**Week 1: Introduction**

**1. What is machine learning?**

"The field of study that gives computers the ability to learn without being explicitly programmed." This is an older, informal definition.

Modern definition:

“A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E”.

**2. Classification:**

**Supervised Learning:**

- In supervised learning, we are given a dataset and already know what correct output should look like, having the idea that there is a relationship between the input and output.

- It consists of regression and classification problem:

+ Regression: We are trying to predicts results within a continuous output, meaning that we are trying to map into variables to some continuous function.

+ Classification: We are instead trying to predict results in a discrete output.

**Unsupervised Learning:**

- Unsupervised learning allows us to approach problems with little or no idea what our result should look like. We can derive structure from data where we don’t necessarily know the effect of the variables.

- We can derive this structure by clustering the data based on the relationships among the variables in the data.

- With unsupervised learning there is no feedback based on the prediction results.

- It consists of clustering (medial social network) and non-clustering problems (Cocktail Party Algorithm).

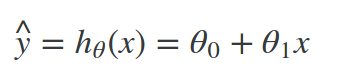
**\* Additional: Page 17, 18, 19 ML book.**

**3. ML: Linear Regression with One Variable**

**Model Representation:**

- Univriate linear regression is used when you want to predict a single output value y from a single input value x. We’re doing supervised learning here, so that means we already have an idea about what the input/output cause and cause and effect should be.

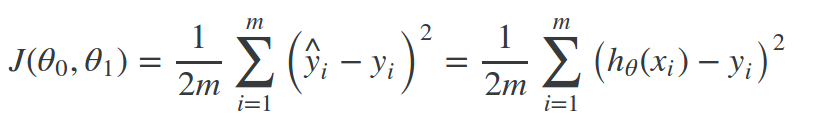
**The Hypothesis Function:**

****

- We are trying to create a function call h(theta) that is trying to map our input data (the x’s) to your output data (the y’s).

**Cost function:**

- We can measure the **accuracy of our hypothesis function by using a cost function**. This takes an averages (actually a fancier version of an average) of all the results of the hypothesis with inputs from x’s compared to the actual output y’s.



- It is the difference between the predicted value and the actual value.

- This function is otherwise called the “Squared error function”, or “Mean squared error”. The mean is halved (1/2m) as a convenience for the computation of the gradient descent, as the derivative term of the square function with cancel out the ½ term.

**ML: Gradient Descent**

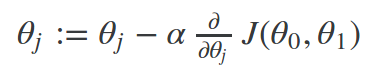
- So we have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in hypothesis function. That’s where gradient descent comes in.

- We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting particular set of parameters. We put theta0 and theta1 on the x and y axis orderly, and the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters.

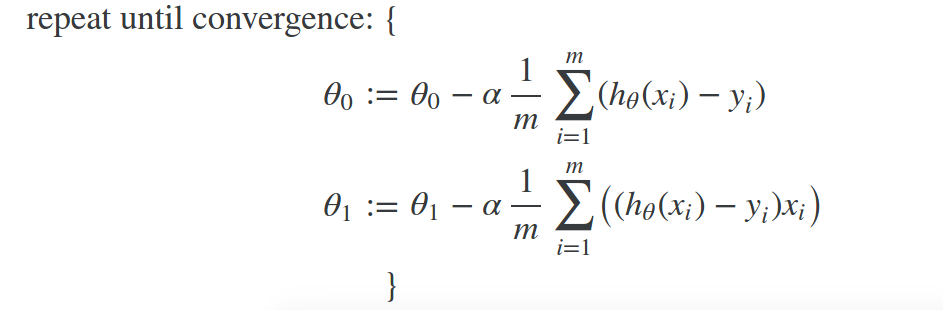
- We will know that **we have succeeded when our cost function is at the very bottom of the pits in our graph (when its value is the minimum)**. The way we do this is by talking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We **make steps down the cost function in the direction with the steepest descent**, and the **size of each step is determined by a parameter alpha, which is called the learning rate**.

