Machine Learning Notes

An Qi Zhang - With credits to the course

July 23, 2015

Note for everything: ***Vectorize AS MUCH AS POSSIBLE****, it will take advantage of the parallel computing abilities of Matlab/Octave/Distributed R/Whatever parallel computing language being used*

Also, general workflow: train with the training set, optimize on the cross-validation set, and test against the test set

1 Linear Regression

- in linear regression we try to fit an equation to a set of points
- we'll get the computer the pick good coefficients for a function to closely resemble the data points we have
- we'll call those coefficients θ
- to get good $\theta's$, we'll use the equation:
- $\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^(i)$
- where:
- θ_j is a coefficient of the total j coefficients for our j "features" or "variables" in our function
- α is the "learning rate" (we'll see in gradient descent that it controls the size of the "step" we take to a optimal point of the function)
- $h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$ which is our "hypothesis" function, x's being the variables/features
- Remember, when updating the θ values, update them simultaneously, that is, at the same time. In Matlab, the easiest way is to multiply them as matrices and/or vectors (this is also known as vectorizing code, and Matlab will try to do the calculation in parallel)

- Now, to check if our $\theta's$ are giving us a good function, we check the "cost" or how accurate our equation is so far
- and by checking this cost with every iteration of updating θ, we are performing *Gradient Descent*. Gradient Descent is similar to newton's method of approximation, with respect to the idea of taking the derivative at a point to get an idea of which direction to go to get closer to the global optima (in other machine learning techniques there will be local optima the equation can get stuck on)
- note: to clarify we use the negative of the derivative, as further shown in this image:

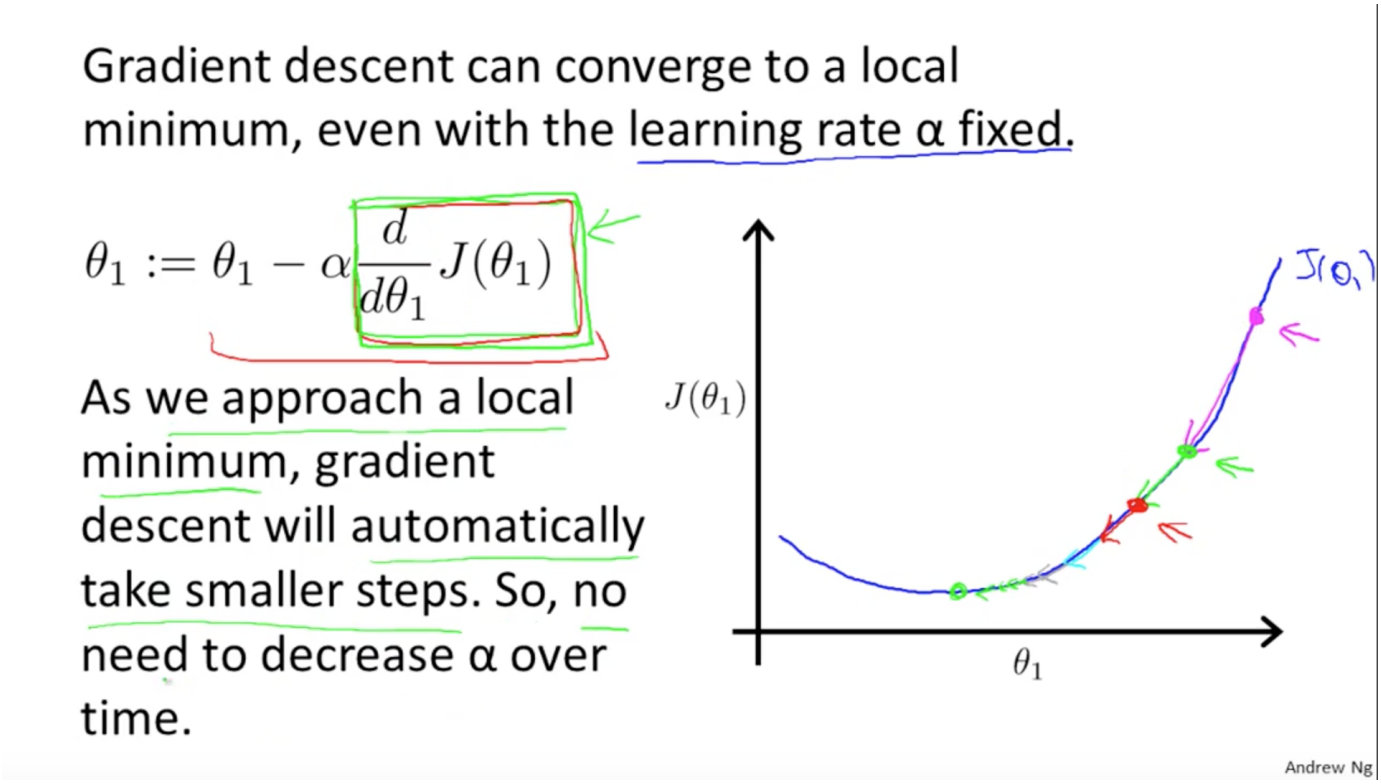

Figure 1: By taking the negative of the derivative, we know which way to go to move towards the global minima

- (note: thinking about it further, gradient descent really does seem like newton's method. I'll have to fact check that later.)
- To elaborate on the alpha, the alpha can be thought of as the "step size" of gradient desc.

- the equation for this cost function is:
- $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$
-
- note: to "normalize" or "keep the 'features' in the same range so that gradient descent will flow more naturally" subtract the mean value, and divide by the standard deviation
- also, if you normalize the features, when you test your resulting function, normalize the input first. Because you normalized the test data, the function is set to work for normalized values. (and use the same mean and standard deviation)

2 Gradient Descent (something like Newton's Method)

- basically partial derivative of Cost Function $J(\theta)$ with respect to each θ
- accidentally already explained this in the previous section.
- But to continue, we can also do multivariate linear regression, where we just have more theta's and more variables (nothing new)

3 Using Matrix Normalization to Find the Optmial Theta Parameters

- include equation $(X^T * X)^{-1} * X^T * y$
- what to do when $X^T * X$ is non invertible?
- in Matlab, use pinv
- remove redundant features (for example, there are two parameters for the same measurement, but in feet and meters)
- there could be too many features

4 Logtistic Regression or "Classification"

- used for classifying things (like spam/not spam)
- $h_\theta(x) = g(\theta^T * x)$
- where sigmoid/logistic function $g(z) = \frac{1}{1+e^{-z}}$
- $h_\theta(x) = \frac{1}{1+e^{\theta^T * x}}$

5 Overfitting (the function tries too hard to fit the training data and does not make a general equation)

- Reduce number of features
  - Manually select useful features
  - Model section algorithm (to pick features automatically)
  - Hard to say if features are all needed or not
- Regularization
  - Keep features, reduce magnitude of θ_j, perhaps keep to -0.03 range to 0.03
  - Works well when many features exist
  - Small values for theta parameters gives "simpler" hypothesis, less prone to overfitting
  - choose a good λ
  - Regularized linear regression: $J(\theta) = \frac{1}{2m}[\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})^2+\lambda \sum_{j=1}^{n} \theta_j^2]$
  - Can use algorithm to automatically chose λ as well
  - Regularized Gradient Descent: (Don't regularize θ_0)
    Repeat {
    $\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$
    $\theta_j := \theta_j - \alpha[\frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j]$
    }
  - the second theta eqn can be simplified to $\theta_j := \theta_j(1-\alpha\frac{\lambda}{m})-\alpha\frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})x_j^{(i)}$
  - α is usually small, there are many training examples (large m), so that usually results in a small value
  - Note, a large λ can result in underfitting, and a small one results in overfitting

6 Normal Equation after regularized

- deriving cost function wrt theta and setting it = 0 results in normal equation

- after regularizing this results in a similar equation, but with adding lambda times a identity matrix with the first element being set to 0 in the inverting part of the equation (this identity matrix being (n+1) by (n+1))
- if this lambda is greater than 0 then the matrix to be inverted will never be "singular" or uninvertable
- there is an advanced optimization part in the second assignment pdf. it uses fminunc (f min unconstrained). read more there. (remember to use @costFunction for using your costFunction)

7 Non-Linear Classification & Neural Networks

- when you have lots of features, and you need to have them become higher order, there might bee too many items
- so logistic and linear regression doesn't quite work as quickly as it would on a smaller feature problem

- We can use a Neuron Model: Logistic unit
- We have "dendrites" (the inputs) the cell body (the function) and the output (our hypothesis or the axon)
- still have x's and parameters. Note that the prof sometimes adds x0 or a0 as a bias unit
- a Neural network can be represented by "layers", like input layers, fed into a "hidden" (neurons) (anything not input or output layer) layer, fed into an output layer (our hypothesis)
- the dimensions of this $\theta^{(j)}$ are $s_{j+1}x(s_j + 1)$, meaning the size/number of units in the next layer by the number of units in the current layer plus 1 (ex. 2 inputs, 4 neurons, so 4 x 3 theta parameter)
- for our course at least, the super scripts will mean which layer the item came from.
- subscript will be which item in the layer
- neural networks are like logistic regression, except, instead of using x inputs, it uses it's own learned inputs. the inputs (x's) are fed into it's a's, which are it's own inputs, which are then fed into a logistic regression layer.
- we still use J of theta and the derivative wrt theta

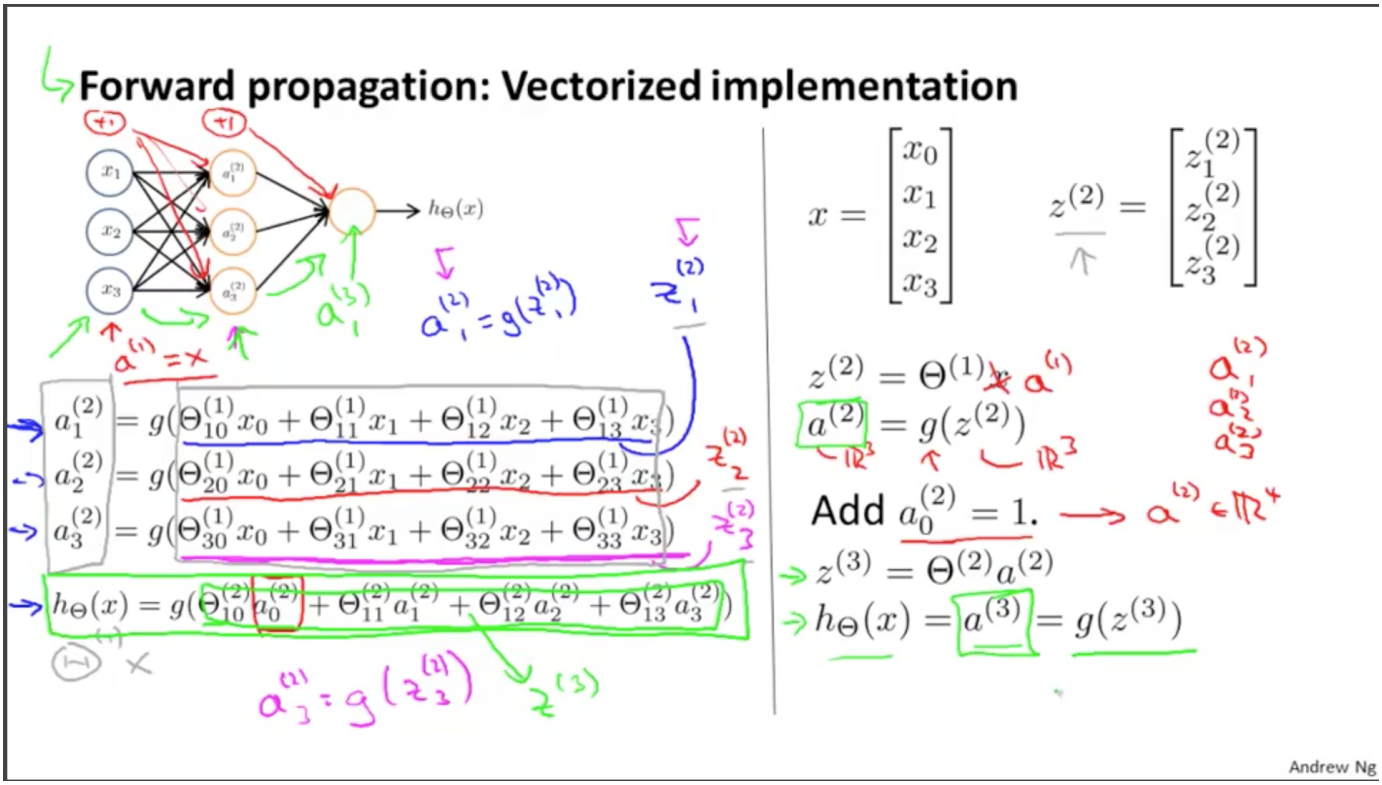

Figure 2: A general neural network. Vectorization of the code makes it faster, especially using a gpu to handle the computations in parallel

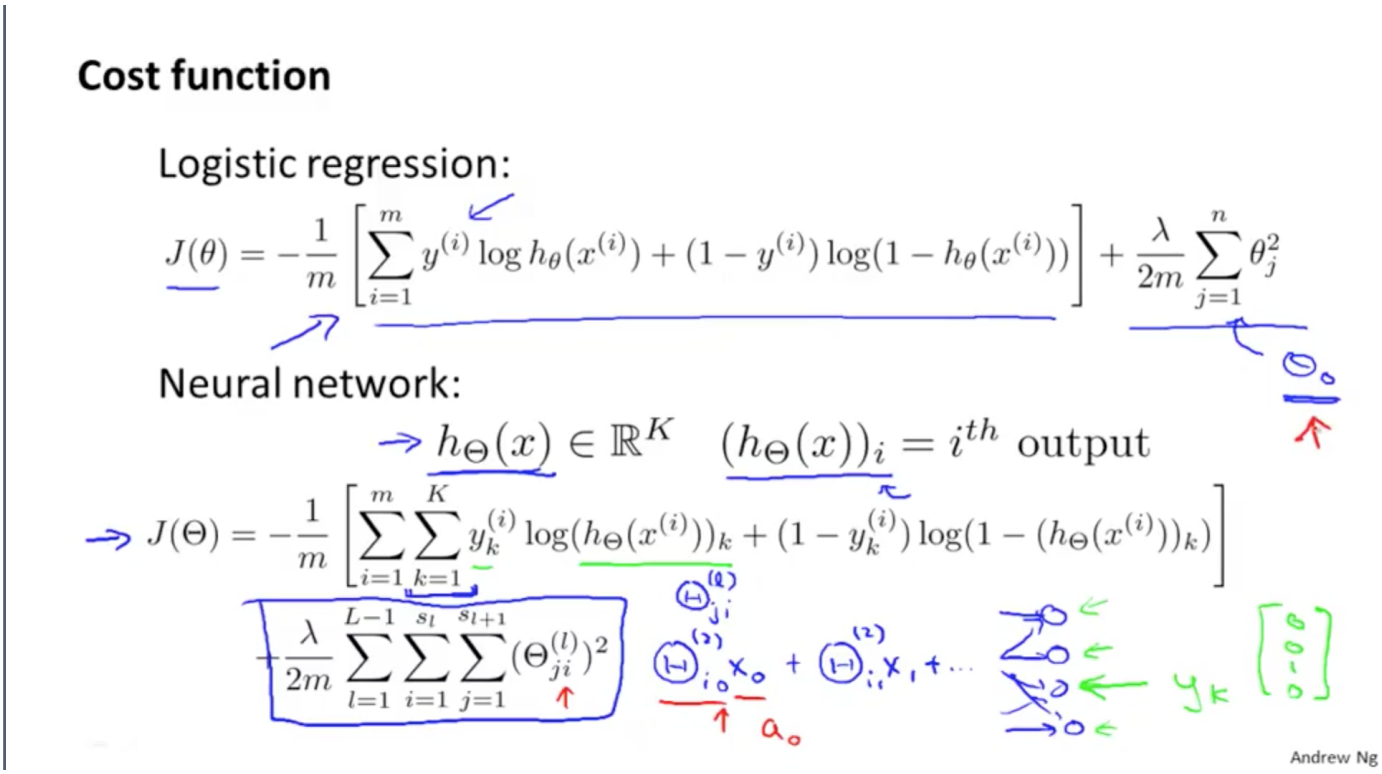

Figure 3: The cost function for a neural net.

