10: Advice for applying Machine Learning

Previous Next Index

Deciding what to try next

- We now know many techniques
    - But, there is a big difference between someone who knows an algorithm vs. someone less familiar and doesn't understand how to apply them
    - Make sure you know how to chose the best avenues to explore the various techniques
    - Here we focus deciding what avenues to try

Debugging a learning algorithm

- So, say you've implemented regularized linear regression to predict housing prices

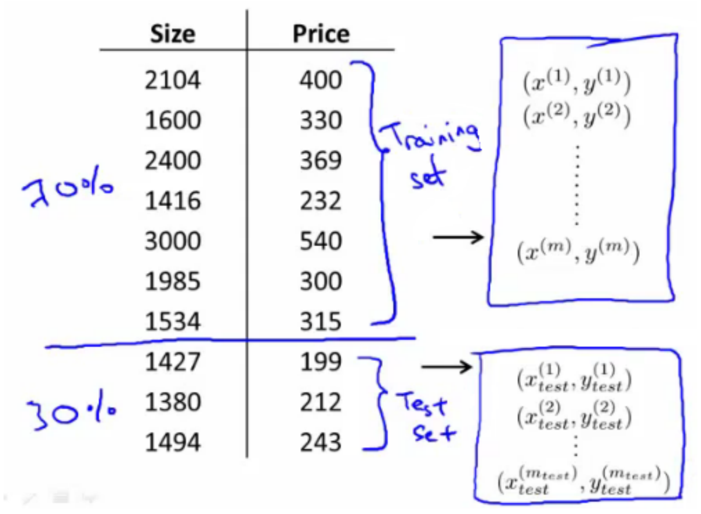
$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{m}\theta_j^2\right]$$

-
    - Trained it
    - But, when you test on new data you find it makes unacceptably large errors in its predictions
    - :-(
- What should you try next?
    - There are many things you can do;
        - **Get more training data**
            - Sometimes more data doesn't help
            - Often it does though, although you should always do some preliminary testing to make sure more data will actually make a difference (discussed later)
        - **Try a smaller set a features**
            - Carefully select small subset
            - You can do this by hand, or use some dimensionality reduction technique (e.g. PCA - we'll get to this later)
        - **Try getting additional features**
            - Sometimes this isn't helpful
            - LOOK at the data
            - Can be very time consuming
        - **Adding polynomial features**
            - You're grasping at straws, aren't you...
        - **Building your own, new, better features** based on your knowledge of the problem
            - Can be risky if you accidentally over fit your data by creating new features which are inherently specific/relevant to your training data
        - **Try decreasing or increasing λ**
            - Change how important the regularization term is in your calculations
    - These changes can become MAJOR projects/headaches (6 months +)
        - Sadly, most common method for choosing one of these examples is to go by gut feeling (randomly)
        - Many times, see people spend huge amounts of time only to discover that the avenue is fruitless
            - No apples, pears, or any other fruit. Nada.
    - There are some simple techniques which can let you rule out half the things on the list
        - Save you a lot of time!
- **Machine learning diagnostics**
    - Tests you can run to see what is/what isn't working for an algorithm
    - See what you can change to improve an algorithm's performance
    - These can take time to implement and understand (week)
        - But, they can also save you spending months going down an avenue which will *never* work

Evaluating a hypothesis

- When we fit parameters to training data, try and minimize the error
    - We might think a low error is good - doesn't necessarily mean a good parameter set
        - Could, in fact, be indicative of overfitting
        - This means you model will fail to generalize
    - How do you tell if a hypothesis is overfitting?
        - Could plot $h_\theta(x)$
        - But with lots of features may be impossible to plot
- Standard way to evaluate a hypothesis is
    - Split data into two portions

        - 1st portion is **training set**
        - 2nd portion is **test set**
    - Typical split might be 70:30 (training:test)



Size	Price
2104	400
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199
1380	212
1494	243

70%
Training set
$(x^{(1)}, y^{(1)})$
$(x^{(2)}, y^{(2)})$
$(x^{(m)}, y^{(m)})$
30%
Test set
$(x^{(1)}_{test}, y^{(1)}_{test})$
$(x^{(2)}_{test}, y^{(2)}_{test})$
$(x^{(m_{test})}_{test}, y^{(m_{test})}_{test})$

- NB if data is ordered, send a random percentage
    - (Or randomly order, then send data)
    - Data is typically ordered in some way anyway
- So a typical **train and test scheme** would be
    - 1) Learn parameters θ from training data, minimizing J(θ) using 70% of the training data]
    - 2) Compute the test error
        - $J_{test}(θ)$ = average square error as measured on the test set

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

    - This is the definition of the **test set error**
- What about if we were using logistic regression
    - The same, learn using 70% of the data, test with the remaining 30%

$$J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} y^{(i)}_{test} \log h_\theta(x^{(i)}_{test}) + (1 - y^{(i)}_{test}) \log h_\theta(x^{(i)}_{test})$$

    - Sometimes there a better way - misclassification error (0/1 misclassification)
        - We define the error as follows

$$err(h_\theta(x), y) = \begin{cases} 1 & \text{if } h_\theta(x) \geq 0.5,\ y = 0 \\ & \text{or if } h_\theta(x) < 0.5,\ y = 1 \\ 0 & \text{otherwise} \end{cases} \Big\} \text{ error}$$

        - Then the test error is

$$\text{Test error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\theta(x^{(i)}_{test}), y^{(i)}_{test})$$

            - i.e. its the fraction in the test set the hypothesis mislabels
- These are the standard techniques for evaluating a learned hypothesis

Model selection and training validation test sets

- How to chose regularization parameter or degree of polynomial (**model selection problems**)
- We've already seen the problem of overfitting
    - More generally, this is why training set error is a poor predictor of hypothesis accuracy for new data (generalization)
- Model selection problem
    - Try to chose the degree for a polynomial to fit data

1. $h_\theta(x) = \theta_0 + \theta_1 x$
2. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$
3. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_3 x^3$

$\vdots$

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

- d = what degree of polynomial do you want to pick
  - An additional parameter to try and determine your training set
    - d =1 (linear)
    - d=2 (quadratic)
    - ...
    - d=10
  - Chose a model, fit that model and get an estimate of how well you hypothesis will generalize
- You could
  - Take model 1, minimize with training data which generates a parameter vector θ^1 (where d =1)
  - Take mode 2, do the same, get a *different* θ^2 (where d = 2)
  - And so on
  - Take these parameters and look at the test set error for each using the previous formula
    - $J_{test}(\theta^1)$
    - $J_{test}(\theta^2)$
    - ...
    - $J_{test}(\theta^{10})$
- You could then
  - See which model has the lowest test set error
- Say, for example, d=5 is the lowest
  - Now take the d=5 model and say, how well does it generalize?
    - You could use $J_{test}(\theta^5)$
    - BUT, this is going to be an optimistic estimate of generalization error, because our parameter is fit to that test set (i.e. specifically chose it because the test set error is small)
    - So not a good way to evaluate if it will generalize
- To address this problem, we do something a bit different for model selection

* Improved model selection
  - Given a training set instead split into three pieces
    - 1 - **Training set** (60%) - m values
    - 2 - **Cross validation** (CV) set (20%)m_{cv}
    - 3 - **Test set** (20%) m_{test}
  - As before, we can calculate
    - Training error
    - Cross validation error
    - Test error

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

Cross Validation error: $$J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\theta(x^{(i)}_{cv}) - y^{(i)}_{cv})^2$$

Test error: $$J_{test}(\theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\theta(x^{(i)}_{test}) - y^{(i)}_{test})^2$$

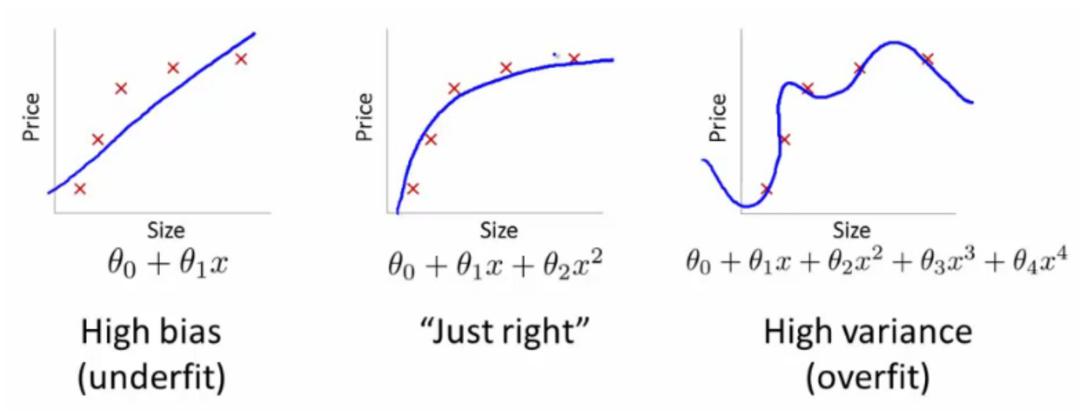
  - So
    - Minimize cost function for each of the models as before
    - Test these hypothesis on the cross validation set to generate the cross validation error
    - Pick the hypothesis with the lowest cross validation error
      - e.g. pick θ^5
    - Finally
      - Estimate generalization error of model using the test set

* Final note
  - In machine learning as practiced today - many people will select the model using the test set and then check the model is

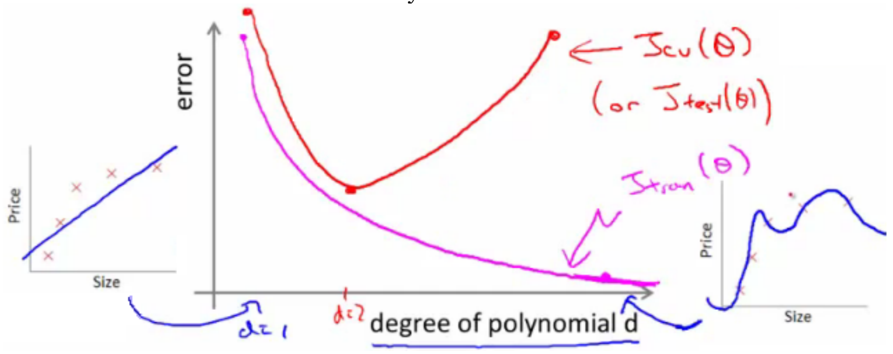
OK for generalization using the test error (which we've said is bad because it gives a bias analysis)
        - With a MASSIVE test set this is maybe OK
    - But considered much better practice to have separate training and validation sets

Diagnosis - bias vs. variance

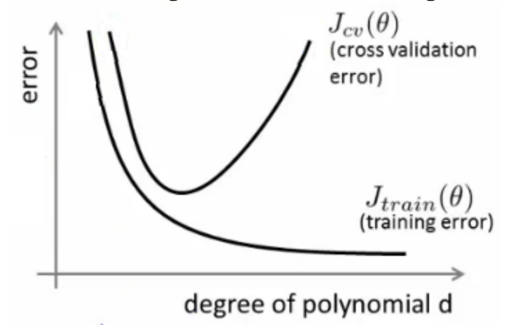
- If you get bad results usually because of one of
    - **High bias** - under fitting problem
    - **High variance** - over fitting problem
- Important to work out which is the problem
    - Knowing which will help let you improve the algorithm
- Bias/variance shown graphically below



