16: Recommender Systems

Previous Next Index

Recommender systems - introduction

- Two motivations for talking about recommender systems
  - **Important application of ML systems**
    - Many technology companies find recommender systems to be absolutely key
    - Think about websites (amazon, Ebay, iTunes genius)
      - Try and recommend new content for you based on passed purchase
      - Substantial part of Amazon's revenue generation
    - Improvement in recommender system performance can bring in more income
    - Kind of a funny problem
      - In academic learning, recommender systems receives a small amount of attention
      - But in industry it's an absolutely crucial tool
  - Talk about the big ideas in machine learning
    - Not so much a technique, but an idea
    - As soon, features are really important
      - There's a big idea in machine learning that for some problems you can learn what a good set of features are
      - So not select those features but learn them
    - Recommender systems do this - try and identify the crucial and relevant features

Example - predict movie ratings

- You're a company who sells movies
  - You let users rate movies using a 1-5 star rating
    - To make the example nicer, allow 0-5 (makes math easier)
- You have five movies
- And you have four users
- Admittedly, business isn't going well, but you're optimistic about the future as a result of your truly outstanding (if limited) inventory

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Love at last	5	5	0	6
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	?

- To introduce some notation
  - n_u - Number of users (called $?^{nu}$ occasionally as we can't subscript in superscript)
  - n_m - Number of movies
  - r(i, j) - 1 if user j has rated movie i (i.e. bitmap)
  - $y^{(i,j)}$ - rating given by user j to move i (defined only if r(i,j) = 1
- So for this example
  - $n_u = 4$
  - $n_m = 5$
  - Summary of scoring
    - Alice and Bob gave good ratings to rom coms, but low scores to action films
    - Carol and Dave game good ratings for action films but low ratings for rom coms
  - We have the data given above

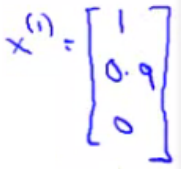
- The problem is as follows
  - Given r(i,j) and $y^{(i,j)}$ - go through and try and predict missing values (?s)
  - Come up with a learning algorithm that can fill in these missing values

Content based recommendation

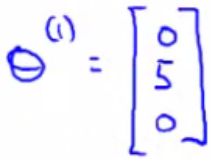
- Using our example above, how do we predict?
  - For each movie we have a feature which measure degree to which each film is a
    - Romance (x_1)
    - Action (x_2)

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs. karate	0	0	5	?	0	0.9

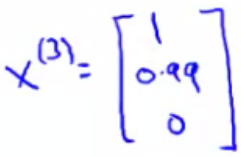
- If we have features like these, each film can be recommended by a feature vector
  - Add an extra feature which is $x_0 = 1$ for each film
  - So for each film we have a [3 x 1] vector, which for film number 1 ("Love at Last") would be



  - i.e. for our dataset we have
    - $\{x^1, x^2, x^3, x^4, x^5\}$
      - Where each of these is a [3x1] vector with an $x_0 = 1$ and then a romance and an action score
  - To be consistent with our notation, n is going to be the number of features NOT counting the x_0 term, so n = 2
- We could treat each rating for each user as a separate linear regression problem
  - For each user j we could learn a parameter vector
  - Then predict that user j will rate movie i with
    - $(\theta^j)^T x^i$ = stars
    - inner product of parameter vector and features
  - So, lets take user 1 (Alice) and see what she makes of the modern classic Cute Puppies of Love (CPOL)
    - We have some parameter vector (θ^1) associated with Alice
      - We'll explain later how we derived these values, but for now just take it that we have a vector



    - CPOL has a parameter vector (x^3) associated with it

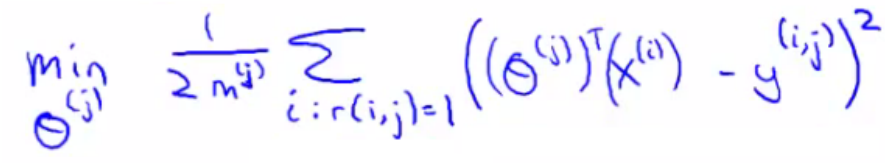


    - Our prediction will be equal to
      - $(\theta^1)^T x^3 = (0 * 1) + (5 * 0.99) + (0 * 0)$
      - = 4.95

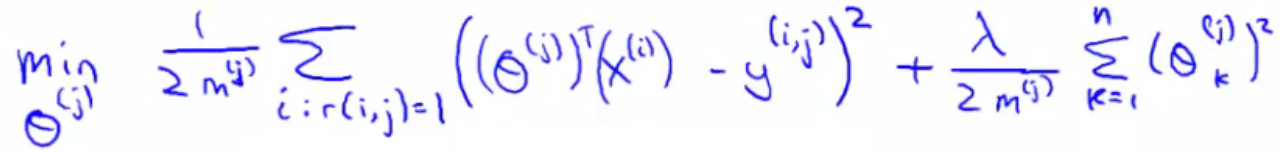
        - Which may seem like a reasonable value
    - All we're doing here is applying a linear regression method for each user
        - So we determine a future rating based on their interest in romance and action based on previous films
    - We should also add one final piece of notation
        - m^j, - Number of movies rated by the user (j)

How do we learn (θ^j)

- Create some parameters which give values as close as those seen in the data when applied



- Sum over all values of i (all movies the user has used) when r(i,j) = 1 (i.e. all the films that the user has rated)
- This is just like linear regression with least-squared error
- We can also add a regularization term to make our equation look as follows



    - The regularization term goes from k=1 through to m, so (θ^j) ends up being an n+1 feature vector
        - Don't regularize over the bias terms (0)
- If you do this you get a reasonable value
- We're rushing through this a bit, but it's just a linear regression problem

- To make this a little bit clearer you can get rid of the m^j term (it's just a constant so shouldn't make any difference to minimization)
    - So to learn (θ^j)

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

    - But for our recommender system we want to learn parameters for *all* users, so we add an extra summation term to this which means we determine the minimum (θ^j) value for every user

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

    - When you do this as a function of each (θ^j) parameter vector you get the parameters for each user
        - So this is our optimization objective -> $J(\theta^1, \ldots, \theta^{nu})$
- In order to do the minimization we have the following gradient descent

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

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

    - Slightly different to our previous gradient descent implementations
        - k = 0 and k != 0 versions
        - We can define the middle term above as

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

  - Difference from linear regression
    - No 1/m terms (got rid of the 1/m term)
    - Otherwise very similar
- This approach is called content-based approach because we assume we have features regarding the content which will help us identify things that make them appealing to a user
  - However, often such features are not available - next we discuss a non-contents based approach!

Collaborative filtering - overview

- The collaborative filtering algorithm has a very interesting property - does feature learning
  - i.e. it can learn for itself what features it needs to learn
- Recall our original data set above for our five films and four raters
  - Here we assume someone had calculated the "romance" and "action" amounts of the films
    - This can be very hard to do in reality
    - Often want more features than just two
- So - let's change the problem and pretend we have a data set where we don't know any of the features associated with the films

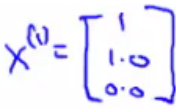
Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	?	?
Romance forever	5	?	?	0	?	?
Cute puppies of love	?	4	0	?	?	?
Nonstop car chases	0	0	5	4	?	?
Swords vs. karate	0	0	5	?	?	?

  - Now let's make a different assumption
    - We've polled each user and found out how much each user likes
      - Romantic films
      - Action films
    - Which has generated the following parameter set

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

    - Alice and Bob like romance but hate action
    - Carol and Dave like action but hate romance
- If we can get these parameters from the users we can infer the missing values from our table
  - Lets look at "Love at Last"
    - Alice and Bob loved it
    - Carol and Dave hated it
  - We know from the feature vectors Alice and Bob love romantic films, while Carol and Dave hate them
    - Based on the factor Alice and Bob liked "Love at Last" and Carol and Dave hated it we may be able to (correctly) conclude that "Love at Last" is a romantic film
- This is a bit of a simplification in terms of the maths, but what we're really asking is
  - "What feature vector should x^1 be so that
    - $(\theta^1)^T x^1$ is about 5

        - $(\theta^2)^T x^1$ is about 5
        - $(\theta^3)^T x^1$ is about 0
        - $(\theta^4)^T x^1$ is about 0
    - From this we can guess that x^1 may be



    - Using that same approach we should then be able to determine the remaining feature vectors for the other films

Formalizing the collaborative filtering problem

- We can more formally describe the approach as follows
    - Given (θ^1, ..., θ^{nu}) (i.e. given the parameter vectors for each users' preferences)
    - We must minimize an optimization function which tries to identify the best parameter vector associated with a film

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

        - So we're summing over all the indices j for where we have data for movie i
        - We're minimizing this squared error
    - Like before, the above equation gives us a way to learn the features for one film
        - We want to learn all the features for *all* the films - so we need an additional summation term