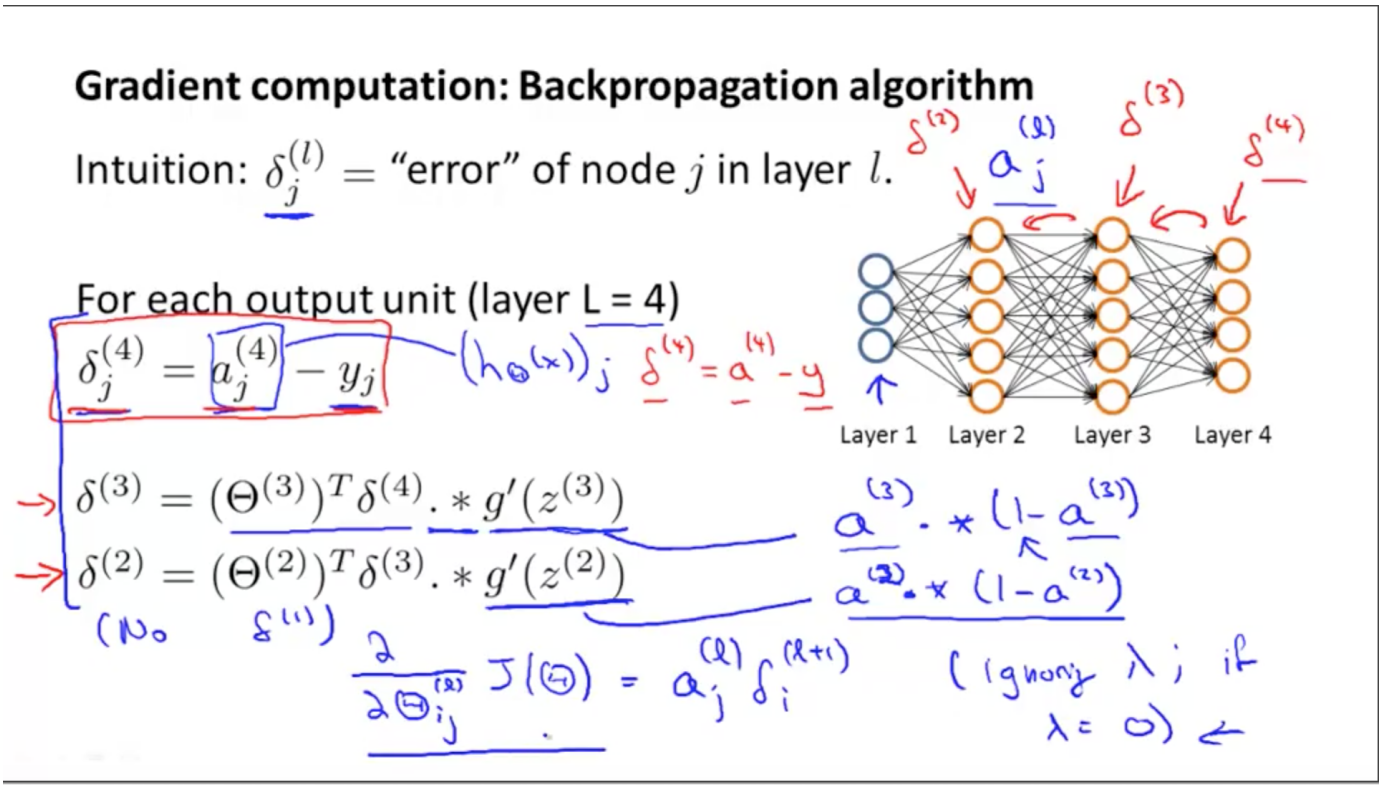

Figure 4: There is a error term, shows how far off we are. Reminds me of calculus class, with the error term of a Taylor/Mclaurien Polynomial

- We use forward propagation to get our $z^{(2)} = \theta^{(1)}a^{(1)}$ and $a^{(2)} = g(z^{(2)})$ (and eventually our hypothesis)
- We use backwards propagation to get our δ(kinda like in calculus with the error term in a Taylor Polynomial)
- We can unroll our theta's into one large theta vector, and also do the same for D (delta).
- In Matlab/Octave, we can use the reshape() function to get back our vectors

7.1 Numerical Gradient Checking - make sure there aren't those subtle errors in your program!

- Basically a little bit of that ϵ/δ or the general calculus derivative with the f(x+h) - f(x) / h, but we also subtract h from the second term. (so the one I knew was one-sided, but this one is two-sided and more accurate)
- Only use Gradient Checking for testing to make sure there isn't an error in the program code
- When actually implementing, use backprop (backpropagation) for learning

7.2 Choosing the Initial Value of θ

- While just using a vector full of zeros works for the previous algorithms, it doesn't work for Neural Nets.
- If they're zero, then at the layer after the input layer, then $a_1^{(2} = a_2^{(2)}$, the terms will both be the same due to this theta of zeros, the delta (δ) values will be the same as well and the θ will be the same (same meaning theta1 and theta2 equal to eachother).
- So, we have *Random Initialization*! We need to pick random values for theta, as well as *breaking symmetry.*
- To break symmetry we init each theta to a random value in $[-\epsilon, \epsilon]$. We make a rand(x,y) random matrix of size x by y, multiply by 2 times $init_\epsilon$, and then subtract this $init_\epsilon$, and now we have one of our thetas.

7.3 Training a Neural Network

- Pick a network architecture (a pattern between neurons. like 3 inputs, 5 neurons in the second layer, and 4 output hypotheses)
- # of input units: Dimension of features $x^{(i)}$
- # of output units: Number of classes (could be 1-10 like the numbers from MNIST)

- Reasonable # default: Just use 1 hidden layer. Or, if > 1 hidden layer, use same # of hidden units in every layer (the more the better)
- # of hidden units usually should be comparable to # of input units or features, maybe even 2x, 3x ... 7x the # of input units

1. Randomly init weights (parameters/Θs)
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 derivaties $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
   (a) could use for loop and do fp and bp at each layer, but you can vectorize it (they say it's advanced, but let's see about that)
   (b) in the for loop, also get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, ..., L$ (l being the current layer that we iterate through with the for loop and L being the last layer number)
   (c) something like $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
   (d) after the for loop, compute the partial derivative terms of the cost function J
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using back-propagation vs. using numerical estimate (the ϵ thing) of $J(\Theta)$. Then disable gradient checking code
6. Use grad descent or an advanced optimization method (with fminunc) with backpropagation to minimize J
7. This $J(\Theta)$ is non-convex (not a bowl) and can get stuck on local minima (unlikely, but could happen).

8 Debugging a Learning Algorithm

- What should you do if your trained hypothesis makes errors in it's predictions?
- > Get more training examples
- > Try a smaller set of features (prevent overfitting)
- > Try more/better features
- > Add polynomial features (Try visualizing first, and try to guess which polynomial would look good)
- > increase or decrease λ
- Try running a Machine Learning Diagnostic

- It takes time to make and understand, but can be very informative as to what to improve

- Try Visualizing the hypothesis
- (better used if we have multiple feature training sets)
- But we can still Evaluate the Hypothesis by splitting a training set into parts
- We can assign 70% of the training set as the training set, and 30% as the test set

- Training/Testing Procedure for Linear Regression
- > Learn θ from training data (minimize J, for that 70% of the training set)
- > Compute test set error J for our test set 30%

- Training/Testing Procedure for logistic Regression
- Learn θ from training set
- Compute test set error J for logistic regression
- Check for Misclassification error (0 or 1)
- err($h_\theta^{(x)}, y$)= 1 if it's greater than 0.5 and y = 0 then there's an error , 0 otherwise
- basically check if there's an error
- comes out to be test error = 1/m(test) * sum(err($h_\theta(x_{test}^{(i)})$)) from i=1 to m(test)

- Model Selection
- Evaluating your hypothesis
- set 60% to training set, 20 to cross validation set, 20 to test set
- Now we can define error J for training, cost validation, and test error with the same J eqn we've used
- Now test different order hypotheses on the cross validation set, and pick the best/lowest J eqn from the different order hypotheses

- Choosing regularization parameter λ
- try different values of lambda, could try doubling lambdas, then check the J value for the best J
- Note: getting more training data is helpful when there's high variance or Jcv is » than Jtrain
- Note: high bias: hypothesis is too simple (underfitting) hypothesis, training data not fit well (J for training and test both high)
- add more features, training,
- Note: high variance: hypothesis is too complex, overfitting hypothesis,
- less features, adding data won't help
- Remember, a high lambda will make the thetas worth less, kinda like removing those features (can underfit and cause high bias, or fix high variance), and a low lambda will make the thetas worth more, like emphasizing them (can make over fit and cause high variance, or fix high bias)
- note: if there's high bias, only adding training examples might not help, since it could be that lambda is too large so our thetas are being penalized and shrunk to approximately zero
- note: if there's hight variance, adding more examples can help since the model is overfitting the training data. Adding more training data will increase the complexity of the training set which makes it harder for the hypothesis to force some kind of weird function that will fit the training examples
- note: to get this visualization, we can graph the error J for test and training sets Vs. the m training set size.

9 Prioritizing What to Work on - Less Math, Mostly Plain Old Logic

9.1 Example: Building a spam classifier

- We could take a look at the different words used, like buy, deal, discount, etc.
- From this, we could have a vector of 1's and 0's to show which words exist in this email.

- For this example, we could pick 100 words
- In practice we could check for the most frequently occurring n (10k to 50k) words
- Now, collecting lots of data might help, or not.
- We can make sophisticated features based on things like the email's routing (address, header, etc.)
- Have sophisticated features to maybe count "discount" and "discounts" as the same word
- Have algorithm to detect misspelled words, like w4tches, m0rtgage, etc.

9.2 Error Analysis

- Start with a simple algorithm that can be implemented quickly and tested on the cross-validation data (quick and dirty, could be a little bit longer that "quick")
- with this, plot learning curves to decide if you need more data, features are likely to help by checking for high bias or variance, AND AVOID THE ROOT OF ALL EVIL THAT IS PREMATURE OPTIMIZATION (Ahem, sorry, got a little carried away there)
- this evidence is more useful than going with a gut feeling and falling prey to premature optimization
- Check training examples in cross validation set that your algorithm makes errors on, check for any trends
- Ex. the spam classifier, manually check the types of the emails, features that would have helped to have
- could categorize emails into specific categories like stealing passwords, pharma, fakes, etc.
- could check # of occurrences of deliberate misspellings, unusual email routing, unusual punctuation, and focus making features on those specific metrics
- Remember, a quick and dirty implementation is good for checking what might be good leads to follow

9.3 The importance of numerical evalutation

- Ex. the spam: should discount/discounts/discounted/discounting be treated as the same word?
- Use "stemming" software?
- but it could make mistakes like universe/university
- Best way? Use it and observe the results for signs of success/failure (higher or lower error percent?)
- Another issue: lowercase vs uppercase (Mom/mom), again, implement then check error
- Rule of thumb: test your idea first, then check the error for improvements
- and remember, test this on the cross validation set, not the test set, you don't want to train the algorithm to be good at the test set!

9.4 Error Metrics for Skewed Classes

- Cancer classification example
- lets say you have only 1% error on the test set
- BUT only 0.5% have cancer. we have skewed classes since we have a small group of positive data in out data set
- Say we made a change and now our error improved by 0.3%, did we do something useful?
- hard to say with this skewed data set. we might have just made our algorithm predict zero more often
- we should check the precision of our algorithm.
- Precision = $\frac{True+ve}{\#predictedpostitive} = \frac{True+ve}{True+ve+False+ve}$
- Now Recall, # of patients with cancer Vs. our predicted amount
- Recall = $\frac{True+ve}{\#actualpostitive} = \frac{True+ve}{True+ve+False-ve}$
- Note: Accuracy is just true +ve plus true -ve divided by total examples

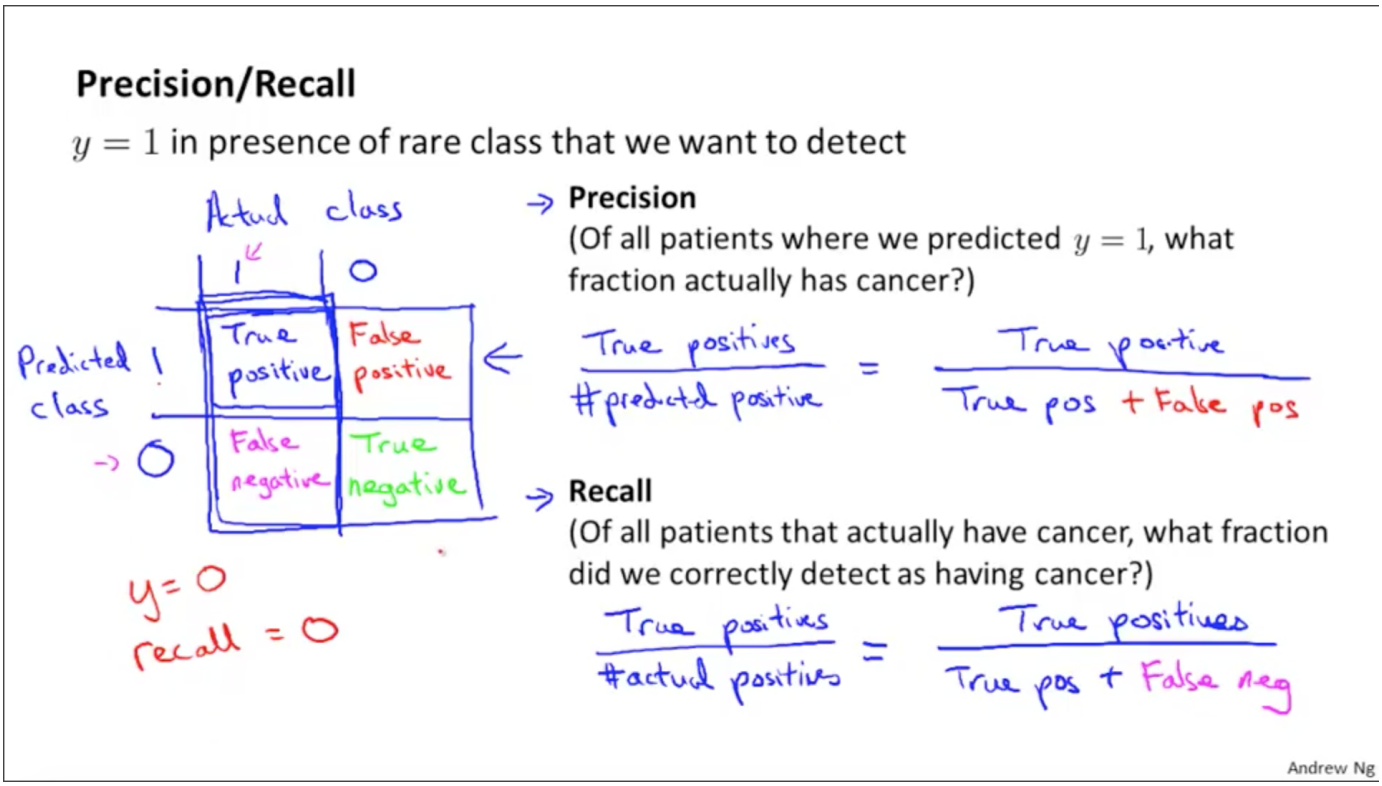

Figure 5: Precision and Recall.

9.5 Trading off precision and recall

- continuing on the cancer example
- how about changing our logistic regression to output 1 if our hypothesis is greater than 0.7 to try to ensure we have a high confidence in our prediction
- This gives higher precision, but lower recall
- But then true positive cases might slip by
- If we want to avoid missing these cases we could change the threshold again! maybe to something like 0.3
- This gives lower precision, and higher recall
- We can make a graph with different thresholds and make a decision with it!

- let's check the (P)recision and (R)ecall for different thresholds, and take the F score of them, and pick the best settings based on it!
- $F_1 Score = 2\frac{PR}{P+R}$
- Note: bad F score -> P and R both = 0, so F = 0, where as a good one will have them all equal 1

9.6 Data for Machine Learning

- Designing a high accuracy learning system
- Ex.
- Let's say we have enough data for feature X for a good prediction
- Counter Ex. Predict housing prices with just X (Not a good thing to try)
- Thus, try a little useful test, where a human expert predicts y using X. If they can do it, then a computer should be able to do it too.
- Rationale Behind Large Data
- Ex use learning algorithm with many params, (regression with many features or neural networks with many hidden units)
- With a large training set, we are unlikely to have overfitting bias, so we can have low variance.
- Lesson: Have good features (one a human could use as well) and a decently sized training set.