- The degree of a model will increase as you move towards overfitting
- Lets define training and cross validation error as before
- Now plot
    - x = degree of polynomial d
    - y = error for both training and cross validation (two lines)
        - CV error and test set error will be very similar



        - This plot helps us understand the error
    - We want to minimize both errors
        - Which is why that d=2 model is the sweet spot
- How do we apply this for diagnostics
    - If cv error is high we're either at the high or the low end of d



    - if d is too small --> this probably corresponds to a high bias problem
    - if d is too large --> this probably corresponds to a high variance problem
- **For the high bias case, we find both cross validation and training error are high**
    - Doesn't fit training data well
    - Doesn't generalize either

- **For high variance, we find the cross validation error is high but training error is low**
  - So we suffer from overfitting (training is low, cross validation is high)
  - i.e. training set fits well
  - But generalizes poorly

Regularization and bias/variance

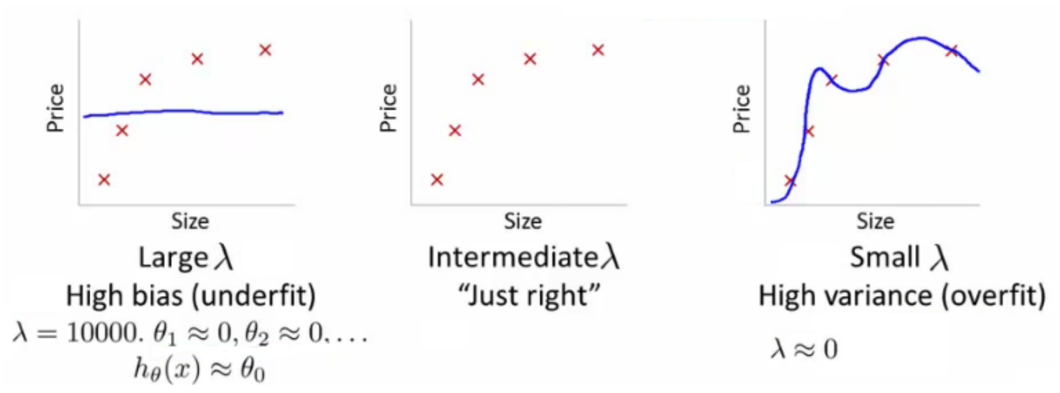
- How is bias and variance effected by regularization?

Linear regression with regularization

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

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

- The equation above describes fitting a high order polynomial with regularization (used to keep parameter values small)
  - Consider three cases
    - **λ = large**
      - All θ values are heavily penalized
      - So most parameters end up being close to zero
      - So hypothesis ends up being close to 0
      - So **high bias -> under fitting data**
    - **λ = intermediate**
      - Only this values gives the fitting which is reasonable
    - **λ = small**
      - Lambda = 0
      - So we make the regularization term 0
      - So **high variance -> Get overfitting** (minimal regularization means it obviously doesn't do what it's meant to)



- How can we automatically chose a good value for λ?
  - To do this we define another function $J_{train}(\theta)$ which is the optimization function *without* the regularization term (average squared errors)

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

  - Define cross validation error and test set errors as before (i.e. without regularization term)
    - So they are 1/2 average squared error of various sets

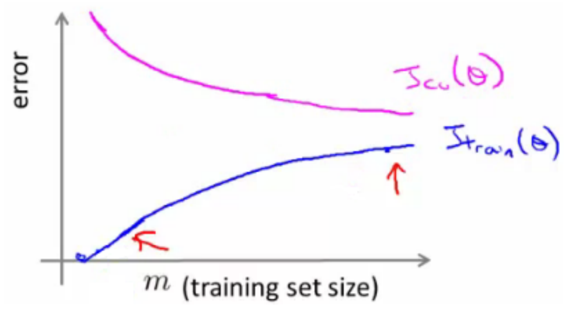
- **Choosing λ**
  - Have a set or range of values to use
  - Often increment by factors of 2 so
    - model(1)= λ = 0
    - model(2)= λ = 0.01
    - model(3)= λ = 0.02
    - model(4) = λ = 0.04
    - model(5) = λ = 0.08

      .
      .
      .

        - model(p) = λ = 10
    - This gives a number of models which have different λ
    - With these models
        - Take each one (pth)
        - Minimize the cost function
        - This will generate some parameter vector
            - Call this $\theta^{(p)}$
        - So now we have a set of parameter vectors corresponding to models with different λ values
    - Take all of the hypothesis and use the cross validation set to validate them
        - Measure average squared error on cross validation set
        - Pick the model which gives the lowest error
        - Say we pick $\theta^{(5)}$
    - Finally, take the one we've selected ($\theta^{(5)}$) and test it with the test set
- **Bias/variance as a function of λ**
    - Plot λ vs.
        - J_{train}
            - When λ is small you get a small value (regularization basically goes to 0)
            - When λ is large you get a large vale corresponding to high bias
        - J_{cv}
            - When λ is small we see high variance
                - Too small a value means we over fit the data
            - When λ is large we end up underfitting, so this is bias
                - So cross validation error is high
    - Such a plot can help show you you're picking a good value for λ

Learning curves

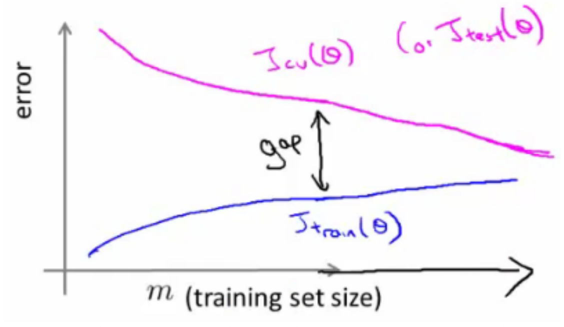
- A learning curve is often useful to plot for algorithmic sanity checking or improving performance
- What is a learning curve?
    - Plot J_{train} (average squared error on training set) or J_{cv} (average squared error on cross validation set)
    - Plot against m (number of training examples)
        - m is a constant
        - So artificially reduce m and recalculate errors with the smaller training set sizes
    - J_{train}
        - Error on smaller sample sizes is smaller (as less variance to accommodate)
        - So as m grows error grows
    - J_{cv}
        - Error on cross validation set
        - When you have a tiny training set your generalize badly
        - But as training set grows your hypothesis generalize better
        - So cv error will decrease as m increases



- What do these curves look like if you have
    - **High bias**
        - e.g. setting straight line to data
        - J_{train}
            - Training error is small at first and grows
            - Training error becomes close to cross validation
            - So the performance of the cross validation and training set end up being similar (but very poor)
        - J_{cv}
            - Straight line fit is similar for a few vs. a lot of data
            - So it doesn't generalize any better with lots of data because the function just doesn't fit the data
                - No increase in data will help it fit
        - The problem with high bias is because cross validation and training error are both high
        - Also implies that if a learning algorithm as high bias as we get more examples the cross validation

error doesn't decrease
- **So if an algorithm is already suffering from high bias, more data does not help**
- So knowing if you're suffering from high bias is good!
- In other words, high bias is a problem with the underlying way you're modeling your data
    - So more data won't improve that model
    - It's too simplistic

- **High variance**
    - e.g. high order polynomial
    - J_{train}
        - When set is small, training error is small too
        - As training set sizes increases, value is still small
        - But slowly increases (in a near linear fashion)
        - Error is still low
    - J_{cv}
        - Error remains high, even when you have a moderate number of examples
        - Because the problem with high variance (overfitting) is your model doesn't generalize
    - An indicative diagnostic that you have high variance is that there's a big gap between training error and cross validation error
    - If a learning algorithm is suffering from high variance, more data is probably going to help



- **So if an algorithm is already suffering from high variance, more data will probably help**
    - Maybe
- These are clean curves
- In reality the curves you get are far dirtier
- But, learning curve plotting can help diagnose the problems your algorithm will be suffering from

What to do next (revisited)

- How do these ideas help us chose how we approach a problem?
    - Original example
        - Trained a learning algorithm (regularized linear regression)
        - But, when you test on new data you find it makes unacceptably large errors in its predictions
        - What should try next?
    - How do we decide what to do?
        - **Get more examples** --> helps to fix high variance
            - Not good if you have high bias

        - **Smaller set of features** --> fixes high variance (overfitting)
            - Not good if you have high bias

        - **Try adding additional features** --> fixes high bias (because hypothesis is too simple, make hypothesis more specific)

        - **Add polynomial terms** --> fixes high bias problem

        - **Decreasing λ** --> fixes high bias

        - **Increases λ** --> fixes high variance

- Relating it all back to neural networks - selecting a network architecture
    - One option is to use a small neural network
        - Few (maybe one) hidden layer and few hidden units
        - Such networks are prone to under fitting
        - But they are computationally cheaper
    - Larger network
        - More hidden layers
            - How do you decide that a larger network is good?

- Using a single hidden layer is good default
  - Also try with 1, 2, 3, see which performs best on cross validation set
  - So like before, take three sets (training, cross validation)
- More units
  - This is computational expensive
  - Prone to over-fitting
    - Use regularization to address over fitting

11: Machine Learning System Design

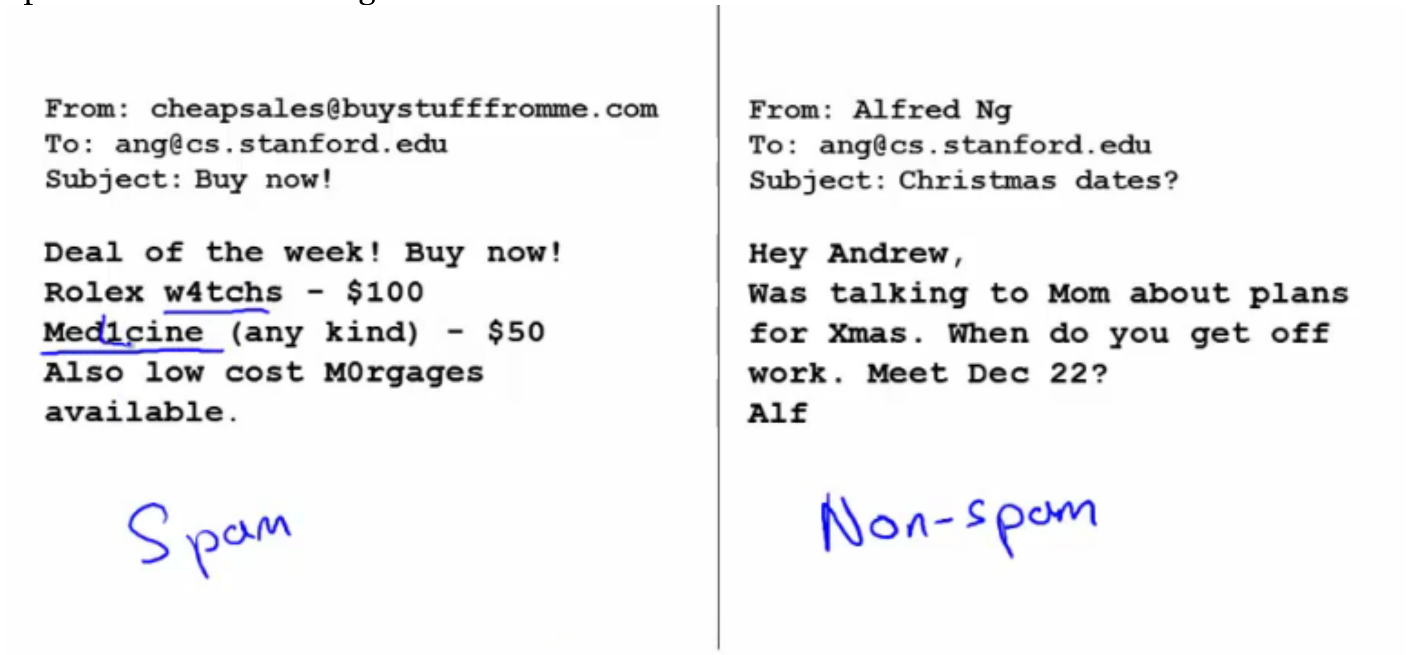
Previous Next Index

Machine learning systems design

- In this section we'll touch on how to put together a system
- Previous sections have looked at a wide range of different issues in significant focus
- This section is less mathematical, but material will be very useful non-the-less
  - Consider the system approach
  - You can understand all the algorithms, but if you don't understand how to make them work in a complete system that's no good!