How does this work with the previous recommendation system

- Content based recommendation systems
    - Saw that if we have a set of features for movie rating you can learn a user's preferences
- Now
    - If you have your users preferences you can therefore determine a film's features
- This is a bit of a chicken & egg problem
- What you can do is
    - Randomly guess values for θ
    - Then use collaborative filtering to generate x
    - Then use content based recommendation to improve θ
    - Use that to improve x
    - And so on
- This actually works
    - Causes your algorithm to converge on a reasonable set of parameters
    - This is collaborative filtering
- We call it collaborative filtering because in this example the users are collaborating together to help the algorithm learn better features and help the system and the other users

Collaborative filtering Algorithm

- Here we combine the ideas from before to build a collaborative filtering algorithm
- Our starting point is as follows
    - If we're given the film's features we can use that to work out the users' preference

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

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

    - If we're given the users' preferences we can use them to work out the film's features

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

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

- One thing you could do is
  - Randomly initialize parameter
  - Go back and forward
- But there's a more efficient algorithm which can solve θ and x simultaneously
  - Define a new optimization objective which is a function of x and θ

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

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

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

- Understanding this optimization objective
  - The squared error term is the same as the squared error term in the two individual objectives above

$$\sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2$$

    - So it's summing over every movie rated by every users
    - Note the ":" means, "for which"
      - Sum over all pairs (i,j) for which r(i,j) is equal to 1
  - The regularization terms
    - Are simply added to the end from the original two optimization functions
- This newly defined function has the property that
  - If you held x constant and only solved θ then you solve the, "Given x, solve θ" objective above
  - Similarly, if you held θ constant you could solve x
- In order to come up with just one optimization function we treat this function as a function of both film features x and user parameters θ
  - Only difference between this in the back-and-forward approach is that we minimize with respect to both x and θ simultaneously
- When we're learning the features this way
  - Previously had a convention that we have an $x_0 = 1$ term
  - When we're using this kind of approach we have no x_0,
    - So now our vectors (both x and θ) are n-dimensional (not n+1)
  - We do this because we are now learning all the features so if the system needs a feature always = 1 then the algorithm can learn one

Algorithm Structure

- **1)** Initialize $\theta^1, \dots, \theta^{nu}$ and $x^1, \dots, x^{nm}$ to small random values
  - A bit like neural networks - initialize all parameters to small random numbers
- **2)** Minimize cost function ($J(x^1, \dots, x^{nm}, \theta^1, \dots, \theta^{nu})$ using gradient descent
  - We find that the update rules look like this

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

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

  - Where the top term is the partial derivative of the cost function with respect to x_k^i while the bottom is the partial derivative of the cost function with respect to θ_k^i
  - So here we regularize EVERY parameters (no longer x_0 parameter) so no special case update rule
- **3)** Having minimized the values, given a user (user j) with parameters θ and movie (movie i) with learned features x, we predict a start rating of $(\theta^j)^T x^i$
  - This is the collaborative filtering algorithm, which should give pretty good predictions for how users like new movies

Vectorization: Low rank matrix factorization

- Having looked at collaborative filtering algorithm, how can we improve this?
  - Given one product, can we determine other relevant products?
- We start by working out another way of writing out our predictions
  - So take all ratings by all users in our example above and group into a matrix Y

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

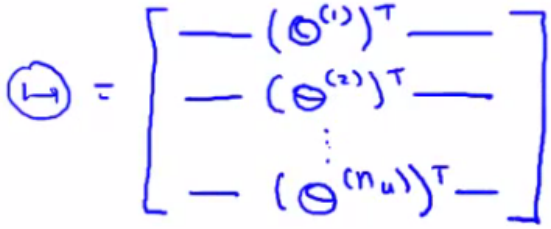
    - 5 movies
    - 4 users
    - Get a [5 x 4] matrix
  - Given [Y] there's another way of writing out all the predicted ratings

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

    - With this matrix of predictive ratings
    - We determine the (i,j) entry for EVERY movie
- We can define another matrix X
  - Just like matrix we had for linear regression
  - Take all the features for each movie and stack them in rows

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

    - Think of each movie as one example
  - Also define a matrix Θ



    - Take each per user parameter vector and stack in rows
- Given our new matrices X and θ
    - We can have a vectorized way of computing the prediction range matrix by doing $X * \theta^T$
- We can given this algorithm another name - **low rank matrix factorization**
    - This comes from the property that the $X * \theta^T$ calculation has a property in linear algebra that we create a **low rank** matrix
        - Don't worry about what a low rank matrix is

Recommending new movies to a user

- Finally, having run the collaborative filtering algorithm, we can use the learned features to find related films
    - When you learn a set of features you don't know what the features will be - lets you identify the features which define a film
    - Say we learn the following features
        - x_1 - romance
        - x_2 - action
        - x_3 - comedy
        - x_4 - ...
    - So we have n features all together
    - After you've learned features it's often very hard to come in and apply a human understandable metric to what those features are
        - Usually learn features which are very meaning full for understanding what users like
- Say you have movie i
    - Find movies j which is similar to i, which you can recommend
    - Our features allow a good way to measure movie similarity
    - If we have two movies x^i and x^j
        - We want to minimize $||x^i - x^j||$
            - i.e. the distance between those two movies
    - Provides a good indicator of how similar two films are in the sense of user perception
        - NB - Maybe ONLY in terms of user perception

Implementation detail: Mean Normalization

- Here we have one final implementation detail - make algorithm work a bit better
- To show why we might need mean normalization let's consider an example where there's a user who hasn't 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	?	?

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

    - Lets see what the algorithm does for this user
        - Say n = 2
        - We now have to learn θ^5 (which is an n-dimensional vector)
    - Looking in the first term of the optimization objective

- There are *no* films for which r(i,j) = 1
- So this term places no role in determining θ^5
- So we're just minimizing the final regularization term

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

This term is irrelevant

We're only minimizing this

Which can for our single example be simplified to this $\frac{\lambda}{2}\left[(\theta_1^{(5)})^2 + (\theta_2^{(5)})^2\right]$

- Of course, if the goal is to minimize this term then

$\theta^{(5)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

  - Why - If there's no data to pull the values away from 0 this gives the min value
- So this means we predict ANY movie to be zero
  - Presumably Eve doesn't hate all movies...
  - So if we're doing this we can't recommend any movies to her either
- Mean normalization should let us fix this problem

How does mean normalization work?

- Group all our ratings into matrix Y as before
  - We now have a column of ?s which corresponds to Eves rating

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

  - Now we compute the average rating each movie obtained and stored in an n_m - dimensional column vector

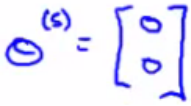
$$\mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix}$$

  - If we look at all the movie ratings in [Y] we can subtract off the mean rating

$$Y = \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$$

    - Means we normalize each film to have an average rating of 0
  - Now, we take the new set of ratings and use it with the collaborative filtering algorithm
    - Learn θ^j and x^i from the mean normalized ratings
- For our prediction of user j on movie i, predict
  - $(\theta^j)^T x^i + \mu_i$

        - Where these vectors are the mean normalized values
        - We have to add μ because we removed it from our θ values
    - So for user 5 the same argument applies, so



    - So on any movie i we're going to predict
        - $(\theta^5)^T x^i + \mu_i$
            - Where $(\theta^5)^T x^i$ = to 0 (still)
            - But we then add the mean (μ_i) which means Eve has an average rating assigned to each movie for here
- This makes sense
    - If Eve hasn't rated any films, predict the average rating of the films based on everyone
        - This is the best we can do

- As an aside - we spoke here about mean normalization for users with no ratings
    - If you have some movies with no ratings you can also play with versions of the algorithm where you normalize the columns
    - BUT this is probably less relevant - probably shouldn't recommend an unrated movie
- To summarize, this shows how you do mean normalization preprocessing to allow your system to deal with users who have not yet made any ratings
    - Means we recommend the user we know little about the best average rated products

17: Large Scale Machine Learning

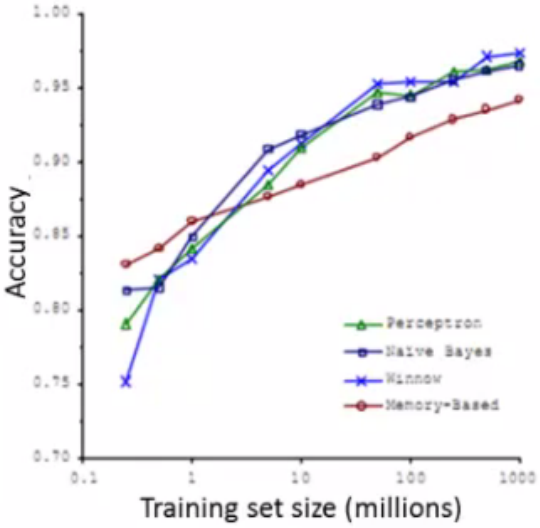
Previous Next Index

Learning with large datasets

- This set of notes look at large scale machine learning - how do we deal with big datasets?
- If you look back at 5-10 year history of machine learning, ML is much better now because we have much more data
  - However, with this increase in data comes great responsibility? No, comes a much more significant computational cost
  - New and exciting problems are emerging that need to be dealt with on both the algorithmic and architectural level

Why large datasets?