9.7 Support Vector Machines

- Let's start by looking at logistic regression
- In it's cost function, when y = 1, the (1-y) term disappears. And when the z is greater than zero the value gets smaller
- To build a SVM from this, we just have an approximation of the cost function and break it into two pieces
- In other words, we can replace the -log h terms with cost1 and cost0
- we will also remove 1/m from the cost J and regularization parameter lambda since it's just a constant
- we can also scale them by setting a constant C = 1/m, and also C = 1/lambda
- Note: SVM doesn't predict probability, it predicts 0 or 1, a true or false

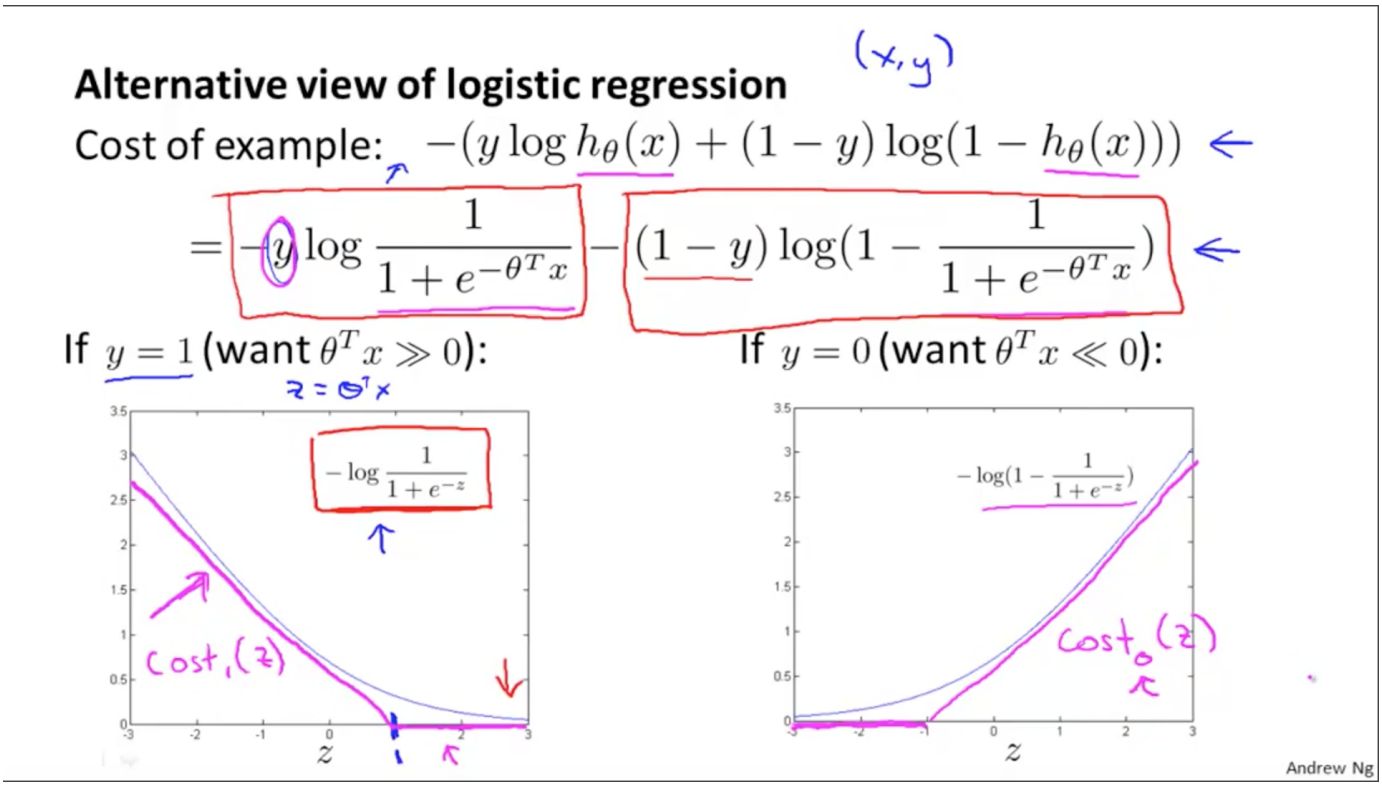

Figure 6: Get SVM from Logistic Regression.

- h = 1 if $\theta^T x >= 0$ and h = 0 otherwise
- making constant C a light number, the summed part goes to zero
- Note: when calculating the cost, for y = 1, we want to try to get theta times x to be >= 1 since this would get a 0 from the cost function of our SVM, and vice versa for y = 0
- note: to make this SVM decision boundary, the intuition to follow is that it tires to maximize the vector projection projected by each testing data point, and create the largest margin

9.8 Kernels

- With non-linear decision boundaries, we usually have had polynomial features
- But is there a better feature set to use?
- We can use x to compute new features depending on proximity landmarks

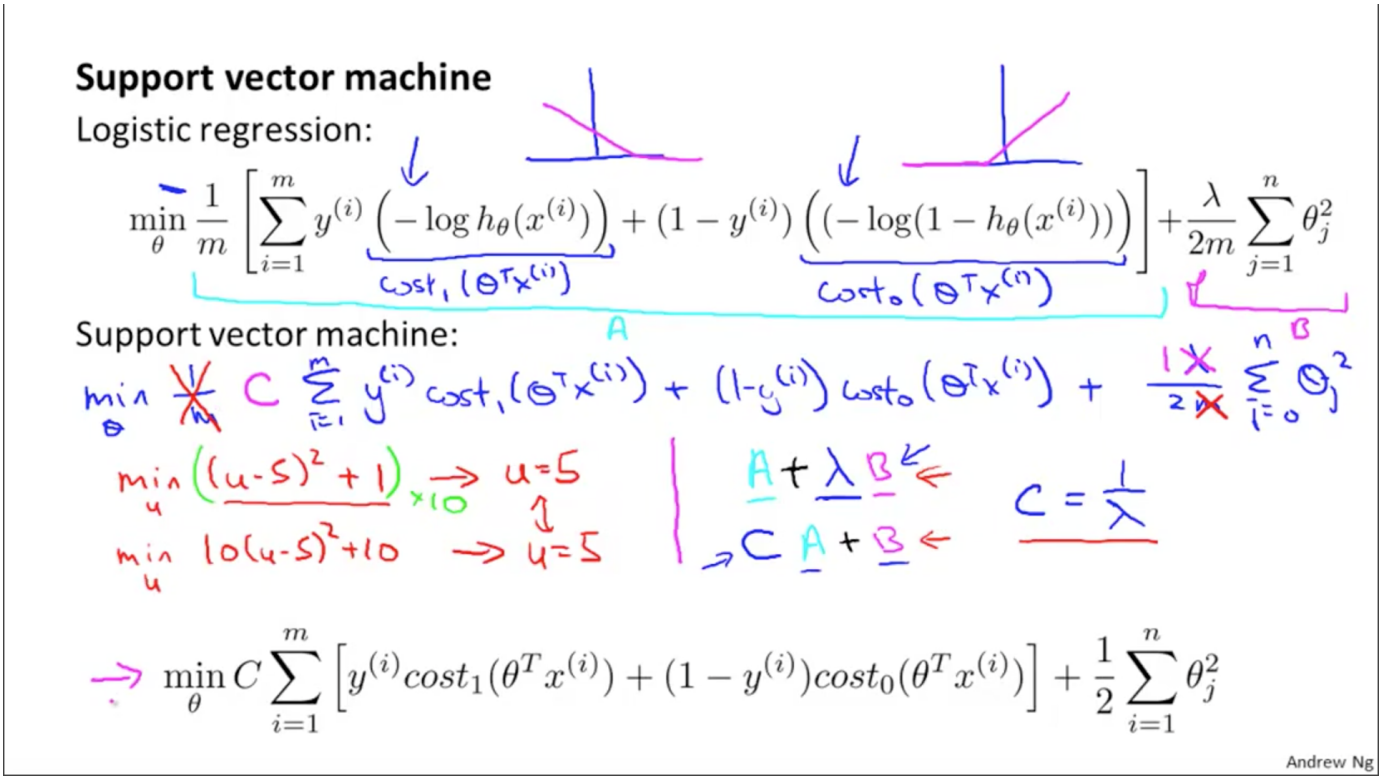

Figure 7: C is 1/m and 1/lambda.

- We calculate kernels of similarity of x and landmarks l, with the eqn $e^{-\frac{\|x-\ell\|^2}{2\sigma^2}}$ (this eqn is a Gaussian kernel)
- Cute trick with calculating the regularization term with theta
- Graphic on effects of SVM parameters

9.9 Using a SVM

- Will need to choose a value for C
- choose a kernel (similarity function) (Can even choose no kernel/linear kernel, with $\theta^T x >= 0$
- choose a kernel (could be gaussian kernel)
- remember to do feature scaling before using the gaussian kernel!
- choose σ^2

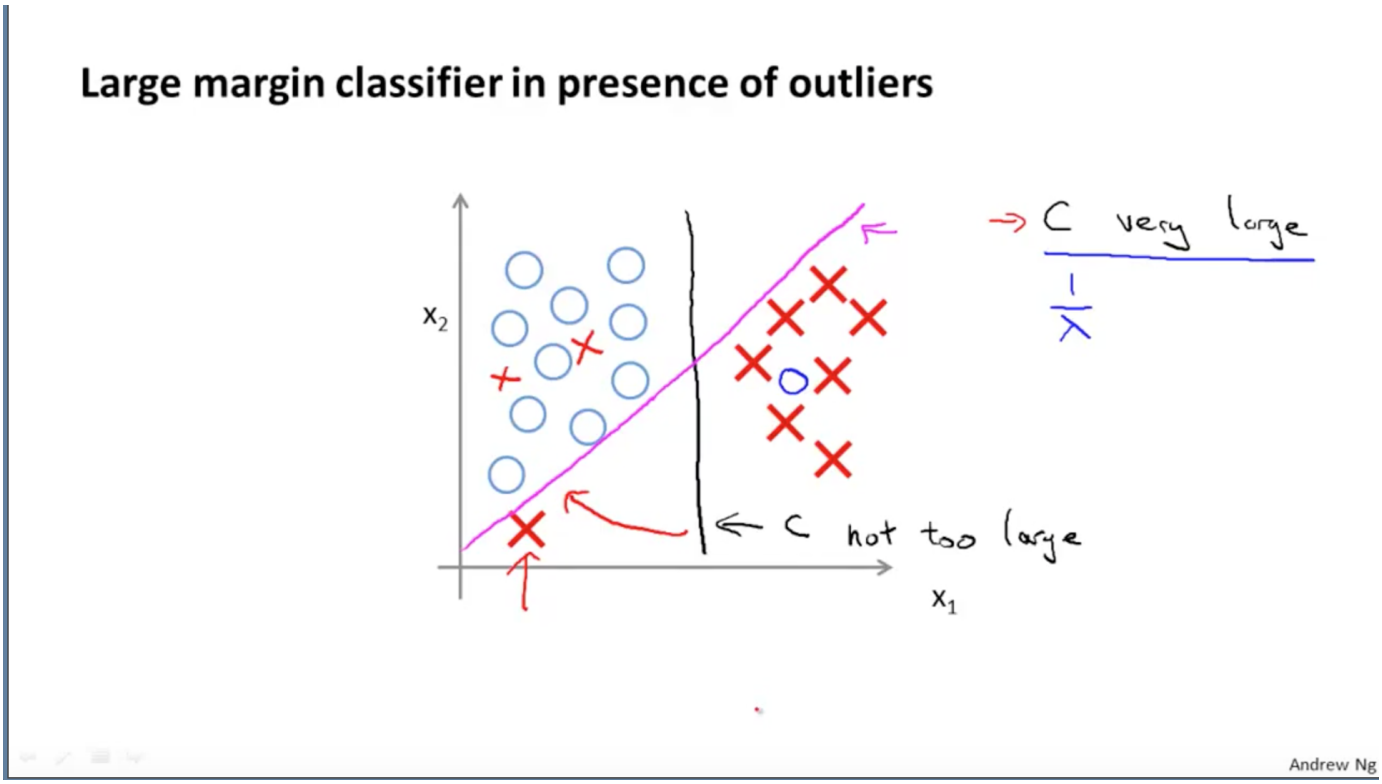

Figure 8: Some explanation on how it works.

- *Multi-class classification*
- SVM packages generally come with multi-class classification functionality, if not, use one vs. all
- if # of features is large, try logistic regression, or SVM without a kernel
- note: SVM - no local minima it could get stuck on compared to neural networks

10 Unsupervised Learning

Unsupervised learning is where we have our data set X, but we don't have a y. That is, we have an algorithm find the relation in the data for us.

10.1 Clustering: K-means Algorithm

- This is the most popular clustering algorithm!

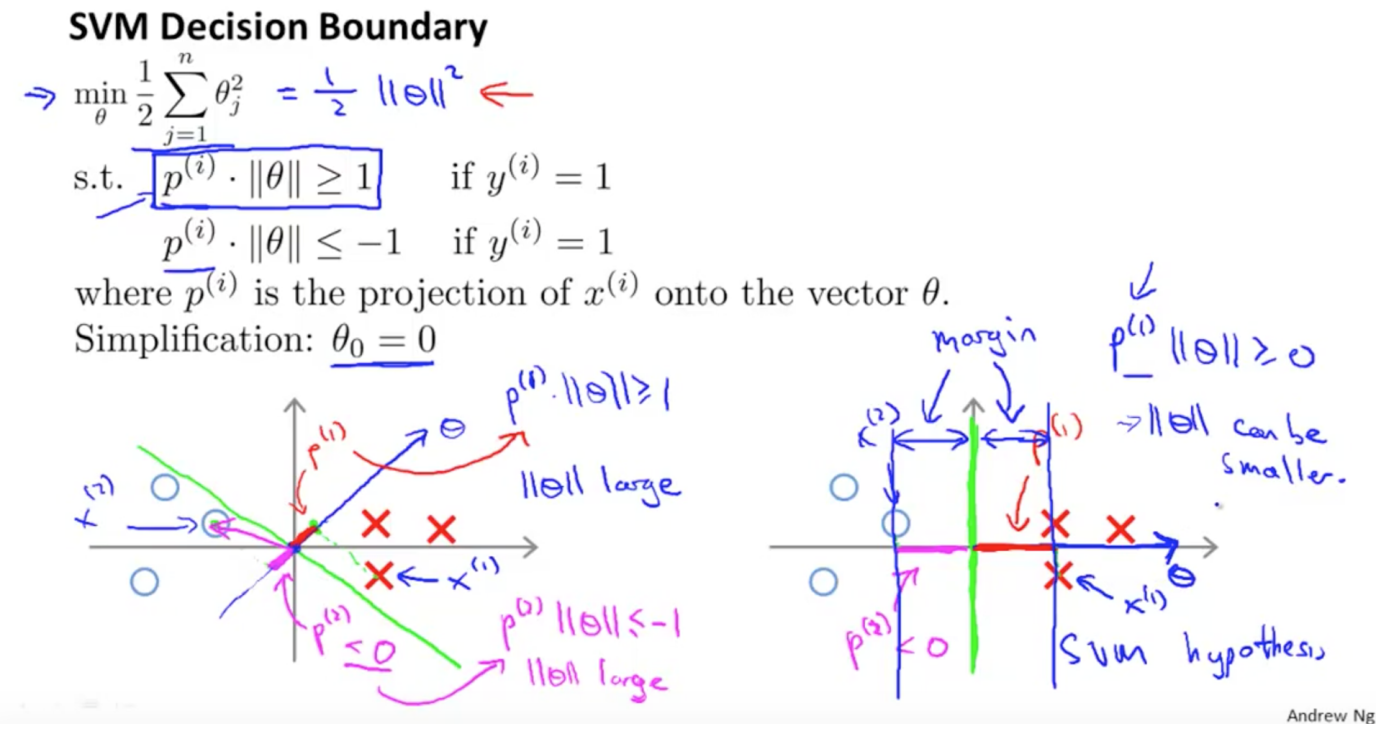

Figure 9: Intuition on SVM.

- (Rough idea of how it works)
- We initialize 2 (some number) cluster centroids randomly
- We go through each point and try to assign each oint to the 2 cluster centroids
- We move the centroids closer to the sets' means.
- We repeat the last 2 steps again until we have them all bundled up into groups

- (More accurately, but still in general)
- We take input K (number of clusters)
- and a training set X
- This time, we don't use $x_0 = 1$

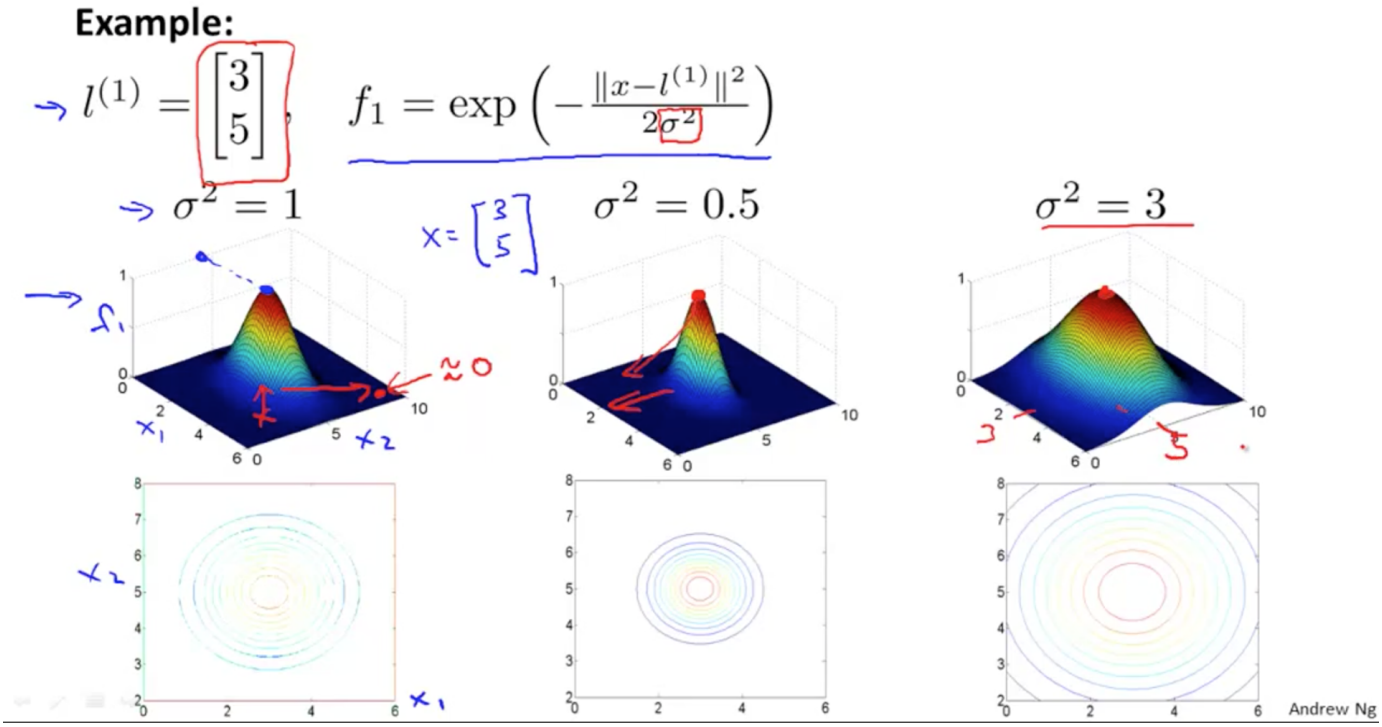

Figure 10: Kernel feature.