Prioritizing what to work on - spam classification example

- The idea of prioritizing what to work on is perhaps the most important skill programmers typically need to develop
  - It's so easy to have many ideas you want to work on, and as a result do none of them well, because doing one well is harder than doing six superficially
    - So you need to make sure you complete projects
    - Get something "shipped" - even if it doesn't have all the bells and whistles, that final 20% getting it ready is often the toughest
    - If you only release when you're totally happy you rarely get practice doing that final 20%
  - So, back to machine learning...
- Building a spam classifier
- Spam is email advertising


```
From: cheapsales@buystufffromme.com
To: ang@cs.stanford.edu
Subject: Buy now!

Deal of the week! Buy now!
Rolex w4tchs - $100
Med1cine (any kind) - $50
Also low cost M0rgages
available.
```
Spam

```
From: Alfred Ng
To: ang@cs.stanford.edu
Subject: Christmas dates?

Hey Andrew,
Was talking to Mom about plans
for Xmas. When do you get off
work. Meet Dec 22?
Alf
```
Non-spam

- What kind of features might we define
  - Spam (1)
    - Misspelled word
  - Not spam (0)

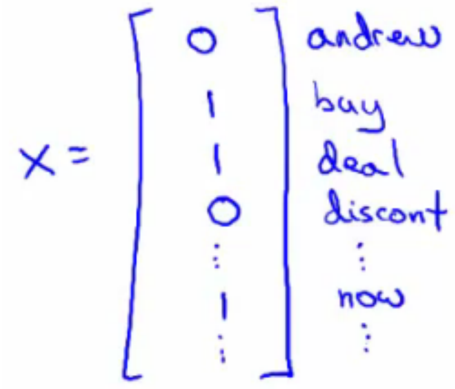
        - Real content
- How do we build a classifier to distinguish between the two
    - Feature representation
        - How do represent x (features of the email)?
            - y = spam (1) or not spam (0)

One approach - choosing your own features

- Chose 100 words which are indicative of an email being spam or not spam
    - Spam --> e.g. buy, discount, deal
    - Non spam --> Andrew, now
    - All these words go into one long vector

- Encode this into a **reference vector**
    - See which words appear in a message

- Define a feature vector x
    - Which is 0 or 1 if a word corresponding word in the reference vector is present or not
        - This is a bitmap of the word content of your email
    - i.e. don't recount if a word appears more than once



$$x = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \\ \vdots \end{bmatrix} \begin{array}{l} \text{andrew} \\ \text{buy} \\ \text{deal} \\ \text{discont} \\ \vdots \\ \text{now} \\ \vdots \end{array}$$

    - In practice its more common to have a training set and pick the most frequently n words, where n is 10 000 to 50 000
        - So here you're not specifically choosing your own features, but you are choosing *how* you select them from the training set data

What's the best use of your time to improve system accuracy?

- Natural inclination is to collect lots of data
    - Honey pot anti-spam projects try and get fake email addresses into spammers' hands, collect loads of spam
    - This doesn't always help though
- Develop sophisticated features based on email routing information (contained in email header)
    - Spammers often try and obscure origins of email
    - Send through unusual routes
- Develop sophisticated features for message body analysis
    - Discount == discounts?
    - DEAL == deal?
- Develop sophisticated algorithm to detect misspelling
    - Spammers use misspelled word to get around detection systems

- Often a research group **randomly focus on one option**
    - May not be the most fruitful way to spend your time

- If you brainstorm a set of options this is **really good**
  - Very tempting to just try something

Error analysis

- When faced with a ML problem lots of ideas of how to improve a problem
  - Talk about error analysis - how to better make decisions
- If you're building a machine learning system often good to start by building a simple algorithm which you can implement quickly
  - Spend at most 24 hours developing an initially bootstrapped algorithm
    - Implement and test on cross validation data
  - Plot learning curves to decide if more data, features etc will help algorithmic optimization
    - Hard to tell in advance what is important
    - Learning curves really help with this
    - Way of avoiding **premature optimization**
      - We should let evidence guide decision making regarding development trajectory
  - **Error analysis**
    - Manually examine the samples (in cross validation set) that your algorithm made errors on
    - See if you can work out why
      - Systematic patterns - help design new features to avoid these shortcomings
    - e.g.
      - Built a spam classifier with 500 examples in CV set
        - Here, error rate is high - gets 100 wrong
      - Manually look at 100 and categorize them depending on features
        - e.g. type of email
      - Looking at those email
        - May find **most common type** of spam emails are pharmacy emails, phishing emails
          - See which type is most common - focus your work on those ones
        - What **features would have helped** classify them correctly
          - e.g. deliberate misspelling
          - Unusual email routing
          - Unusual punctuation
          - May fine some "spammer technique" is causing a lot of your misses
            - Guide a way around it
  - Importance of **numerical evaluation**
    - Have a way of numerically evaluated the algorithm
    - If you're developing an algorithm, it's really good to have some performance calculation which gives a single real number to tell you how well its doing
    - e.g.
      - Say were deciding if we should treat a set of similar words as the same word
      - This is done by stemming in NLP (e.g. "Porter stemmer" looks at the etymological stem of a word)
      - This may make your algorithm better or worse
        - Also worth consider weighting error (false positive vs. false negative)
          - e.g. is a false positive really bad, or is it worth have a few of one to improve performance a lot
      - Can use numerical evaluation to compare the changes
        - See if a change improves an algorithm or not
    - A single real number may be hard/complicated to compute

        - But makes it much easier to evaluate how changes impact your algorithm
    - You should do error analysis on the cross validation set instead of the test set

Error metrics for skewed analysis

- Once case where it's hard to come up with good error metric - skewed classes
- Example
    - Cancer classification
        - Train logistic regression model $h_\theta(x)$ where
            - Cancer means y = 1
            - Otherwise y = 0
        - Test classifier on test set
            - Get 1% error
                - So this looks pretty good..
            - But only 0.5% have cancer
                - Now, 1% error looks very bad!
    - So when one number of examples is very small this is an example of skewed classes
        - LOTS more of one class than another
        - So standard error metrics aren't so good
- Another example
    - Algorithm has 99.2% accuracy
    - Make a change, now get 99.5% accuracy
        - Does this really represent an improvement to the algorithm?
    - Did we do something useful, or did we just create something which predicts y = 0 more often
        - Get very low error, but classifier is still not great

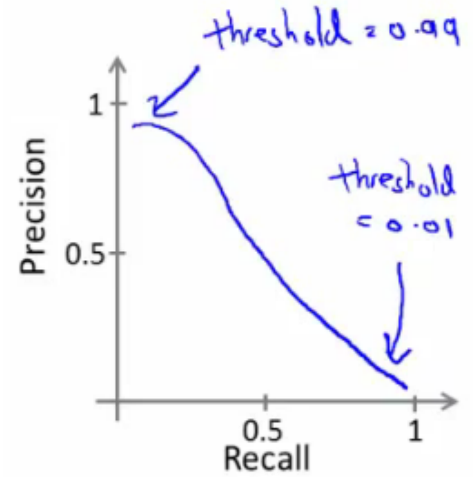
Precision and recall

- Two new metrics - **precision** and **recall**
    - Both give a value between 0 and 1
    - Evaluating classifier on a test set
    - For a test set, the actual class is 1 or 0
    - Algorithm predicts some value for class, predicting a value for each example in the test set
        - Considering this, classification can be
            - True positive (we guessed 1, it was 1)
            - False positive (we guessed 1, it was 0)
            - True negative (we guessed 0, it was 0)
            - False negative (we guessed 0, it was 1)
    - **Precision**
        - *How often does our algorithm cause a false alarm?*
        - Of all patients we predicted have cancer, what fraction of them *actually* have cancer
            - = true positives / # predicted positive
            - = true positives / (true positive + false positive)
        - High precision is good (i.e. closer to 1)
            - You want a big number, because you want false positive to be as close to 0 as possible
    - **Recall**
        - *How sensitive is our algorithm?*
        - Of all patients in set that actually have cancer, what fraction did we correctly detect

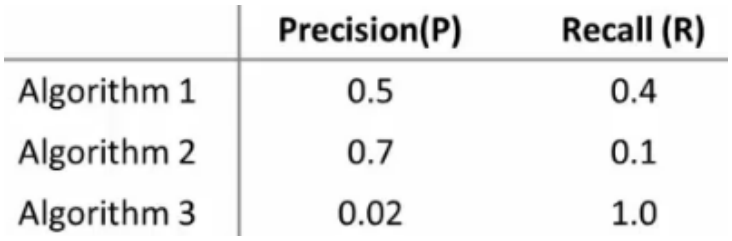
            - = true positives / # actual positives
            - = true positive / (true positive + false negative)
        - High recall is good (i.e. closer to 1)
            - You want a big number, because you want false negative to be as close to 0 as possible
    - By computing precision and recall get a better sense of how an algorithm is doing
        - This can't really be gamed
        - Means we're much more sure that an algorithm is good
    - Typically we say the presence of a rare class is what we're trying to determine (e.g. positive (1) is the existence of the rare thing)

Trading off precision and recall

- For many applications we want to control the trade-off between precision and recall
- Example
    - Trained a logistic regression classifier
        - Predict 1 if $h_\theta(x) >= 0.5$
        - Predict 0 if $h_\theta(x) < 0.5$
    - This classifier may give some value for precision and some value for recall
    - Predict 1 only if very confident
        - One way to do this modify the algorithm we could modify the prediction threshold
            - Predict 1 if $h_\theta(x) >= 0.8$
            - Predict 0 if $h_\theta(x) < 0.2$
        - Now we can be more confident a 1 is a true positive
        - But classifier has lower recall - predict y = 1 for a smaller number of patients
            - Risk of false negatives
    - Another example - avoid false negatives
        - This is probably worse for the cancer example
            - Now we may set to a lower threshold
                - Predict 1 if $h_\theta(x) >= 0.3$
                    - Predict 0 if $h_\theta(x) < 0.7$
            - i.e. 30% chance they have cancer
            - So now we have have a higher recall, but lower precision
                - Risk of false positives, because we're less discriminating in deciding what means the person has cancer
- This threshold defines the trade-off
    - We can show this graphically by plotting precision vs. recall



  - This curve can take many different shapes depending on classifier details
  - Is there a way to automatically chose the threshold
    - Or, if we have a few algorithms, how do we compare different algorithms or parameter sets?