The gradient descent algorithm:

repeat until convergence:



**Gradient Descent for Linear Regression:**

****

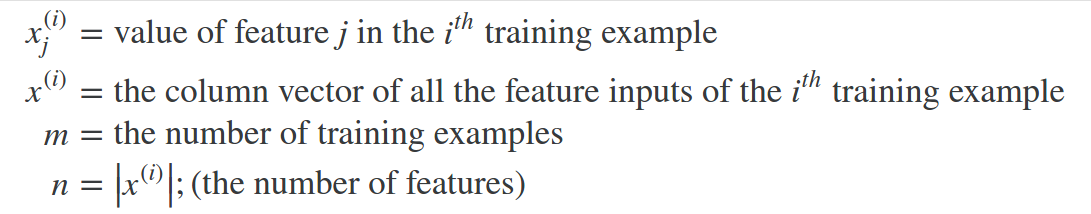
where m is the size of the training set, theta0 a constant that will be changing simultaneously with theta1 and xi, yi are the values of the given training set (theta).

=> The point of all this is that if we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate.

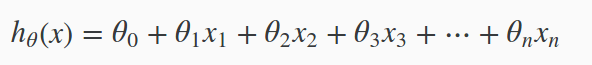
**Week 2:**

**1. Linear regression with Multiple Variables:**

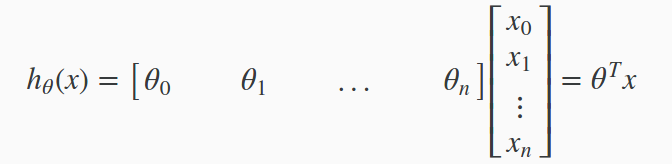
- As known as “multivariate linear regression”.



- Now define the multivariable form of the hypothesis function accommodating these multiple features is as follows:

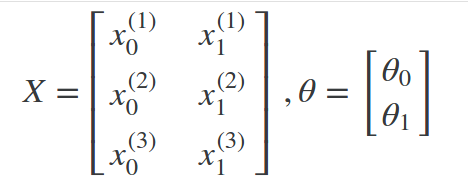


Using definition of matrix multiplication, our multivariable hypothesis can be concisely represented as:



This is a vectorization of our hypothesis function.

The training examples are store in X row-wise, like such:

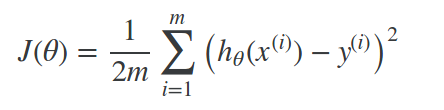


We can calculate the hypothesis as a column vector of size (m,1) with:

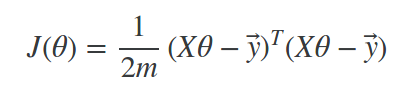


**Cost function:**

- For the parameter vector theta, the cost function is:



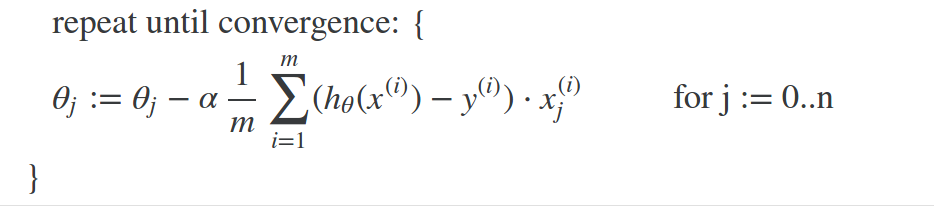
-The vectorized version is:



where y- denotes the vector of all y values.