- Now randomly init(ialize) the K centroids
- Repeat {
- for $i = 1$ to m
- $c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$
- for $k = 1$ to K
- μ_k := average (mean) of points assigned to cluster k
- }
- The first loop assigns x's to clusters they're closest to (and minimizes J wrt c's, and holds μ's)
- The second gets the mean and moves the centroids to the means (chooses μ's that minimize J)
- If a cluster has no points, we usually remove it. Or, some times we can re-init it.

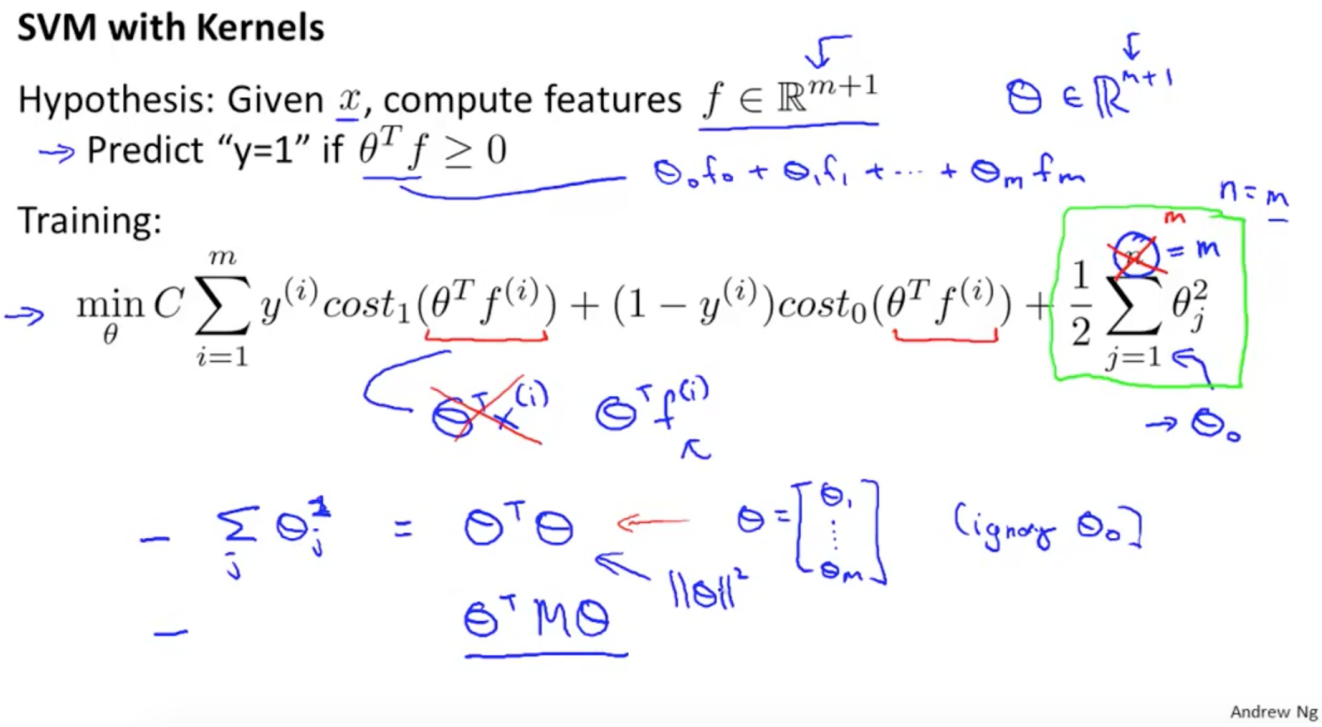

Figure 11: Kernel feature.

- K-means for non-separated clusters
- It will try to make different segments, even if it just looks like a big group a points on graph.

10.2 Optimization Objective

- K-means optimization objective
- Notation We Will Use:
  - $c^{(i)}$ = index of cluster (1 to K) to which $x^{(i)}$ is currently assigned
  - μ_k = cluster centroid number k
  - $\mu_{c^{(i)}}$ = cluster centroid of cluster to which $x^{(i)}$ has been assigned

10.3 Random Initialization

- There's actually a good (well, better i should say) way to randomly pick K centroids.

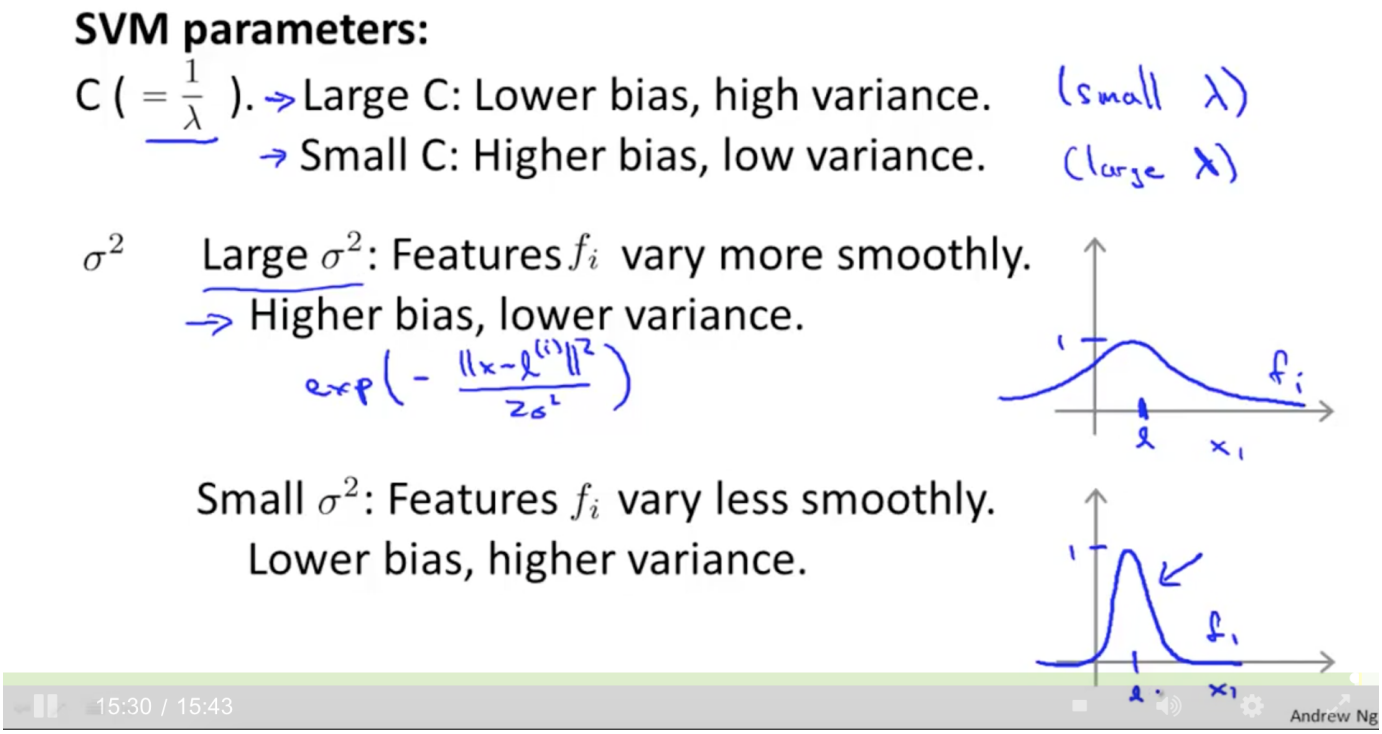

Figure 12: Kernel feature.

- Depending on the init, K-means can end up at different local optima, and not your desired global optima
- Solution? We init K-means many times to try to ensure that we reach the global optima
- After that, we just pick the one with the lowest J!
- We can just use a for loop to do this 50-1000 times (try to see if there's a way to vectorize this code).
- But when you have 100's of clusters, this is unlikely to be of much help. The first try might as well be as good as it gets.

10.4 Choosing the Number of Clusters

- It might be hard to try to figure out. (could be ambiguous)
- Could try the *Elbow Method*

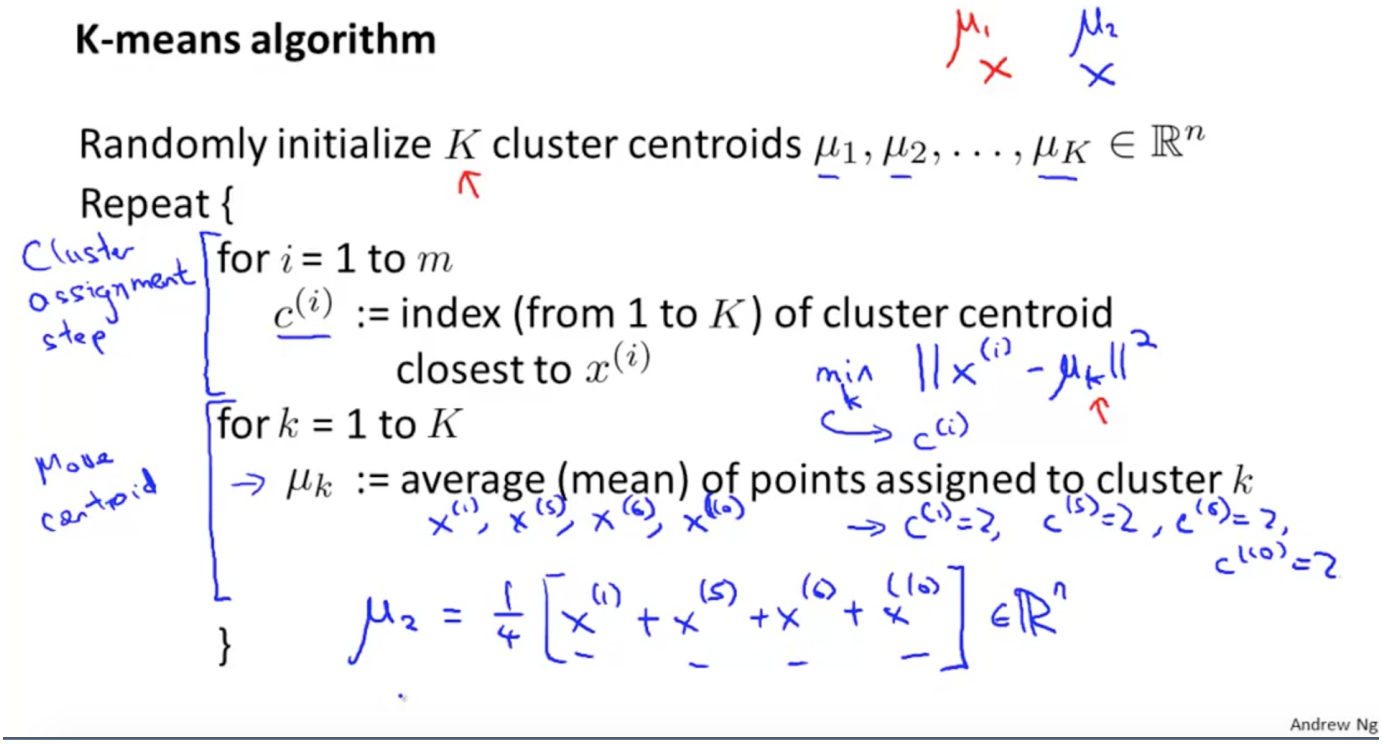

Figure 13: This can also be explained mathematically, but it's out of scope of the course. Maybe I'll add it if I get the chance

- So try and graph the cost Vs. different K's (num of clusters)
- Called elbow method since it looks something like a elbow usually
- But it's usually not helpful since it usually is a smoother curve and we can't easily find this elbow point that acts as the optimum
- So, we could try *Choosing the value of K*
- For example, if we have T-shirt sizes based on weight and height, we can try 3 clusters, S, M, L or 5 clusters, XS, S, M, L, XL

10.5 Dimensionality Reduction

- *Motivation I: Data Compression*
- Ex. 2d to 1d, all the points on the line can be used to represent the 2d data it came from

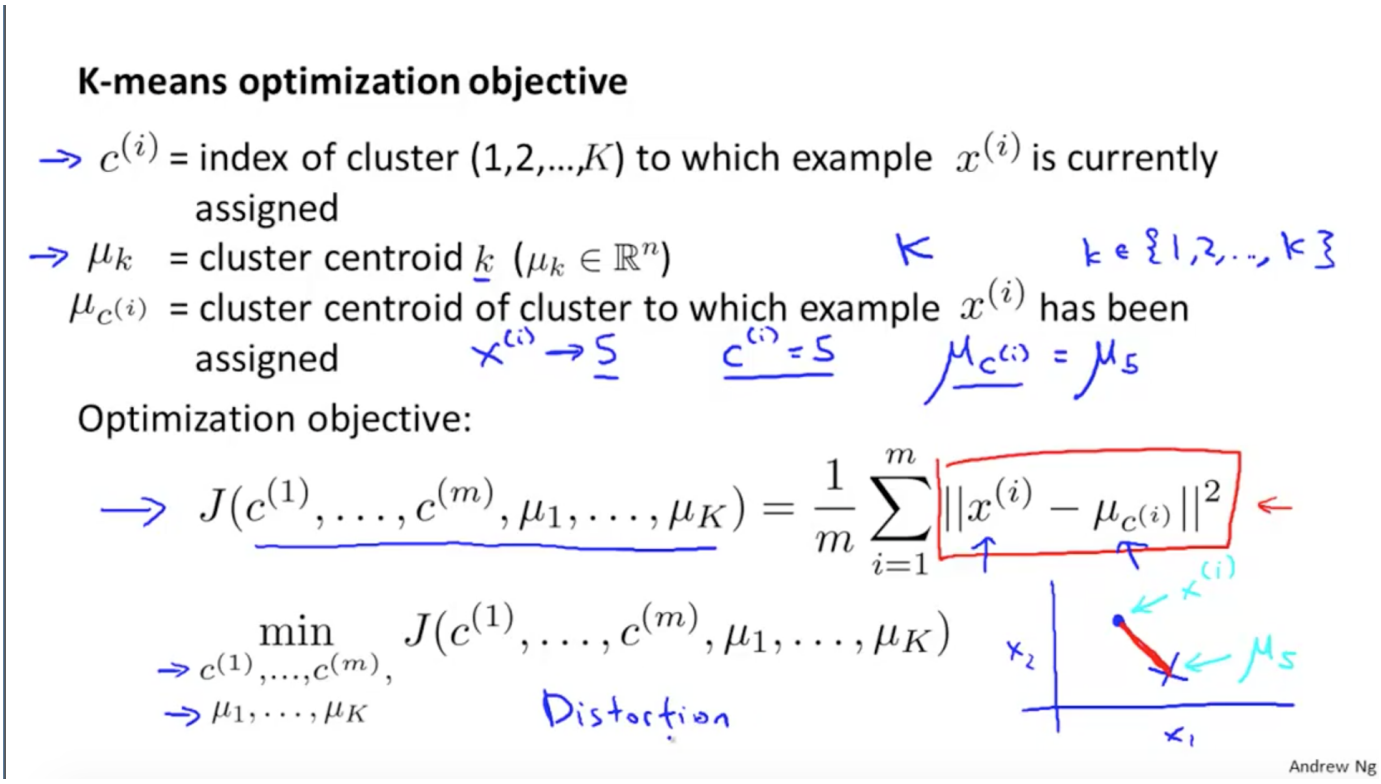

Figure 14: J, the optimization or "distortion" for this clustering algorithm

- Ex. 3d to 2d, all the points can be projected to some 2d plane
- *Motivation II: Data Visualization*
- Hard to visualize many features at once
- We can reduce this many dimensional thing into a 2d problem to visualize

10.6 Principal Component Analysis Problem Formulation

- In general, reduce n dimensional data to k dimensional data. We find k vectors, (u1, u2... uk) and project the data onto them while also minimizing the projection error
- PCA is NOT linear regression
- *Data preprocessing*
- good to do feature scaling/mean normalization