	Precision(P)	Recall (R)
Algorithm 1	0.5	0.4
Algorithm 2	0.7	0.1
Algorithm 3	0.02	1.0

  - How do we decide which of these algorithms is best?
    - We spoke previously about using a single real number evaluation metric
    - By switching to precision/recall we have two numbers
    - Now comparison becomes harder
      - Better to have just one number
    - How can we convert P & R into one number?
      - One option is the average - (P + R)/2
        - This is not such a good solution
          - Means if we have a classifier which predicts y = 1 all the time you get a high recall and low precision
          - Similarly, if we predict Y rarely get high precision and low recall
          - So averages here would be 0.45, 0.4 and 0.51
            - 0.51 is best, despite having a recall of 1 - i.e. predict y=1 for everything
        - So average isn't great
      - **F_1Score (fscore)**
        - = 2 * (PR/ [P + R])
        - Fscore is like taking the average of precision and recall giving a higher weight to the lower value
      - Many formulas for computing comparable precision/accuracy values
        - If P = 0 or R = 0 the Fscore = 0
        - If P = 1 and R = 1 then Fscore = 1
        - The remaining values lie between 0 and 1
- Threshold offers a way to control trade-off between precision and recall
- Fscore gives a single real number evaluation metric
  - If you're trying to automatically set the threshold, one way is to try a range of threshold values and evaluate them on your cross validation set
    - Then pick the threshold which gives the best fscore.

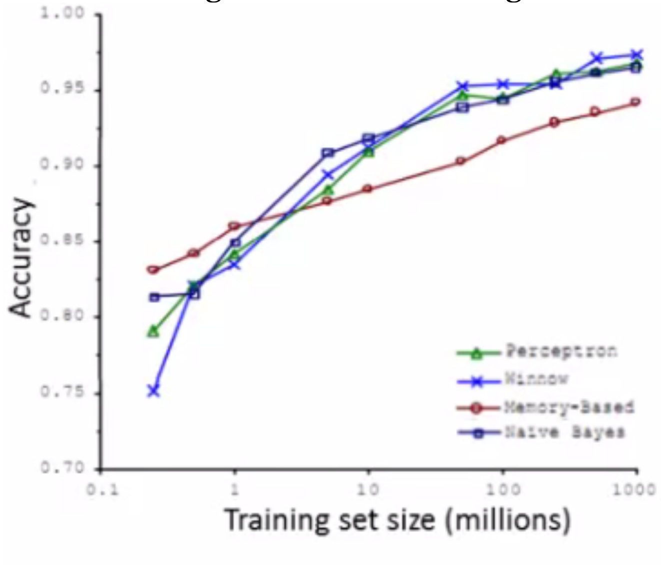
Data for machine learning

- Now switch tracks and look at how much data to train on
- On early videos caution on just blindly getting more data
  - Turns out under certain conditions getting more data is a very effective way to improve performance

Designing a high accuracy learning system

- There have been studies of using different algorithms on data
  - Data - confusing words (e.g. two, to or too)
  - Algorithms

        - Perceptron (logistic regression)
        - Winnow
            - Like logistic regression
            - Used less now
        - Memory based
            - Used less now
            - Talk about this later
        - Naive Bayes
            - Cover later
    - Varied training set size and tried algorithms on a range of sizes



    - What can we conclude
        - Algorithms give remarkably similar performance
        - As training set sizes increases accuracy increases
        - Take an algorithm, give it more data, should beat a "better" one with less data
        - Shows that
            - Algorithm choice is pretty similar
            - More data helps

- When is this true and when is it not?
    - If we can correctly assume that features x have enough information to predict y accurately, then more data will probably help
        - A useful test to determine if this is true can be, "given x, can a human expert predict y?"
    - So lets say we use a learning algorithm with many parameters such as logistic regression or linear regression with many features, or neural networks with many hidden features
        - These are powerful learning algorithms with many parameters which can fit complex functions
            - Such algorithms are low bias algorithms
                - Little systemic bias in their description - flexible
        - Use a small training set
            - Training error should be small
        - Use a very large training set
            - If the training set error is close to the test set error
            - Unlikely to over fit with our complex algorithms
            - So the test set error should also be small
    - Another way to think about this is we want our algorithm to have low bias and low variance

- Low bias --> use complex algorithm
- Low variance --> use large training set

12: Support Vector Machines (SVMs)

Previous Next Index

Support Vector Machine (SVM) - Optimization objective

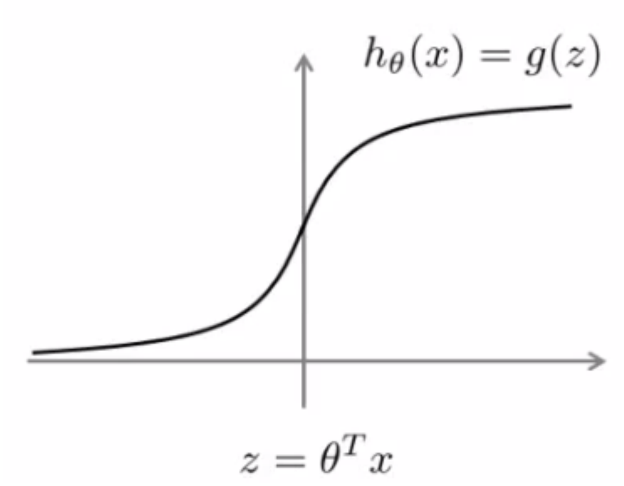
- So far, we've seen a range of different algorithms
  - With supervised learning algorithms - performance is pretty similar
    - What matters more often is;
      - The amount of training data
      - Skill of applying algorithms
- One final supervised learning algorithm that is widely used - **support vector machine (SVM)**
  - Compared to both logistic regression and neural networks, a SVM sometimes gives a cleaner way of learning non-linear functions
  - Later in the course we'll do a survey of different supervised learning algorithms

An alternative view of logistic regression

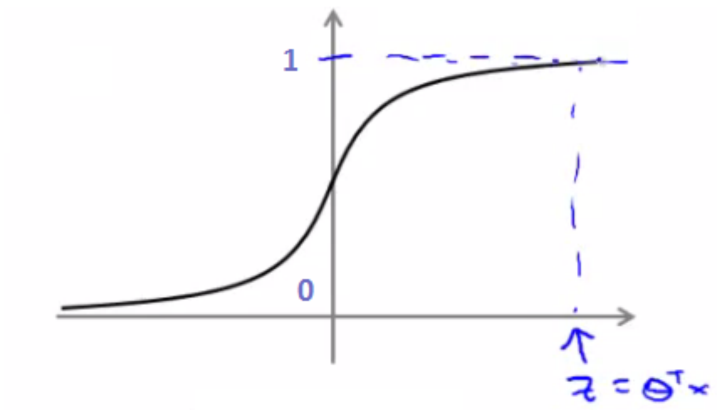
- Start with logistic regression, see how we can modify it to get the SVM
  - As before, the logistic regression hypothesis is as follows

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

  - And the sigmoid activation function looks like this



  - In order to explain the math, we use z as defined above
- What do we want logistic regression to do?
  - We have an example where y = 1
    - Then we hope $h_\theta(x)$ is close to 1
    - With $h_\theta(x)$ close to 1, $(\theta^T x)$ must be **much larger** than 0



- Similarly, when y = 0
  - Then we hope $h_\theta(x)$ is close to 0
  - With $h_\theta(x)$ close to 0, $(\theta^T x)$ must be **much less** than 0
- This is our classic view of logistic regression
  - Let's consider another way of thinking about the problem

- Alternative view of logistic regression
  - If you look at cost function, each example contributes a term like the one below to the overall cost function

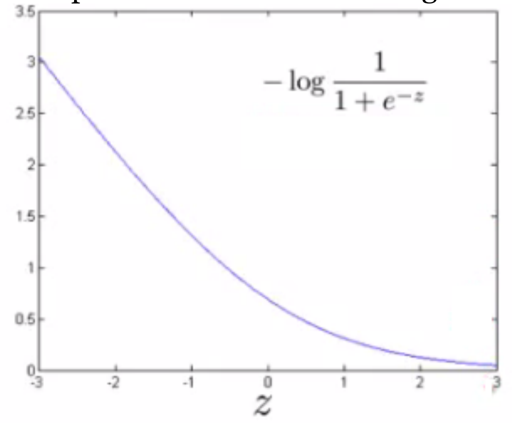
$$-(y \log h_\theta(x) + (1-y) \log(1-h_\theta(x)))$$

    - For the overall cost function, we sum over all the training examples using the above function, and have a 1/m term
- If you then plug in the hypothesis definition ($h_\theta(x)$), you get an expanded cost function equation;

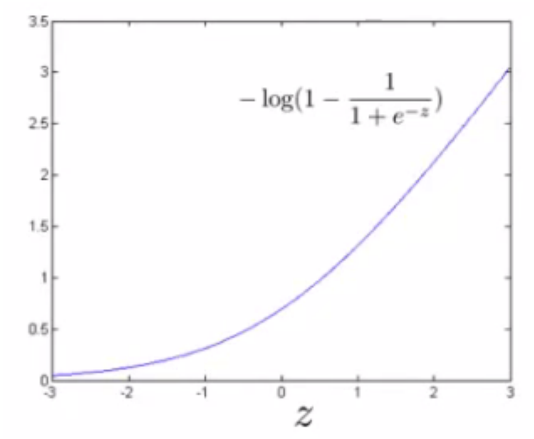
$$= -y \log \frac{1}{1+e^{-\theta^T x}} - (1-y) \log(1 - \frac{1}{1+e^{-\theta^T x}})$$

  - So each training example contributes that term to the cost function for logistic regression

- If y = 1 then only the first term in the objective matters
  - If we plot the functions vs. z we get the following graph