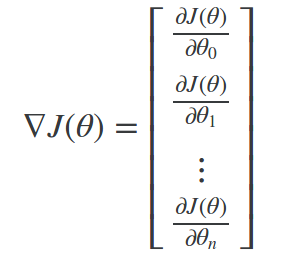
**Gradient Descent for Multiple Variables:**

- The gradient equation itself is generally the same form; we just have to repeat it for our ‘n’ features:

Matrix Notation: The Gradient Descent rule can be expressed as:

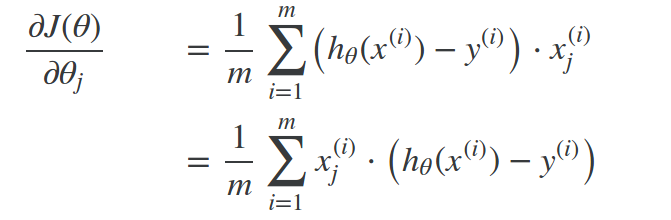


where:

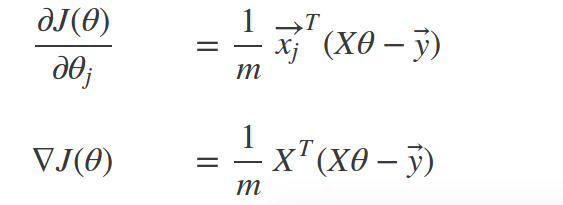


is a column vector of the form.

- The j-th component of the gradient is the summation of the product of two terms:



In other way:



=> Finally:



**Future Normalization:**

- We can speed up gradient descent by having each of our input values in roughly the same range. This is because theta will descend quickly on small ranges and slowly on the large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

- The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. Ideally: -1 <= x <= 1 or -0.5 <= x <= 0.5 (-3 <=x <=3).

- Two techniques to help with this are **feature scaling** and **mean normalization**. Future scaling involves dividing the input values by the range of the input variables, resulting in the new range of 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable, resulting in a new average value of the input variable of just zero. To implement both of these techniques, adjust your input values as shown:

**x = (x – u)/s**

where u is the mean of all the values for feature and s is the range of values (max – min) or s is the standard deviation.

- Note that dividing by the range or dividing by the standard deviation, give different results.

**Gradient Descent Tips:**

**- Debugging gradient descent:** Make a plot with the number of iterations on the x-axis. Now plot the cost function, J(theta) over the number of iterations of gradient descent. If J(theta) ever increases, then you probably need to decrease alpha.

**- Automatic convergence test:** Declare convergence if J(theta) decreases by less than E in one iteration, where E is some small value such as 1e-3. However in practice it is difficult to choose this threshold value.

=> It has been proven that if learning rate alpha is sufficiently small, then J(theta) will decrease on every iteration (Andrew Ng recommends decreasing alpha by multiples of 3).

**Polynomial Regression:**

- Our hypothesis function need not be linear (a straight line) if that does not fit the data well.

- We can change the behavior or curve of our hypothesis function by making it quadratic, cubic or square root function (or any other form).

For ex:

or:

- One **important thing** to keep in mind is, if you choose your features this way then feature scaling becomes very important.

Ex: If x1 has range 1-1e3 then range of x1^2 become 1-1e6 and that of x1^3 becomes 1-1e9.

**Week 3: Logistic Regression (Classification)**

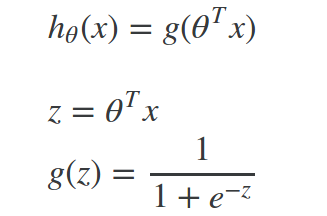
**1. Binary Classification:**

- Instead of our output vector y being continuous range of values, it will only be 0 or 1:

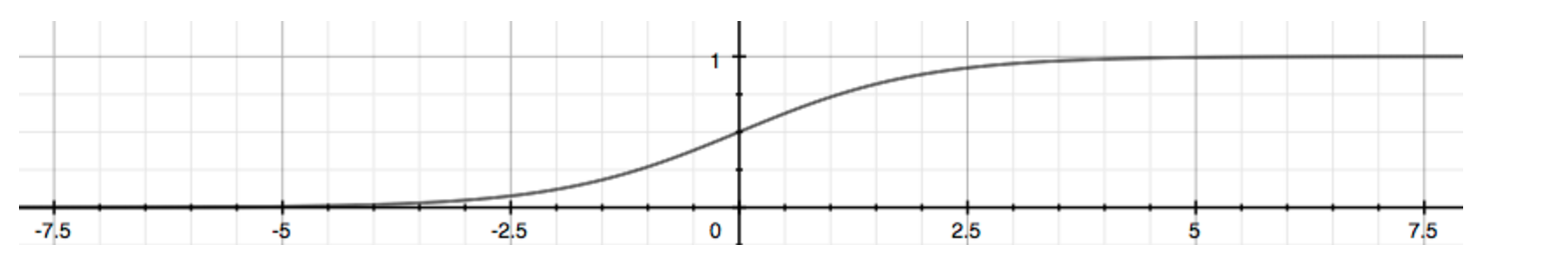
**y = {0,1}** with 0: negative class and 1: positive class.

- Hypothesis Representation: 0 <= htheta(x) <= 1

=> Our new form uses the “Sigmoid (Logistic) Function”:

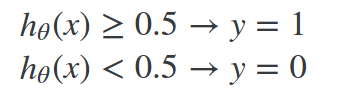


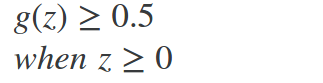
- The function g(z) shown here, maps any real number to the (0,1) interval, making it useful for transforming an arbitrary function into a function better suited for classification.



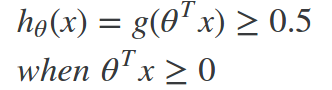
**2. Decision boundary:**

- We got:

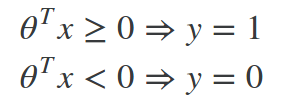
****

****

=>



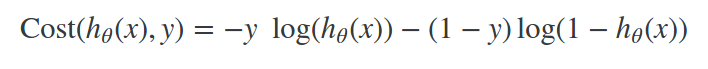
From these statements we now can say:



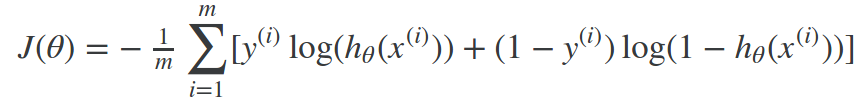
=> In conclusion, the **decision boundary** is the line that separates the area where y = 0 and y = 1. It is **created** by our hypothesis function.

**3. Simplified Cost Function and Gradient Descent:**

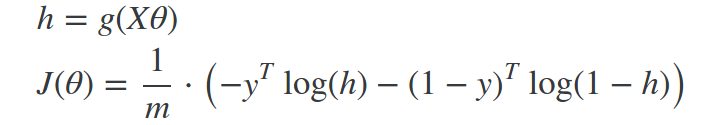
- We can compress our cost function’s two conditional cases into one case:



=>

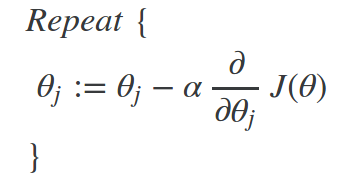


- A vectorized implementation is:

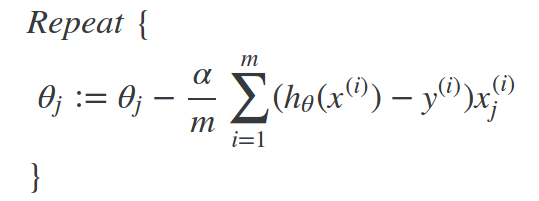


**Gradient Descent:**

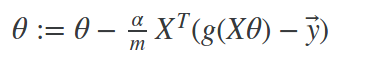
- Remember that the general form of gradient descent is:



=> We can work out the derivative part using calculus to get:



A vectorized implementation is:



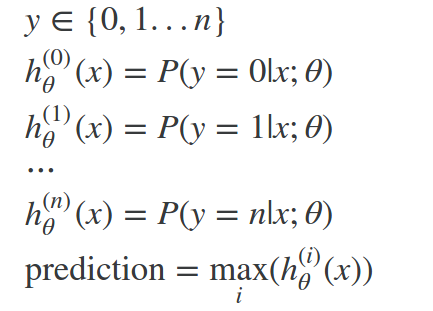
**3. Advanced Optimization:**

“Conjugate gradient”, “BFGS” and “L-BFGS” are more sophisticated, faster ways to optimize theta that can be used instead of gradient descent. A. Ng suggests **not to write these some sophisticated algorithms yourself but uses libraries instead**, as they’re already tested and highly optimized.

**4. Multiclass Classification: One-vs-all**

- Now we will approach the classification of data into more then two categories. Instead of y = {0,1} we will explain our definition so that y = {0,1,…,n}.

- In this case we provide our problem into n+1 (+1 because the index stars at 0) binary classification problems, in each one we predict the probability that ‘y’ is a member of one of our classes.



=> We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that return the highest value as our prediction.

**5. Regularization**

- Regularization is designed to address the problem of overfitting.

- **High bias or underfitting** is when the form of your hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that too simple or uses too few features.

- At the other extreme, **overfitting or high variance** is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

- There are 2 main options to address the issue of overfitting:

+ **Reduce the number of futures:**

a) Manually select which features to keep.

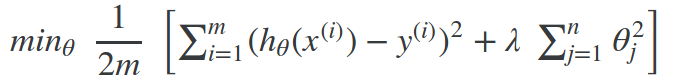
b) Use a model selection algorithm (studied later in the course).

**+ Regularization:** Keep all the features, but reduce the parameters theta.

=> Regularization works well when we have a lot of slightly useful features.

**Regularized Linear Regression:**

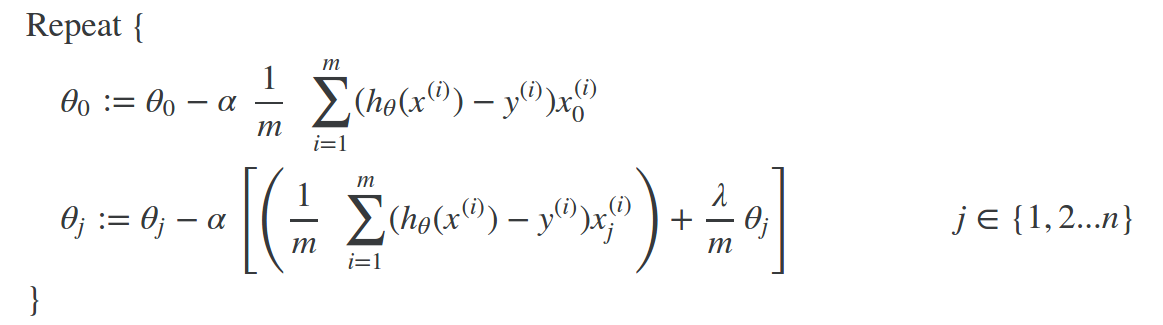
Cost function:

****

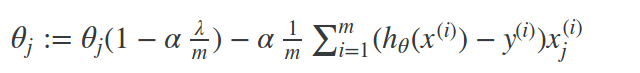
**-** The lambda is the regularization parameter . It determines how much the costs of our theta parameters are inflated.

- Using th cost function with the extra summation, we can smooth the output of your hypothesis function to the reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting.

**-** Gradient Descent:



with some manipulation our update rule can also be represented as:

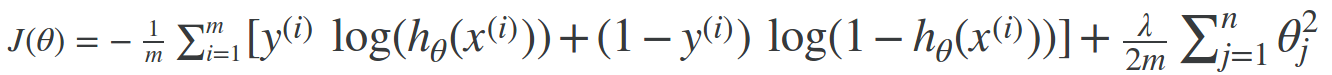


- The first term will always be less than 1. Intuitively you can see it as reducing the value of theta by some amount on every update.