- just like before, if features are on different scales, subtract the mean, and divide by the standard deviation
- Now for the algorithm
- We compute the "covariance matrix"
- $\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T$ (here, the left sigma is a matrix, and x is nx1, so the result is a n x n)
- Compute "eigenvectors" of matrix Σ:
- the Octave command is [U,S,V] = svd(Sigma); (could use the eig() func. but svd is more stable)
- the columns of U go up to n
- we can take the first k columns of the U matrix (n x k now)
- now transpose it (k x n) and multiply by x (n x 1) so now we have vector z
- in code, something like: Ureduced = U(:,1:k); and z = Ureduced'*x;

- *Reconstruction from Compressed Representation*
- To get x back, we can multiply this Ureduced matrix (n x k) by the z vector (k x 1), so we get a n x 1 vector back
- Of course this x will be approximately close to the original

10.7 Choosing the Number of Principal Components

- get average square projection error divided by the total variation less than 0.01
- so that "99% of variance is retained"
- We can also use 95-90% to get more compression
- Now to choose k try PCA with k = 1 .... some number
- compute U, z etc.
- now check the variance
- As you can imagine, this is horribly inefficient
- So the other way is, looking at octave's [U,S,V] = svd(Sigma), to sum k of S's diagonal elements, divide by the sum of n (all of S's) elements, and subtract this from 1, and see if this is greater or equal to 0.99
- After this, we can also check the variance retained by running the calculations with k incase anyone asks what it was (to check)

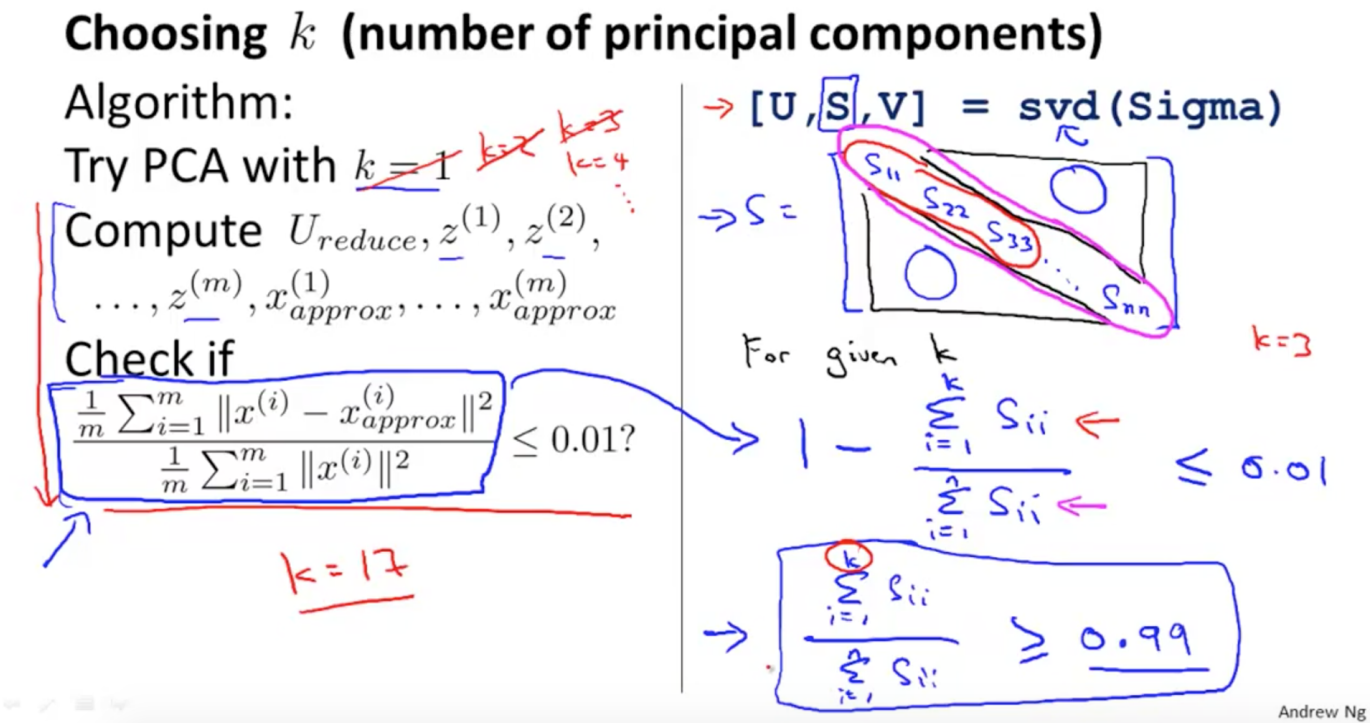

Figure 15: To the right, is a good way to check what's a good k without too much work (If you work iwth matlab/octave)

10.8 Advice for Applying PCA

- *supervised learning speedup*
- We can reduce the dimensions of the data with PCA (let's ignore the y's first)
- let's say we took 10,000 feature training examples and made them into 1000 feature examples
- now let's match the y's back to these compressed z examples
- Note: this mapping of x->z should only be defined with PCA on the training set
- now use this mapping for the cross validation set and test set

- Recap: Compression

- reduce memory/disk needed to store data
- speeds up learning algorithm
- OR for visualization (where k = 2 or 3)

- A *BAD USE OF PCA*: to prevent overfitting
- This won't really work (well... it might in some odd cases) but it won't in general
- Just use regularization

- Note: before just throwing in PCA for ML, try your implementation first without PCA, and then see if you need PCA or not.
- Recap: compress data, visualize, and make training data smaller

11 Anomaly Detection

11.1 Problem Motivation

- Density Estimation is one technique
- An application is Fraud Detection
- Quick review of Gaussian Distribution
- *Parameter Estimation*
- Try to guess where the parameters of μ and σ^2 (and tell if it's far away from the mean on the gaussian distribution)
- So remember Data Management in high school, calculate mu and sigma
- $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
- $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)^2$
- Remember that in stats, that sometimes m-1 is used (I already forgot why (normalization i think)) but when m is very large, it doesn't matter if you subtract 1

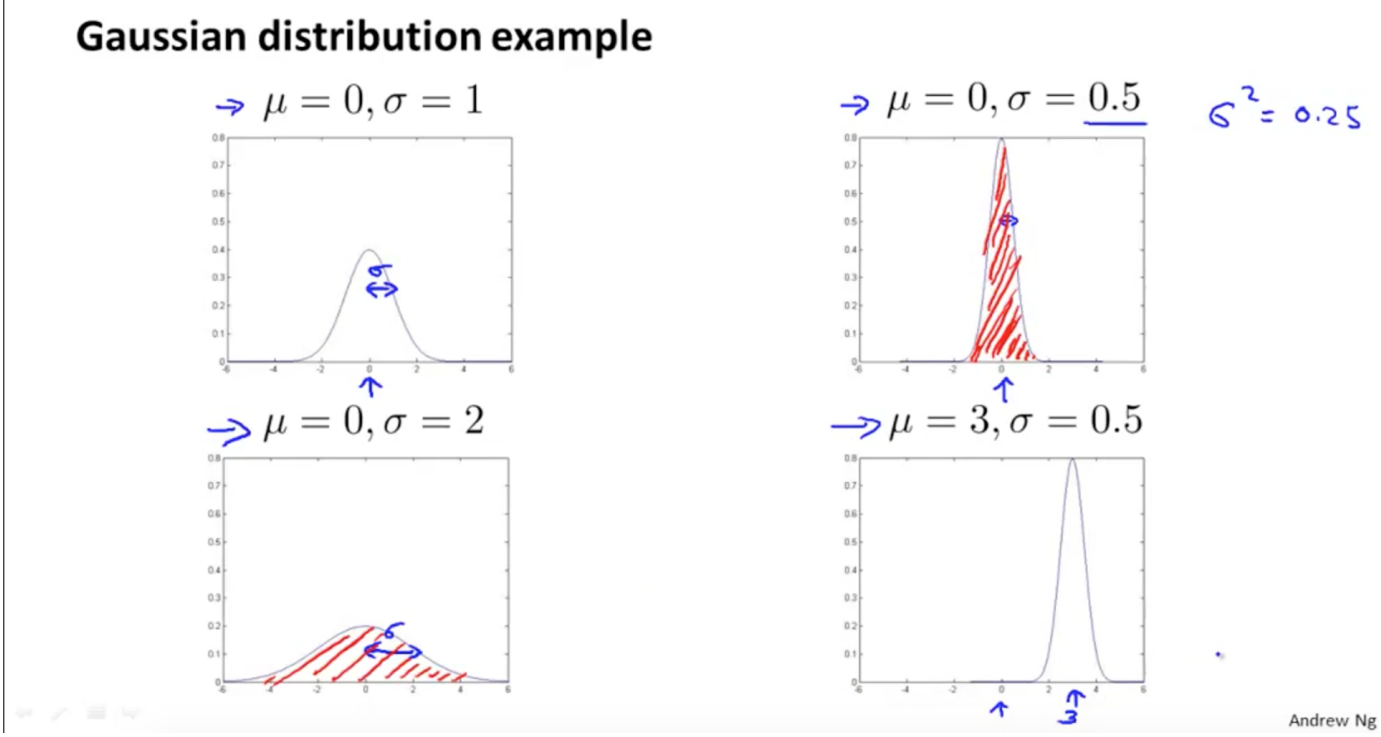

Figure 16: Quick Review on Gaussian Distribution. Note: the area always integrates to 1.

11.2 Algorithm

- *Density Estimation*
- we have the training set x1, 2, 3... etc. and each example is $x \in (R)^n$
- So, x_i $\mathcal{N}(\mu_i, \sigma_i^2)$, with i being the training examples
- and now p(x)
- $= p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2)...p(x_n; \mu_n, \sigma_n^2)$
- OR $\prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$
- Now with this, we can make the algorithm:
    1. Choose features x_i that you think might be indicative of anomalous examples

2. Fit parameters $\mu_1, ..., \mu_n, \sigma_1^2, ..., \sigma_n^2$, in otherwords, use the equations for mu and sigma
3. Given new example x, compute p(x)
   $p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{sqrt2\pi\sigma_j} exp(-\frac{(x_j-\mu_j)^2}{2\sigma_j^2})$
   Add your condition: Anomaly if p(x) $< \epsilon$

11.3 Building an Anomaly Detection System

- train the program with mostly non-anomalous data
- include a few examples in the x-validation and test set
- Note: as mentioned before, good evaluation metrics are use the true positives and stuff, precisions/recall, and F-score.
- Can also can use validation set to pick a good ϵ

11.4 Anomaly Detection vs. Supervised Learning

- Anomaly detection: Use if -
- - small # of positive examples (y=1) (0-20 is a common amount)
- - large # of negative examples (y=0)
- - Many diff "types" of anomalies. Hard for an algorithm to learn from positive examples what anomalies look like, so easier to just learn what negative examples look like and just flag things that are different
- -future examples may not look like the examples we have trained on
- Ex. - Fraud detection (easier if only a few occurrences of fraud in your company)
- Ex. - Manufacturing (again, easier if only a few occurrences, otherwise be better to use Supervised Learning)
- Ex. - Monitoring machines in data center (again, same argument)
- Supervised Learning: Use if -
- - large # of positive and negative examples
- - we have enough positive examples and are confident future positive examples will be similar
- Ex. - Spam Classification
- Ex. - Weather prediction
- Ex. - Cancer Classification

11.5 Choosing What Features to Use

- Note: This is mainly about choosing features with values that will act out when an anomaly occurs
- Note: first, try to plot a histogram (hist function in Matlab/Octave) to see it it looks gausian (it's ok if it doesn't look gaussian, it'll probably still work)
- That, or try taking the log, square root, cube root, etc. of the histogram to see if that makes it look gaussian
- Note: the histogram command is something like hist(x) or hist(x, numOfBins) And the run it on a vector or matrix x

- Picking features
- try some Error analysis for anomaly detection
- want p(x) large for normal examples of x and small for anomalous examples. (to fit the gaussian curve)
- Common Problem: p(x) is large comparable (similar magnitude) for normal and anomalous examples
- So try to find a feature that would give better information.

- Examples: Monitoring computers in a data center
- choose features that might take unusually large or small values in the event of an anomaly
- let's say we have:
- x1 = memory use of computer
- x2 = number of disk accesses/sec
- x3 = CPU load
- x4 = network traffic
- this is good, but we might get more info from connecting some info together, like x5 = $\frac{CPUload}{networktraffic}$ or x6 = $\frac{(CPUload)^2}{networktraffic}$

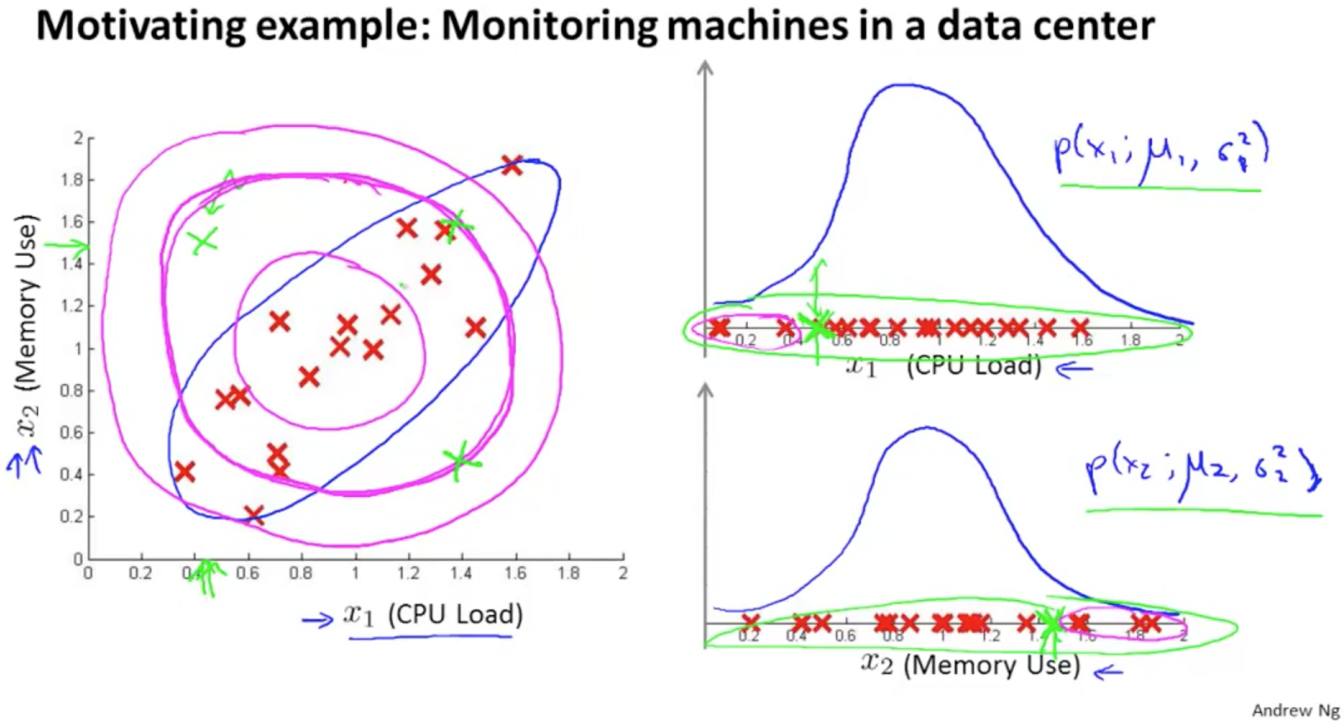

Figure 17: When we move to a 3D plot, it's clear that the 2 green points are anomalies

11.6 Multivariate Gaussian Distribution

- Sometimes you might have 2 features that each score 0.2, but it isn't high enough to be flagged as an anomaly, but it would be if it were on a 3D plot
- *Multivariate Gaussian (Normal) Distribution*
- $x \in \mathbb{R}^n$ don't model $p(x_1), p(x_2)...$ etc. separately
- model them altogether
- Parameters: $\mu \in \mathbb{R}^n, \Sigma \in \mathbb{R}^{nxn}$ (covariance matrix)

11.7 Anomaly Detection using the Multivariate Gaussian Distribution

- remember, we have parameters $\mu \in \mathbb{R}^n, \Sigma \in \mathbb{R}^{nxn}$
- But to get this p we use:

- $p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}} exp(1\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu))$
- and to get μ and Σ
- $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
- $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu)(x^{(i)}-\mu)^T$
    1. fit model p(x) by setting μ and Σ
    2. given new example x, compute p(x), and flag as anomaly if p(x) $< \epsilon$
- turns out our old not multivariate model is a variation of the multivariate Gaussian, just with the Σ with zeros everywhere except the diagonal
- *Original model vs. Multivariate Gaussian*
- Original Model:
- might need to manually create features to spot anomalies (like that thing with the cpu load and memory use)
- computationally cheaper, also scales better to large n (could be 100's of 1000's)
- it works even if m (training set size) is small
- Multivariate Model:
- automatically capture correlations between features
- but computationally more expensive
- must have $m > n$, or else Σ is non-invertable
- better to use only if m is much greater than n (maybe 10 times n)
- also won't work if there are redundant features, like duplicated features
- (in other words, linearly dependent features aren't good)