- One of best ways to get high performance is take a low bias algorithm and train it on a lot of data
  - e.g. Classification between confusable words
  - We saw that so long as you feed an algorithm lots of data they all perform pretty similarly



  - So it's good to learn with large datasets
- But learning with large datasets comes with its own computational problems

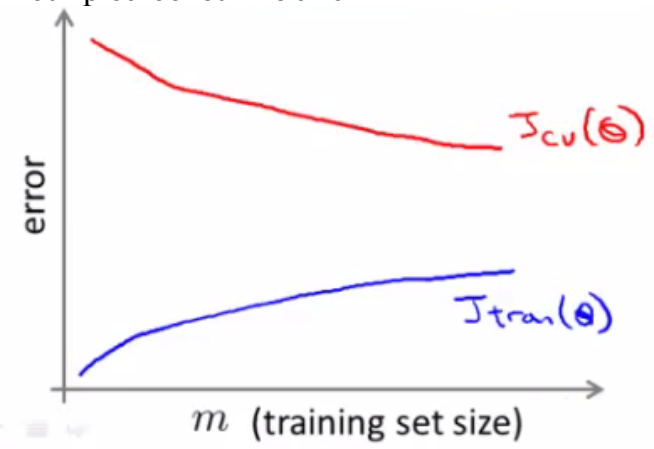
Learning with large datasets

- For example, say we have a data set where m = 100, 000, 000
  - This is pretty realistic for many datasets
    - Census data
    - Website traffic data
  - How do we train a logistic regression model on such a big system?

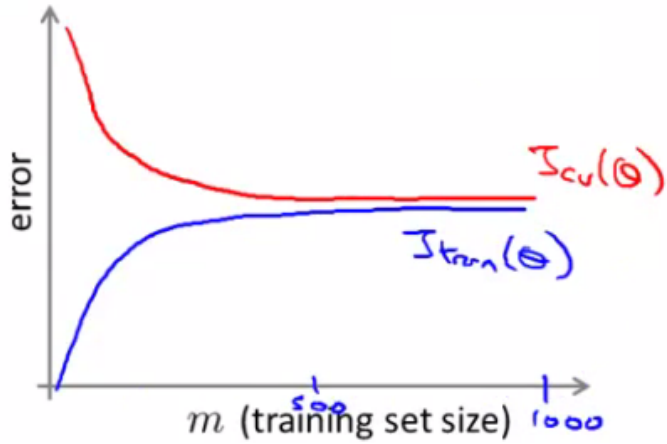
$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

    - So you have to sum over 100,000,000 terms per step of gradient descent
- Because of the computational cost of this massive summation, we'll look at more efficient ways around this
  - - Either using a different approach
  - - Optimizing to avoid the summation
- First thing to do is ask if we can train on 1000 examples instead of 100 000 000

    - Randomly pick a small selection
    - Can you develop a system which performs as well?
        - Sometimes yes - if this is the case you can avoid a lot of the headaches associated with big data
- To see if taking a smaller sample works, you can sanity check by plotting error vs. training set size
    - If our plot looked like this



    - Looks like a **high variance problem**
        - More examples should improve performance
    - If plot looked like this



    - This looks like a **high bias problem**
        - More examples may not actually help - save a lot of time and effort if we know this *before hand*
        - One natural thing to do here might be to;
            - Add extra features
            - Add extra hidden units (if using neural networks)

Stochastic Gradient Descent

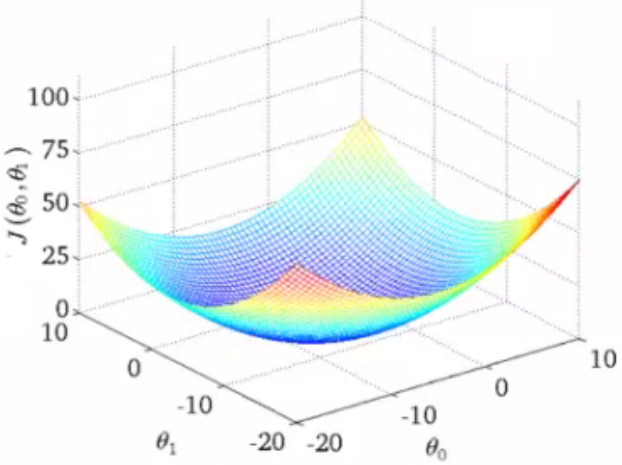
- For many learning algorithms, we derived them by coming up with an optimization objective (cost function) and using an algorithm to minimize that cost function
    - When you have a large dataset, gradient descent becomes very expensive
    - So here we'll define a different way to optimize for large data sets which will allow us to scale the algorithms
- Suppose you're training a linear regression model with gradient descent
    - **Hypothesis**

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

    - **Cost function**

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

- If we plot our two parameters vs. the cost function we get something like this



- Looks like this bowl shape surface plot

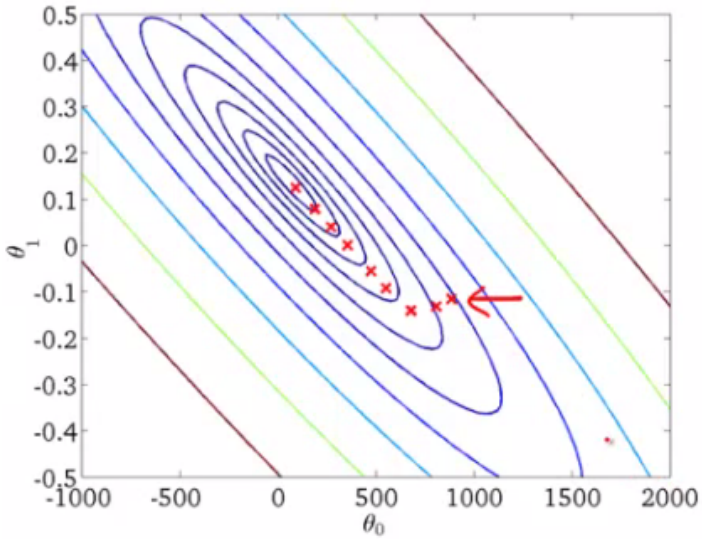
- **Quick reminder - how does gradient descent work?**

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

$$(\text{for every } j = 0, \ldots, n) \quad \}$$

- In the inner loop we repeatedly update the parameters θ

- We will use linear regression for our algorithmic example here when talking about **stochastic gradient descent**, although the ideas apply to other algorithms too, such as
  - Logistic regression
  - Neural networks
- Below we have a contour plot for gradient descent showing iteration to a global minimum



- As mentioned, if *m* is large gradient descent can be very expensive
- Although so far we just referred to it as gradient descent, this kind of gradient descent is called **batch gradient descent**
  - This just means we look at all the examples at the same time
- Batch gradient descent is not great for huge datasets
  - If you have 300,000,000 records you need to read in all the records into memory from disk because

you can't store them all in memory
- By reading all the records, you can move one step (iteration) through the algorithm
- Then repeat for EVERY step
  - This means it take a LONG time to converge
  - Especially because disk I/O is typically a system bottleneck anyway, and this will inevitably require a *huge* number of reads

- What we're going to do here is come up with a different algorithm which only needs to look at single example at a time

Stochastic gradient descent

- Define our cost function slightly differently, as

$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

  - So the function represents the cost of θ with respect to a specific example (x^i, y^i)
    - And we calculate this value as one half times the squared error on that example
  - Measures how well the hypothesis works on a single example
- The overall cost function can now be re-written in the following form;

$$J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

  - This is equivalent to the batch gradient descent cost function
- With this slightly modified (but equivalent) view of linear regression we can write out how stochastic gradient descent works

- **1) - Randomly shuffle**

Randomly shuffle (reorder)
training examples

- **2) - Algorithm body**

Repeat {
  for $i := 1, \ldots, m${

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

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

  }
}

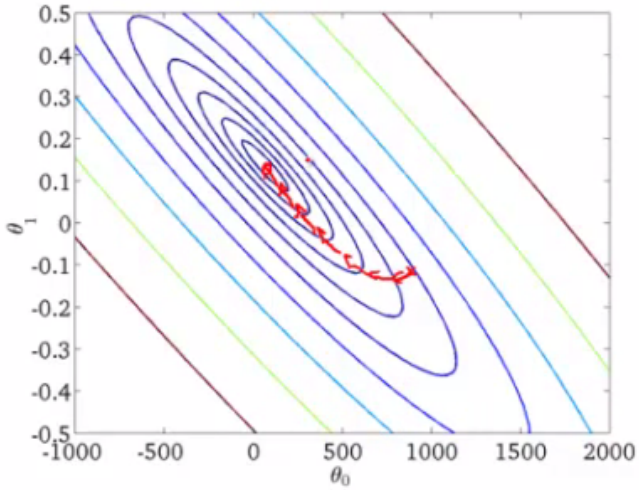
- *So what's going on here?*
  - The term