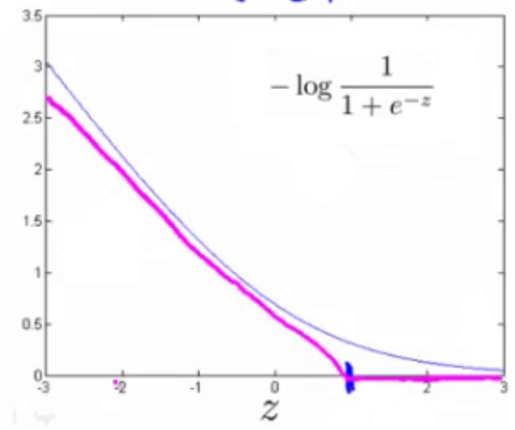
    - This plot shows the cost contribution of an example when y = 1 given z
      - So if z is big, the cost is low - this is good!
      - But if z is 0 or negative the cost contribution is high
      - This is why, when logistic regression sees a positive example, it tries to set θ^T x to be a very large term
- If y = 0 then only the second term matters
  - We can again plot it and get a similar graph



- Same deal, if z is small then the cost is low
  - But if s is large then the cost is massive

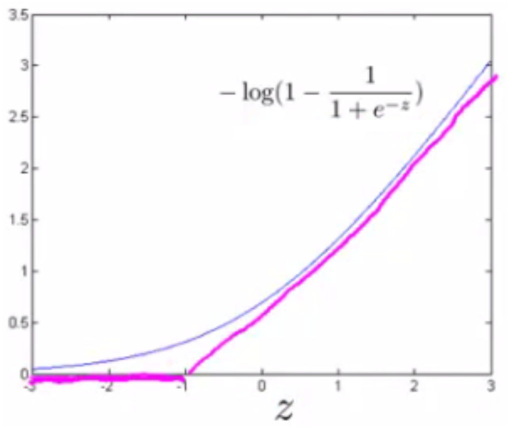
SVM cost functions from logistic regression cost functions

- To build a SVM we must redefine our cost functions
  - When y = 1
    - Take the y = 1 function and create a new cost function
    - Instead of a curved line create two straight lines (magenta) which acts as an approximation to the logistic regression y = 1 function



      - Take point (1) on the z axis
        - Flat from 1 onwards
        - Grows when we reach 1 or a lower number
      - This means we have two straight lines
        - Flat when cost is 0
        - Straight growing line after 1
    - So this is the new y=1 cost function
      - Gives the SVM a computational advantage and an easier optimization problem
      - We call this function **$\text{cost}_1(z)$**

- Similarly
  - When y = 0
    - Do the equivalent with the y=0 function plot



  - We call this function **$\text{cost}_0(z)$**
- So here we define the two cost function terms for our SVM graphically
  - How do we implement this?

The complete SVM cost function

- As a comparison/reminder we have logistic regression below

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

  - If this looks unfamiliar its because we previously had the - sign outside the expression
- For the SVM we take our two logistic regression y=1 and y=0 terms described previously and replace with
  - $\text{cost}_1(\theta^T x)$
  - $\text{cost}_0(\theta^T x)$
- So we get

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

SVM notation is slightly different

- In convention with SVM notation we rename a few things here
- 1) Get rid of the 1/m terms
  - This is just a slightly different convention
  - By removing 1/m we should get the same optimal values for
    - 1/m is a constant, so should get same optimization
    - e.g. say you have a minimization problem which minimizes to u = 5
      - If your cost function * by a constant, you still generates the minimal value
      - That minimal value is different, but that's irrelevant
- 2) For logistic regression we had two terms;
  - Training data set term (i.e. that we sum over m) = **A**
  - Regularization term (i.e. that we sum over n) = **B**
    - So we could describe it as A + λB
    - Need some way to deal with the trade-off between regularization and data set terms
    - Set different values for λ to parametrize this trade-off
  - Instead of parameterization this as A + λB
    - For SVMs the convention is to use a different parameter called C
    - So do CA + B
    - If C were equal to 1/λ then the two functions (CA + B and A + λB) would give the same value

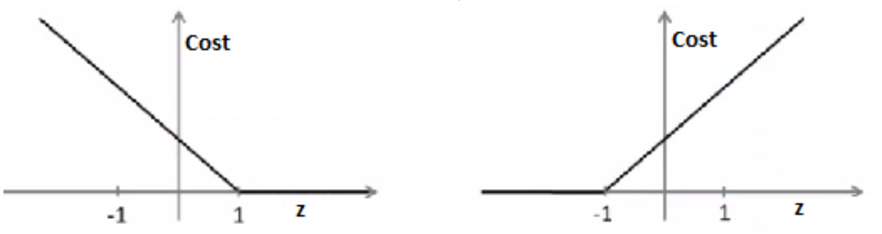
- So, our overall equation is

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

- Unlike logistic, $h_\theta(x)$ doesn't give us a probability, but instead we get a direct prediction of 1 or 0
  - So if θ^T x is equal to or greater than 0 --> $h_\theta(x) = 1$
  - Else --> $h_\theta(x) = 0$

Large margin intuition

- Sometimes people refer to SVM as **large margin classifiers**
  - We'll consider what that means and what an SVM hypothesis looks like
  - The SVM cost function is as above, and we've drawn out the cost terms below



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

  - Left is $cost_1$ and right is $cost_0$
  - What does it take to make terms small
    - If y =1
      - $cost_1(z) = 0$ only when z >= 1
    - If y = 0
      - $cost_0(z) = 0$ only when z <= -1
  - Interesting property of SVM
    - If you have a positive example, you only really *need* z to be greater or equal to 0
      - If this is the case then you predict 1
    - SVM wants a bit more than that - doesn't want to *just* get it right, but have the value be quite a bit bigger than zero
      - Throws in an extra safety margin factor
- Logistic regression does something similar
- What are the consequences of this?
  - Consider a case where we set C to be huge
    - C = 100,000
    - So considering we're minimizing CA + B
      - If C is huge we're going to pick an A value so that A is equal to zero
      - What is the optimization problem here - how do we make A = 0?
    - Making A = 0
      - If y = 1
        - Then to make our "A" term 0 need to find a value of θ so (θ^T x) is greater than or equal to 1
      - Similarly, if y = 0
        - Then we want to make "A" = 0 then we need to find a value of θ so (θ^T x) is equal to or less than -1
    - So - if we think of our optimization problem a way to ensure that this first "A" term is equal to 0, we re-factor our optimization problem into just minimizing the "B" (regularization) term,

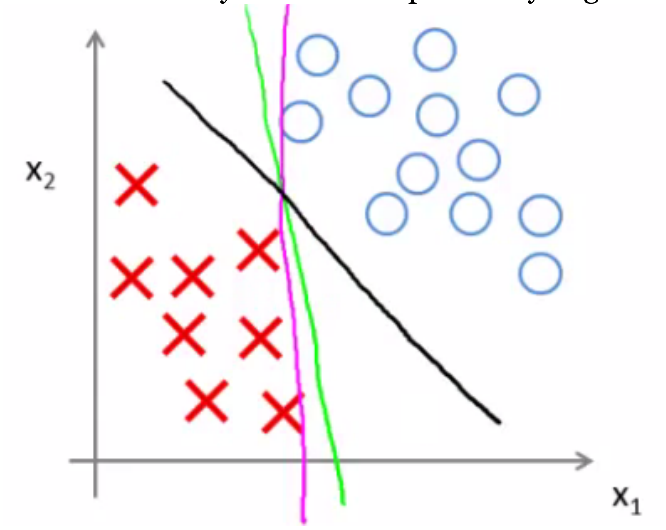
because
- When A = 0 --> A*C = 0
- So we're minimizing B, under the constraints shown below

$$\min \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

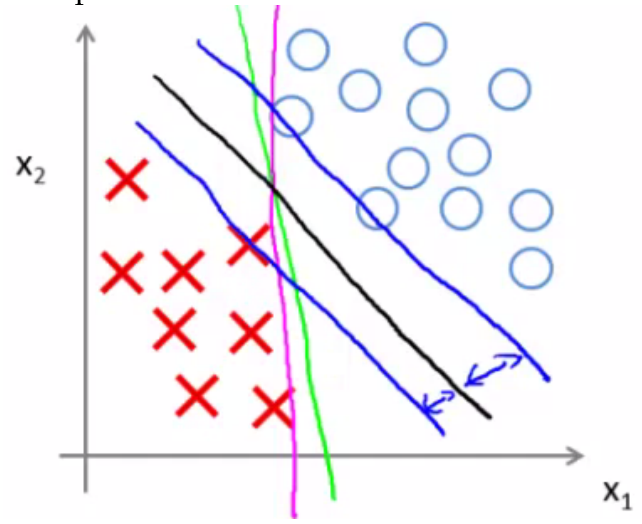
$$\text{s.t.} \quad \theta^T x^{(i)} \geq 1 \quad \text{if} \quad y^{(i)} = 1$$

$$\theta^T x^{(i)} \leq -1 \quad \text{if} \quad y^{(i)} = 0$$

- Turns out when you solve this problem you get interesting decision boundaries

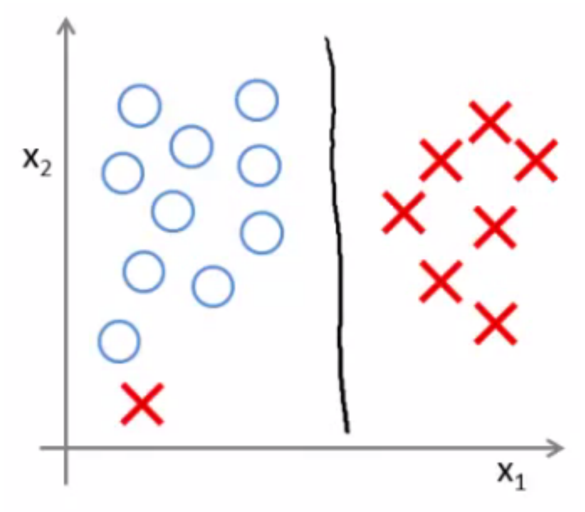


- The green and magenta lines are functional decision boundaries which could be chosen by logistic regression
  - But they probably don't generalize too well
- The black line, by contrast is the the chosen by the SVM because of this safety net imposed by the optimization graph
  - More robust separator
- Mathematically, that black line has a larger minimum distance (margin) from any of the training examples

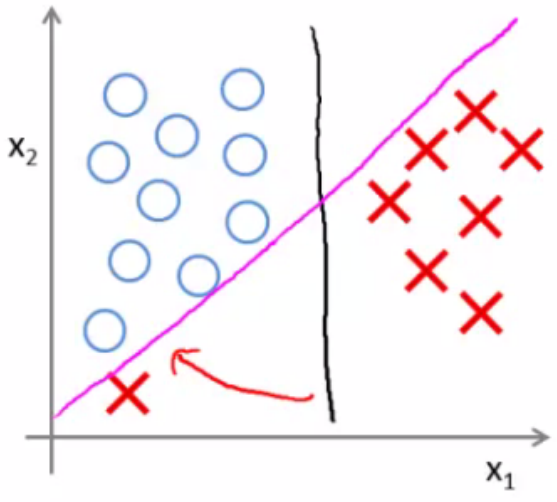


- By separating with the largest margin you incorporate robustness into your decision making process

- We looked at this at when C is very large
  - SVM is more sophisticated than the large margin might look
    - If you were just using large margin then SVM would be very sensitive to outliers



