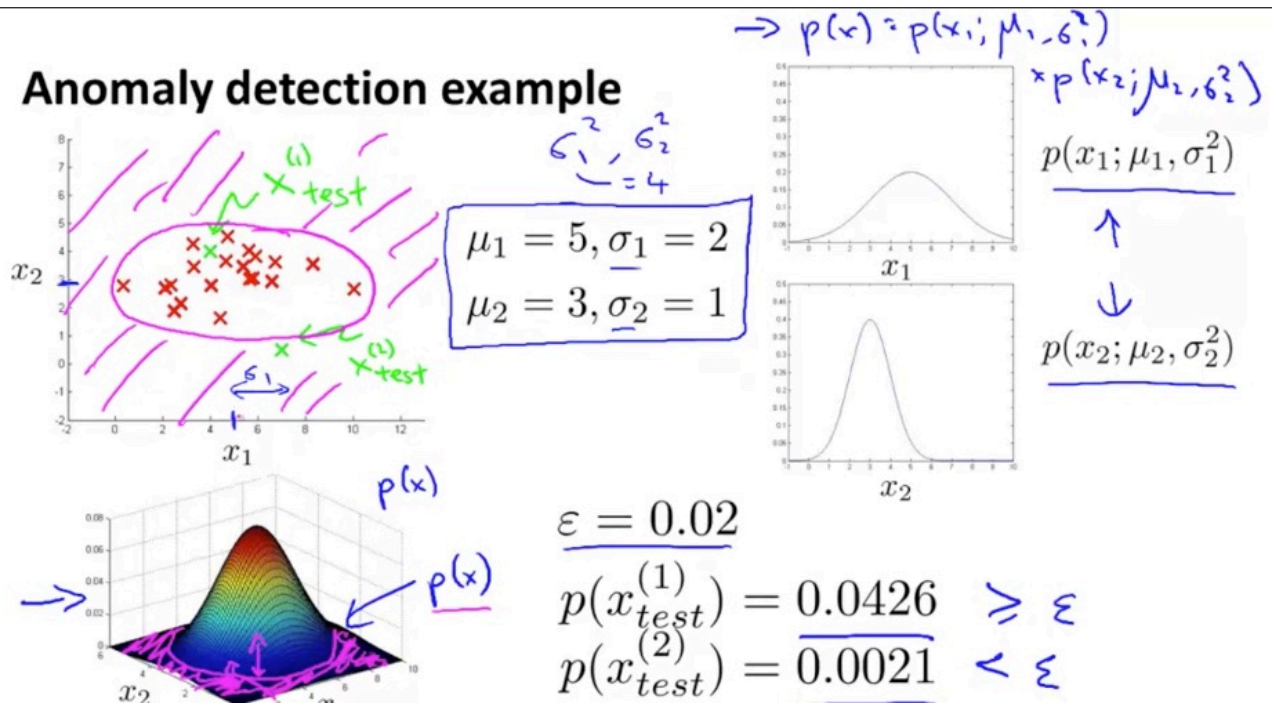
3. Given new example x, compute $p(x)$:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

Anomaly if $p(x) < \varepsilon$

- Q: However, won't these probabilities be tiny after all the multiplications? 80% * 80% = 64%. Do we have an epsilon for each? (.05 raised to the number of features?). An average of 5% error on each feature maybe?

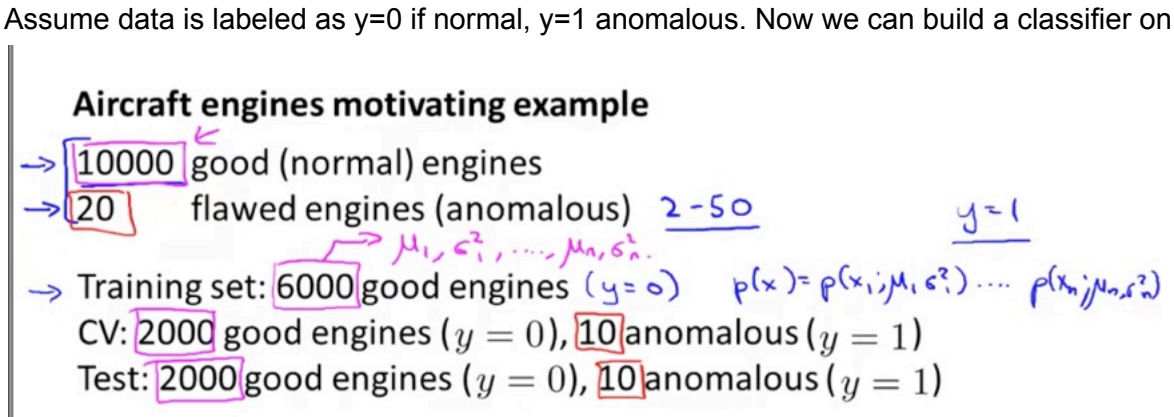
Anomaly detection example



Anomaly Detection System

Learning algos, want to quantify their performance. (General idea: have a number to manage.)

- Assume data is labeled as y=0 if normal, y=1 anomalous. Now we can build a classifier on this.



- Again, cross validation set helps us figure out the right regularization parameter.

Algorithm evaluation

→ Fit model $p(x)$ on training set $\{x^{(1)}, \ldots, x^{(m)}\}$

→ On a cross validation/test example x, predict

$$y = \begin{cases} 1 & \text{if } p(x) < \varepsilon \text{ (anomaly)} \\ 0 & \text{if } p(x) \geq \varepsilon \text{ (normal)} \end{cases}$$

Possible evaluation metrics:

- True positive, false positive, false negative, true negative
- Precision/Recall
- F_1-score

Can also use cross validation set to choose parameter ε

- Remember the F1 score -- tradeoff between true/false pos/neg
- Yep, this is just another type of supervised learning. It's skewed though, b/c y=0 is much more common than y=1
  - Gotcha! Remember we can't just use test accuracy. Because of the skew in the data. Use F1 score (precision/recall). We need to figure out how the true/false pos/neg impact things. Because if you just say everything is ok, you get 99% accuracy.
  - Want to maximize F1 score.

Anomaly Detection vs. Supervised Learning

- Anomaly when you have a small number of positive examples. Makes sense.
- Anomalies aren't predictable. Look different from what we have. For supervised learning, new inputs will look similar to ones we already have.

Anomaly detection	vs.	Supervised learning
→ Very small number of positive examples ($y = 1$). (0-20 is common).		Large number of positive and negative examples.
→ Large number of negative ($y = 0$) examples. $p(x)$		
→ Many different "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.		Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set.

- Spam - supervised. Not many unique types of spam.

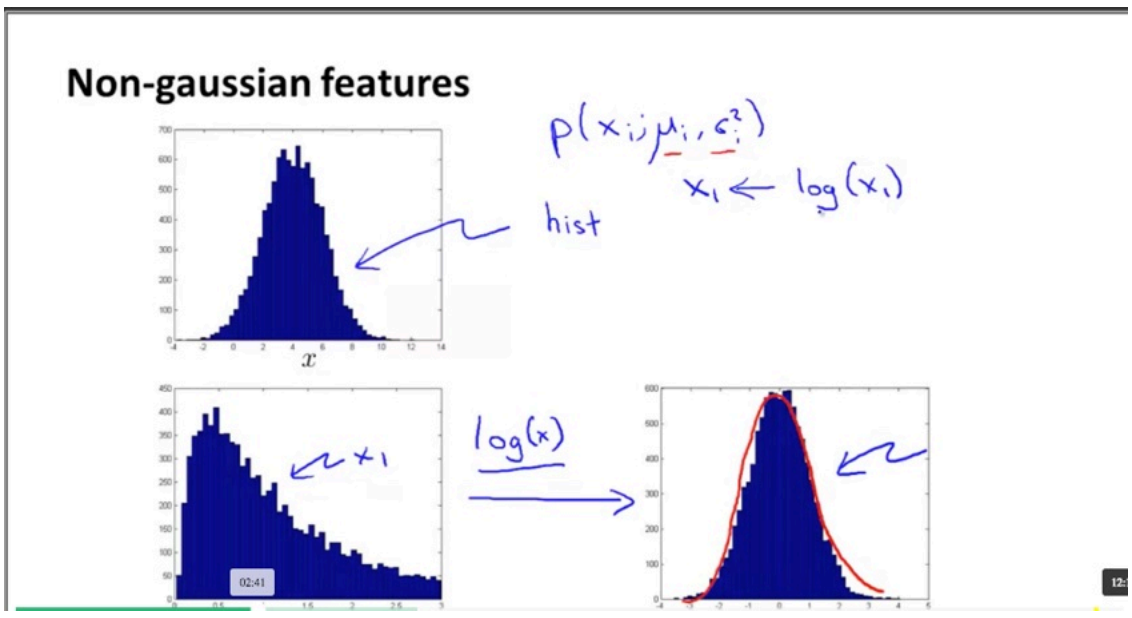
Anomaly detection	vs.	Supervised learning
→ • Fraud detection $y=1$		• Email spam classification
• Manufacturing (e.g. aircraft engines)		• Weather prediction (sunny/rainy/etc).
• Monitoring machines in a data center		• Cancer classification
⋮		⋮

5:59

Choosing what features to use

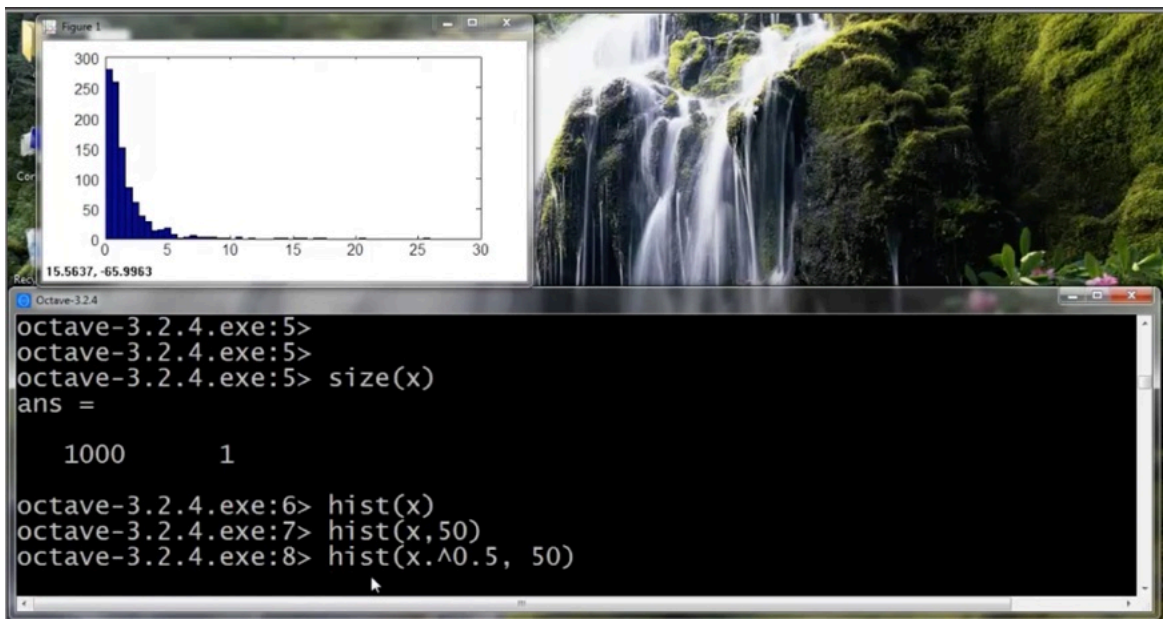
Prediction: Should we get rid of correlations within the data? Use PCA to simplify?

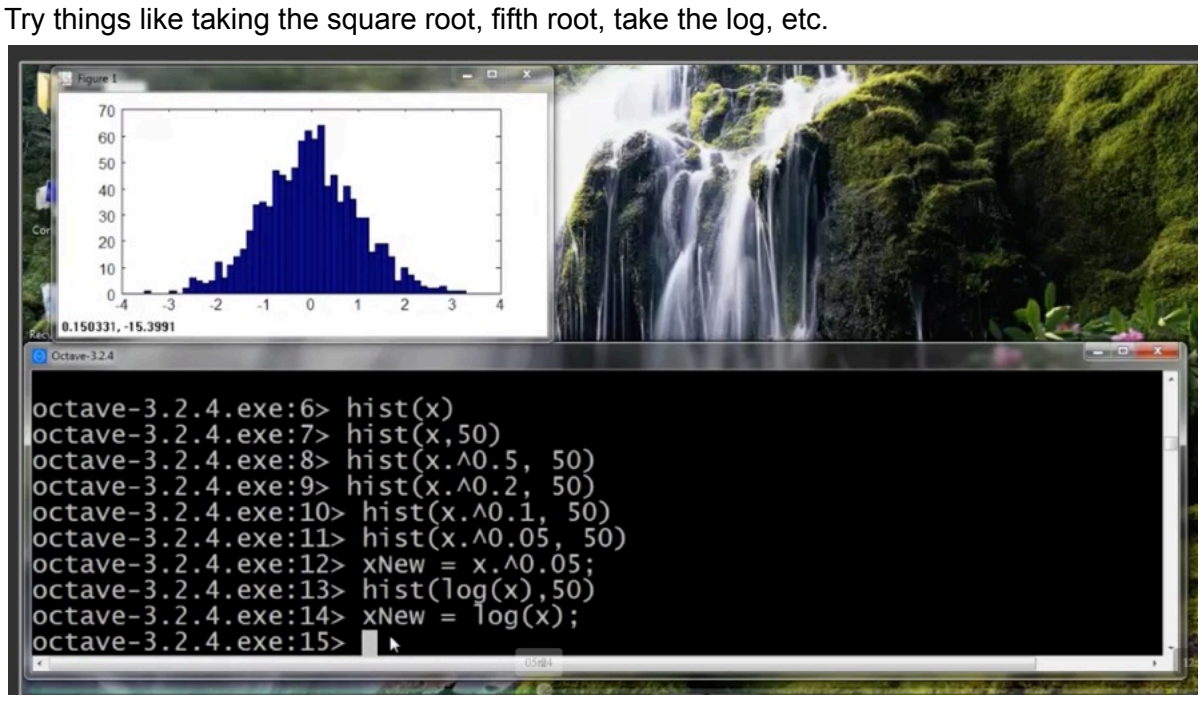
Q: What if features are non-Gaussian? (Use transformations to convert to a Gaussian data set):