11.8 Problem Formulation

- VERY USEFUL FOR COMPANIES LIKE AMAZON, NETFLIX, etc. (can recommend products to consumers to make lots of $$$)
- Some notation we'll use for now is:
- n_u = no. users
- n_m = no. movies
- $r(i,j) = 1$ if user j has rated movie i
- $y^{(i,j)}$ = rating given by user j to movie i (defined only if $r(i,j) = 1$)

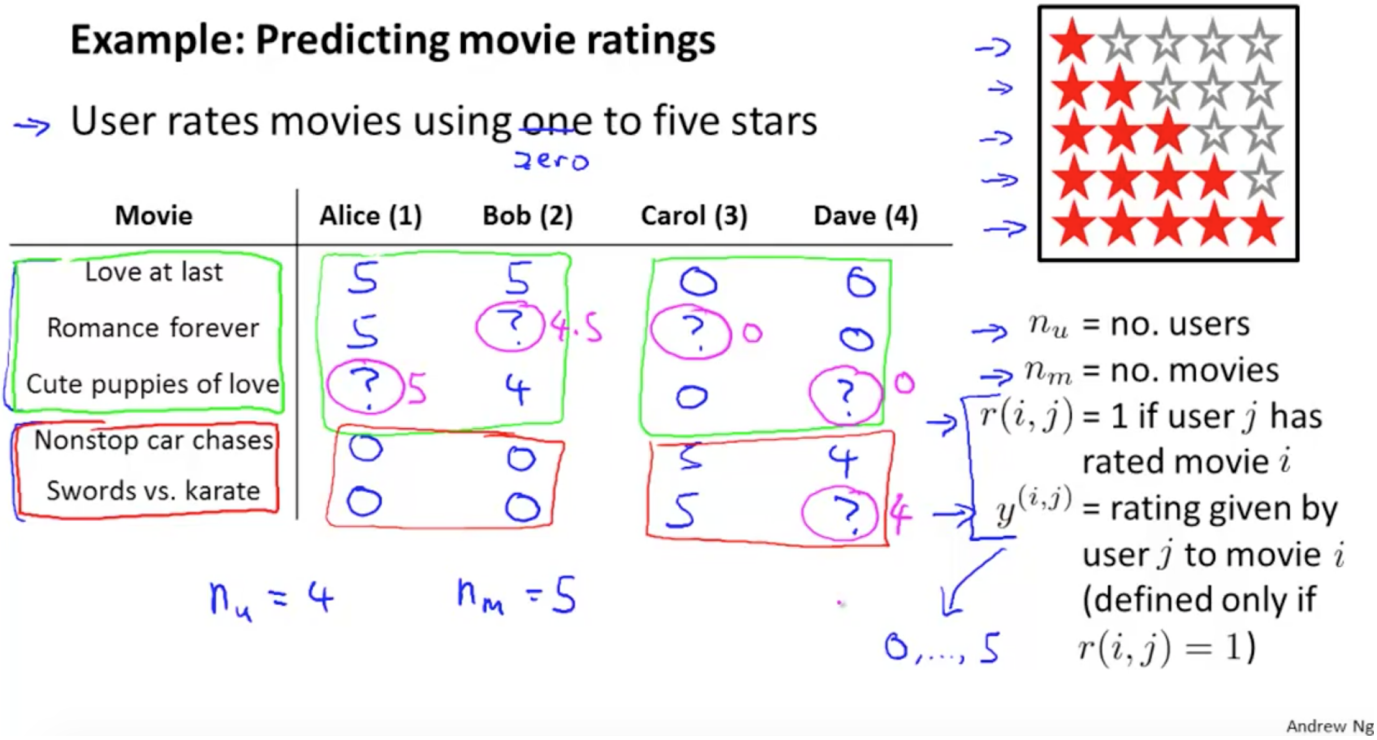

Figure 18: An example

11.9 Content Based Recommendations

- Let's say we had ratings for the genres of each movie, (like a movie had a rating for romance, and action), we would then have a feature vector
- the rest becomes linear regression

11.10 Collaborative Filtering

- (Initialize thetas to random values, just like in neural networks) guess a theta term first, the minimize theta, then x, then theta, then x, etc.

11.11 Collaborative Filtering Algorithm

- I'll be lazy for this part and just include some snapshots

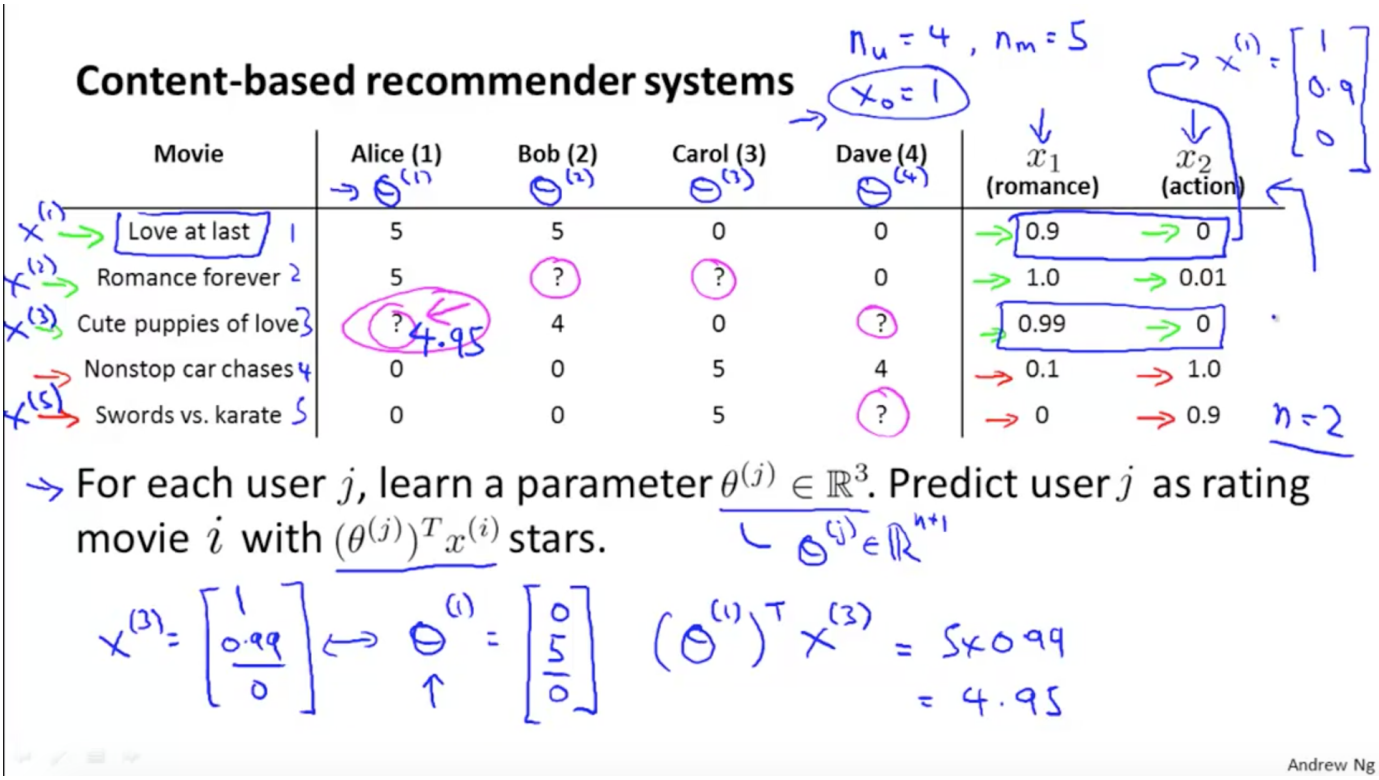

Figure 19: How to predict ratings

11.12 Vectorization: Low Rank Matrix Factorization

11.13 Implementational Detail: Mean Normalization

12 Large Scale Machine Learning

12.1 Learning With Large Datasets

- large data sets are useful for machine learning
- but there are issues that arise
- one obvious one might be that it takes a large time to perform computations on 100 million examples
- But before using large amounts of training examples, check it it's going to help
- (plot Jcv and Jtrain as a function of # on training examples like before, and check if there's high variance first)

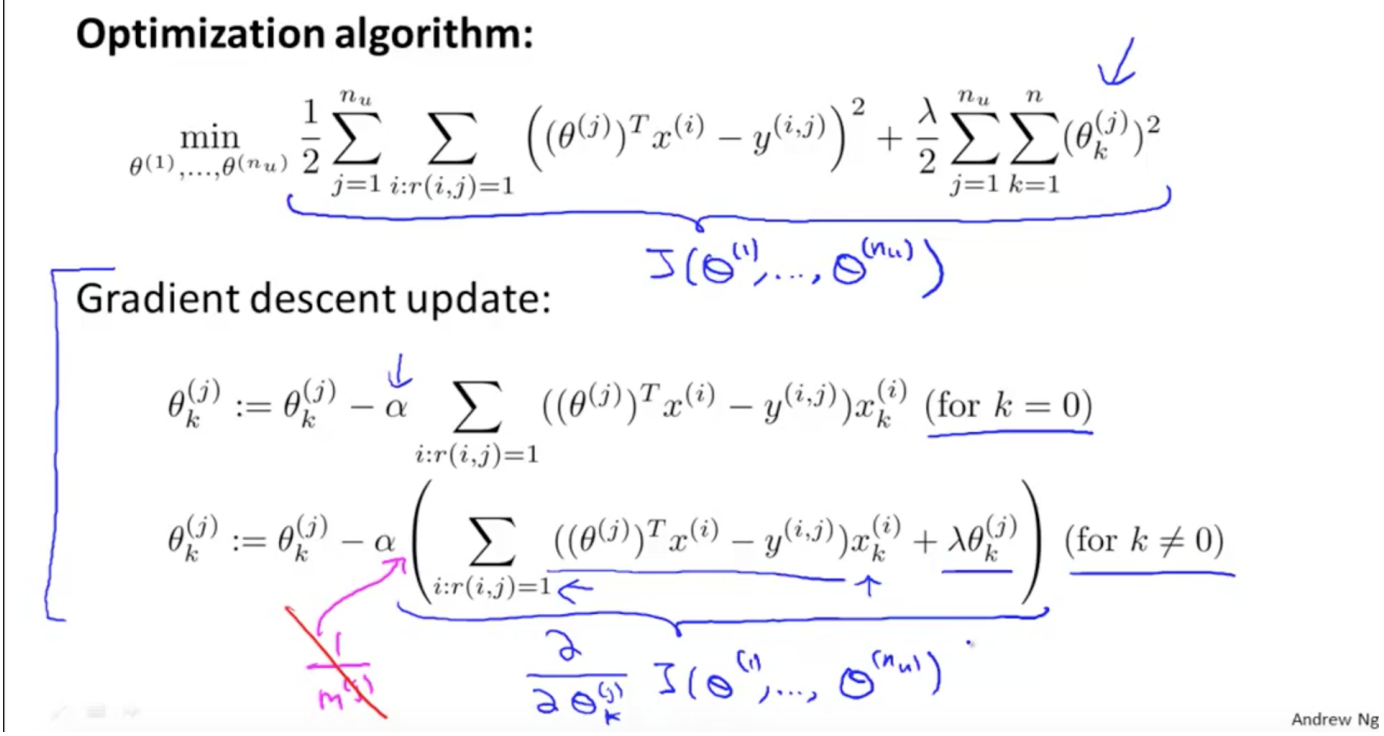

Figure 20: How to optimize the fn. Remeber, it's possbile to also use an implementation of a more complex algoirthm than gradient descent.

- (and as before, with high bias, large amounts of bias won't be fixed by large training #s (can add more neurons or something))

12.2 Stochastic Gradient Descent

- in normal gradient descent, or "batch" gradient set, we would need to compute the gradient descent for all m training examples (takes time for a large m)
- now, for large data sets, we use stochastic gradient descent
- the steps are:
  1. Randomly Shuffle Dataset
  2. Repeat for i = 1,...,m $\theta_j = \theta_j - \alpha(h_\theta(s^{(i)}) - y^{(i)})x_j^{(i)}$ Note: the h-y is the derivative of the cost function (for j=0,...,n)

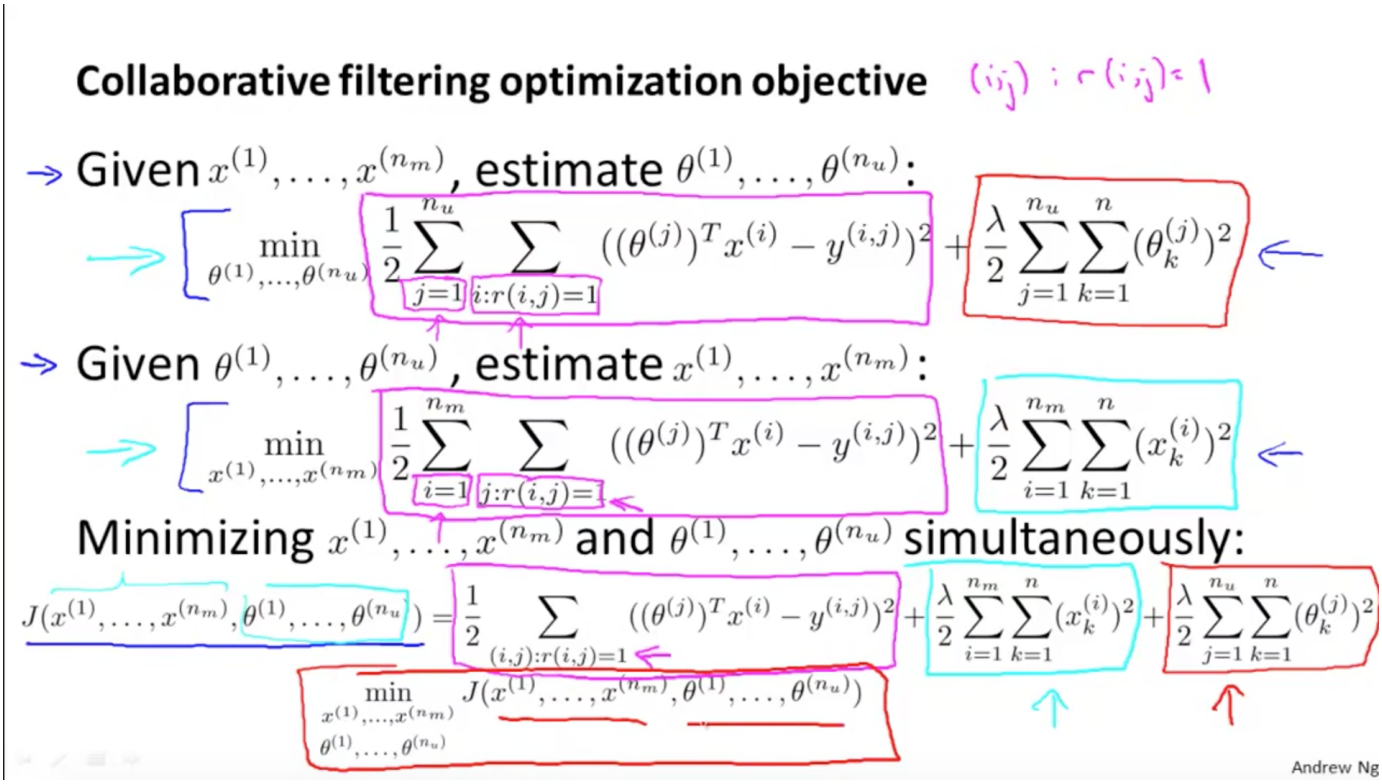

Figure 21: What to optimize

- the algorithm goes through the examples one at a time, and improves theta one at a time to build progress (will finish in one go through of all m examples)
- it generally lands near the global minimum, which is good enough
- compared to batch gradient descent, this is faster, since after batch takes only one gradient step towards the minimum after computing all m terms, and needs to compute them again to compute it again to take another small step.