- You would risk making a ridiculous hugely impact your classification boundary
    - A single example might not represent a good reason to change an algorithm
    - If C is very large then we *do* use this quite naive maximize the margin approach



    - So we'd change the black to the magenta
- But if C is reasonably small, or a not too large, then you stick with the black decision boundary
- What about non-linearly separable data?
    - Then SVM still does the right thing if you use a normal size C
    - So the idea of SVM being a large margin classifier is only really relevant when you have no outliers and you can easily linearly separable data
- Means we ignore a few outliers

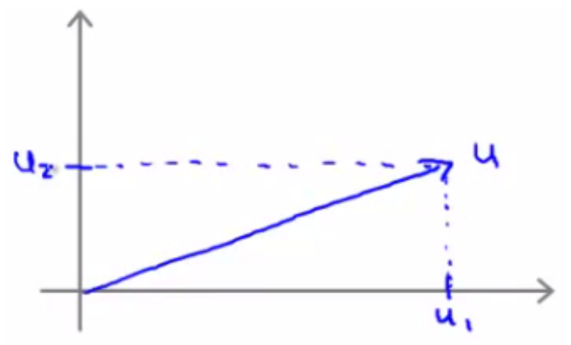
Large margin classification mathematics (optional)

Vector inner products

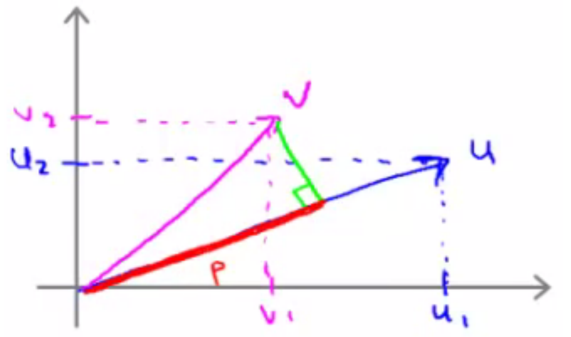
- Have two (2D) vectors u and v - what is the inner product ($u^T v$)?

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

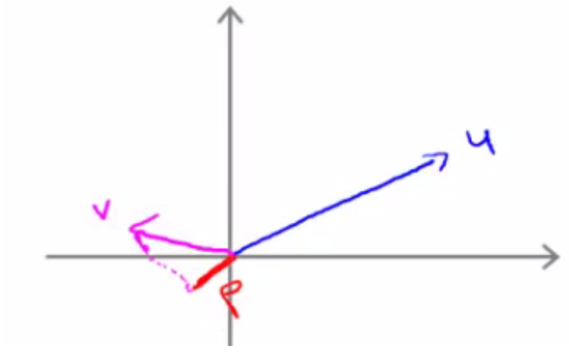
    - Plot u on graph
        - i.e u_1 vs. u_2



- One property which is good to have is the **norm** of a vector
    - Written as ||u||
        - This is the euclidean length of vector u
    - So ||u|| = SQRT(u_1^2 + u_2^2) = real number
        - i.e. length of the arrow above
        - Can show via Pythagoras
- For the inner product, take v and orthogonally project down onto u
    - First we can plot v on the same axis in the same way (v_1 vs v_1)
    - Measure the length/magnitude of the projection



    - So here, the green line is the projection
        - p = length along u to the intersection
        - p is the magnitude of the projection of vector v onto vector u
- Possible to show that
    - $u^T v$ = p * ||u||
        - So this is one way to compute the inner product
    - $u^T v = u_1v_1 + u_2v_2$
    - So therefore
        - **p * ||u|| = $u_1v_1 + u_2v_2$**
        - This is an important rule in linear algebra
    - We can reverse this too
        - So we could do
            - $v^T u = v_1u_1 + v_2u_2$
            - Which would obviously give you the same number
- p can be negative if the angle between them is 90 degrees or more



    - So here p is negative
- Use the vector inner product theory to try and understand SVMs a little better

SVM decision boundary

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$
$$\text{s.t.} \quad \theta^T x^{(i)} \geq 1 \quad \text{if } y^{(i)} = 1$$
$$\theta^T x^{(i)} \leq -1 \quad \text{if } y^{(i)} = 0$$

- For the following explanation - two simplification
  - Set $\theta_0 = 0$ (i.e. ignore intercept terms)
  - Set n = 2 - (x_1, x_2)
    - i.e. each example has only 2 features
- Given we only have two parameters we can simplify our function to

$$\frac{1}{2}(\theta_1^2 + \theta_2^2)$$

- And, can be re-written as

$$\frac{1}{2}\left(\sqrt{\theta_1^2 + \theta_2^2}\right)^2$$

  - Should give same thing
- We may notice that

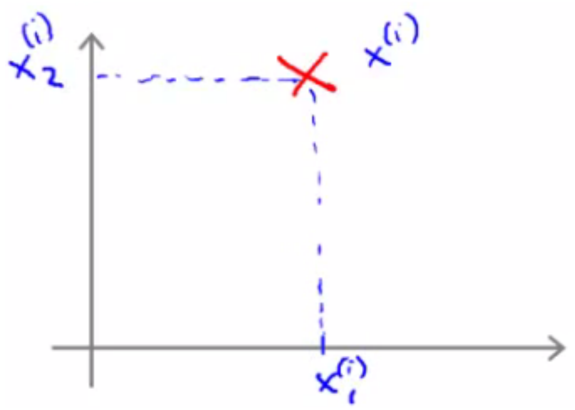
$$\frac{1}{2}\left(\underbrace{\sqrt{\theta_1^2 + \theta_2^2}}_{= \|\theta\|}\right)^2$$

  - The term in red is the norm of θ
    - If we take θ as a 2x1 vector
    - If we assume $\theta_0 = 0$ its still true
- So, finally, this means our optimization function can be re-defined as

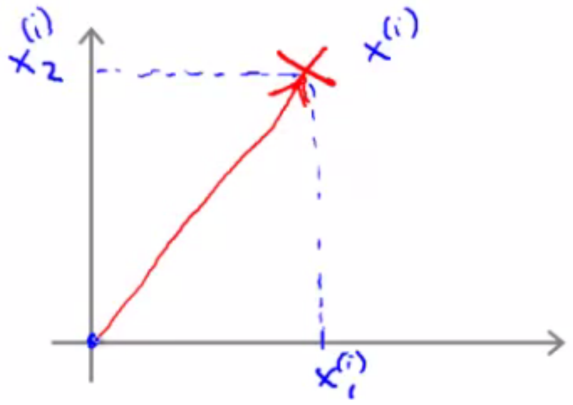
$$= \frac{1}{2} \|\theta\|^2$$

- So the SVM is minimizing the squared norm

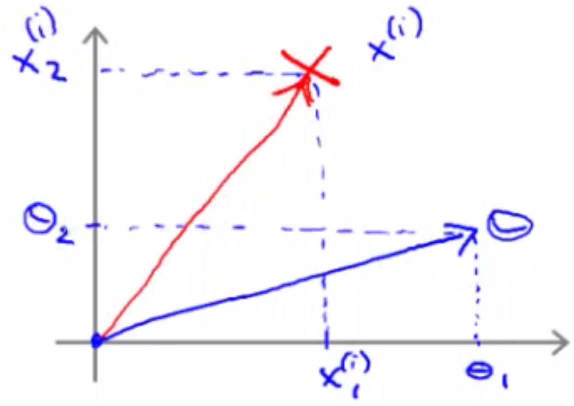
- Given this, what are the ($\theta^T x$) parameters doing?
  - Given θ and given example x what is this equal to
    - We can look at this in a comparable manner to how we just looked at u and v
  - Say we have a single positive training example (red cross below)



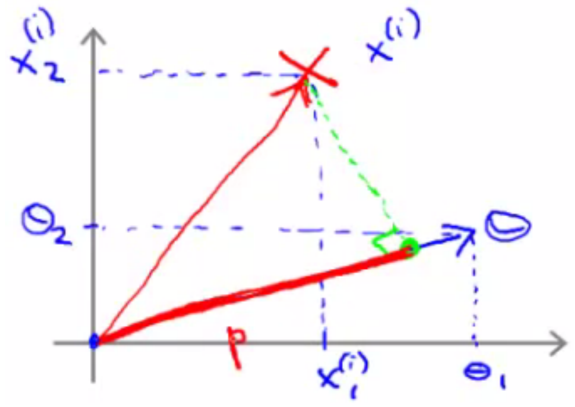
- Although we haven't been thinking about examples as vectors it can be described as such



- Now, say we have our parameter vector θ and we plot that on the same axis



- The next question is what is the inner product of these two vectors

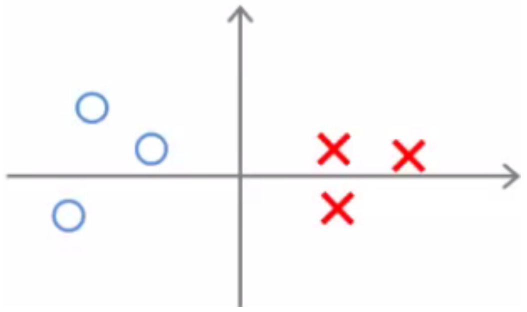


  - p, is in fact p^i, because it's the length of p for example i
    - Given our previous discussion we know
      $(\theta^T x^i) = p^i * ||\theta||$
      $= \theta_1 x^i_1 + \theta_2 x^i_2$
    - So these are both equally valid ways of computing $\theta^T x^i$

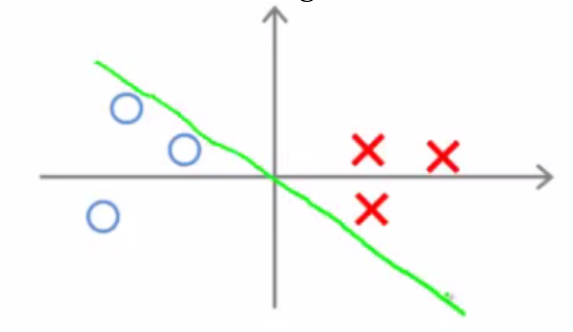
- What does this mean?
    - The constraints we defined earlier
        - $(\theta^T x) >= 1$ if y = 1
        - $(\theta^T x) <= -1$ if y = 0
    - Can be replaced/substituted with the constraints
        - $p^i * ||\theta|| >= 1$ if y = 1
        - $p^i * ||\theta|| <= -1$ if y = 0
    - Writing that into our optimization objective

$$\min_\theta \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\|\theta\|^2$$
$$\text{s.t.} \quad p^{(i)} \cdot \|\theta\| \geq 1 \quad \text{if } y^{(i)} = 1$$
$$p^{(i)} \cdot \|\theta\| \leq -1 \quad \text{if } y^{(i)} = 1$$

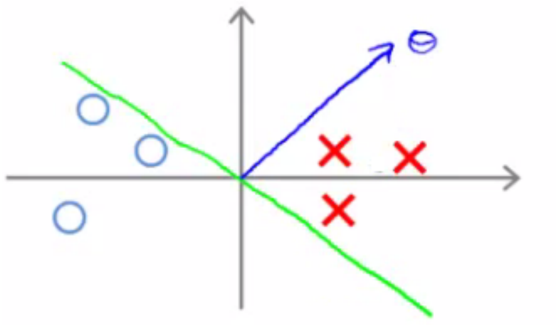
- So, given we've redefined these functions let us now consider the training example below