$$h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

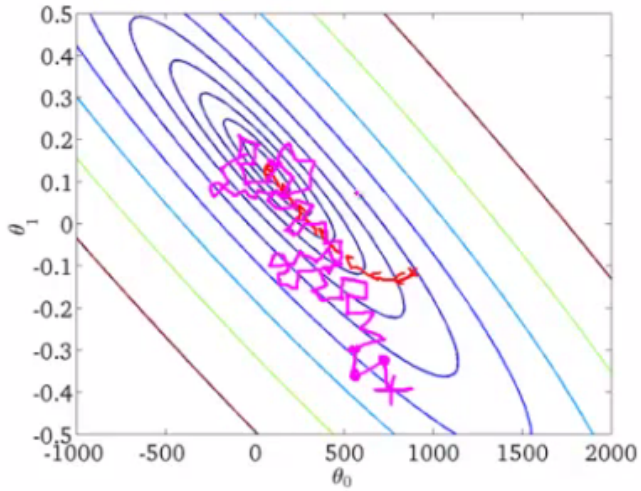
  - Is the same as that found in the summation for batch gradient descent
  - It's possible to show that this term is equal to the partial derivative with respect to the parameter θ_j of the cost(θ, (x^i, y^i))

$$\frac{\partial}{\partial \theta_j} cost(\theta, (x^{(i)}, y^{(i)}))$$

- What stochastic gradient descent algorithm is doing is scanning through each example
  - The inner for loop does something like this...
    - Looking at example 1, take a step with respect to the cost of just the 1st training example
      - Having done this, we go on to the second training example
    - Now take a second step in parameter space to try and fit the second training example better
      - Now move onto the third training example
    - And so on...
    - Until it gets to the end of the data
  - We may now repeat this who procedure and take multiple passes over the data
- The **randomly shuffling** at the start means we ensure the data is in a random order so we don't bias the movement
  - Randomization should speed up convergence a little bit
- Although stochastic gradient descent is a lot like batch gradient descent, rather than waiting to sum up the gradient terms over all *m* examples, we take just one example and make progress in improving the parameters already
  - Means we update the parameters on EVERY step through data, instead of at the end of each loop through all the data
- What does the algorithm do to the parameters?
  - As we saw, batch gradient descent does something like this to get to a global minimum



  - With stochastic gradient descent every iteration is much faster, but every iteration is flitting a single example



    - What you find is that you "generally" move in the direction of the global minimum, but not

always
        - Never actually converges like batch gradient descent does, but ends up wandering around some region close to the global minimum
            - In practice, this isn't a problem - as long as you're close to the minimum that's probably OK
- One final implementation note
    - May need to loop over the entire dataset 1-10 times
    - If you have a truly massive dataset it's possible that by the time you've taken a single pass through the dataset you may already have a perfectly good hypothesis
        - In which case the inner loop might only need to happen 1 if m is very very large
- If we contrast this to batch gradient descent
    - We have to make k passes through the entire dataset, where k is the number of steps needed to move through the data

Mini Batch Gradient Descent

- **Mini-batch gradient descent** is an additional approach which can work even faster than stochastic gradient descent
- To summarize our approaches so far
    - Batch gradient descent: Use all m examples in each iteration
    - Stochastic gradient descent: Use 1 example in each iteration
    - Mini-batch gradient descent: Use b examples in each iteration
        - b = mini-batch size
- So just like batch, except we use tiny batches
    - Typical range for b = 2-100 (10 maybe)
- For example
    - b = 10
    - Get 10 examples from training set
    - Perform gradient descent update using the ten examples

Mini-batch algorithm

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 $j = 0, \ldots, n$) } }

- We for-loop through b-size batches of m
- Compared to batch gradient descent this allows us to get through data in a much more efficient way
    - After just b examples we begin to improve our parameters
    - Don't have to update parameters after *every* example, and don't have to wait until you cycled through all the data

Mini-batch gradient descent vs. stochastic gradient descent

- Why should we use mini-batch?

    - Allows you to have a vectorized implementation
    - Means implementation is much more efficient
    - Can partially parallelize your computation (i.e. do 10 at once)
- A disadvantage of mini-batch gradient descent is the optimization of the parameter *b*
    - But this is often worth it!

- To be honest, stochastic gradient descent and batch gradient descent are just specific forms of batch-gradient descent
    - For mini-batch gradient descent, b is somewhere in between 1 and m and you can try to optimize for it!

Stochastic gradient descent convergence

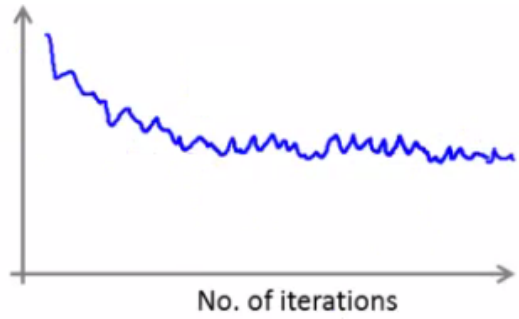
- We now know about stochastic gradient descent
    - But how do you know when it's done!?
    - How do you tune learning rate alpha (α)?

Checking for convergence

- With batch gradient descent, we could plot cost function vs number of iterations
    - Should decrease on every iteration
    - This works when the training set size was small because we could sum over all examples
        - Doesn't work when you have a massive dataset
    - With stochastic gradient descent
        - We don't want to have to pause the algorithm periodically to do a summation over all data
        - Moreover, the whole point of stochastic gradient descent is to *avoid* those whole-data summations
- For stochastic gradient descent, we have to do something different
    - Take cost function definition

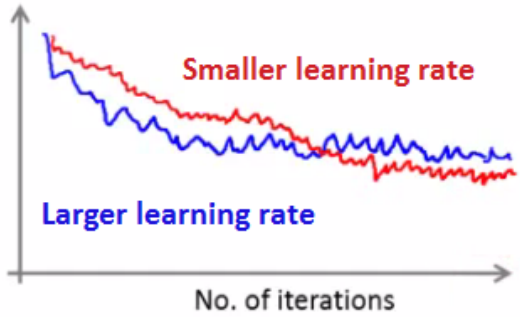
$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

        - One half the squared error on a single example
    - While the algorithm is looking at the example (x^i, y^i), but *before* it has updated θ we can compute the cost of the example (cost(θ, (x^i, y^i))
        - i.e. we compute how well the hypothesis is working on the training example
            - Need to do this before we update θ because if we did it after θ was updated the algorithm would be performing a bit better (because we'd have just used (x^i, y^i) to improve θ)
    - To check for the convergence, every 1000 iterations we can plot the costs averaged over the last 1000 examples
        - Gives a running estimate of how well we've done on the last 1000 estimates
        - By looking at the plots we should be able to check convergence is happening
- What do these plots look like
    - In general
        - Might be a bit noisy (1000 examples isn't that much)
    - If you get a figure like this



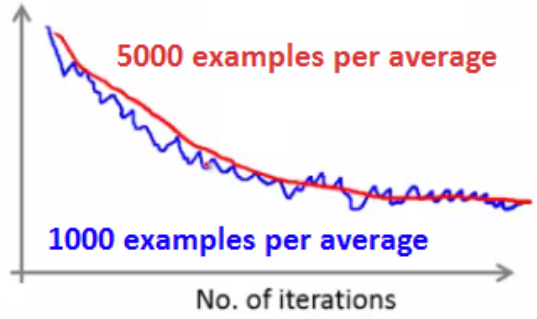
- That's a pretty decent run
- Algorithm may have convergence

- If you use a smaller learning rate you may get an even better final solution



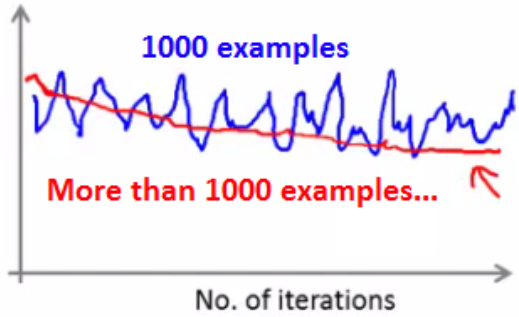
- This is because the parameter oscillate around the global minimum
- A smaller learning rate means smaller oscillations

- If you average over 1000 examples and 5000 examples you may get a smoother curve



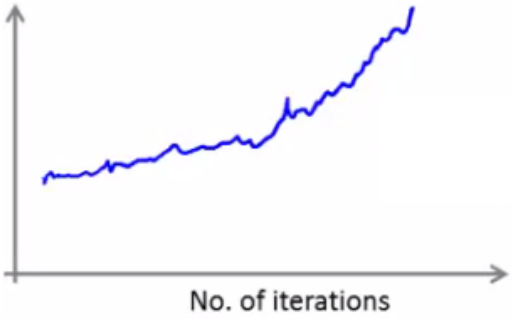
- This disadvantage of a larger average means you get less frequent feedback

- Sometimes you may get a plot that looks like this



- Looks like cost is not decreasing at all
- But if you then increase to averaging over a larger number of examples you do see this general trend
  - Means the blue line was too noisy, and that noise is ironed out by taking a greater number of entires per average
- Of course, it may not decrease, even with a large number