12.3 Mini-Batch Gradient Descent

- So: Batch grad desc: all m examples
- Stochastic grad desc: Use 1 example each iteration
- Mini-Batch grad desc: Use b examples each iteration (b = mini-batch size)

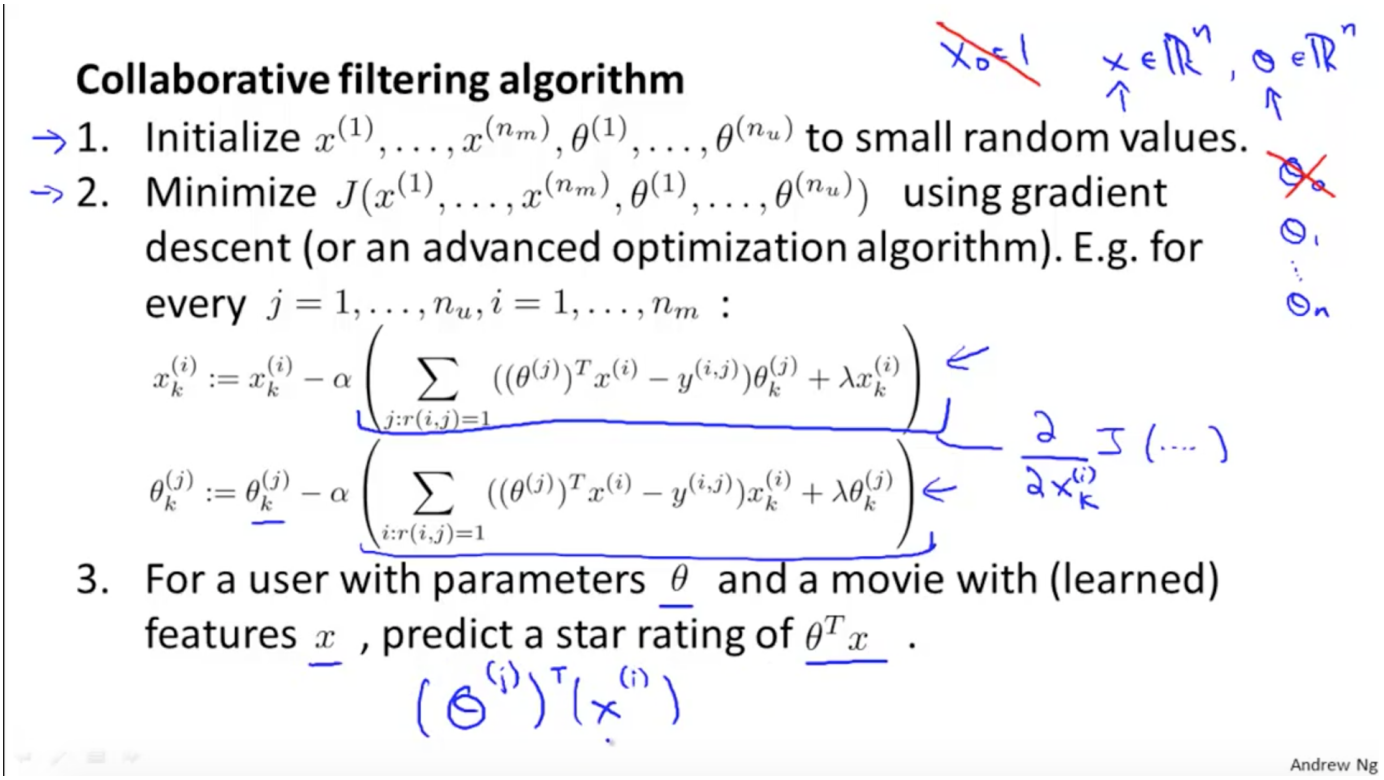

Figure 22: What to do

- Now, this can be faster than Stochastic grad desc because you can use a vectorized approach to calculating those b examples at the same time (some parallel computing, or a good linear algebra library)

12.4 Stochastic Gradient Descent Convergence

- we check the cost during learning (calc the cost during the learning)
- then, maybe every 1000 iterations, plot the cost (averaged over those 1000 examples)
- To try to make the algorithm converge better near the end of the training, you can try to make alpha equal to some constant divided by another constant plus the number of iterations, which will lower alpha as more iterations pass

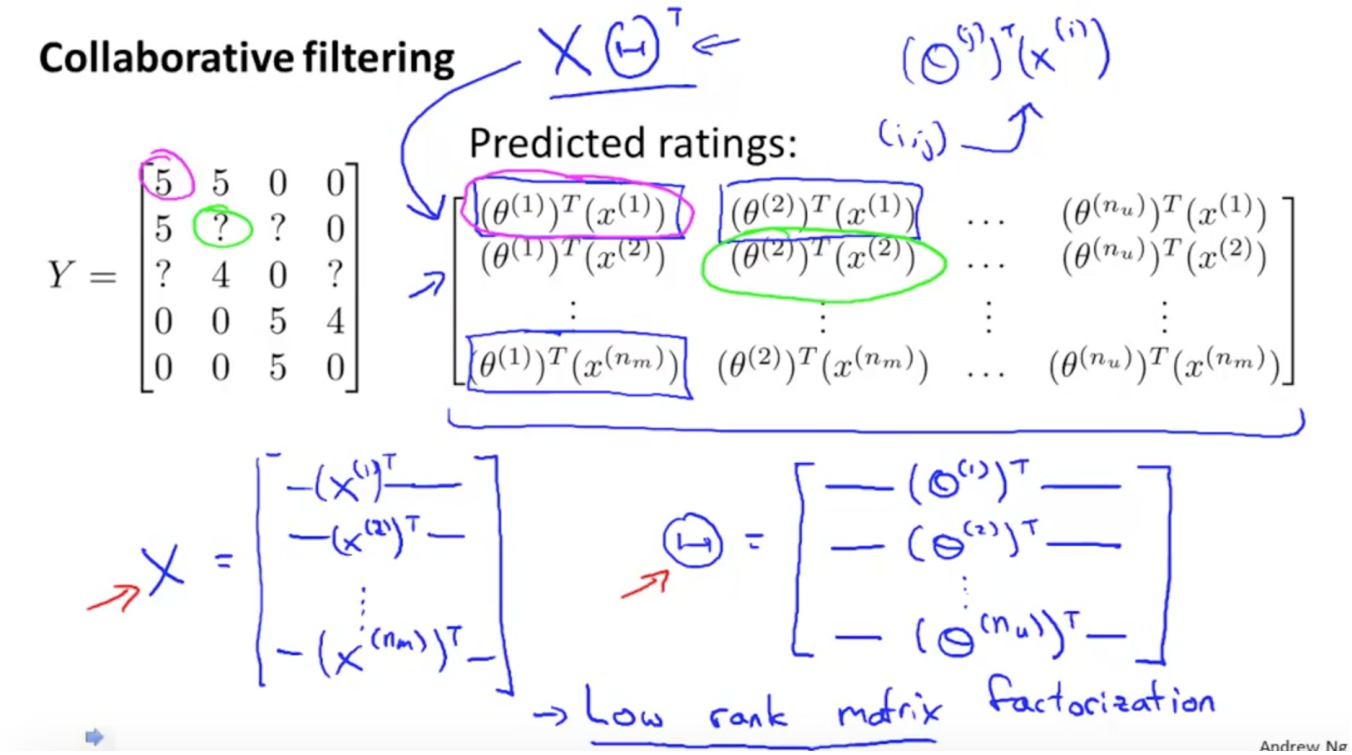

Figure 23: Uses Low Rank Matrix Factorization

12.5 Online Learning

- keep learning from a continuous stream of data (like people online on a shipping website, choosing to use your shipping service)
- maybe we would want to try to optimize the price to get the most sales
- We could make the site repeat the following forever (while it's on)
- – Get (x,y) corresponding to the user
- – Update theta using (x,y):
- – $\theta_j = \theta_j - \alpha(h_\theta - y)x_j (j = 0, ..., n)$
- This can adapt to changing user preference since it just keeps getting examples from users that visit the site

12.6 Map Reduce and Data Parallelism

- *Map-reduce*

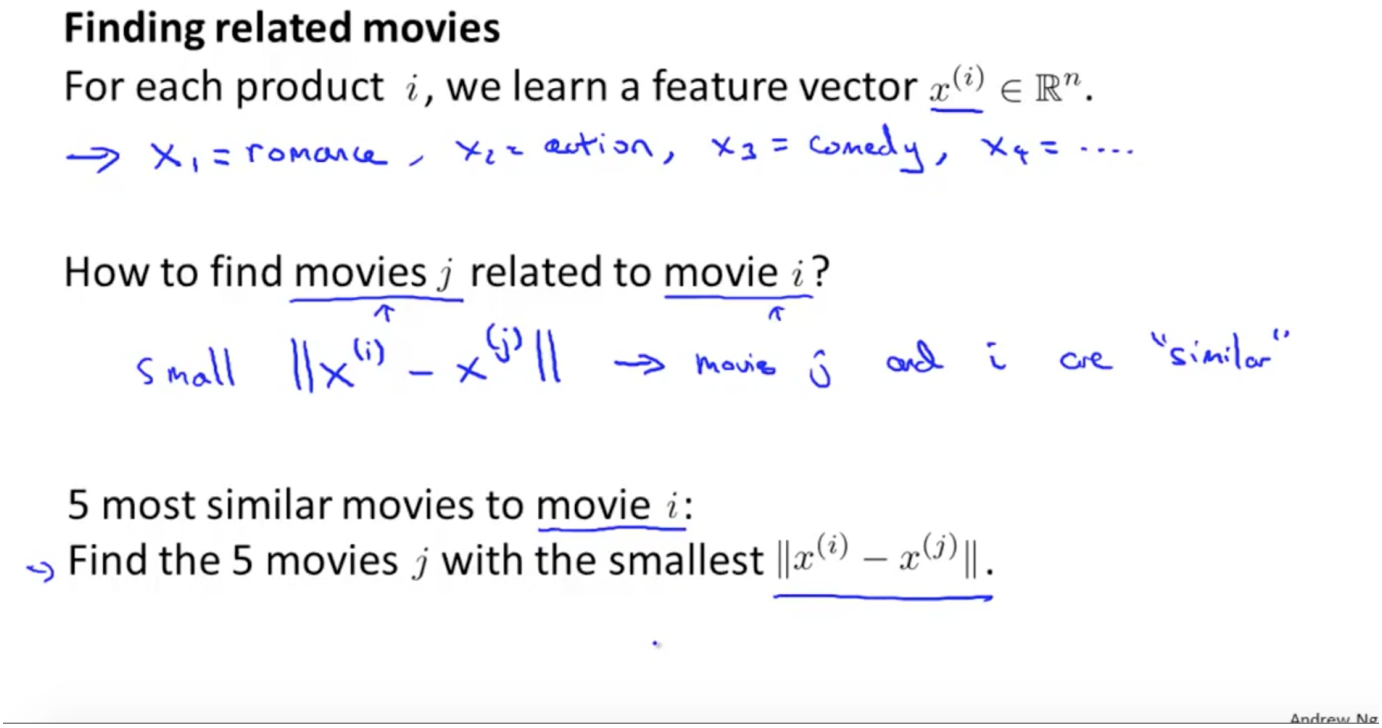

Figure 24: How to find related items: check the distance between the two items (since we're dealing with vectors)

- Ex. we're doing batch grad desc with m = 400.
- let's say we have multiple computers, maybe 4 computers
- we could have computer 1 use $(x^{(1)}, y^{(1)}, ..., x^{(100)}, y^{(100)})$
- $temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$
- then comp. 2 use $(x^{(101)}, y^{(101)}, ..., x^{(200)}, y^{(200)})$
- $temp_j^{(2)} = \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$
- and so on and so forth for all 4 machines
- so we would get $\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (temp_j^{(1)} + temp_j^{(2)} + temp_j^{(3)} + temp_j^{(4)})$

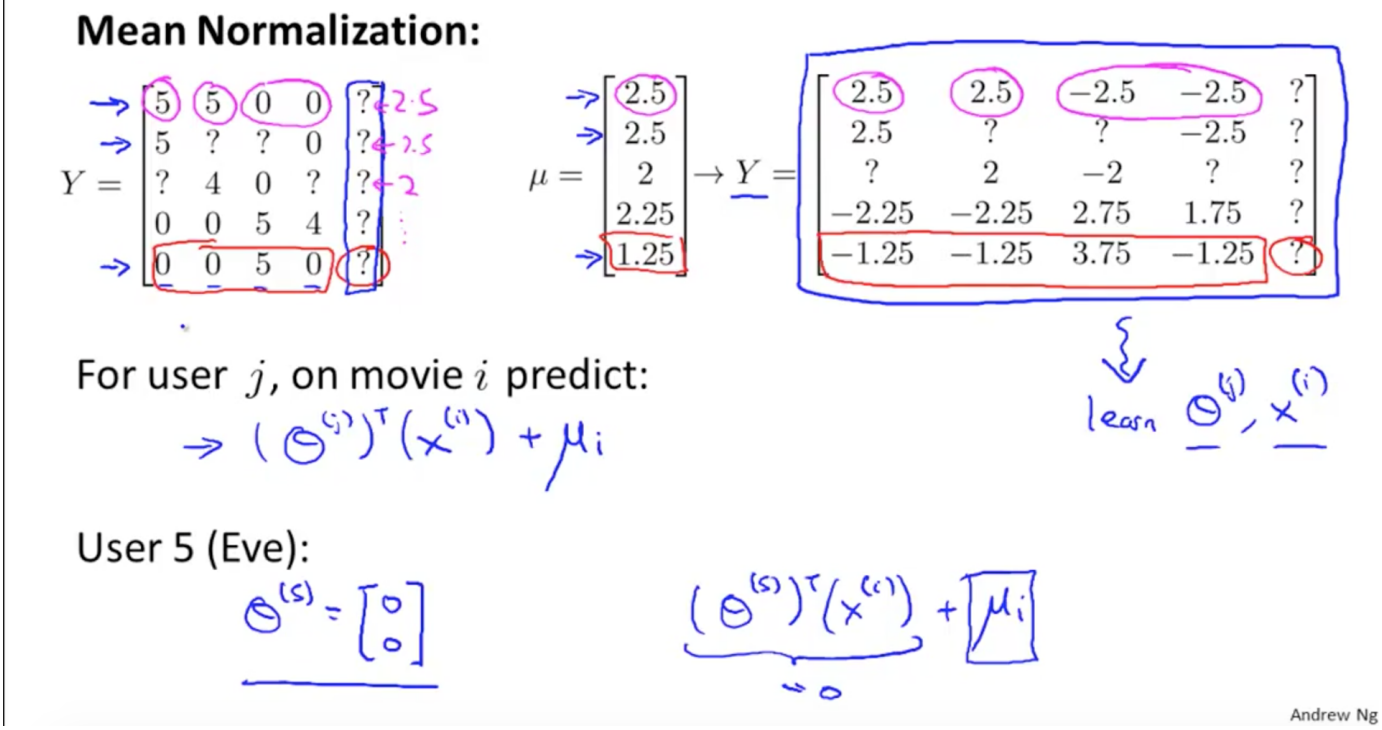

Figure 25: Normalize the means. Helps with recommending movies to a user that has not rated any movies.

- one thing that might limit it is propagation delay between the computers back to the head computer. but, this would provide a 4x speedup (in theory, in reality, we have some delay between computers)
- ---
- So, to use Map-reduce(distributed computing) and summation over training sets
- just distribute the summation/vectorized program over multiple computers (or CORES!)
- with this, network latency/propagation delay is less of an issue since the components are in the same machine
- (A good open source implementation of Map-reduce is Hadoop, check it out)

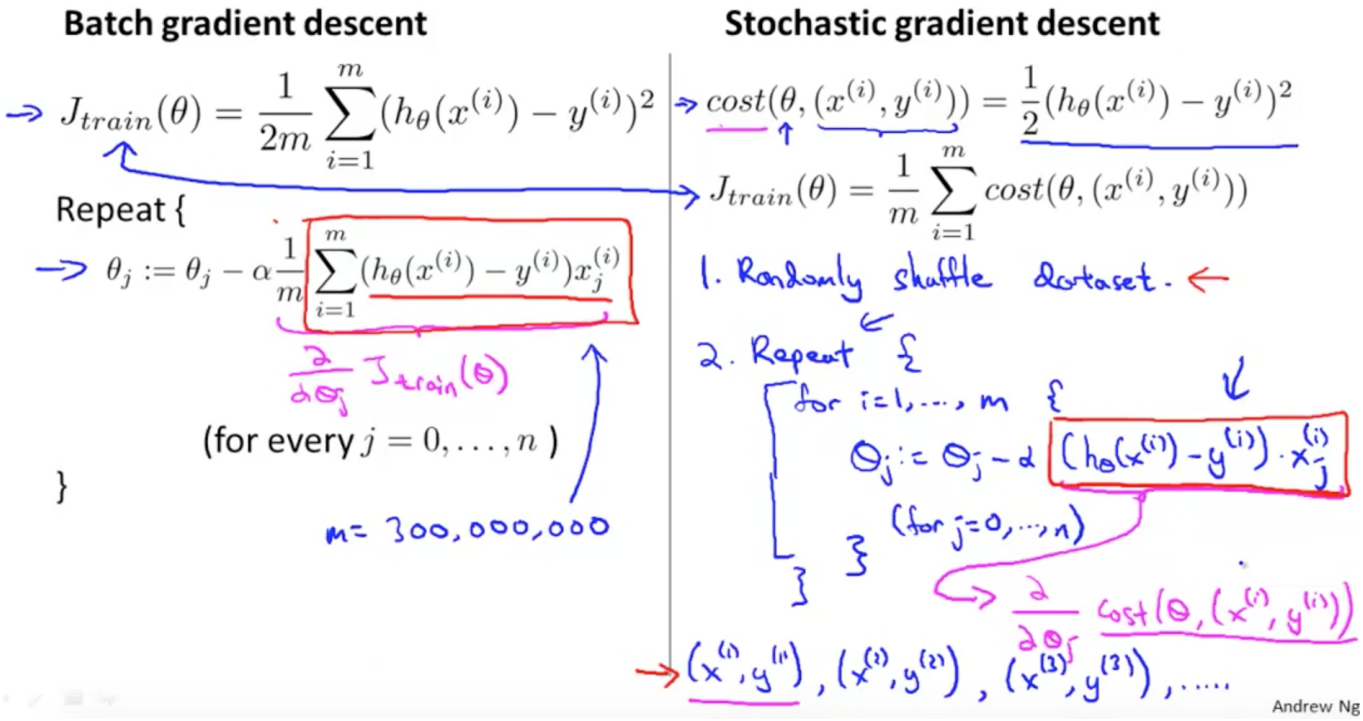

Figure 26: The algorithm, and how it compares to normal "Batch" Grad Descent

13 Applications of Machine Learning

13.1 Problem Description and Pipeline

- *The Photo OCR problem*: get computers to read text from images
- the Photo OCR pipeline, given an image to work on
  1. Text detection - detect where the text is
  2. Character segmentation - separate the characters
  3. Character classification - figure out what each character is
- we can made different modules to do each step. the first module would call the next one when it is done, and so one

13.2 Sliding Windows

- going back to detecting things, like pedestrians in a picture, how do we detect things?