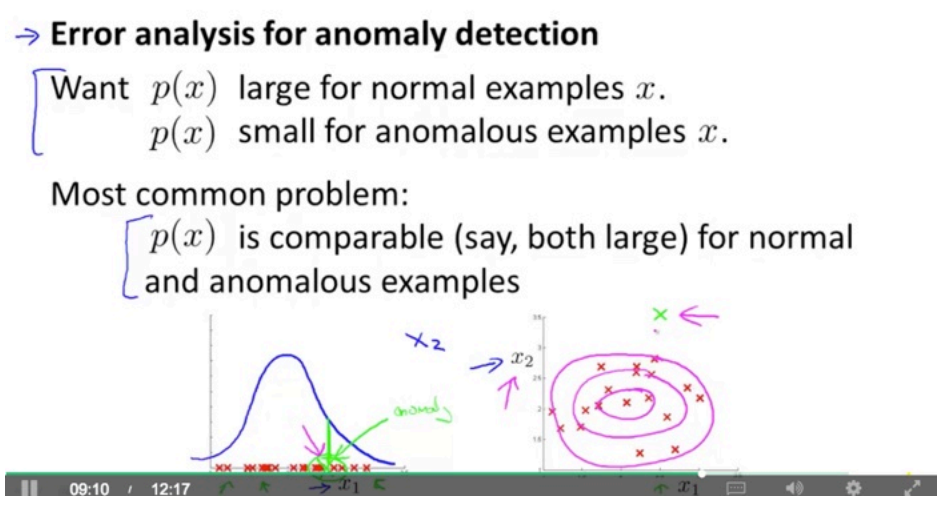
- The transform could be log(x), log(x + 1), sqrt(x), etc.

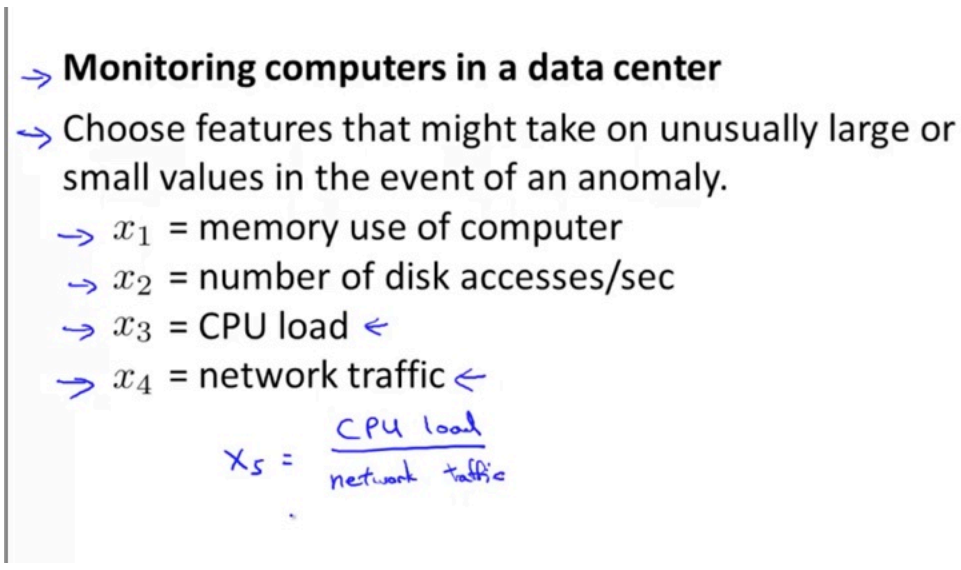
How to do this:

- First, get a histogram of data.
- 

- Try things like taking the square root, fifth root, take the log, etc.
- 

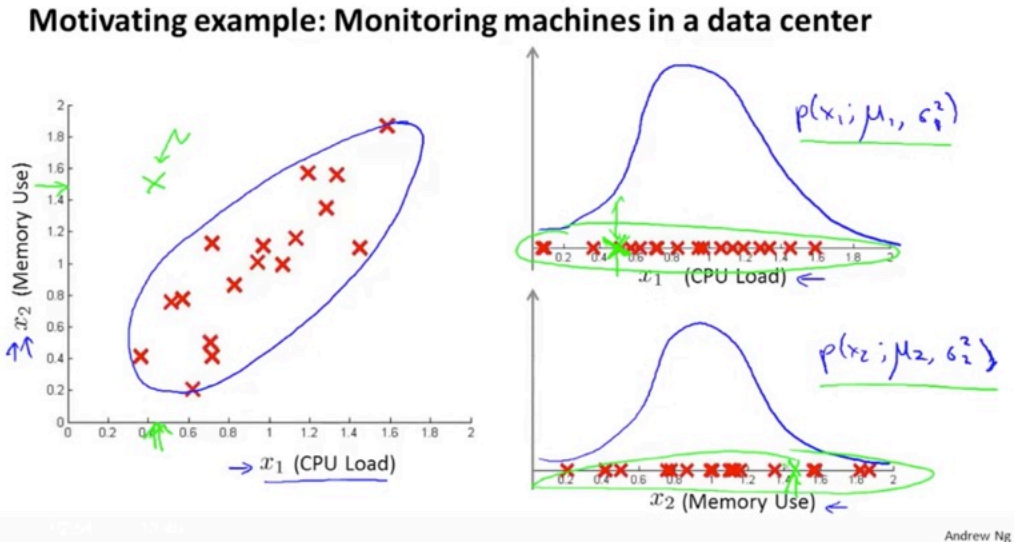
- If your anomalous examples are buried inside the regular ones, look at that error case and make a new feature which helps distinguish it.

- 

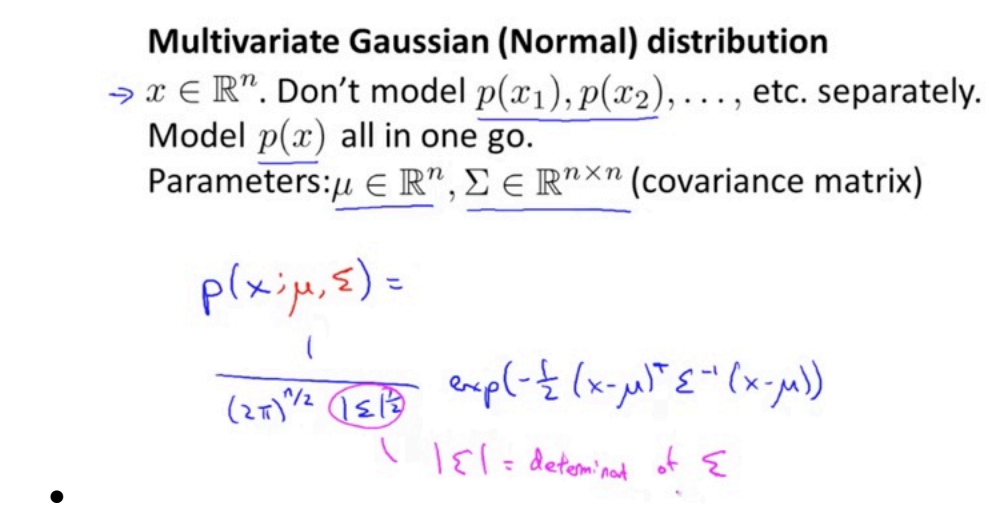


- Example: create that new feature x5 which is a new feature with some interesting properties.

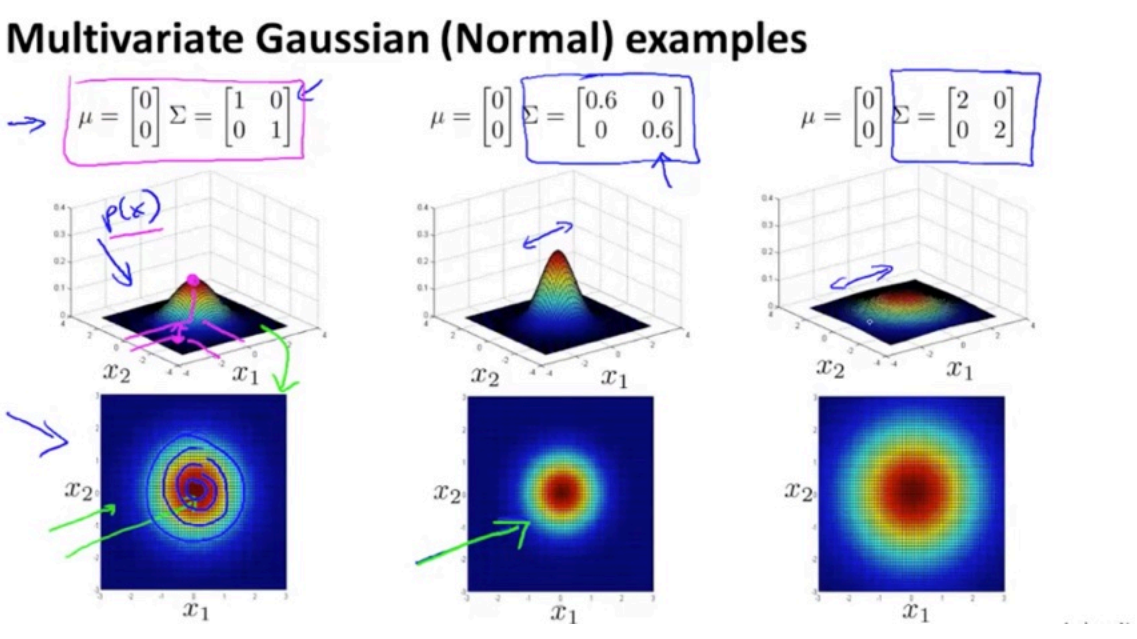
Multivariate Gaussian Distribution



- Let's say that each variable is fairly reasonable on its own. But the combination is rare.
- Solution: Don't model each feature independently -- model everything together. Ah (avoid independence?)



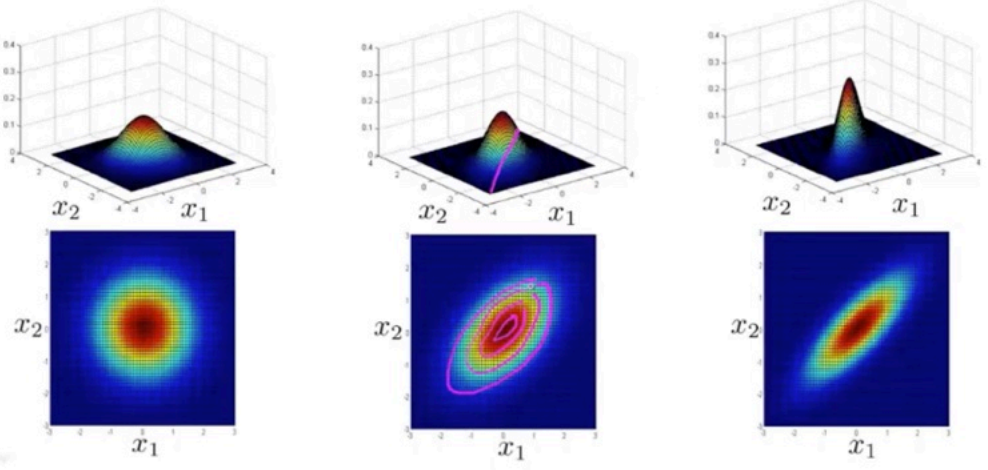
- Use the covariance matrix again. TODO: Figure out intuition behind covariance matrices (eigenvectors inside for PCA). Can find correlations existing inside the data. Sigma, the "smear" of the covariance matrix.



Can vary smear in each dimension. Can find correlations inside data. Get the probability / correlation between the two. We are reshaping the space.

Multivariate Gaussian (Normal) examples

$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ $\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$ $\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}$

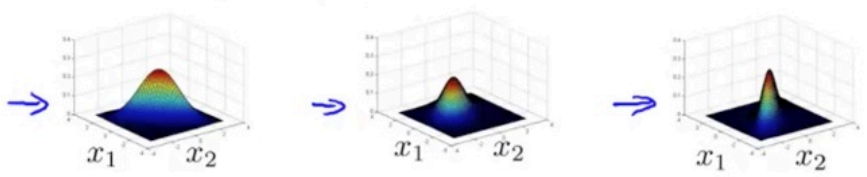


Andrew Ng

Multivariate Gaussian (Normal) distribution

Parameters μ, Σ $\mu \in \mathbb{R}^n$ $\Sigma \in \mathbb{R}^{n \times n}$

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$$



Parameter fitting:

Given training set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ $x \in \mathbb{R}^n$

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \qquad \Sigma = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)(x^{(i)} - \mu)^T$$

Anomaly detection algo:

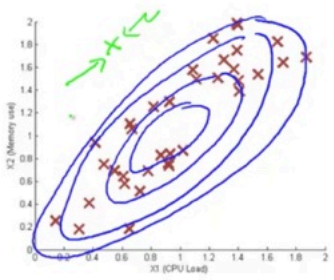
- Fit model to multivariate Gaussian distribution
- If p(x) < e, it's anomaly

Anomaly detection with the multivariate Gaussian

1. Fit model $p(x)$ by setting

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

$$\Sigma = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)(x^{(i)} - \mu)^T$$



2. Given a new example x, compute

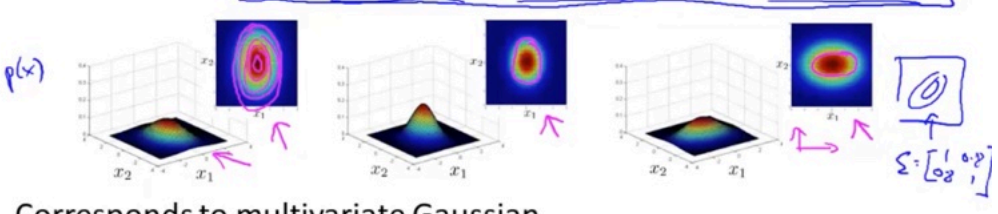
$$p(x) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$$

Flag an anomaly if $p(x) < \varepsilon$

Relationship to original model: We are making the contours axis-aligned.