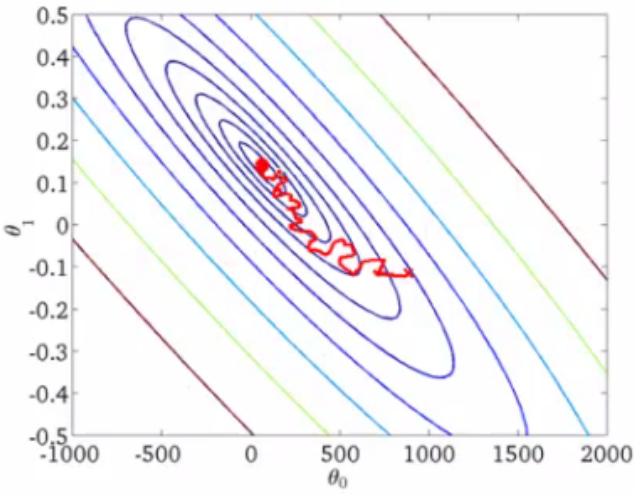
- If you see a curve the looks like its increasing then the algorithm may be displaying divergence



- Should use a smaller learning rate

Learning rate

- We saw that with stochastic gradient descent we get this wandering around the minimum
  - In most implementations the learning rate is held constant
- However, if you want to converge to a minimum you can slowly decrease the learning rate over time
  - A classic way of doing this is to calculate α as follows
    α = const1/(iterationNumber + const2)
  - Which means you're guaranteed to converge somewhere
    - You also need to determine const1 and const2
  - BUT if you tune the parameters well, you can get something like this



Online learning

- New setting
  - Allows us to model problems where you have a continuous stream of data you want an algorithm to learn from
  - Similar idea of stochastic gradient descent, in that you do slow updates
  - Web companies use various types of online learning algorithms to learn from traffic
    - Can (for example) learn about user preferences and hence optimize your website
- Example - Shipping service
  - Users come and tell you origin and destination
  - You offer to ship the package for some amount of money ($10 - $50)
  - Based on the price you offer, sometimes the user uses your service (y = 1), sometimes they don't (y = 0)
  - Build an algorithm to optimize what price we offer to the users
    - Capture
      - Information about user
      - Origin and destination
    - Work out

        - What the probability of a user selecting the service is
      - We want to optimize the price
    - To model this probability we have something like
      - p(y = 1|x; θ)
        - Probability that y =1, given x, parameterized by θ
      - Build this model with something like
        - Logistic regression
        - Neural network
    - If you have a website that runs continuously an online learning algorithm would do something like this
      - User comes - is represented as an (x,y) pair where
        - x - feature vector including price we offer, origin, destination
        - y - if they chose to use our service or not
      - The algorithm updates θ using just the (x,y) pair

      - So we basically update all the θ parameters every time we get some new data
    - While in previous examples we might have described the data example as (x^i, y^i) for an online learning problem we discard this idea of a data "set" - instead we have a continuous stream of data so indexing is largely irrelevant as you're not storing the data (although presumably you could store it)
- If you have a major website where you have a massive stream of data then this kind of algorithm is pretty reasonable
  - You're free of the need to deal with all your training data
- If you had a small number of users you could save their data and then run a normal algorithm on a dataset
- An online algorithm can adapt to changing user preferences
  - So over time users may become more price sensitive
  - The algorithm adapts and learns to this
  - So your system is dynamic

Another example - product search

- Run an online store that sells cellphones
  - You have a UI where the user can type in a query like, "Android phone 1080p camera"
  - We want to offer the user 10 phones per query
- How do we do this
  - For each phone and given a specific user query, we create a feature vector (x) which has data like features of the phone, how many words in the user query match the name of the phone, how many words in user query match description of phone
    - Basically how well does the phone match the user query
  - We want to estimate the probability of a user selecting a phone
  - So define
    - y = 1 if a user clicks on a link
    - y = 0 otherwise
  - So we want to learn
    - p(y = 1|x ; θ) <- this is the problem of learning the predicted **click through rate** (CTR)
  - If you can estimate the CTR for any phone we can use this to show the highest probability phones first
  - If we display 10 phones per search, it means for each search we generate 10 training examples of data
    - i.e. user can click through one or more, or none of them, which defines how well the prediction performed
- Other things you can do
  - Special offers to show the user

    - Show news articles - learn what users like
    - Product recommendation
- These problems could have been formulated using standard techniques, but they are the kinds of problems where you have so much data that this is a better way to do things

Map reduce and data parallelism

- Previously spoke about stochastic gradient descent and other algorithms
    - These could be run on one machine
    - Some problems are just too big for one computer
    - Talk here about a different approach called Map Reduce
- Map reduce example
    - We want to do batch gradient descent

$$\text{Batch gradient descent: } \theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

        - Assume m = 400
            - Normally m would be more like 400 000 000
            - If m is large this is really expensive
    - Split training sets into different subsets
        - So split training set into 4 pieces
    - **Machine 1** : use $(x^1, y^1), \ldots, (x^{100}, y^{100})$
        - Uses first quarter of training set
        - Just does the summation for the first 100

$$temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

$$\text{Machine 2: Use } (x^{(101)}, y^{(101)}), \ldots, (x^{(200)}, y^{(200)}).$$
$$temp_j^{(2)} = \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

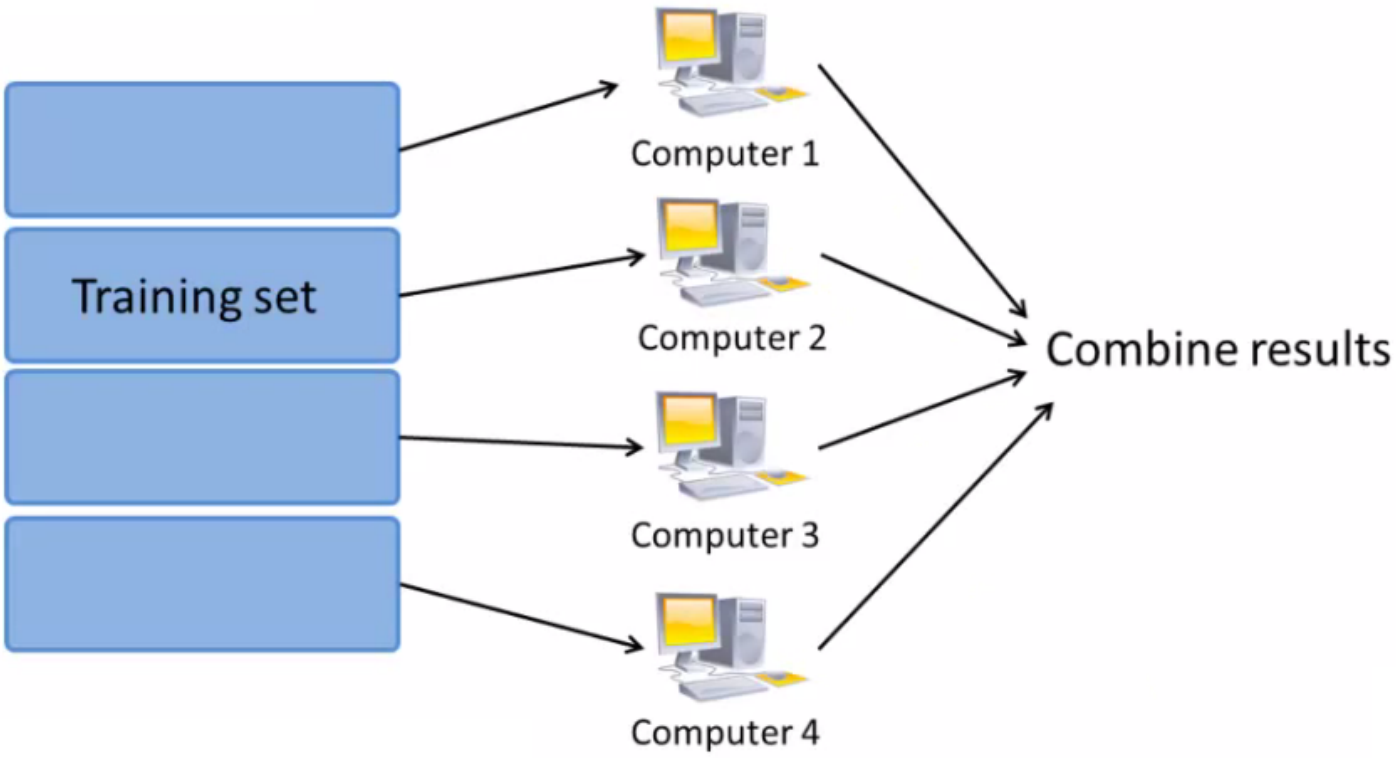
$$\text{Machine 3: Use } (x^{(201)}, y^{(201)}), \ldots, (x^{(300)}, y^{(300)}).$$
$$temp_j^{(3)} = \sum_{i=201}^{300} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

$$\text{Machine 4: Use } (x^{(301)}, y^{(301)}), \ldots, (x^{(400)}, y^{(400)}).$$
$$temp_j^{(4)} = \sum_{i=301}^{400} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