    - Given this data, what boundary will the SVM choose? Note that we're still assuming $\theta_0 = 0$, which means the boundary has to pass through the origin (0,0)
        - Green line - small margins

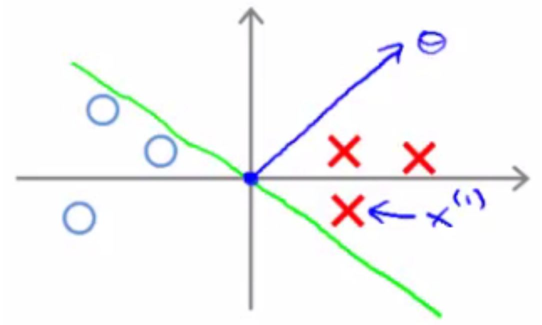


            - SVM would not chose this line
                - Decision boundary comes very close to examples
                - Lets discuss *why* the SVM would **not** chose this decision boundary
    - Looking at this line
        - We can show that θ is at 90 degrees to the decision boundary

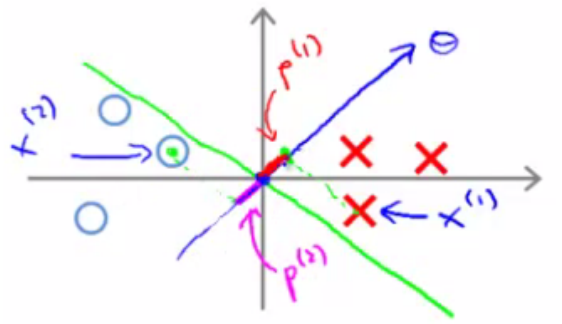


            - **θ is always at 90 degrees to the decision boundary** (can show with linear algebra, although we're not going to!)

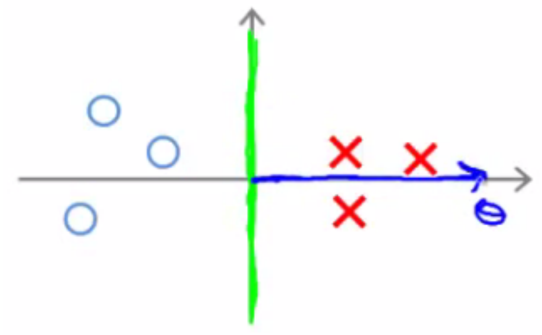
- So now lets look at what this implies for the optimization objective
  - Look at first example (x^1)



  - Project a line from x^1 on to to the θ vector (so it hits at 90 degrees)
    - The distance between the intersection and the origin is (**p^1**)
  - Similarly, look at second example (x^2)
    - Project a line from x^2 into to the θ vector
    - This is the magenta line, which will be **negative** (p^2)
  - If we overview these two lines below we see a graphical representation of what's going on;

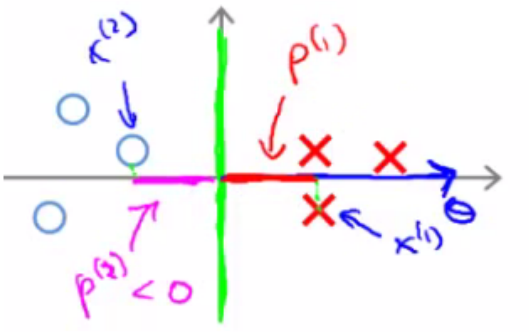


  - We find that both these p values are going to be pretty small
  - If we look back at our optimization objective
    - We know we need p^1 * ||θ|| to be bigger than or equal to 1 for positive examples
      - If p is small
        - Means that ||θ|| must be pretty large
    - Similarly, for negative examples we need p^2 * ||θ|| to be smaller than or equal to -1
      - We saw in this example p^2 is a small negative number
        - So ||θ|| must be a large number
  - Why is this a problem?
    - The optimization objective is trying to find a set of parameters where the norm of theta is small
      - So this doesn't seem like a good direction for the parameter vector (because as p values get smaller ||θ|| must get larger to compensate)
        - So we should make p values larger which allows ||θ|| to become smaller
- So lets chose a different boundary

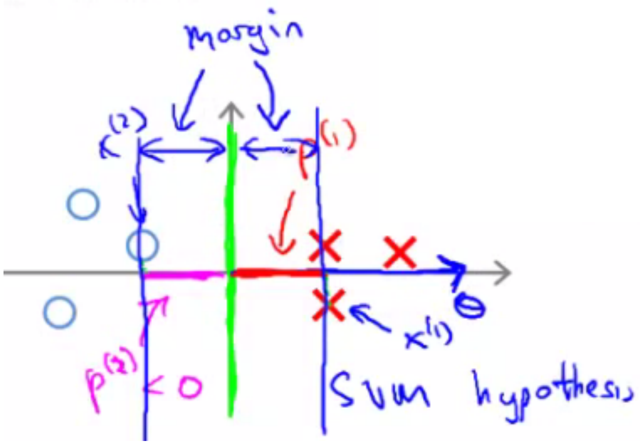


  - Now if you look at the projection of the examples to θ we find that p^1 becomes large and ||θ|| can

  become small
  - So with some values drawn in



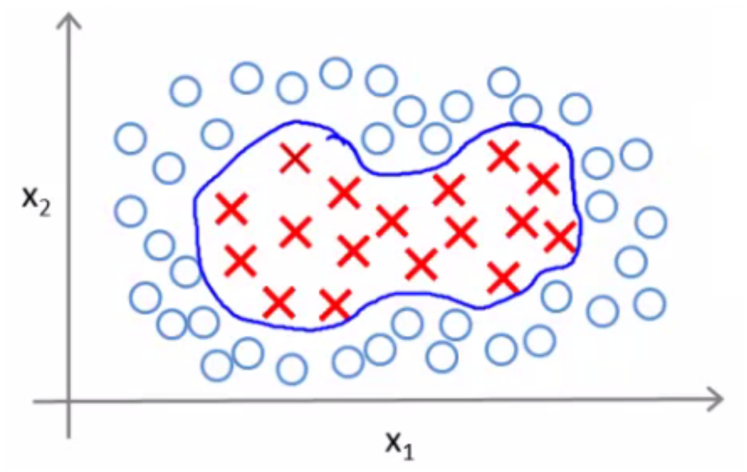
  - This means that by choosing this second decision boundary we can make ||θ|| smaller
    - Which is why the SVM choses this hypothesis as better
    - This is how we generate the large margin effect



    - The magnitude of this margin is a function of the p values
      - So by maximizing these p values we minimize ||θ||
- Finally, we did this derivation assuming $\theta_0 = 0$,
  - If this is the case we're entertaining only decision boundaries which pass through (0,0)
  - If you allow θ_0 to be other values then this simply means you can have decision boundaries which cross through the x and y values at points other than (0,0)
  - Can show with basically same logic that this works, and even when θ_0 is non-zero when you have optimization objective described above (when C is very large) that the SVM is looking for a large margin separator between the classes

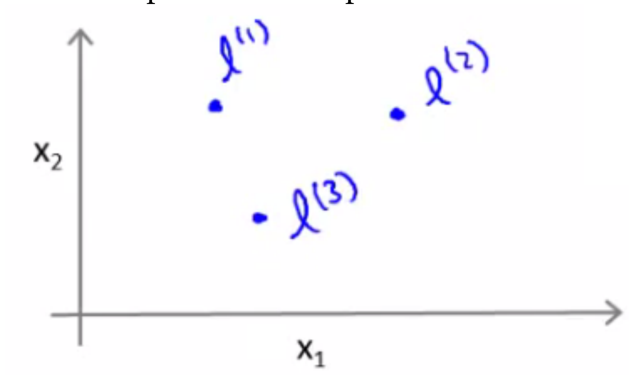
Kernels - 1: Adapting SVM to non-linear classifiers

- What are kernels and how do we use them
  - We have a training set
  - We want to find a non-linear boundary



- Come up with a complex set of polynomial features to fit the data
    - Have $h_\theta(x)$ which
        - Returns 1 if the combined weighted sum of vectors (weighted by the parameter vector) is less than or equal to 0
        - Else return 0
    - Another way of writing this (new notation) is
        - That a hypothesis computes a decision boundary by taking the sum of the parameter vector multiplied by a **new feature vector f**, which simply contains the various high order x terms
        - e.g.
            - $h_\theta(x) = \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3$
            - Where
                - $f_1 = x_1$
                - $f_2 = x_1 x_2$
                - $f_3 = \ldots$
                - i.e. not specific values, but each of the terms from your complex polynomial function
    - Is there a better choice of feature f than the high order polynomials?
        - As we saw with computer imaging, high order polynomials become computationally expensive

- New features
    - Define three features in this example (ignore x_0)
    - Have a graph of x_1 vs. x_2 (don't plot the values, just define the space)
    - Pick three points in that space



- These points l^1, l^2, and l^3, were chosen manually and are called **landmarks**
    - Given x, define f1 as the similarity between (x, l^1)
        - $= \exp(- (|| x - l^1 ||^2) / 2\sigma^2)$

$$\exp\left(-\frac{\|x-l^{(1)}\|^2}{2\sigma^2}\right)$$
=

      - **|| x - l^1 ||** is the euclidean distance between the point x and the landmark l^1 squared
        - Disussed more later
      - If we remember our statistics, we know that
        - σ is the **standard deviation**
        - σ^2 is commonly called the **variance**
    - Remember, that as discussed

$$\|x-l^{(1)}\|^2=\sum_{j=1}^{n}(x_j-l_j^{(1)})^2$$

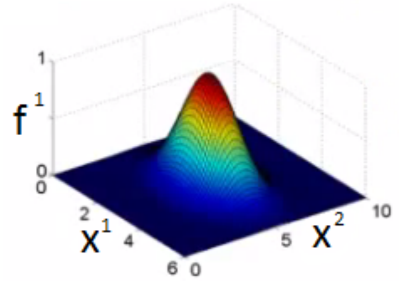
  - So, f2 is defined as
    - f2 = similarity(x, l^1) = exp(- (|| x - l^2 ||^2) / 2σ^2)
  - And similarly
    - f3 = similarity(x, l^2) = exp(- (|| x - l^1 ||^2) / 2σ^2)
  - This similarity function is called a **kernel**
    - This function is a **Gaussian Kernel**
  - So, instead of writing similarity between x and l we might write
    - f1 = k(x, l^1)

Diving deeper into the kernel

- So lets see what these kernels do and why the functions defined make sense
  - Say x is close to a landmark
    - Then the squared distance will be ~0
      - So

$$f_1 \approx \exp\left(-\frac{0^2}{2\sigma^2}\right)$$

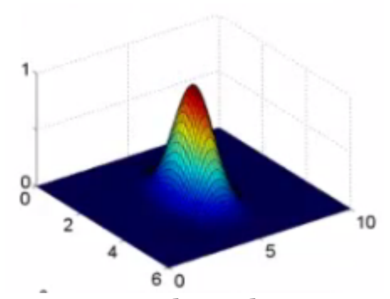
        - Which is basically e^{-0}
          - Which is close to 1
    - Say x is far from a landmark
      - Then the squared distance is big
        - Gives e$^{\text{-large number}}$
          - Which is close to zero
    - Each landmark defines a new features
- If we plot f1 vs the kernel function we get a plot like this
  - Notice that when x = [3,5] then f1 = 1
  - As x moves away from [3,5] then the feature takes on values close to zero
  - So this measures how close x is to this landmark



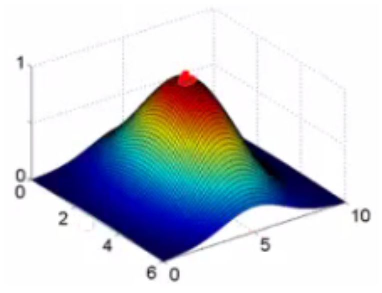
What does σ do?