Relationship to original model

Original model: $p(x) = p(x_1; \mu_1, \sigma_1^2) \times p(x_2; \mu_2, \sigma_2^2) \times \cdots \times p(x_n; \mu_n, \sigma_n^2)$

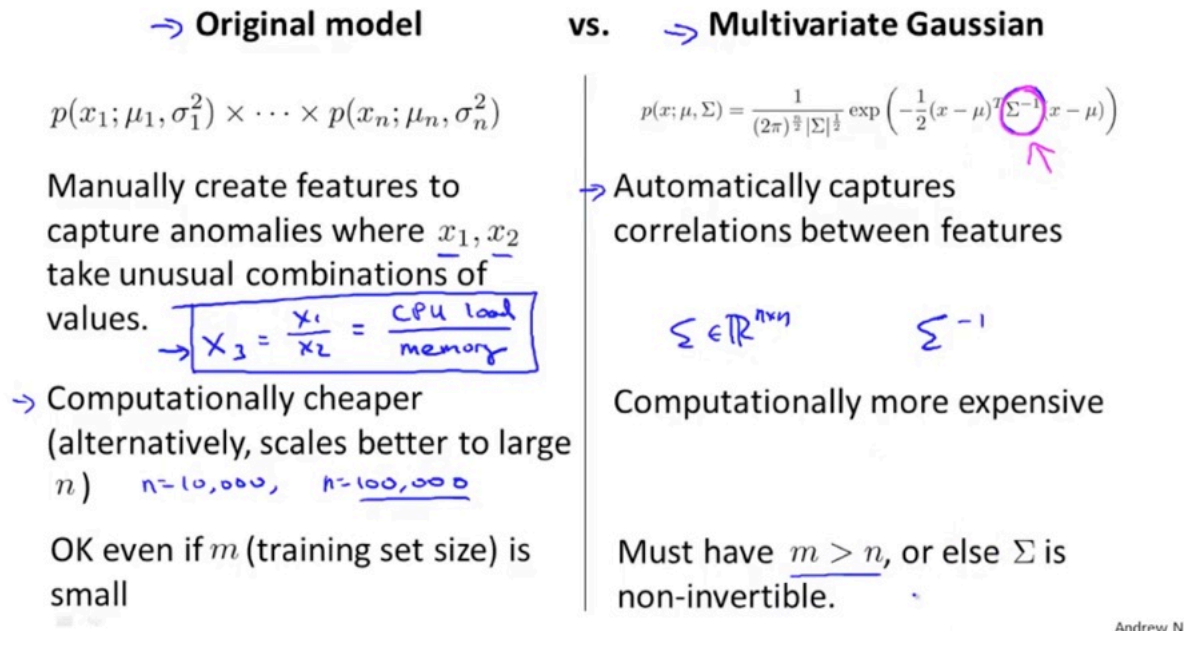


Corresponds to multivariate Gaussian

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$$

where $\Sigma =$

- We have a diagonal matrix for Sigma (e.g., no correlation. Gotcha).

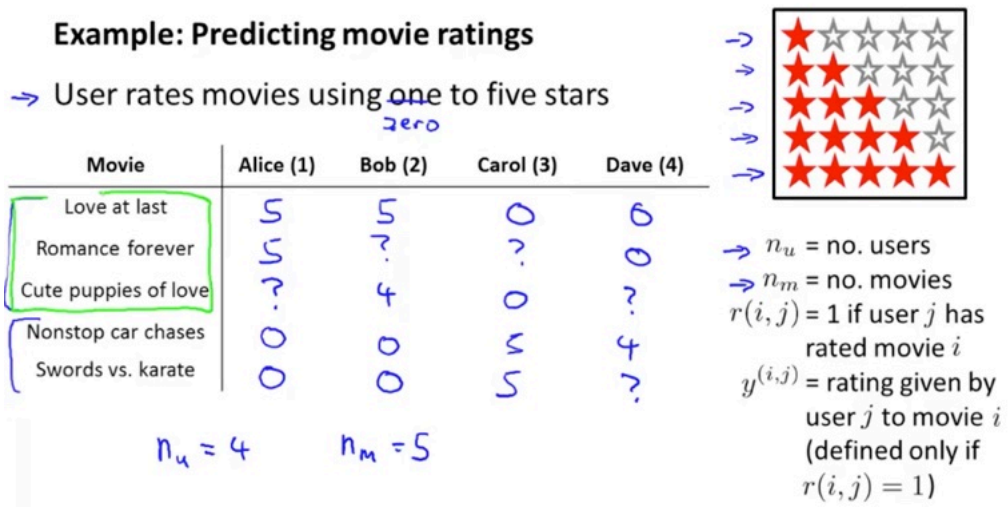


Data is 10x the # of features. Can avoid manually creating features. Neat.

- If you have redundant features (x1 = x2, or x3 = x1 + x2), the non-invertible.
- Or too few data points (m)

If you have features that are normally correlated (x1 ~ x2), you can create features like x1/x2 which will swing wildly if they are off course.
Ah. Remember, the sigma^2 should capture 95% of the samples (if we are eyeballing things)

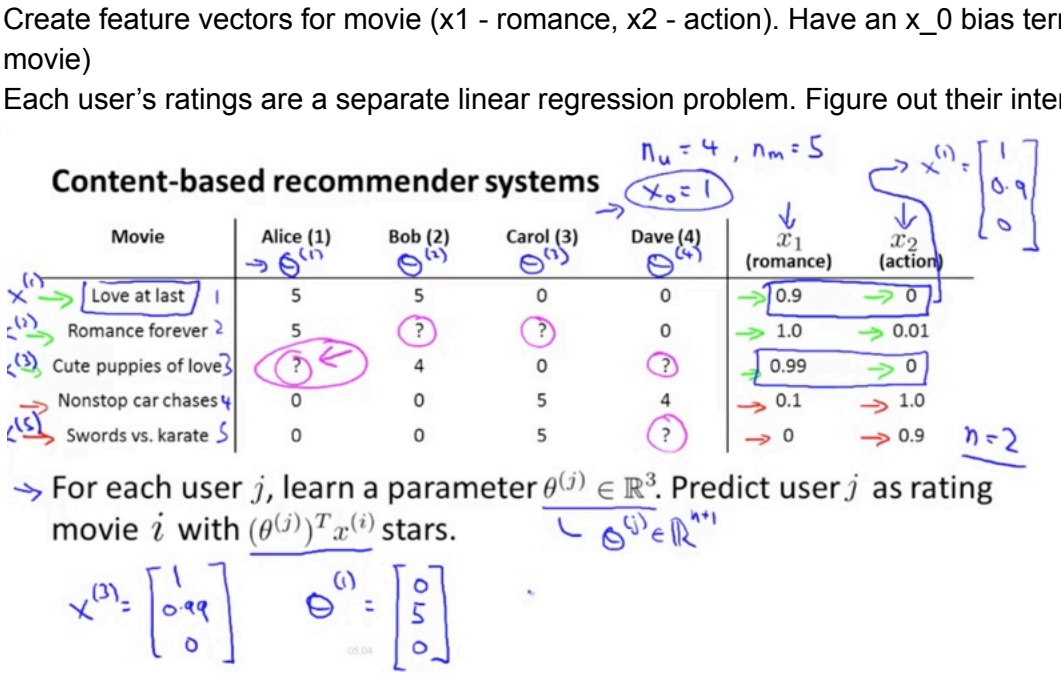
Recommendation System



Recommendation problem: predict how an existing user will rate a new movie.

Approach:

- Create feature vectors for movie (x1 - romance, x2 - action). Have an x_0 bias term (maybe someone hates/loves every movie)
- Each user's ratings are a separate linear regression problem. Figure out their internal utility function (based on x1, x2).



- Thought: Alice may not have seen many movies. Idea: create clusters of users. Then assign user to a cluster, and use that to get a better model and train it that way. (Is that collaborative filtering?)

Problem formulation

- $r(i,j) = 1$ if user j has rated movie i (0 otherwise)
- $y^{(i,j)}$ = rating by user j on movie i (if defined)
- $\theta^{(j)}$ = parameter vector for user j
- $x^{(i)}$ = feature vector for movie i
- For user j, movie i, predicted rating: $(\theta^{(j)})^T(x^{(i)})$
- $m^{(j)}$ = no. of movies rated by user j

To learn $\theta^{(j)}$:

$$\min_{\theta^{(j)}} \frac{1}{2m^{(j)}} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T(x^{(i)}) - y^{(i,j)}\right)^2 + \frac{\lambda}{2m^{(j)}} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

Optimization objective:

To learn $\theta^{(j)}$ (parameter for user j):

$$\min_{\theta^{(j)}} \frac{1}{2} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

To learn $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(n_u)}$:

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

Optimization algorithm:

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \underbrace{\frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2}_{J(\theta^{(1)},\ldots,\theta^{(n_u)})}$$

Gradient descent update:

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} \quad (\text{for } k = 0)$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \right) \quad (\text{for } k \neq 0)$$

Ok. Just do linear regression on each person as a unique problem.

- “Content-based recommendation” -- using features of CONTENT to determine regression. But not always feasible -- you might not have the information tagged in such a way.

Collaborative Filtering

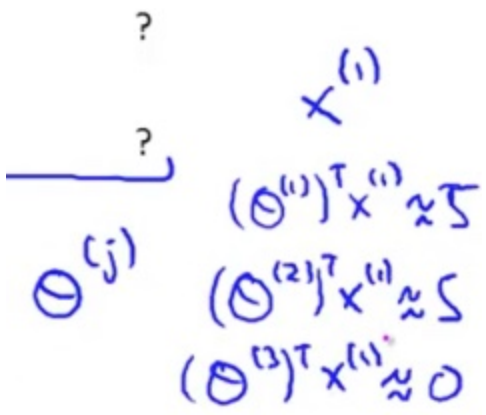
Goal: Learn features automatically.

Problem motivation

Movie	Alice (1) $\theta^{(1)}$	Bob (2) $\theta^{(2)}$	Carol (3) $\theta^{(3)}$	Dave (4) $\theta^{(4)}$	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	?	?
Romance forever	5	?	?	0	?	?
Cute puppies of love	?	4	0	?	?	?
Nonstop car chases	0	0	5	4	?	?
Swords vs. karate	0	0	5	?	?	?

$$\theta^{(1)} = \begin{bmatrix}0\\5\\0\end{bmatrix}, \theta^{(2)} = \begin{bmatrix}0\\5\\0\end{bmatrix}, \theta^{(3)} = \begin{bmatrix}0\\0\\5\end{bmatrix}, \theta^{(4)} = \begin{bmatrix}0\\0\\5\end{bmatrix}$$

- We don’t know the implicit features x1, x2.
  - If users can specify how much they like action, romance, etc. (they love action action movies, they love romantic movies, etc.)
  - Ah… can solve for the features for each movie. Assuming theta is set (don’t like this assumption, but ok...), find the hidden features



- Choose features x_i that minimizes the squared error. Have regularization term to prevent features from getting too large. (Just like solving for a different theta… we have swapped what theta and x are. Just a different unknown.)

Optimization algorithm

Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$, to learn $x^{(i)}$:

$$\min_{x^{(i)}} \frac{1}{2} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2$$

Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$, to learn $x^{(1)}, \ldots, x^{(n_m)}$:

$$\min_{x^{(1)}, \ldots, x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

-
- Sum over every user (use data from all users to update the parameters)

Collaborative filtering

Given $x^{(1)}, \ldots, x^{(n_m)}$ (and movie ratings), can estimate $\theta^{(1)}, \ldots, \theta^{(n_u)}$

Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$, can estimate $x^{(1)}, \ldots, x^{(n_m)}$

Guess $\Theta \rightarrow X \rightarrow \Theta \rightarrow X \rightarrow \Theta \rightarrow X \rightarrow \cdots$

- Pick a random theta, then get the features, then get a better theta, then better features…
  - There is a more efficient algo that does both (best theta, best x):

Collaborative filtering optimization objective

Given $x^{(1)}, \ldots, x^{(n_m)}$, estimate $\theta^{(1)}, \ldots, \theta^{(n_u)}$:

$$\min_{\theta^{(1)}, \ldots, \theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$, estimate $x^{(1)}, \ldots, x^{(n_m)}$:

$$\min_{x^{(1)}, \ldots, x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

Minimizing $x^{(1)}, \ldots, x^{(n_m)}$ and $\theta^{(1)}, \ldots, \theta^{(n_u)}$ simultaneously:

$$J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

$$\min_{x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}} J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$$

-
- There is redundancy in the error terms. So just combine everything together. Total cost function in terms of features (x) and parameters theta.
- Convention: no bias term here (interesting -- why?).
  - Ah… we can choose a feature x1 = 1 if needed. It can decide to have a bias term if we need.
  - Gradient update -- partial derivative of the cost function

Collaborative filtering algorithm

1. Initialize $x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}$ to small random values.
2. Minimize $J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$ using gradient descent (or an advanced optimization algorithm). E.g. for every $j = 1, \ldots, n_u, i = 1, \ldots, n_m$:

$$x_k^{(i)} := x_k^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k^{(j)} + \lambda x_k^{(i)} \right)$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \right)$$

3. For a user with parameters θ and a movie with (learned) features x, predict a star rating of $\theta^T x$.

Andrew Ng

- Ultimately, we create a linear predictive model of a user's movie ratings based on the features of the movie. We learn a theta (weighting) for each user.
  - Again, initialize to small random values -- symmetry breaking