    - So now we have these four temp values, and each machine does 1/4 of the work
    - Once we've got our temp variables
        - Send to to a centralized master server
        - Put them back together
        - Update θ using

$$\theta_j := \theta_j - \alpha \frac{1}{400} \left(temp_j^{(1)} + temp_j^{(2)} + temp_j^{(3)} + temp_j^{(4)} \right)$$
$$(j = 0, \ldots, n)$$

            - This equation is doing the same as our original batch gradient descent algorithm
- More generally map reduce uses the following scheme (e.g. where you split into 4)



- The bulk of the work in gradient descent is the summation
  - Now, because each of the computers does a quarter of the work at the same time, you get a 4x speedup
  - Of course, in practice, because of network latency, combining results, it's slightly less than 4x, but still good!
- Important thing to ask is
  - "Can algorithm be expressed as computing sums of functions of the training set?"
    - Many algorithms can!
- Another example
  - Using an advanced optimization algorithm with logistic regression
  - $$J_{train}(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) - (1-y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$
    - Need to calculate cost function - see we sum over training set
    - So split training set into x machines, have x machines compute the sum of the value over 1/xth of the data
  - $$\frac{\partial}{\partial \theta_j} J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$
    - These terms are also a sum over the training set
    - So use same approach
- So with these results send temps to central server to deal with combining everything

- More broadly, by taking algorithms which compute sums you can scale them to very large datasets through parallelization
  - Parallelization can come from
    - Multiple machines
    - Multiple CPUs
    - Multiple cores in each CPU

    - So even on a single compute can implement parallelization
- The advantage of thinking about Map Reduce here is because you don't need to worry about network issues
    - It's all internal to the same machine
- Finally caveat/thought
    - Depending on implementation detail, certain numerical linear algebra libraries can automatically parallelize your calculations across multiple cores
    - So, if this is the case and you have a good vectorization implementation you can not worry about local Parallelization and the local libraries sort optimization out for you

Hadoop

- Hadoop is a good open source Map Reduce implementation
- Represents a top-level Apache project develop by a global community of developers
    - Large developer community all over the world
- Written in Java
- Yahoo has been the biggest contributor
    - Pushed a lot early on
- Support now from Cloudera

Interview with Cloudera CEO Mike Olson (2010)

- Seeing a change in big data industry (Twitter, Facebook etc) - relational databases can't scale to the volumes of data being generated
    - ***Q: Where the tech came from?***
        - Early 2000s - Google had too much data to process (and index)
            - Designed and built Map Reduce
                - Buy and mount a load of rack servers
                - Spread the data out among these servers (with some duplication)
                - Now you've stored the data and you have this processing infrastructure spread among the data
                    - Use local CPU to look at local data
                    - Massive data parallelism
                - Published as a paper in 2004
                    - At the time wasn't obvious why it was necessary - didn't support queries, transactions, SQL etc
            - When data was at "human" scale relational databases centralized around a single server was fine
            - But now we're scaling by Moore's law in two ways
                - More data
                - Cheaper to store
- **Q: How do people approach the issues in big data?**
    - People still use relational databases - great if you have predictable queries over structured data
        - Data warehousing still used - long term market
    - But the data people want to work with is becoming more complex and bigger
        - Free text, unstructured data doesn't fit will into tables
        - Do sentiment analysis in SQL isn't really that good
        - So to do new kinds of processing need a new type of architecture
    - Hadoop lets you do *data* processing - not transactional processing - on the big scale
    - Increasingly things like NoSQL is being used
    - Data centers are starting to chose technology which is aimed at a specific problem, rather than trying to shoehorn problems into an ER issue
    - Open source technologies are taking over for developer facing infrastructures and platforms
- **Q: What is Hadoop?**
    - Open source implementation of Map reduce (Apache software)

    - Yahoo invested a lot early on - developed a lot the early progress
    - Is two things
        - **HDFS**
            - Disk on ever server
            - Software infrastructure to spread data
        - **Map reduce**
            - Lets you push code down to the data in parallel
    - As size increases you can just add more servers to scale up
- **Q: What is memcached?**
    - Ubiquitous invisible infrastructure that makes the web run
    - You go to a website, see data being delivered out of a MySQL database
        - BUT, when infrastructure needs to scale querying a disk EVERY time is too much
    - Memcache is a memory layer between disk and web server
        - Cache reads
        - Push writes through incrementally
        - Is the glue that connects a website with a disk-backend
    - Northscale is commercializing this technology
    - New data delivery infrastructure which has pretty wide adoption
- **Q: What is Django?**
    - Open source tool/language
- **Q: What are some of the tool sets being used in data management? What is MySQL drizzle?**
    - Drizzle is a re-implementation of MySQL
        - Team developing Drizzle feels they learned a lot of lessons when building MySQL
        - More modern architecture better targeted at web applications
    - NoSQL
        - Distributed hash tables
        - Idea is instead of a SQL query and fetching a record, go look up something from a store by name
            - Go pull a record by name from a store
        - So now systems being developed to support that like
            - MongoDB
            - CouchDB
    - Hadoop companion projects
        - Hive
            - Lets you use SQL to talk to a Hadoop cluster
        - HBase
            - Sits on top of HDFS
            - Gives you key-value storage on top of HDFS - provides abstraction from distributed system
    - Good time to be working in big data
        - Easy to set up small cluster through cloud system
        - Get a virtual cluster through Rackspace or Cloudera

18: Application Example OCR

Previous Next Index

Problem description and pipeline

- Case study focused around photo OCR
- Three reasons to do this
  - 1) Look at how a **complex system** can be put together
  - 2) The idea of a machine learning **pipeline**
    - What to do next
    - How to do it
  - 3) Some more interesting ideas
    - Applying machine learning to tangible problems
    - **Artificial data synthesis**

What is the photo OCR problem?

- Photo OCR = photo optical character recognition
  - With growth of digital photography, lots of digital pictures
  - One idea which has interested many people is getting computers to understand those photos
  - The photo OCR problem is getting computers to read text in an image
    - Possible applications for this would include
      - Make searching easier (e.g. searching for photos based on words in them)
      - Car navigation
- OCR of documents is a comparatively easy problem
  - From photos it's really hard

OCR pipeline

- 1) Look through image and find text
- 2) Do character segmentation
- 3) Do character classification
- 4) *Optional* some may do spell check after this too
  - We're not focussing on such systems though

- **Pipelines** are common in machine learning
  - Separate modules which may each be a machine learning component or data processing component
- If you're designing a machine learning system, pipeline design is one of the most important questions
  - Performance of pipeline and each module often has a big impact on the overall performance a problem
  - You would often have different engineers working on each module
    - Offers a natural way to divide up the workload

Sliding window image analysis

- How do the individual models work?
- Here focus on a sliding windows classifier

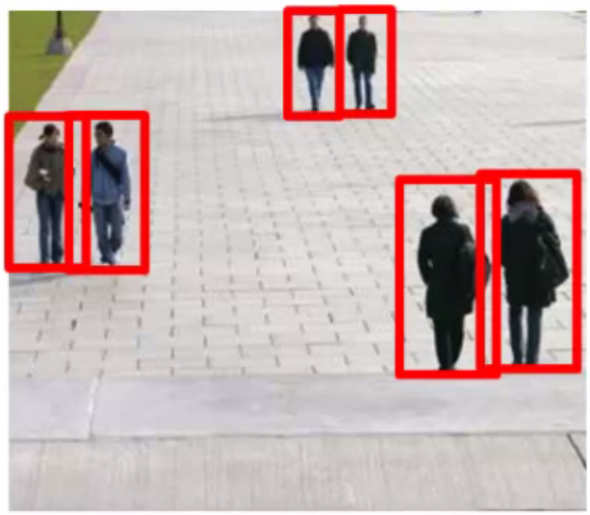
- As mentioned, stage 1 is **text detection**
  - Unusual problem in computer vision - different rectangles (which surround text) may have different aspect ratios (aspect ratio being height : width)
    - Text may be short (few words) or long (many words)
    - Tall or short font
    - Text might be straight on