- **σ^2** is a parameter of the Gaussian kernel

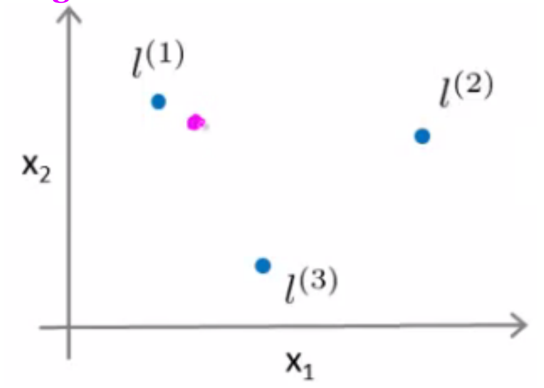
  - Defines the steepness of the rise around the landmark
- Above example $\sigma^2 = 1$
- Below $\sigma^2 = 0.5$



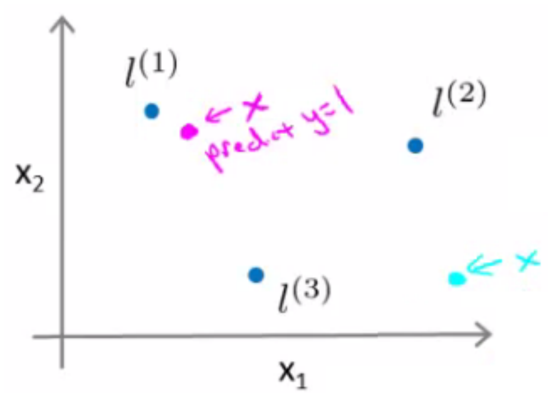
  - We see here that as you move away from 3,5 the feature f1 falls to zero much more rapidly
- The inverse can be seen if $\sigma^2 = 3$



- Given this definition, what kinds of hypotheses can we learn?
  - With training examples x we predict "1" when
  - $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 >= 0$
    - For our example, lets say we've already run an algorithm and got the
      - $\theta_0 = -0.5$
      - $\theta_1 = 1$
      - $\theta_2 = 1$
      - $\theta_3 = 0$
    - Given our placement of three examples, what happens if we evaluate an example at the **magenta dot** below?

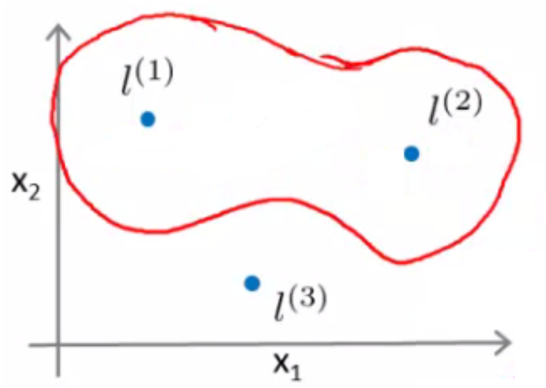


    - Looking at our formula, we know f1 will be close to 1, but f2 and f3 will be close to 0
      - So if we look at the formula we have
        - $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 >= 0$
        - $-0.5 + 1 + 0 + 0 = 0.5$
          - 0.5 is greater than 1
    - If we had **another point** far away from all three



- This equates to -0.5
  - So we predict 0

- Considering our parameter, for points near l^1 and l^2 you predict 1, but for points near l^3 you predict 0
- Which means we create a non-linear decision boundary that goes a lil' something like this;



  - Inside we predict y = 1
  - Outside we predict y = 0
- So this show how we can create a non-linear boundary with landmarks and the kernel function in the support vector machine
  - But
    - How do we get/chose the landmarks
    - What other kernels can we use (other than the Gaussian kernel)

Kernels II

- Filling in missing detail and practical implications regarding kernels
- Spoke about picking landmarks manually, defining the kernel, and building a hypothesis function
  - Where do we get the landmarks from?
  - For complex problems we probably want lots of them

Choosing the landmarks

- Take the training data
- For each example place a landmark at exactly the same location
- So end up with m landmarks
  - One landmark per location per training example
  - Means our features measure how close to a training set example something is
- Given a new example, compute all the f values
  - Gives you a feature vector f (f_0 to f_m)
    - f_0 = 1 always
- A more detailed look at generating the f vector
  - If we had a training example - features we compute would be using (x^i, y^i)

- So we just cycle through each landmark, calculating how close to that landmark actually x^i is
    - f_1^i, = $k(x^i, l^1)$
    - f_2^i, = $k(x^i, l^2)$
    - ...
    - f_m^i, = $k(x^i, l^m)$
- Somewhere in the list we compare x to itself... (i.e. when we're at f_i^i)
    - So because we're using the Gaussian Kernel this evalues to 1
- Take these m features (f_1, f_2 ... f_m) group them into an [m +1 x 1] dimensional vector called f
    - f^i is the f feature vector for the ith example
    - And add a 0th term = 1

- Given these kernels, how do we use a support vector machine

SVM hypothesis prediction with kernels

- Predict y = 1 if (θ^T f) >= 0
    - Because θ = [m+1 x 1]
    - And f = [m +1 x 1]
- So, this is how you make a prediction assuming you already have θ
    - How do you get θ?

SVM training with kernels

- Use the SVM learning algorithm

$$\min_\theta C \sum_{i=1}^{m} y^{(i)} cost_1(\theta^T f^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

    - Now, we minimize using f as the feature vector instead of x
    - By solving this minimization problem you get the parameters for your SVM
- In this setup, m = n
    - Because number of features is the number of training data examples we have
- One final mathematic detail (not crucial to understand)
    - If we ignore θ_0 then the following is true

$$\sum_{j=1}^{n} \theta_j^2 = \theta^T \theta$$

    - What many implementations do is

$$\theta^T \mathbf{M} \theta$$

        - Where the matrix M depends on the kernel you use
        - Gives a slightly different minimization - means we determine a rescaled version of θ
        - Allows more efficient computation, and scale to much bigger training sets
        - If you have a training set with 10 000 values, means you get 10 000 features
            - Solving for all these parameters can become expensive
            - So by adding this in we avoid a for loop and use a matrix multiplication algorithm instead
- You can apply kernels to other algorithms
    - But they tend to be very computationally expensive
    - But the SVM is far more efficient - so more practical
- Lots of good off the shelf software to minimize this function

- **SVM parameters (C)**

- Bias and variance trade off
- Must chose C
  - C plays a role similar to 1/LAMBDA (where LAMBDA is the regularization parameter)
- Large C gives a hypothesis of **low bias high variance** --> overfitting
- Small C gives a hypothesis of **high bias low variance** --> underfitting

- **SVM parameters (σ^2)**
  - Parameter for calculating f values
    - Large σ^2 - f features vary more smoothly - higher bias, lower variance
    - Small σ^2 - f features vary abruptly - low bias, high variance

SVM - implementation and use

- So far spoken about SVM in a very abstract manner
- What do you need to do this
  - Use SVM software packages (e.g. liblinear, libsvm) to solve parameters θ
  - Need to specify
    - Choice of parameter C
    - Choice of kernel

Choosing a kernel

- We've looked at the **Gaussian kernel**
  - Need to define σ (σ^2)
    - Discussed σ^2
  - When would you chose a Gaussian?
    - If n is small and/or m is large
      - e.g. 2D training set that's large
  - If you're using a Gaussian kernel then you may need to implement the kernel function
    - e.g. a function
      fi = kernel(x1,x2)
      - Returns a real number
    - Some SVM packages will expect you to define kernel
    - Although, some SVM implementations include the Gaussian and a few others
      - Gaussian is probably most popular kernel
  - NB - make sure you perform **feature scaling** before using a Gaussian kernel
    - If you don't features with a large value will dominate the f value
- Could use no kernel - **linear kernel**
  - Predict y = 1 if (θ^T x) >= 0
    - So no f vector
    - Get a standard linear classifier
  - Why do this?
    - If n is large and m is small then
      - Lots of features, few examples
      - Not enough data - risk overfitting in a high dimensional feature-space
- Other choice of kernel
  - Linear and Gaussian are most common
  - Not all similarity functions you develop are valid kernels
    - Must satisfy **Merecer's Theorem**
    - SVM use numerical optimization tricks
      - Mean certain optimizations can be made, but they must follow the theorem
  - **Polynomial Kernel**
    - We measure the similarity of x and l by doing one of
      - $(x^T l)^2$
      - $(x^T l)^3$

        - $(x^T l+1)^3$
    - General form is
        - $(x^T l+\text{Con})^D$
    - If they're similar then the inner product tends to be large
    - Not used that often
    - Two parameters
        - Degree of polynomial (D)
        - Number you add to l (Con)
    - Usually performs worse than the Gaussian kernel
    - Used when x and l are both non-negative
- **String kernel**
    - Used if input is text strings
    - Use for text classification
- **Chi-squared kernel**
- **Histogram intersection kernel**

Multi-class classification for SVM

- Many packages have built in multi-class classification packages
- Otherwise use one-vs all method
- Not a big issue

Logistic regression vs. SVM

- When should you use SVM and when is logistic regression more applicable
- If n (features) is large vs. m (training set)
    - e.g. text classification problem
        - Feature vector dimension is 10 000
        - Training set is 10 - 1000
        - Then use logistic regression or SVM with a linear kernel
- If n is small and m is intermediate
    - n = 1 - 1000
    - m = 10 - 10 000
    - Gaussian kernel is good
- If n is small and m is large
    - n = 1 - 1000
    - m = 50 000+
        - SVM will be slow to run with Gaussian kernel
    - In that case
        - Manually create or add more features
        - Use logistic regression of SVM with a linear kernel
- Logistic regression and SVM with a linear kernel are pretty similar
    - Do similar things
    - Get similar performance
- A lot of SVM's power is using diferent kernels to learn complex non-linear functions
- For all these regimes a well designed NN should work
    - But, for some of these problems a NN might be slower - SVM well implemented would be faster
- SVM has a convex optimization problem - so you get a global minimum
- It's not always clear how to chose an algorithm
    - Often more important to get enough data
    - Designing new features
    - Debugging the algorithm
- SVM is widely perceived a very powerful learning algorithm