Low Rank Matrix Factorization

Put everyone's ratings into a matrix

Collaborative filtering

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & ? \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}$$

Predicted ratings:

$$\begin{bmatrix} (\theta^{(1)})^T(x^{(1)}) & (\theta^{(2)})^T(x^{(1)}) & \ldots & (\theta^{(n_u)})^T(x^{(1)}) \\ (\theta^{(1)})^T(x^{(2)}) & (\theta^{(2)})^T(x^{(2)}) & \ldots & (\theta^{(n_u)})^T(x^{(2)}) \\ \vdots & \vdots & \vdots & \vdots \\ (\theta^{(1)})^T(x^{(n_m)}) & (\theta^{(2)})^T(x^{(n_m)}) & \ldots & (\theta^{(n_u)})^T(x^{(n_m)}) \end{bmatrix}$$

Y = actual ratings for user (col) and movie (row)
Predictions are based on the features and the user's weights
Can vectorize this. Movie features in rows (transpose). Then Theta matrix as well:

Collaborative filtering

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & ? & ? & 0 \\ ? & 4 & 0 & ? \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}$$

Predicted ratings:

$$\begin{bmatrix} (\theta^{(1)})^T(x^{(1)}) & (\theta^{(2)})^T(x^{(1)}) & \ldots & (\theta^{(n_u)})^T(x^{(1)}) \\ (\theta^{(1)})^T(x^{(2)}) & (\theta^{(2)})^T(x^{(2)}) & \ldots & (\theta^{(n_u)})^T(x^{(2)}) \\ \vdots & \vdots & \vdots & \vdots \\ (\theta^{(1)})^T(x^{(n_m)}) & (\theta^{(2)})^T(x^{(n_m)}) & \ldots & (\theta^{(n_u)})^T(x^{(n_m)}) \end{bmatrix}$$

$$X = \begin{bmatrix} -(x^{(1)})^T- \\ -(x^{(2)})^T- \\ \vdots \\ -(x^{(n_m)})^T- \end{bmatrix} \qquad \Theta = \begin{bmatrix} -(\theta^{(1)})^T- \\ -(\theta^{(2)})^T- \\ \vdots \\ -(\theta^{(n_u)})^T- \end{bmatrix}$$

Andrew Ng

Then with the data/operations mix we want, we do X Theta^T

"Low rank matrix factorization"

Again, the intuition here is that we are just pushing data through operations. Rewrite it to make it easier. Thinking about "linear functions on basis vectors" doesn't help. We are munging data here.

Let $X = \begin{bmatrix} - & (x^{(1)})^T & - \\ & \vdots & \\ - & (x^{(n_m)} & - \end{bmatrix}$, $\Theta = \begin{bmatrix} - & (\theta^{(1)})^T & - \\ & \vdots & \\ - & (\theta^{(n_u)} & - \end{bmatrix}$.

What is another way of writing the following:

$$\begin{bmatrix} (x^{(1)})^T(\theta^{(1)}) & \dots & (x^{(1)})^T(\theta^{(n_u)}) \\ \vdots & \ddots & \vdots \\ (x^{(n_m)})^T(\theta^{(1)}) & \dots & (x^{(n_m)})^T(\theta^{(n_u)}) \end{bmatrix}$$

X (as operation) working on Theta Transpose (change the horizontal rows to data columns)

- $X\Theta$
- $X^T\Theta$
- $X\Theta^T$

Correct Response

- Find related movies
  - Collect the x1 = romance, x2=action … features. We have a score for each.
  - Find other movies where ||x(i) - x(j)|| -- distance between them is small.

Finding related movies

For each product i, we learn a feature vector $x^{(i)} \in \mathbb{R}^n$.

→ x_1 = romance, x_2 = action, x_3 = comedy, x_4 = ...

How to find movies j related to movie i?

small $\|x^{(i)} - x^{(j)}\|$ → movie j and i are "similar"

5 most similar movies to movie i:

Find the 5 movies j with the smallest $\|x^{(i)} - x^{(j)}\|$.

  -
  - Again, this is “just” Pythagorean theorem once we have the feature vector.

Users who have not rated any movies -- get the average preference for each vector? (prediction)

- Ah… we get zero’s (because regularization). We are rewarded for having no parameters (since there is no error term in the front)
-

Users who have not rated any movies

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	Eve (5)
Love at last	5	5	0	0	?
Romance forever	5	?	?	0	?
Cute puppies of love	?	4	0	?	?
Nonstop car chases	0	0	5	4	?
Swords vs. karate	0	0	5	?	?

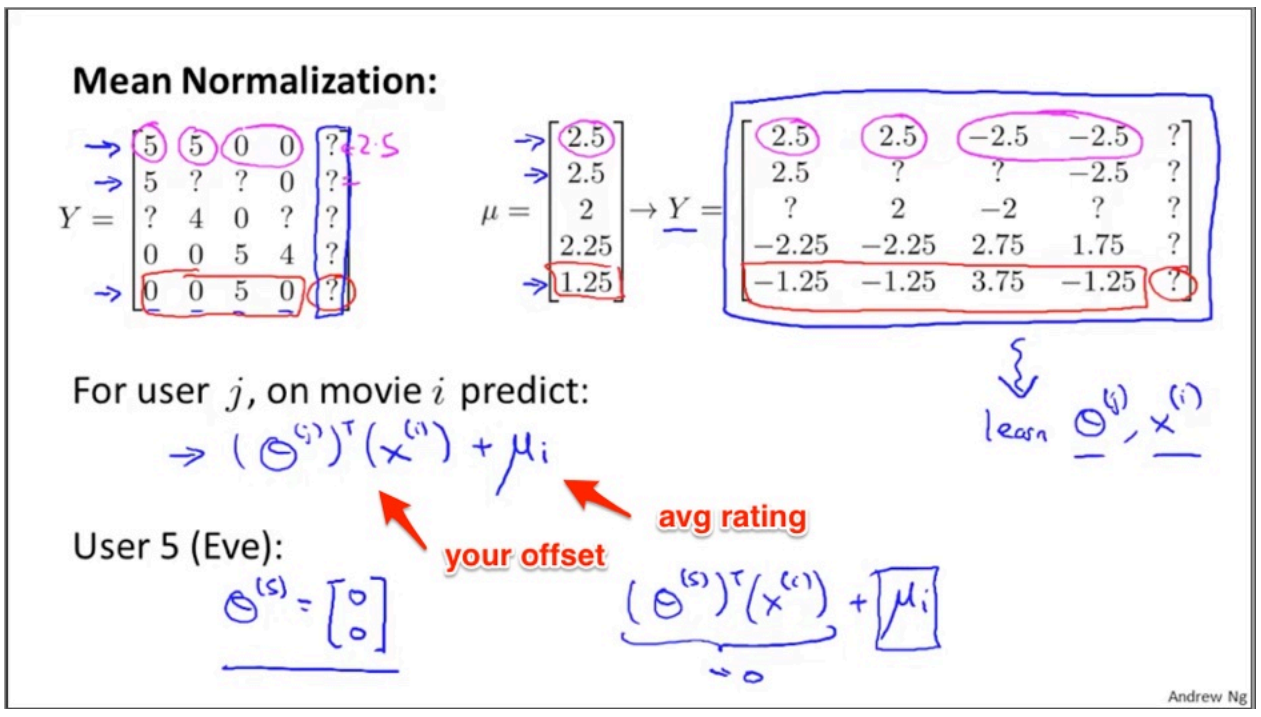
$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix}$$

$$\min_{\substack{x^{(1)},\dots,x^{(n_m)} \\ \theta^{(1)},\dots,\theta^{(n_u)}}} \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

$n=2$ $\theta^{(5)} \in \mathbb{R}^2$ $\theta^{(5)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ $\frac{\lambda}{2}\left[(\theta_1^{(5)})^2 + (\theta_2^{(5)})^2\right]$ ←

$(\theta^{(5)})^T x^{(i)} = 0$

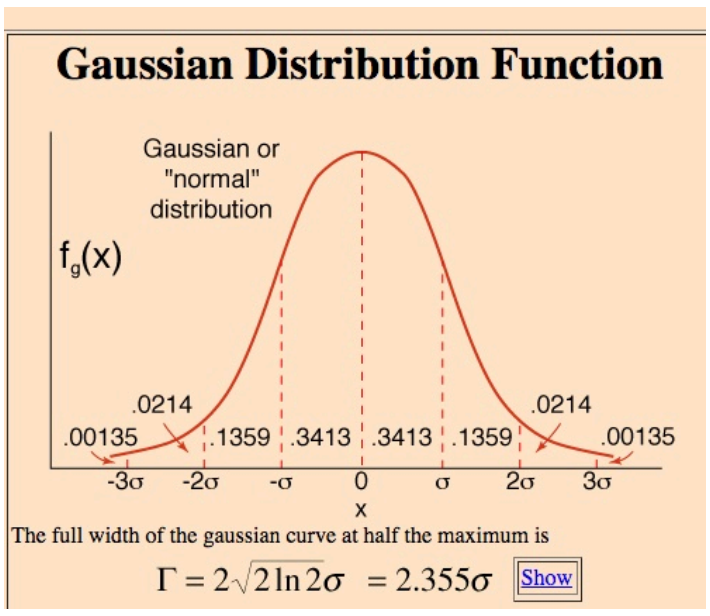
- Mean Normalization:
  - Take matrix of movie ratings, subtract average, so avg rating is 0 (midpoint)
  - Then filter on the normalized version.
  - Your prediction is:
    - Your offset plus the mean (aha!)



- Then, if you have no offset term, we expect you to rate it the average rating

Homework

Remember Gaussian Distribution:



following equations. To estimate the mean, you will

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)},$$

and for the variance you will use:

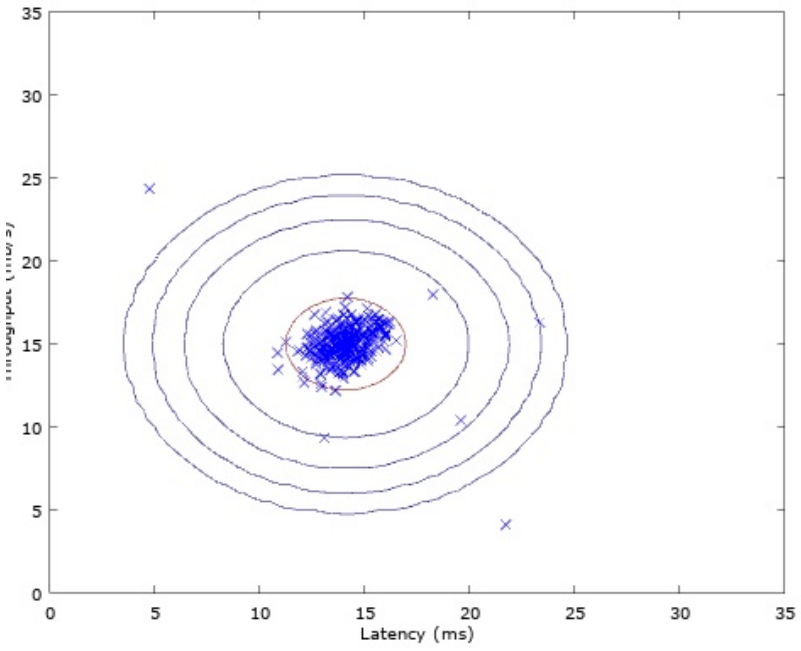
$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(x_i^{(j)} - \mu_i)^2.$$

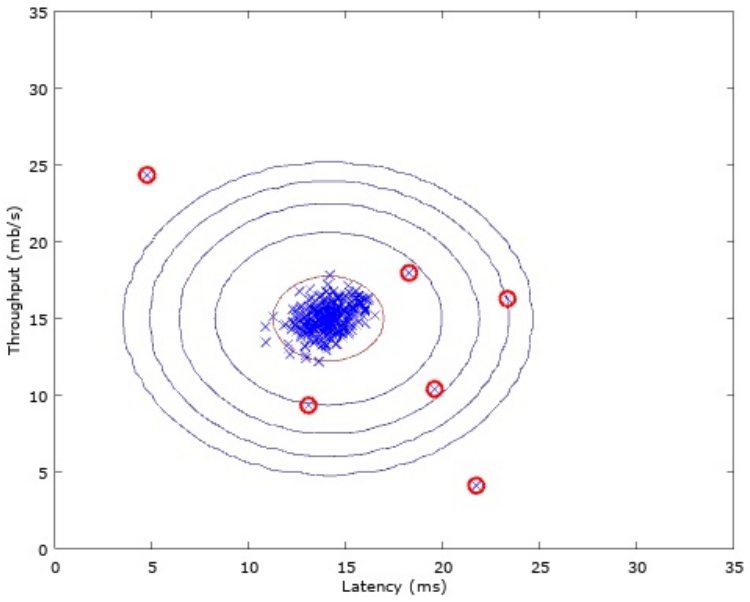
- Just getting the outliers using statistics

```
function [mu sigma2] = estimateGaussian(X)
%ESTIMATEGAUSSIAN This function estimates the parameters of a
%Gaussian distribution using the data in X
%   [mu sigma2] = estimateGaussian(X),
%   The input X is the dataset with each n-dimensional data point in one row
%   The output is an n-dimensional vector mu, the mean of the data set
%   and the variances sigma^2, an n x 1 vector
%

% Useful variables
[m, n] = size(X);

% You should return these values correctly
mu = zeros(n, 1);
sigma2 = zeros(n, 1);

% ====================== YOUR CODE HERE ======================
% Instructions: Compute the mean of the data and the variances
%               In particular, mu(i) should contain the mean of
%               the data for the i-th feature and sigma2(i)
%               should contain variance of the i-th feature.
%

mu = mean(X)
sigma2 = var(X) * (m - 1) / m

% =============================================================


end
```

ML uses non-standard stats (sigh)

-
- Remember, the Gaussian gives a probability of that exact value. The integral of all values within 1 sigma is 68%, and integral of all values at 2 sigma is 95%. A specific probability of an exact value will be small however.