    - Slanted

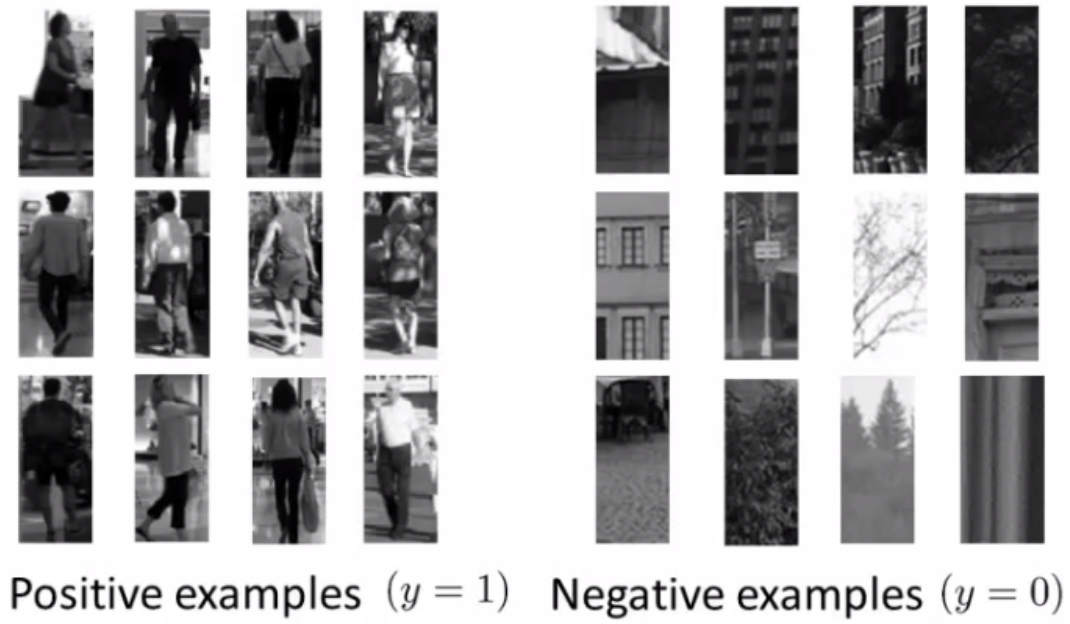
  - Let's start with a simpler example

Pedestrian detection

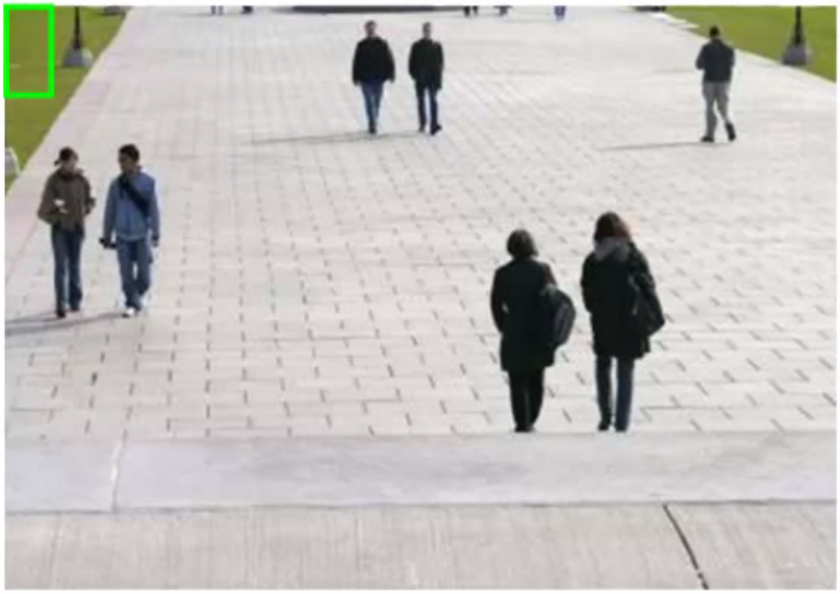
- Want to take an image and find pedestrians in the image



- This is a slightly simpler problem because the aspect ration remains pretty constant
- Building our detection system
  - Have 82 x 36 aspect ratio
    - This is a typical aspect ratio for a standing human
  - Collect training set of positive and negative examples

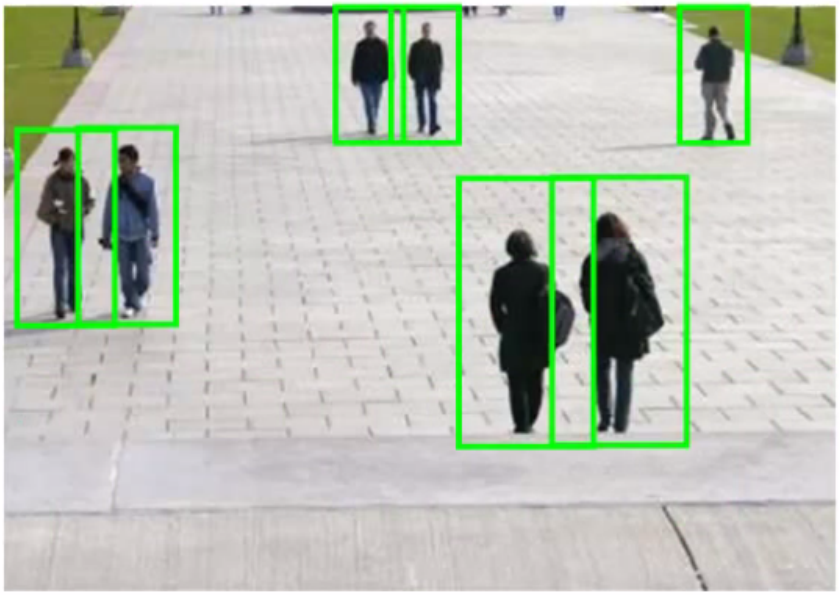


    - Could have 1000 - 10 000 training examples
    - Train a neural network to take an image and classify that image as pedestrian or not
        - Gives you a way to train your system
- Now we have a new image - how do we find pedestrians in it?
    - Start by taking a rectangular 82 x 36 patch in the image



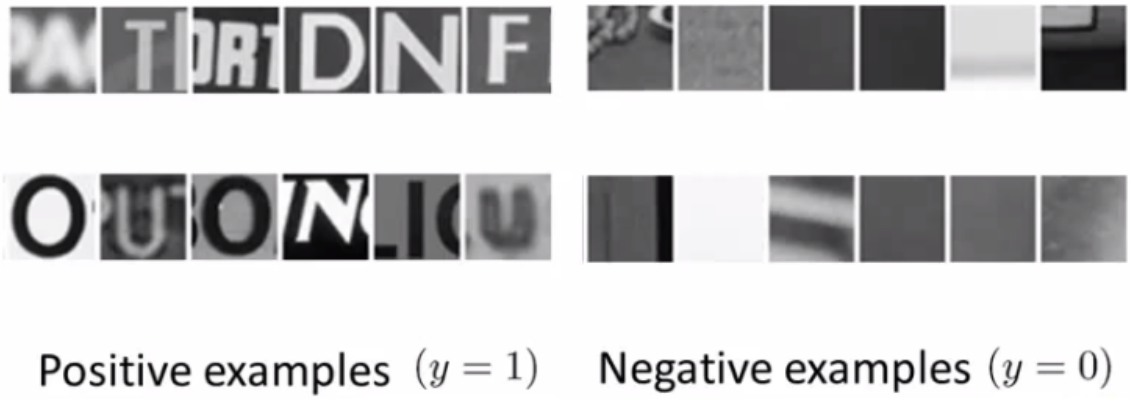
        - Run patch through classifier - hopefully in this example it will return y = 0
    - Next slide the rectangle over to the right a little bit and re-run
        - Then slide again
        - The amount you slide each rectangle over is a parameter called the step-size or stride
            - Could use 1 pixel
                - Best, but computationally expensive
            - More commonly 5-8 pixels used
        - So, keep stepping rectangle along all the way to the right
            - Eventually get to the end
        - Then move back to the left hand side but step down a bit too
        - Repeat until you've covered the whole image
    - Now, we initially started with quite a small rectangle
        - So now we can take a larger image patch (of the same aspect ratio)
        - Each time we process the image patch, we're resizing the larger patch to a smaller image, then running that smaller image through the classifier
    - Hopefully, by changing the patch size and rastering repeatedly across the image, you eventually recognize all

the pedestrians in the picture

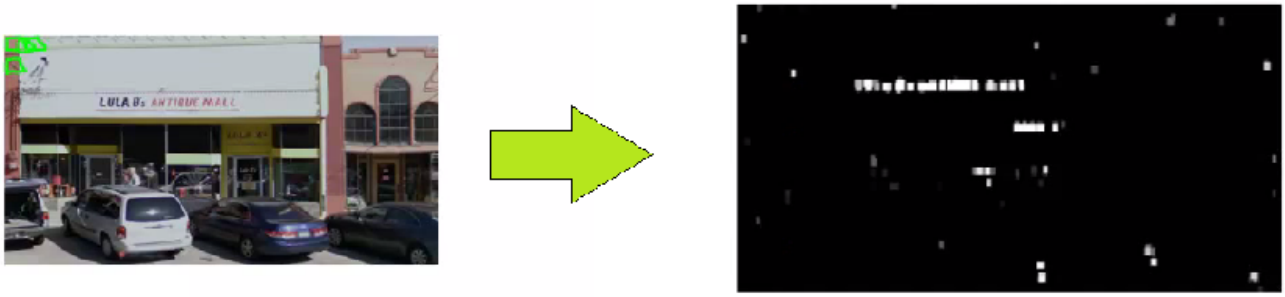


Text detection example

- Like pedestrian detection, we generate a labeled training set with
  - Positive examples (some kind of text)
  - Negative examples (not text)



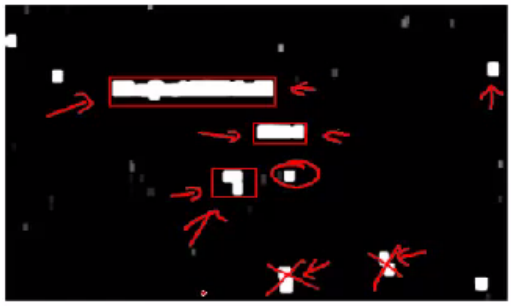
- Having trained the classifier we apply it to an image
  - So, run a sliding window classifier at a fixed rectangle size
  - If you do that end up with something like this