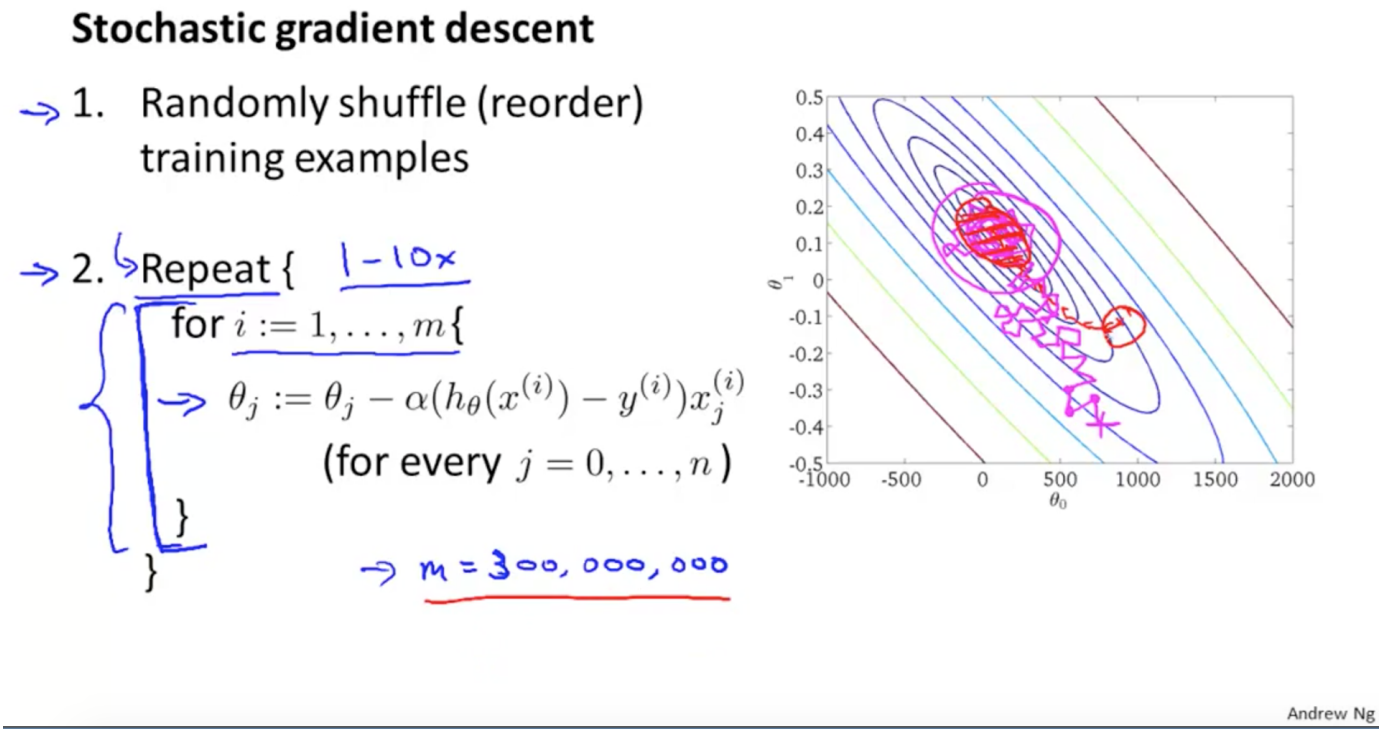

Figure 27: A visualization.

- we could have general size for the image we're searching for, and take the image, and check a box of the size we decided on, and slide it a few pixels over and check with our classifier.
- we keep doing this with these "*image patches*" of the size we decided on
- if the items are at different distances, we would need to check different sizes of rectangles due to the different distances the items are away
- the result might be something like a matrix of probabilities of where text might be. to "clean it up", we could run an algorithm to see if there are other high probability areas to the left or right, and if there are, we would make this area a high probability area.
- Now, for character segmentation, we try to check for a split between characters
- and again, we do use the sliding window technique, and run our classifier on each window

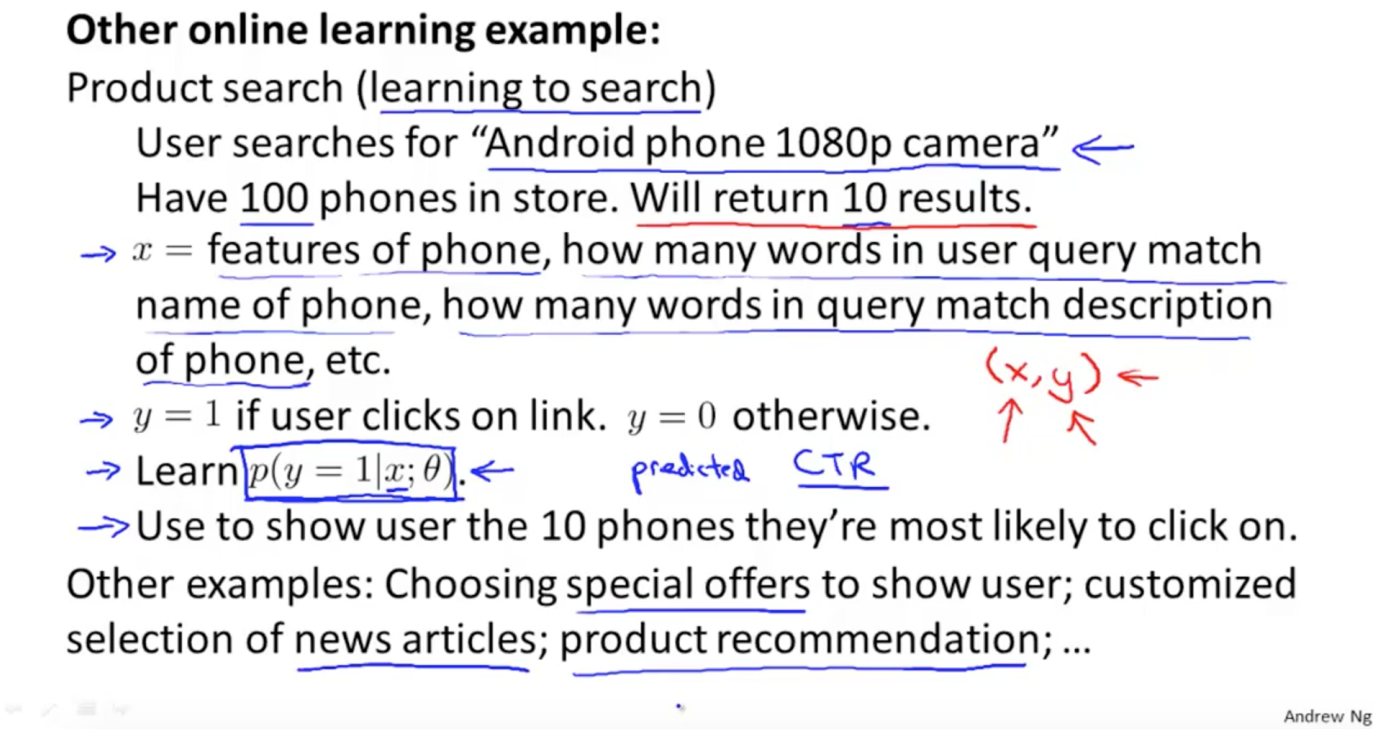

Figure 28: A good example.

13.3 Getting Lots of Data and Artificial Data

- *Artificial data synthesis for photo OCR*
- *Ex. 1* make artificial data of recognizing characters
- just take different fonts, make some text/characters, and then distort them.
- you can also take items from your character training set, you could make different distortions and have those as different examples. But make sure your distortions are reasonable
- *Ex. 2 synthesizing data by introducing distortions: Speech Recognition*
- introduce some small distortions into your speech examples
- **NOTE:** Just adding random noise usually DOES NOT HELP (like adding random gaussian dots on a character image)

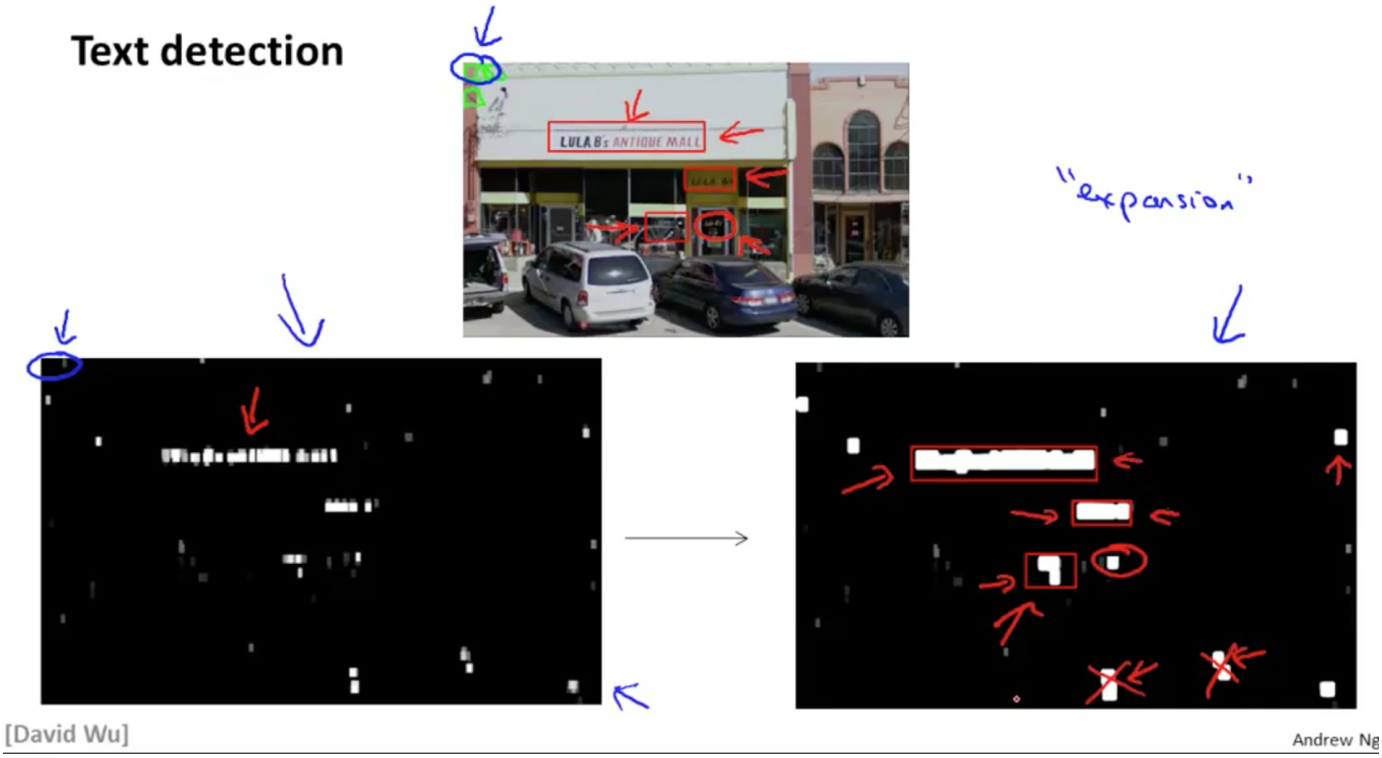

Figure 29: Ex. on what detecting text might look like

- Note: make sure you have a low bias classifier before trying to do these things.
- just plot learning curves (cost vs # of training examples)
- to lower bias, you can try adding more features/# of hidden units in a neural net until you have a low bias classifier
- Note: ask if it's possible to get 10x as much data you currently have (sometimes it's easy for them, so ask)
- - then if needed, maybe try adding artificial data
- - maybe try collecting/labeling data yourself (try to figure out how long this would take, since this can take a few days, but it's only a few days)
- - "Crowd source" (e.g. Amazon Mechanical Turk (very popular at the moment))

13.4 Ceiling Analysis: What Part of the Pipeline to Work on Next

- *Ceiling Analysis*
- we'll use the text detection example to explain
- Now: Which part of the pipeline should be focus on the most?
- We check each components and their accuracy
- We could check the overall system and it's accuracy
- Then, we could manually give the next part a perfect data set we made, so that when we check the next part's accuracy, we have an accurate way to check the accuracy
- recap: feed in our perfect results we manually made to evaluate the true accuracy of each module, without being affected by the previous module's accuracy
- a simple work flow/pipeline might be:
  1. Preprocessing (remove background)
  2. Face detection
     - Eyes segmentation
     - Nose segmentation
     - Mouth segmentation
  3. Logistic regression
- then we feed in out perfect data to each component and check each part's algorithm
- True Story: some specific scientists spent 18 months trying to improve the preprocessing, but then came to realise that it didn't really have much effect on the accuracy of the system
- Moral: DO CEILING ANALYSIS TO MAKE SURE WHAT YOU ARE DOING IS WORTH IT

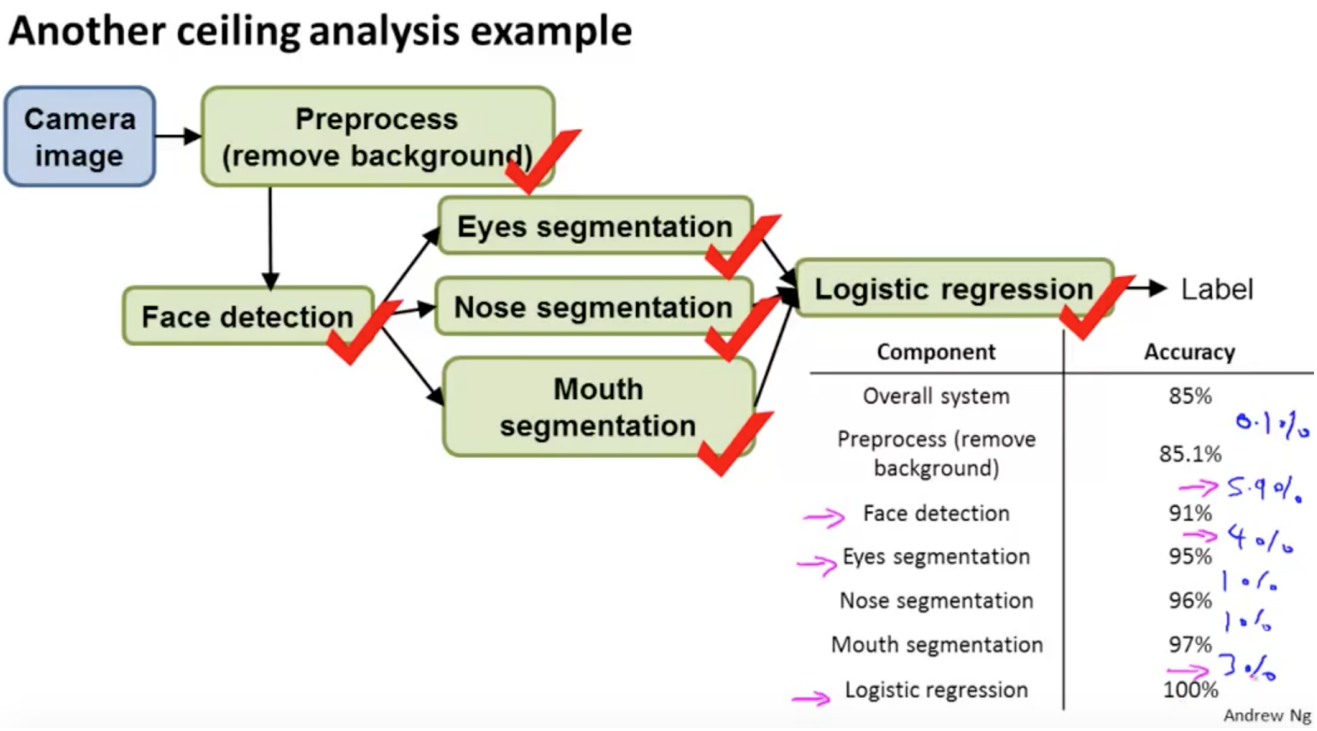

Figure 30: A good example.