- We find the best epsilon (threshold) that gives us the a good precision / recall. Balance between true pos, false pos:

threshold classifies correctly and incorrectly.

The F_1 score is computed using precision ($prec$) and recall (rec):

$$F_1 = \frac{2 \cdot prec \cdot rec}{prec + rec}, \tag{3}$$

You compute precision and recall by:

$$prec = \frac{tp}{tp + fp} \tag{4}$$

$$rec = \frac{tp}{tp + fn}, \tag{5}$$

where

- tp is the number of true positives: the ground truth label says it's an anomaly and our algorithm correctly classified it as an anomaly.
- fp is the number of false positives: the ground truth label says it's not an anomaly, but our algorithm incorrectly classified it as an anomaly.
- fn is the number of false negatives: the ground truth label says it's an anomaly, but our algorithm incorrectly classified it as not being anomalous.

```
function [bestEpsilon bestF1] = selectThreshold(yval, pval)
%SELECTTHRESHOLD Find the best threshold (epsilon) to use for selecting
%outliers
%   [bestEpsilon bestF1] = SELECTTHRESHOLD(yval, pval) finds the best
%   threshold to use for selecting outliers based on the results from a
%   validation set (pval) and the ground truth (yval).
%

bestEpsilon = 0;
bestF1 = 0;
F1 = 0;

stepsize = (max(pval) - min(pval)) / 1000;
for epsilon = min(pval):stepsize:max(pval)

    % ====================== YOUR CODE HERE ======================
    % Instructions: Compute the F1 score of choosing epsilon as the
    %               threshold and place the value in F1. The code at the
    %               end of the loop will compare the F1 score for this
    %               choice of epsilon and set it to be the best epsilon if
    %               it is better than the current choice of epsilon.
    %
    % Note: You can use predictions = (pval < epsilon) to get a binary vector
    %       of 0's and 1's of the outlier predictions

    predictions = (pval < epsilon);
    tp = sum((predictions == 1) & (yval == 1));
    fp = sum((predictions == 1) & (yval == 0));
    fn = sum((predictions == 0) & (yval == 1));
    prec = tp / (tp + fp);
    rec = tp / (tp + fn);

    F1 = 2 * prec * rec / (prec + rec);

    % =============================================================

    if F1 > bestF1
       bestF1 = F1;
       bestEpsilon = epsilon;
    end
end

end
```

- Aha: There's a lot of juggling X, Theta, and the size of various matrices
  - [5 x ?] [4 x 3]
    - If we want to match, we need 4. Think of it as "4 params, 4 data points". That's where they meet
    - [5 x 4][4 x 3]
    - The result is a [5 x 3], the outside elements. Have to reverse engineer what we want a bit.

```
% end

E = (X * Theta' - Y) .* R;      % get errors, filtered to R = 1
J = (1/2) * sum(sum(E .^ 2));


X_grad = E * Theta;
Theta_grad = E' * X;

% =============================================================

grad = [X_grad(:); Theta_grad(:)];

```

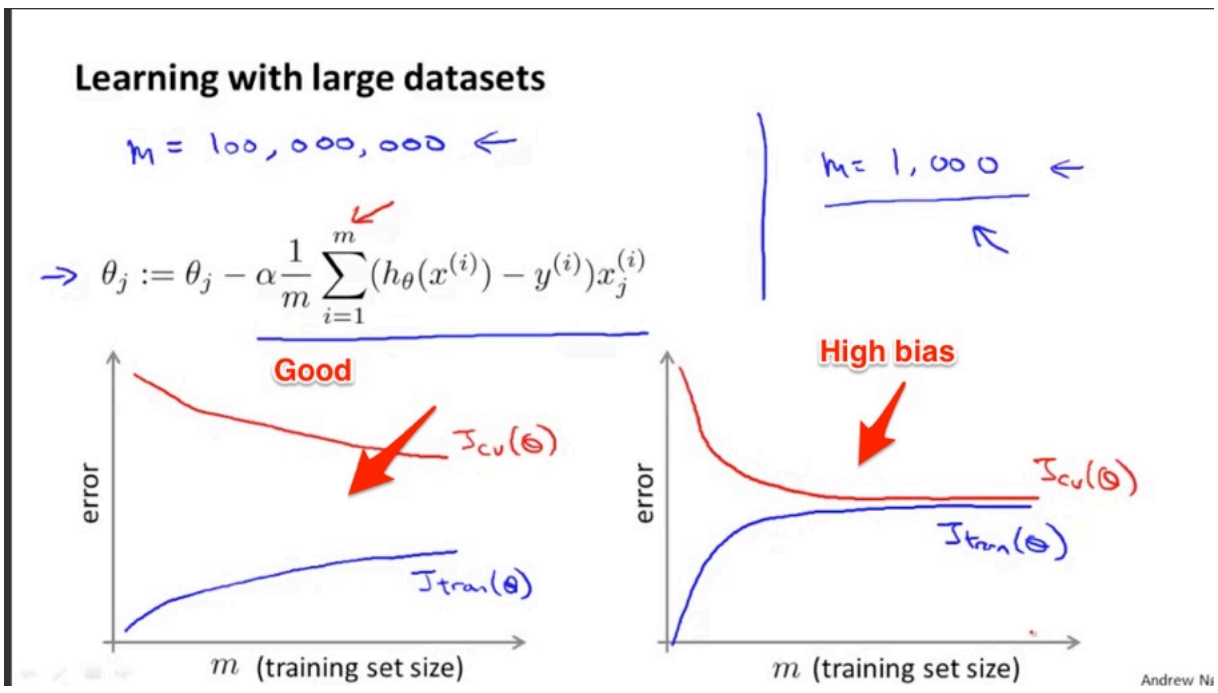
> **Implementation Note:** You can get full credit for this assignment without using a vectorized implementation, but your code will run much more slowly (a small number of hours), and so we recommend that you try to vectorize your implementation.

- Cool! Can train on your own movie preferences.
    - Would be neat to have a web app online where you can get the trained output (and show the features that led to it).

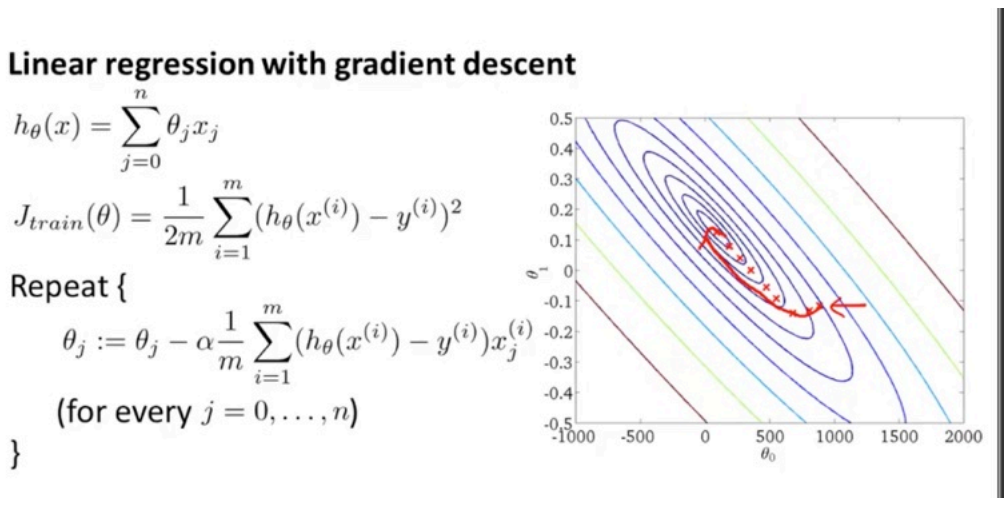
```
Training collaborative filtering...
Iteration    100 | Cost: 7.211082e+04
Recommender system learning completed.

Program paused. Press enter to continue.

Top recommendations for you:
Predicting rating 8.5 for movie Star Wars (1977)
Predicting rating 8.4 for movie Titanic (1997)
Predicting rating 8.3 for movie Shawshank Redemption, The (1994)
Predicting rating 8.2 for movie Raiders of the Lost Ark (1981)
Predicting rating 8.2 for movie Schindler's List (1993)
Predicting rating 8.1 for movie Good Will Hunting (1997)
Predicting rating 8.1 for movie Usual Suspects, The (1995)
Predicting rating 8.1 for movie Empire Strikes Back, The (1980)
Predicting rating 8.0 for movie Braveheart (1995)
Predicting rating 8.0 for movie Wrong Trousers, The (1993)


Original ratings provided:
Rated 4 for Toy Story (1995)
Rated 3 for Twelve Monkeys (1995)
Rated 5 for Usual Suspects, The (1995)
Rated 4 for Outbreak (1995)
Rated 5 for Shawshank Redemption, The (1994)
Rated 3 for While You Were Sleeping (1995)
Rated 5 for Forrest Gump (1994)
Rated 2 for Silence of the Lambs, The (1991)
Rated 4 for Alien (1979)
Rated 5 for Die Hard 2 (1990)
Rated 5 for Sphere (1998)
```

Week 10 - Large Scale Machine Learning

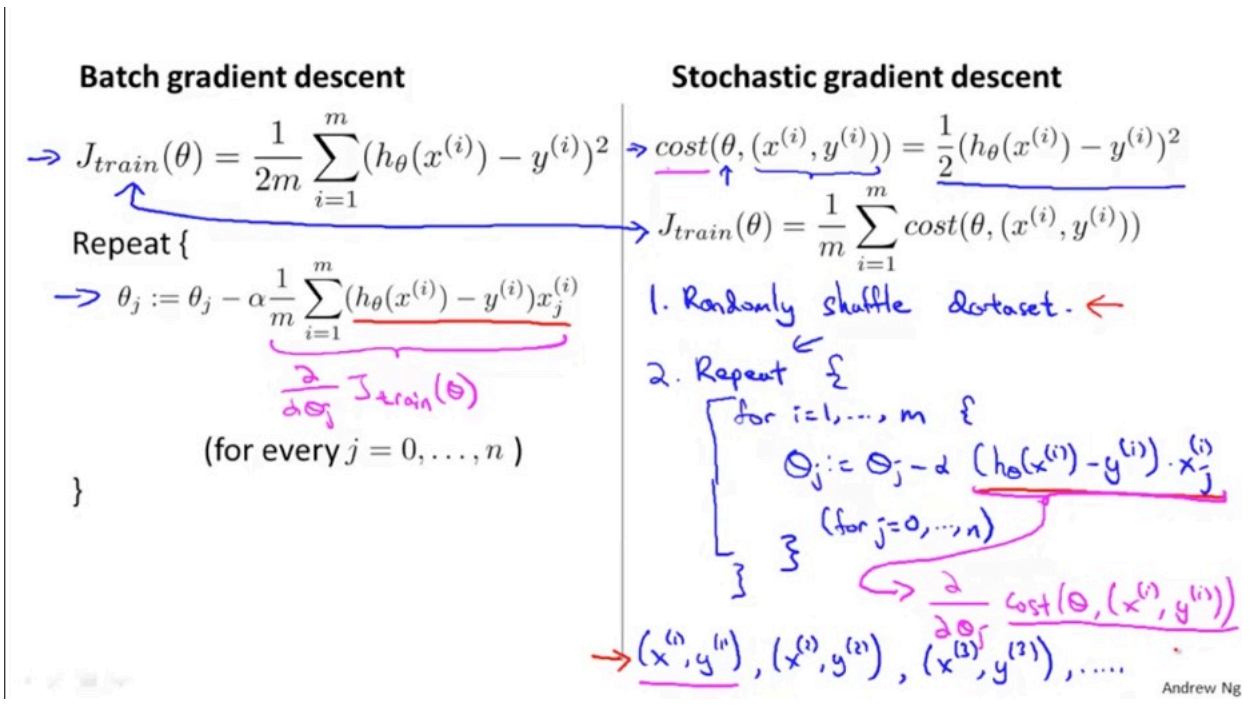
- Machine learning better now b/c of larger data sets
    - Train low-bias algo on lots of data
    - “Not who has the best algo, but the best data”
    - Large data sets (m=100M)
        - Gradient-descent O(n) to compute derivative [impact of each sample on derivative]
    - Why not train on m=1000 to start?
        - Sanity check, make sure variance/error is higher when m gets larger



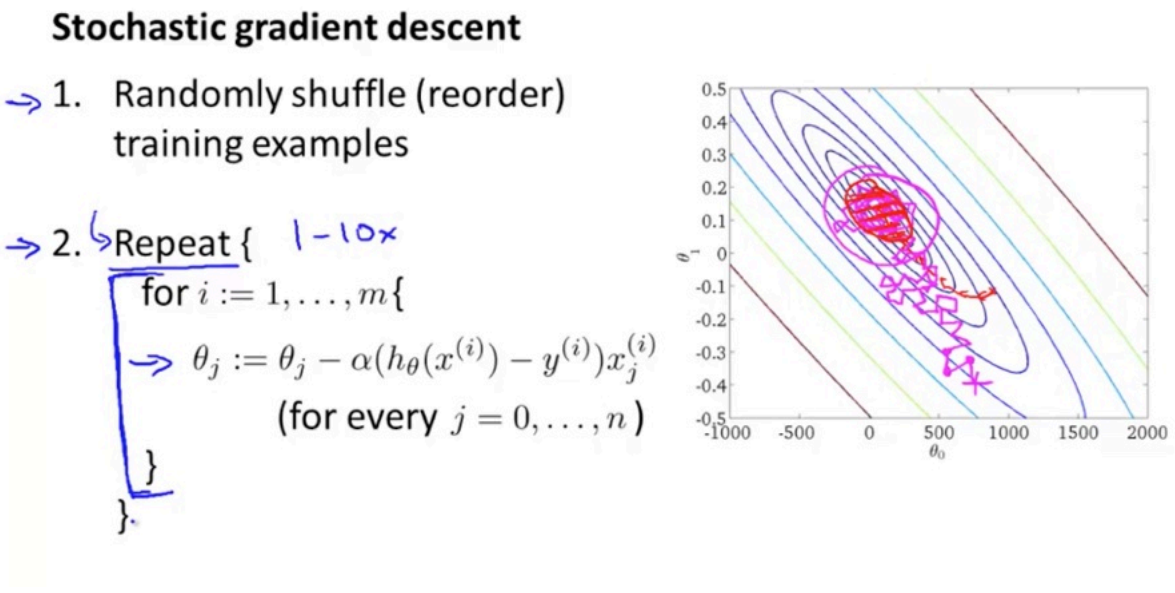
    - Huh: Have to remember the role of these curves. Want a gap between the training and cross-validation. Otherwise the “bias” / overfitting is happening?