  - White region show where text detection system thinks text is
    - Different shades of gray correspond to probability associated with how sure the classifier is the section contains text
      - Black - no text
      - White - text
    - For text detection, we want to draw rectangles around all the regions where there is text in the image
  - Take classifier output and apply an **expansion algorithm**

        - Takes each of white regions and expands it
        - How do we implement this
            - Say, for every pixel, is it within some distance of a white pixel?
            - If yes then colour it white

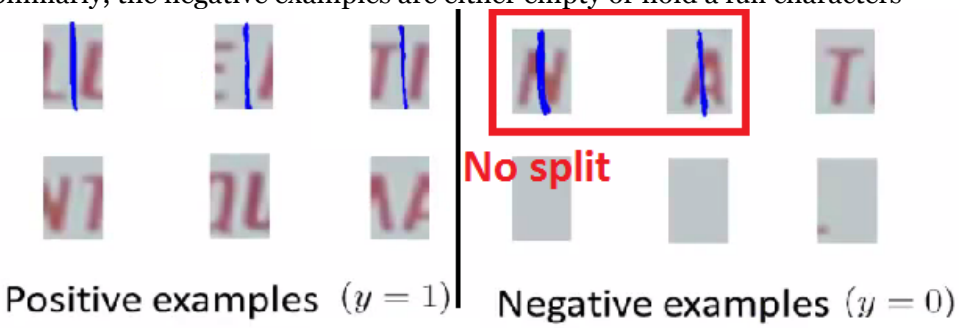
    - Look at connected white regions in the image above
        - Draw rectangles around those which make sense as text (i.e. tall thin boxes don't make sense)



    - This example misses a piece of text on the door because the aspect ratio is wrong
        - Very hard to read

Stage two is character segmentation

- Use supervised learning algorithm
- Look in a defined image patch and decide, is there a split between two characters?
    - So, for example, our first training data item below looks like there is such a split
    - Similarly, the negative examples are either empty or hold a full characters



- We train a classifier to try and classify between positive and negative examples
    - Run that classifier on the regions detected as containing text in the previous section
- Use a 1-dimensional sliding window to move along text regions
    - Does each window snapshot look like the split between two characters?
        - If yes insert a split
        - If not move on
    - So we have something that looks like this

Character classification

- Standard OCR, where you apply standard supervised learning which takes an input and identify which character we decide it is
  - Multi-class characterization problem

Getting lots of data: Artificial data synthesis

- We've seen over and over that one of the most reliable ways to get a high performance machine learning system is to take a low bias algorithm and train on a massive data set
  - Where do we get so much data from
  - In ML artifice data synthesis
    - Doesn't apply to every problem
    - If it applies to your problem can be a great way to generate loads of data
- Two main principles
  - 1) Creating data from scratch
  - 2) If we already have a small labeled training set can we amplify it into a larger training set

Character recognition as an example of data synthesis

- If we go and collect a large labeled data set will look like this
  - Goal is to take an image patch and have the system recognize the character
  - Treat the images as gray-scale (makes it a bit easer)

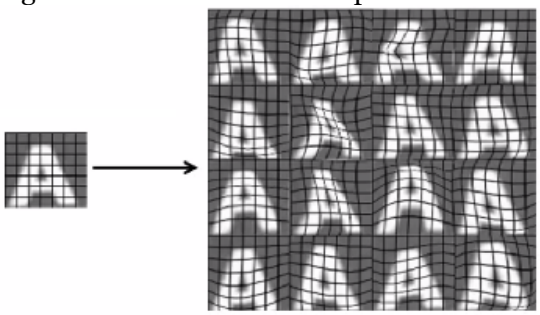
Real data

- How can we amplify this
  - Modern computers often have a big font library
  - If you go to websites, huge free font libraries
  - For more training data, take characters from different fonts, paste these characters again random backgrounds
- After some work, can build a synthetic training set

Synthetic data

- Random background
- Maybe some blurring/distortion filters
- Takes thought and work to make it look realistic
  - If you do a sloppy job this won't help!
  - So unlimited supply of training examples
- This is an example of creating new data from scratch

- Other way is to introduce distortion into existing data
  - e.g. take a character and warp it



  - 16 new examples
  - Allows you amplify existing training set
  - This, again, takes though and insight in terms of deciding how to amplify

Another example: speech recognition

- Learn from audio clip - what were the words
  - Have a labeled training example
  - Introduce audio distortions into the examples
- So only took one example
  - Created lots of new ones!
- When introducing distortion, they should be reasonable relative to the issues your classifier may encounter

Getting more data

- Before creating new data, make sure you have a low bias classifier
  - Plot learning curve
- If not a low bias classifier increase number of features
  - Then create large artificial training set
- Very important question: How much work would it be to get 10x data as we currently have?
  - Often the answer is, "Not that hard"
  - This is often a huge way to improve an algorithm
  - Good question to ask yourself or ask the team
- How many minutes/hours does it take to get a certain number of examples
  - Say we have 1000 examples
  - 10 seconds to label an example
  - So we need another 9000 - 90000 seconds
  - Comes to a few days (25 hours!)

- Crowd sourcing is also a good way to get data
  - Risk or reliability issues
  - Cost
  - Example
    - E.g. Amazon mechanical turks

Ceiling analysis: What part of the pipeline to work on next

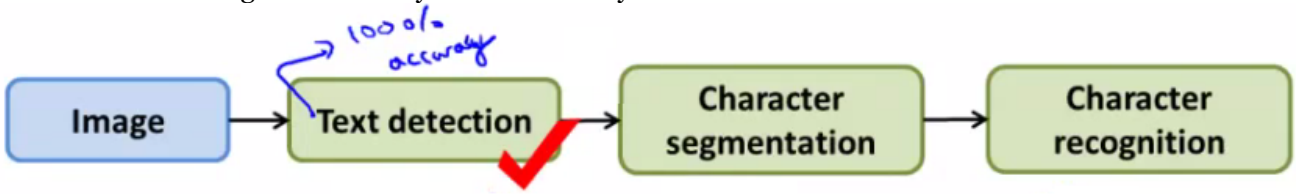
- Through the course repeatedly said one of the most valuable resources is developer time
  - Pick the right thing for you and your team to work on
  - Avoid spending a lot of time to realize the work was pointless in terms of enhancing performance

Photo OCR pipeline

- Three modules
  - Each one could have a small team on it
  - Where should you allocate resources?
- Good to have a single real number as an evaluation metric
  - So, character accuracy for this example
  - Find that our test set has 72% accuracy

Ceiling analysis on our pipeline

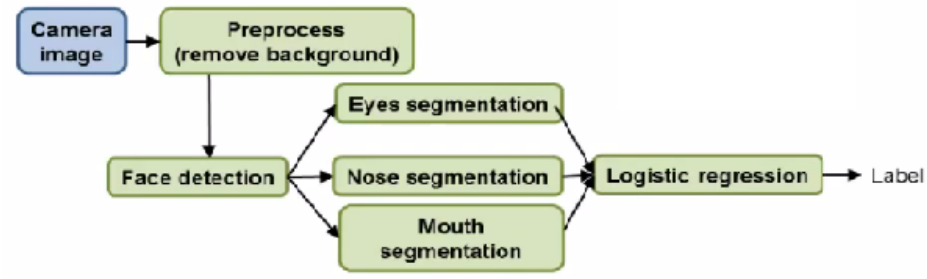
- We go to the first module
  - Mess around with the test set - manually tell the algorithm where the text is
  - Simulate if your text detection system was 100% accurate
    - So we're feeding the character segmentation module with 100% accurate data now
  - How does this change the accuracy of the overall system



  - Accuracy goes up to 89%
- Next do the same for the character segmentation
  - Accuracy goes up to 90% now
- Finally doe the same for character recognition
  - Goes up to 100%
- Having done this we can qualitatively show what the upside to improving each module would be
  - Perfect text detection improves accuracy by 17%!
    - Would bring the biggest gain if we could improve
  - Perfect character segmentation would improve it by 1%
    - Not worth working on
  - Perfect character recognition would improve it by 10%
    - Might be worth working on, depends if it looks easy or not
- The "ceiling" is that each module has a ceiling by which making it perfect would improve the system overall

Other example - face recognition

- NB this is not how it's done in practice



    - Probably more complicated than is used in practice
- How would you do ceiling analysis for this
    - Overall system is 85%
    - Perfect background -> 85.1%
        - Not a crucial step
    - + Perfect face detection -> 91%
        - Most important module to focus on
    - + Perfect eyes ->95%
    - + Perfect Nose -> 96%
    - + Perfect Mouth -> 97%
    - + Perfect logistic regression -> 100%
- Cautionary tale
    - Two engineers spent 18 months improving background pre-processing
        - Turns out had no impact on overall performance
        - Could have saved three years of man power if they'd done ceiling analysis