- Gradient descent walks towards the local min. But computing the derivative term in O(n).
  - “Batch gradient descent”. Need to iterate through all records to compute derivative. Then take a small step.
  - “Stochastic gradient descent” - my guess -- take a sample of points and compute the gradient there.
    - Randomly shuffle the data set
    - Make a step that just fits the first example, then take a step, then look the second example, then take a step, and so on.
    - Don’t scan through all m=300M examples before adjusting parameters. Update with each example.



- Iterations are faster, but the direction isn’t perfectly optimized. Might zig-zag a bit.
- Doesn’t converge on exact global min, but get parameter near the global min.



-
- Take 1-10 passes through data set.

Mini-batch gradient descent

- Batch: use all m examples
- Mini-batch: use b (< m) samples to compute new gradient
  - Use 10 examples at a time (for example)
- Gradient: use 1 sample to compute new gradient

Mini-batch gradient descent

Say $b = 10, m = 1000.$

Repeat {

for $i = 1, 11, 21, 31, \ldots, 991$ {

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

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

}

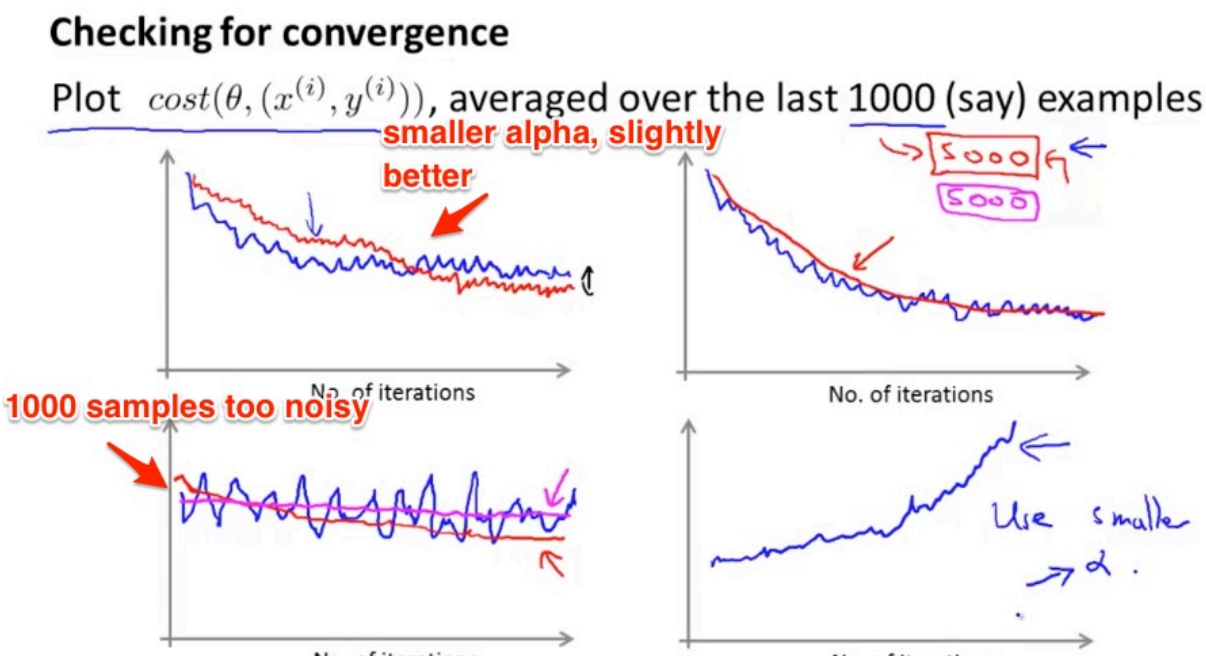
}

- Can vectorize the batch calculation (parallelize the computation). Cool.
- Tradeoff speed and accuracy, like always

Stochastic Gradient Descent Convergence

Check for convergence

- Get a running estimate of our cost (avg of last 1000 samples) to see if we're converging
  - Having a smaller learning rate, might get smaller oscillations around global min. Slightly better theta.
- If you take a data point every 5000 samples, slower feedback.
  - But faster samples may be too noisy. See that it's improving as we go over 5000 examples
- Algo could be diverging (use a smaller alpha)



- Can modify alpha (learning rate) as we go. As you get closer, smaller alpha. Not needed often times.

Learning rate α is typically held constant. Can slowly decrease α over time if we want θ to converge. (E.g. $\alpha = \frac{\text{const1}}{\text{iterationNumber} + \text{const2}}$) $\alpha \to 0$

Andrew N

Advanced Topics - Online Learning (aka continuous stream of data)

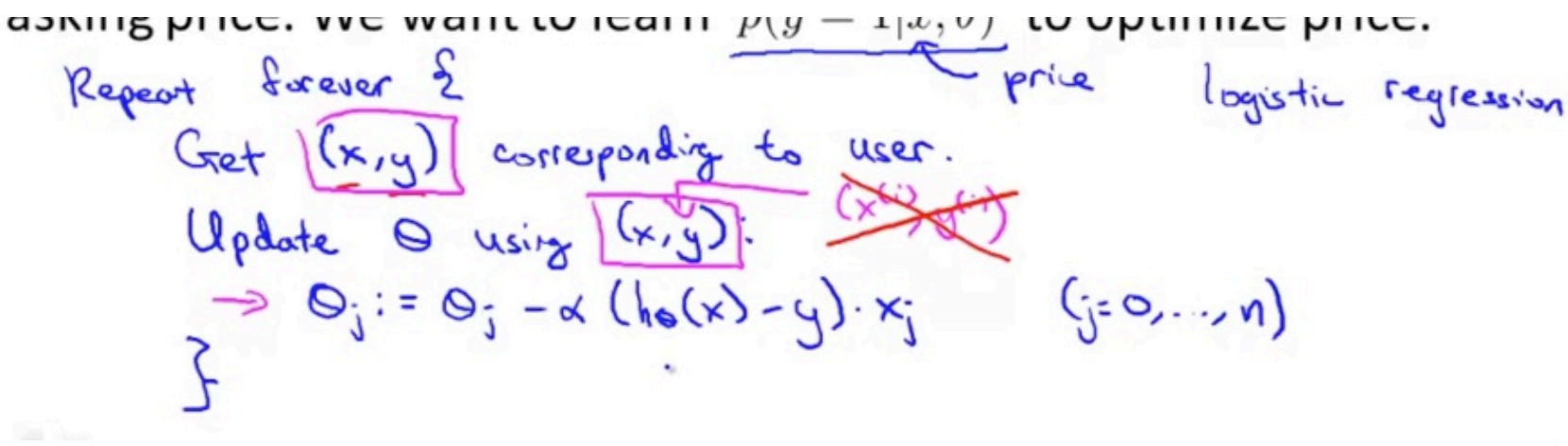
- Example: Based on price you offer, sometimes users use service (y=1) and other times do not (y=0)

Online learning

Shipping service website where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping service ($y = 1$), sometimes not ($y = 0$).

Features x capture properties of user, of origin/destination and asking price. We want to learn $p(y = 1|x; \theta)$ to optimize price.

-
- Logistic regression -- want a probability user will buy (y=1) given their feature vector (x) and params for regression (Theta)



- Continuous stream of users… they keep adjusting the parameter theta. Neat.
    - If only small number of users, then just save data set and analyze offline.
- This adapts to changing user preference.
    - Aha: Basically we have a running average for what theta should be

Example: Product Search

- Figure out 10 phones we should show based on search
    - Get features (properties of phone, number of keywords that match, etc.)
    - Then get probability user will click (y=1) given features x & theta
- Learn the predicted Click Through Rate

Other online learning example:

Product search (learning to search)

User searches for "Android phone 1080p camera"

Have 100 phones in store. Will return 10 results.

→ $x =$ features of phone, how many words in user query match name of phone, how many words in query match description of phone, etc.

→ $y = 1$ if user clicks on link. $y = 0$ otherwise.

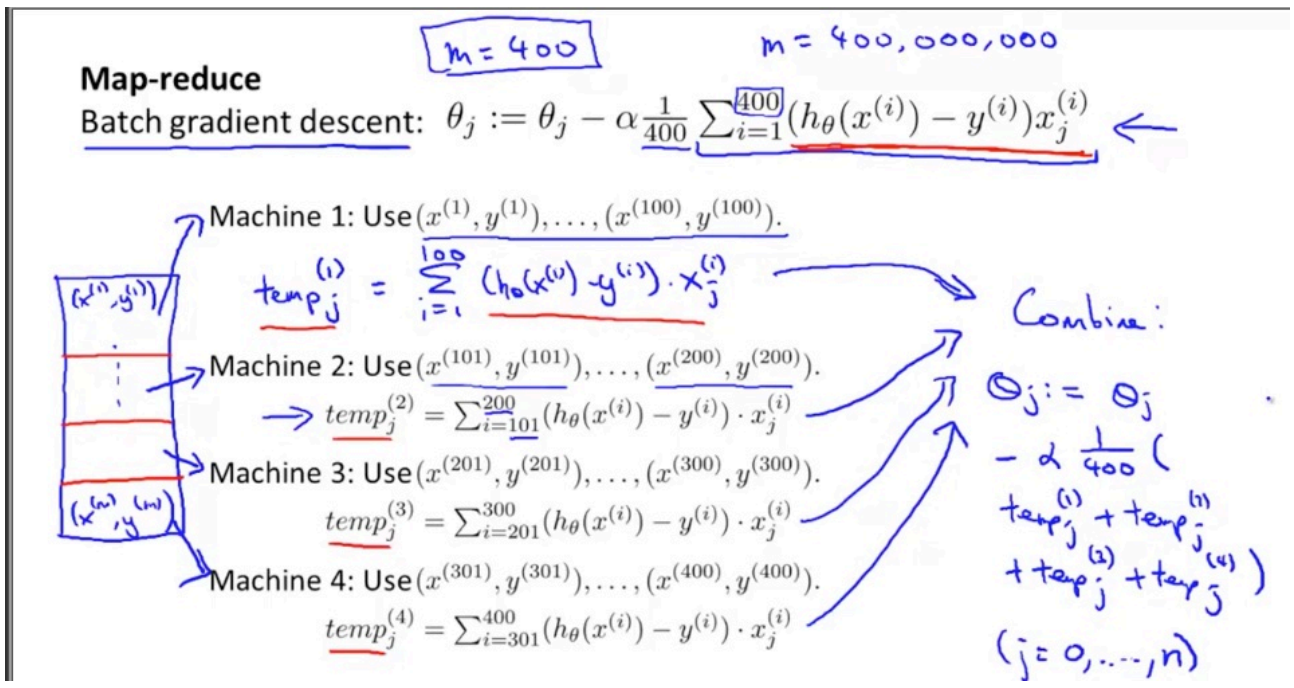
→ Learn $p(y = 1|x; \theta)$. predicted CTR

→ Use to show user the 10 phones they're most likely to click on.

- Show special offers, news items, etc. What is most likely to be clicked on based on user properties (browser, OS, mobile, country, etc…)
    - Or, can save data and analyze offline. Or just learn continuously for huge data sets.
    - We see this happening with YouTube recommendations

Map-Reduce & Data Parallelism

- Split the samples onto various machines. Use map/reduce for parallelism. OK. The calculation is easily parallelized because it's a linear computation of the various data points. Cool.



- In other words, we are vectorizing across machines. (The ideas appear over and over :-)... something is slow and linear, make it run in parallel.)
    - Can parallelize across machines, or cores

Week 11 - Photo OCR

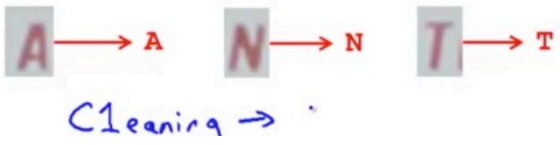
The Photo OCR problem

Photo OCR pipeline

1. Text detection

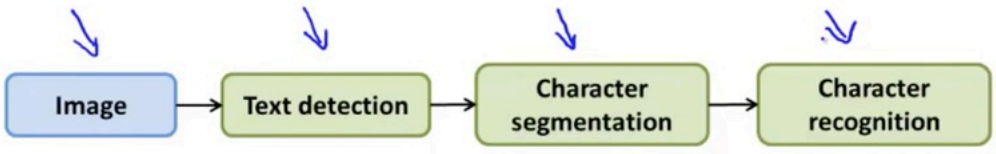
2. Character segmentation

3. Character classification

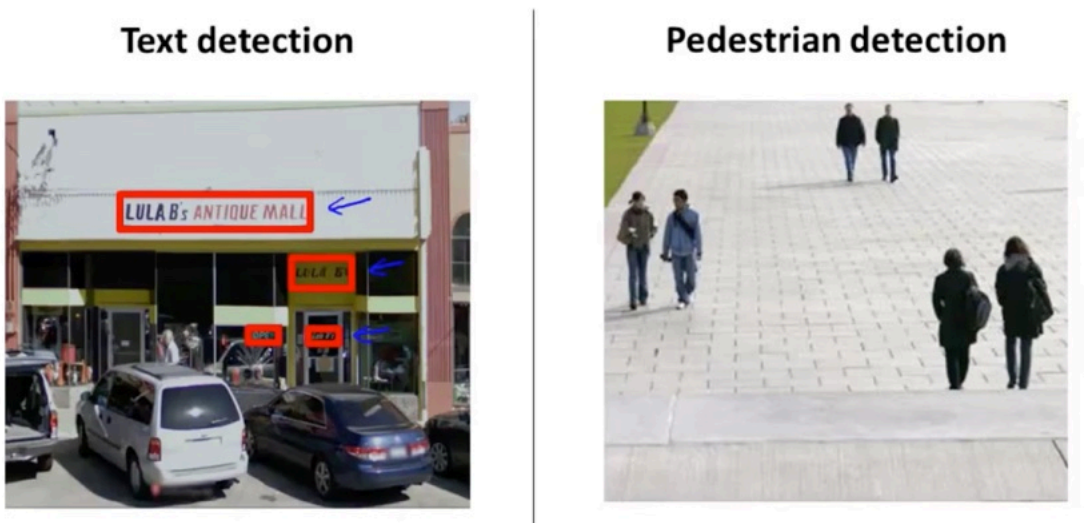


Machine learning pipeline

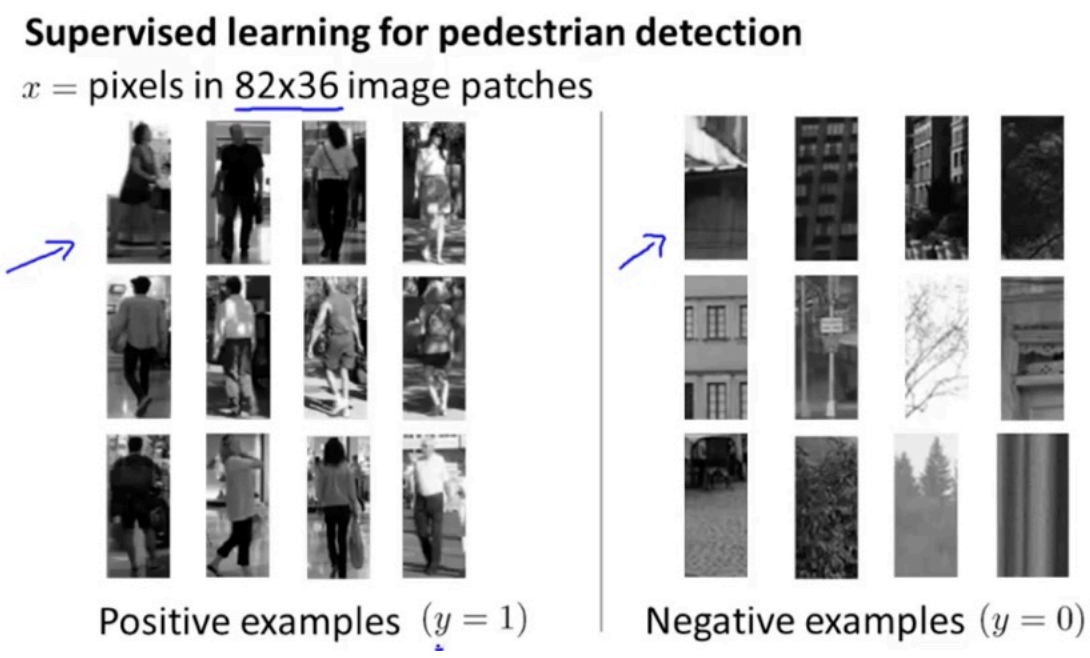
Photo OCR pipeline



- Thought: When we flatten a pixel area (20x20) into a single 400-item array, we can find correlations between #1 and #21, for example. The notion of a row/col grid is our own visualization.



- Simpler problem, detect pedestrians - aspect ratio is the same



- Train NN to take an image patch and classify y=1 or y=0.
- Then, create a window, attempt to classify, and slide around.

- Can take a larger patch, then resize down to 82x36, and classify.

Text detection

Positive examples $(y = 1)$

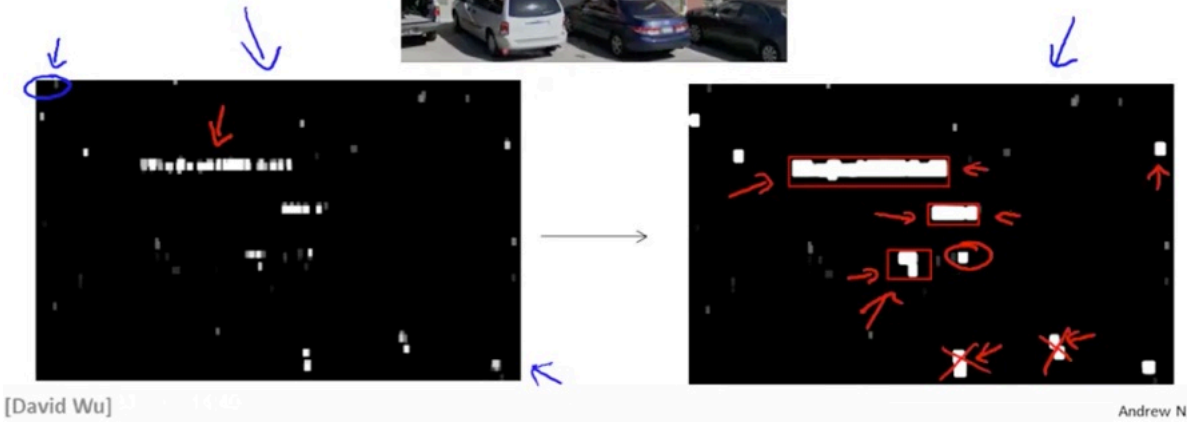
Negative examples $(y = 0)$

Text detection

[David Wu]

- Highlight probability text appears (0 to 1, black to grey to white)
- Then do the “expansion”. If you are within 5 px of an activated item, you are white too.

Text detection

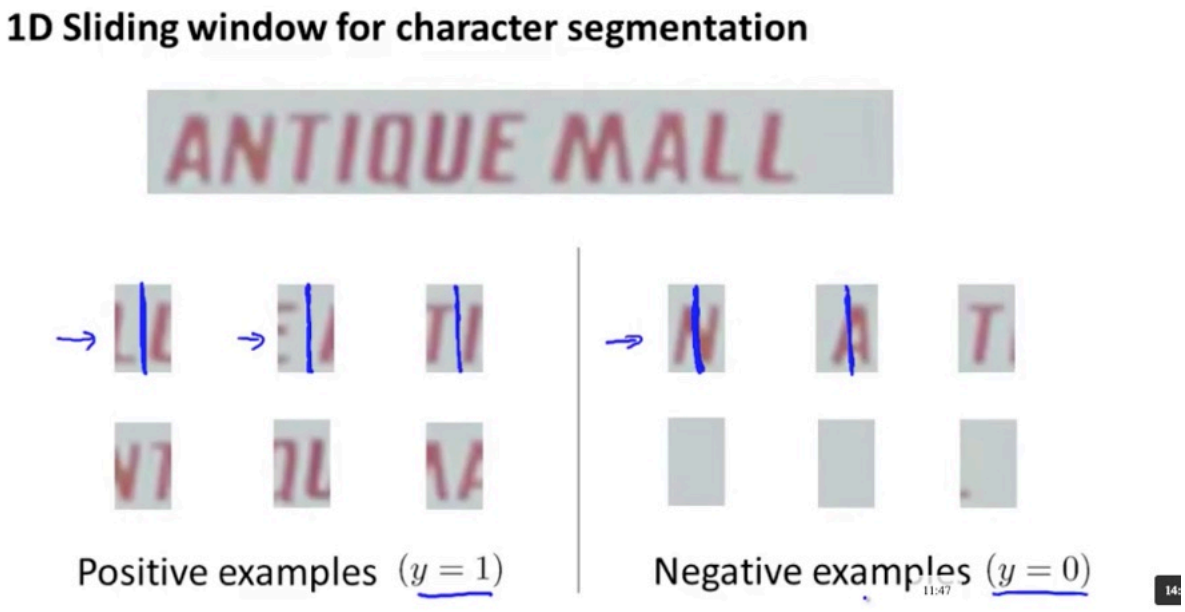


[David Wu]

Andrew N

Then draw rectangles around regions with right aspect ratio.

Try to detect a split between two characters (y=1)

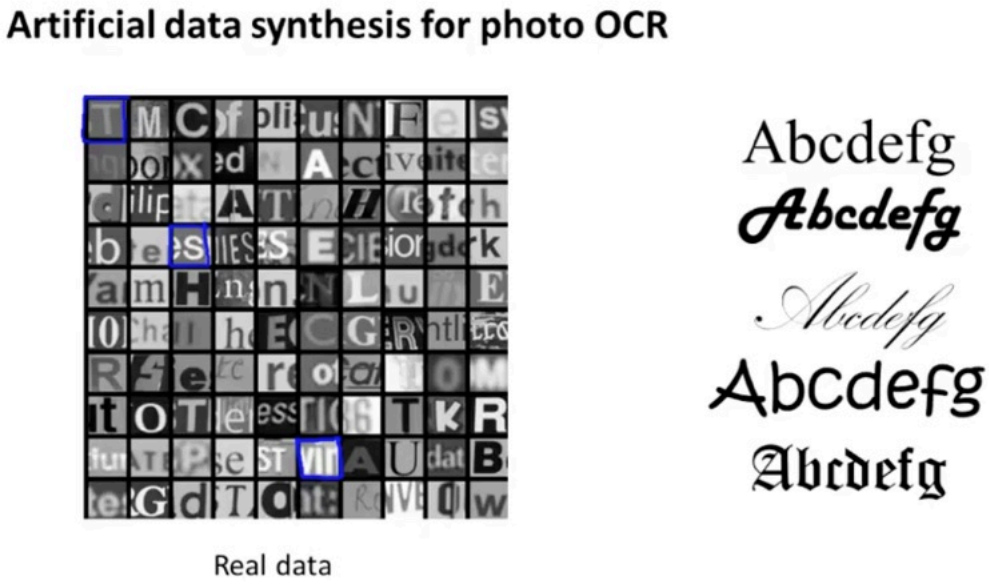


- “Should we split here?” (y=1 yes, y=0 no)

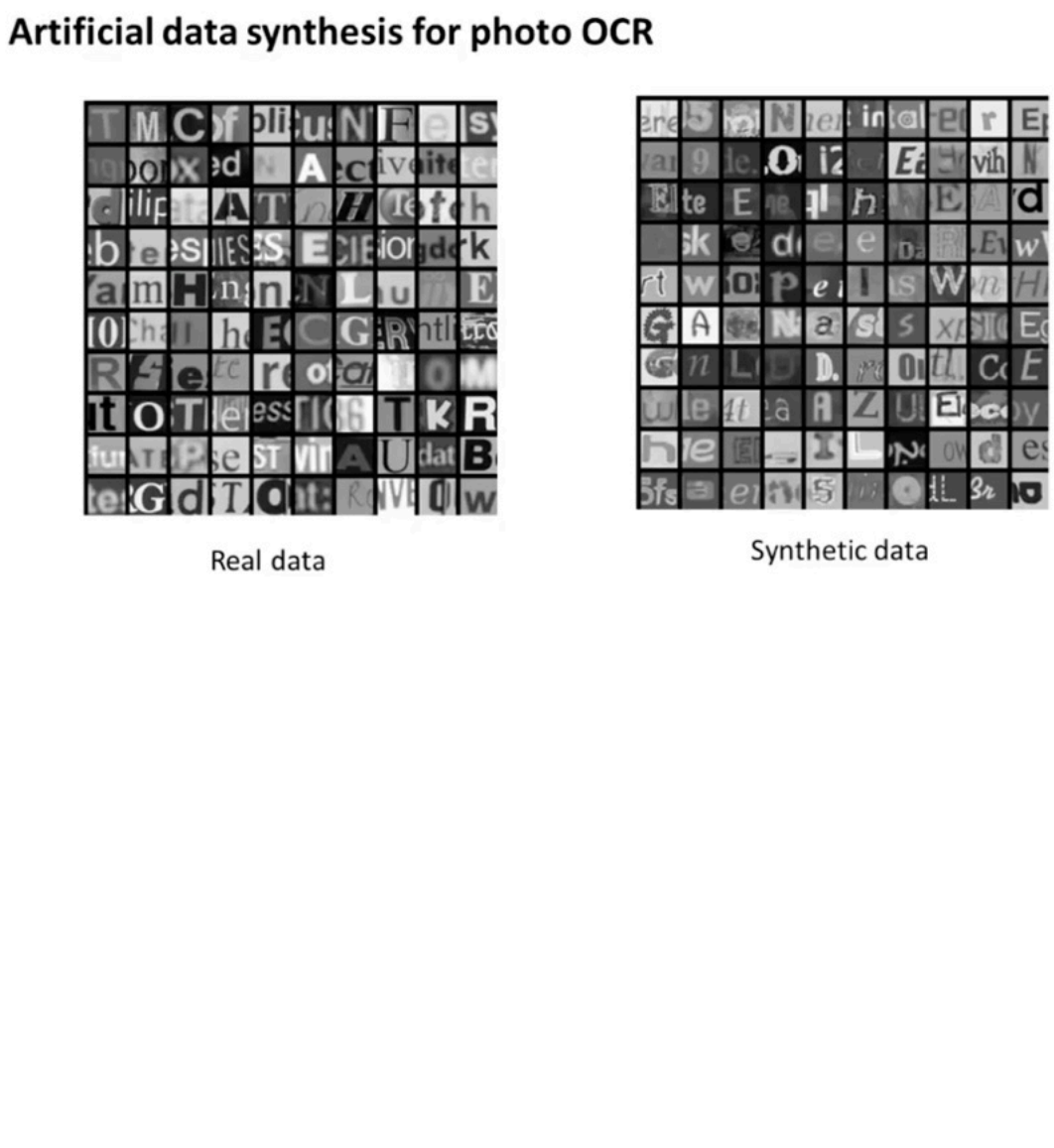
Gettings lots of data / artificial data

Artificial data synthesis

- Idea: imagine taking every font we can find, rendering it to images, blurring it, overlay it on various backgrounds, change aspect ratio, distort it, etc.. We generated the data set (instead of having to look for tagged data).
- Insight: using color doesn’t help that much. (This is key… color is interesting, but is it really needed? Greyscale has many values, do we need more precision than that? Even for self-driving cars.)

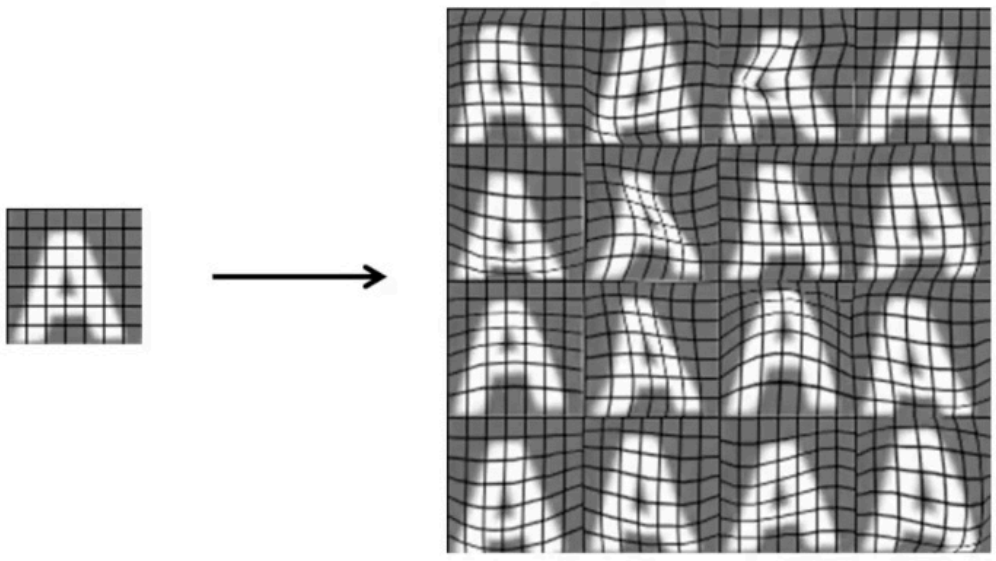


-
    - Yep, paste fonts against random backgrounds.



-

Synthesizing data by introducing distortions



-
    - Same idea, generate new artificial samples. Cool. (With Fourier Transform, could separate in interesting ways. Deconvolve.)
- He uses phrase “Amplify data set”.

Synthesizing data by introducing distortions: Speech recognition

Original audio:

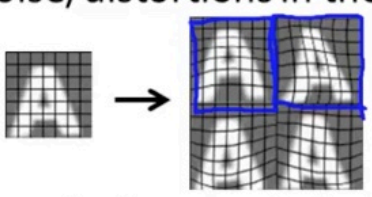
Audio on bad cellphone connection

Noisy background: Crowd

Noisy background: Machinery

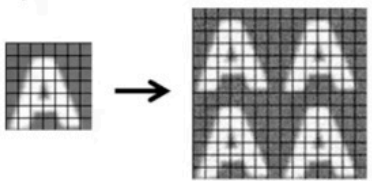
Synthesizing data by introducing distortions

→ Distortion introduced should be representation of the type of noise/distortions in the test set.



→ Audio:
Background noise,
bad cellphone connection

Usually does not help to add purely random/meaningless noise to your data.



x_i = intensity (brightness) of pixel i

$x_i \leftarrow x_i +$ random noise

- Don’t want random noise, want distortions representative of data.

Make sure you have a low-bias classifier before you add more data.

- “How much work to get 10x as much data as we currently have?” (A nice general question: how much work to get 10x as much traffic, performance, sales, etc. as we have?)

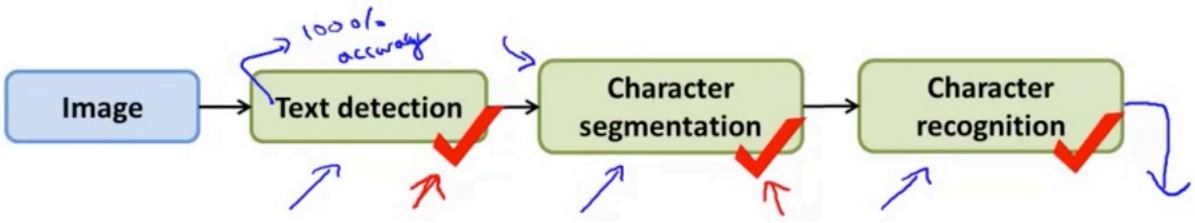
Discussion on getting more data

1. Make sure you have a low bias classifier before expending the effort. (Plot learning curves). E.g. keep increasing the number of features/number of hidden units in neural network until you have a low bias classifier.
2. "How much work would it be to get 10x as much data as we currently have?"
   - Artificial data synthesis
   - Collect/label it yourself
   - "Crowd source" (E.g. Amazon Mechanical Turk)

Ceiling Analysis - What part of the pipeline to work on next

Prediction: Find the limiting reagent. Which step is performing the worst?

Estimating the errors due to each component (ceiling analysis)



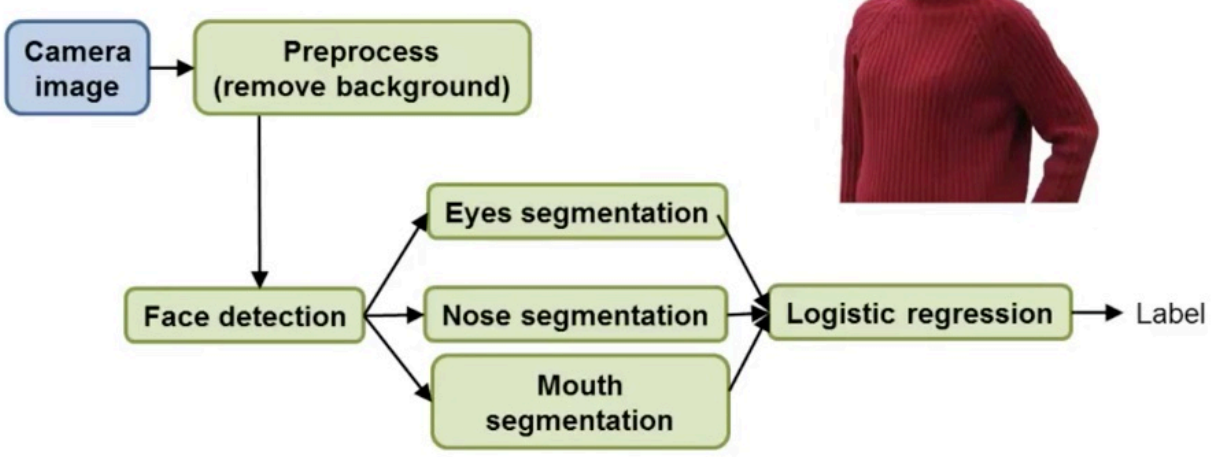
What part of the pipeline should you spend the most time trying to improve?

Component	Accuracy
Overall system	72%
Text detection	89%
Character segmentation	90%
Character recognition	100%

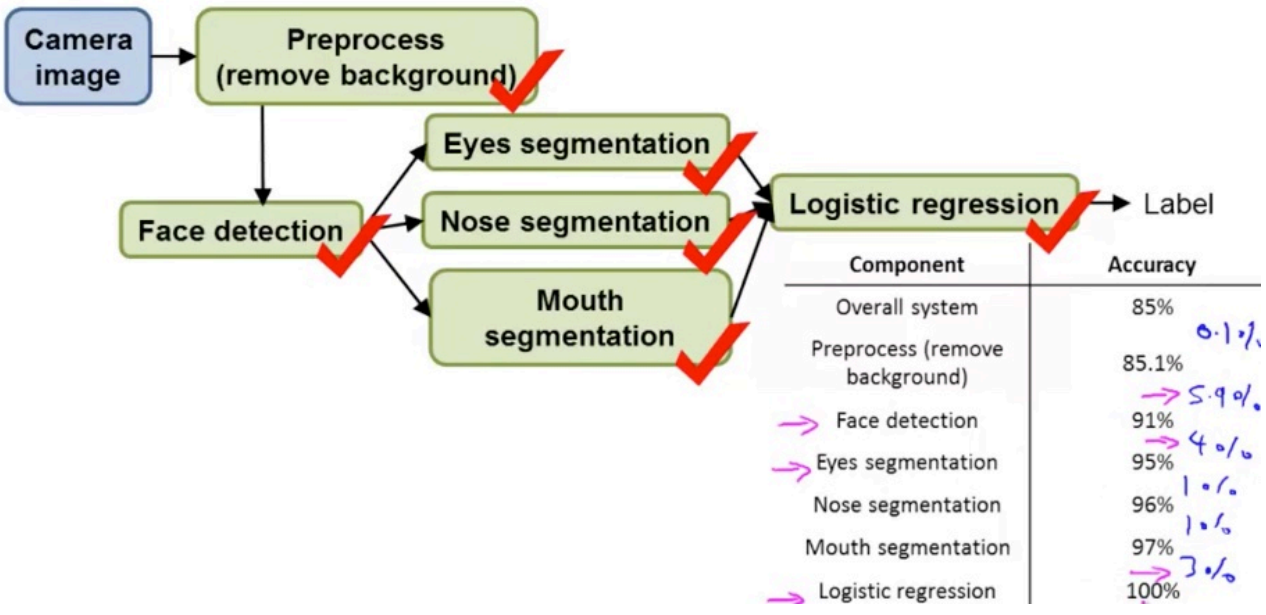
- Pretend each step has perfect accuracy (with your test set) and see what the accuracy of the system is. Checkbox, just give it the correct answers. See how the system behaves.
- Idea: Unit test the components. Given perfect behavior for the others, figure out how well we perform. Give it perfect text detection (keep the other steps as they were), we gain 17% accuracy. Then with perfect character segmentation we gain 1%. Then we have an integration test. We are mocking the behavior for each step.

Another ceiling analysis example

Face recognition from images
(Artificial example)



Another ceiling analysis example

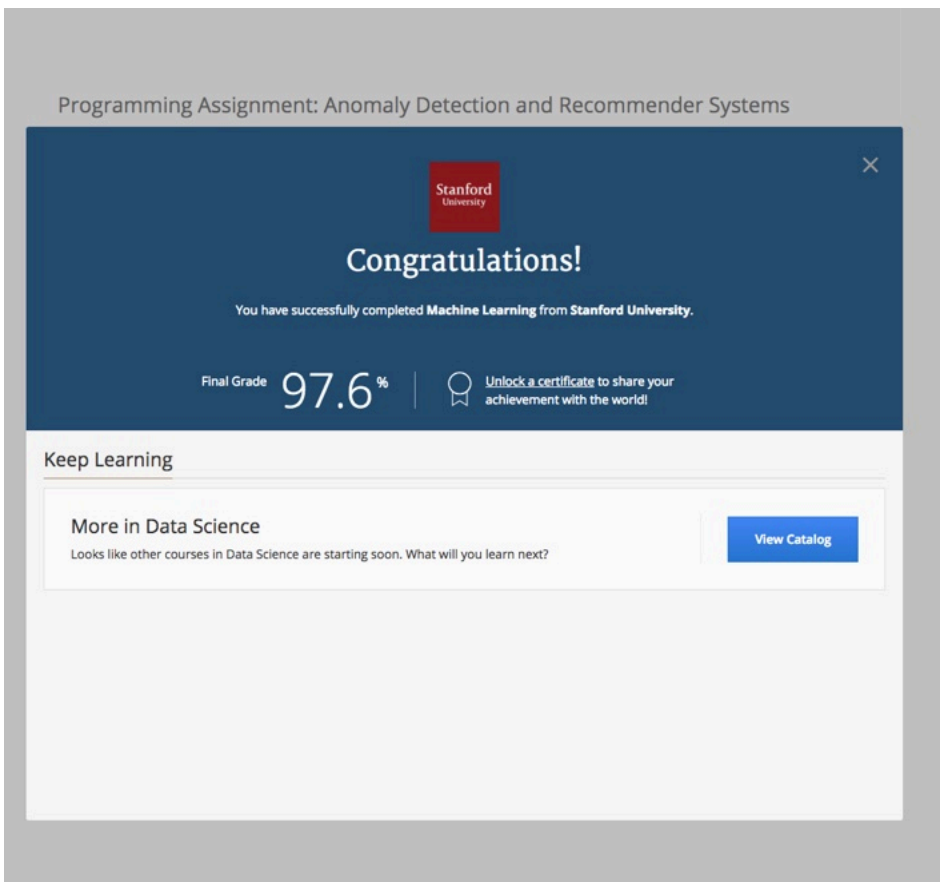


Component	Accuracy
Overall system	85%
Preprocess (remove background)	85.1%
Face detection	91%
Eyes segmentation	95%
Nose segmentation	96%
Mouth segmentation	97%
Logistic regression	100%

- This is a variation of Amdahl's Law. Overall performance can only be improved based on how much that part contributes.

Overall Thoughts

- Good course, got an intuition for the various algorithms. Practically I would never implement these myself in Octave, need to get familiar with a toolkit/library (TensorFlow, BigML.com, etc.).
- Much of machine learning seems to be applied stats.
- Wow. Feels good to have finished the last assignment. Makes me want to always have a course going in the background!



Summary: Main topics

Supervised Learning $(x^{(i)}, y^{(i)})$
- Linear regression, logistic regression, neural networks, SVMs

Unsupervised Learning $x^{(i)}$
- K-means, PCA, Anomaly detection

Special applications/special topics
- Recommender systems, large scale machine learning.

Advice on building a machine learning system
- Bias/variance, regularization; deciding what to work on next: evaluation of learning algorithms, learning curves, error analysis, ceiling analysis.