- Notice that the second term is now exactly the same as it was before.

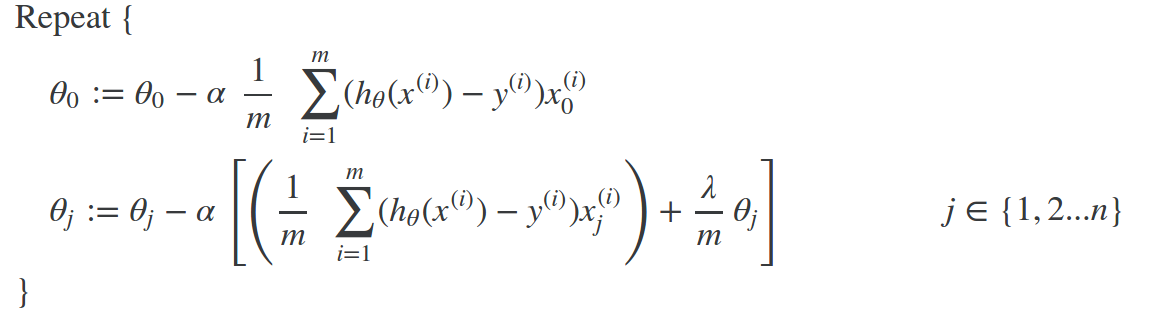
**Regularized Logistic Regression:**

- Cost function:



**=>** The second term means to explicitly exclude the bias term theta0.

- Gradient Descent:

 **=>** This is identical to gradient descent function presented for linear regression.

**Week 4: Neutral Network: Representation**

**1. Non-linear Hypotheses**

**-** Neural networks offers an alternate way to perform machine learning when we have complex hypotheses with many features.

- Neural networks are limited imitations of how our brains work. They’ve had big recent resurgence because of advances in computer hardware.

- There is evidence that the brains uses only one”learning algorithm” for all its difference functions. Scientists have tried cutting (in an animal brain) the connection between the ears and the auditory cortex and rewriting the optical nerve with the auditory cortex to find that the auditory cortex literally learns to see.

=> This principle is called “neuroplasticity” and has many examples and experimental evidence.

**2. Model Representation:**

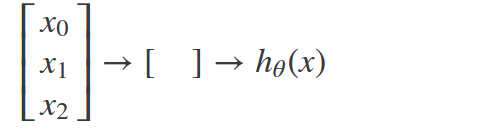
- At a very simple level, neurons are basically computational unit that take input (dendrites) as electrical input (spikes) that are channeled to outputs (axons).

- In our model, our dendrites are like the input features x1,x2,…,xn and the output is the result of our hypothesis function:

+ In this model our x0 input node is sometimes called the “bias unit”. It always equal to 1.

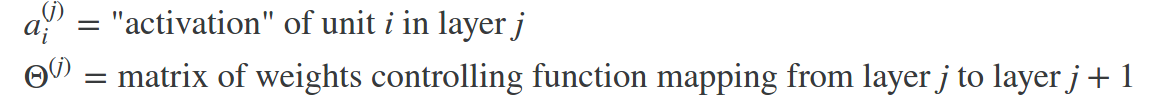
+ In neutral networks, we use the same logistic function as in classification and call it sigmoid (logistic) activation function.

+ Our “theta” parameters are sometimes instead called “weights” in the neutral networks model.

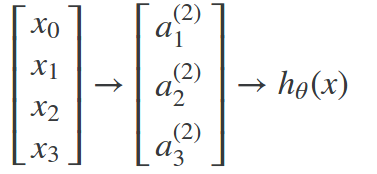


- Intermediate layers of nodes between the input and output layers called the “hidden layer”.

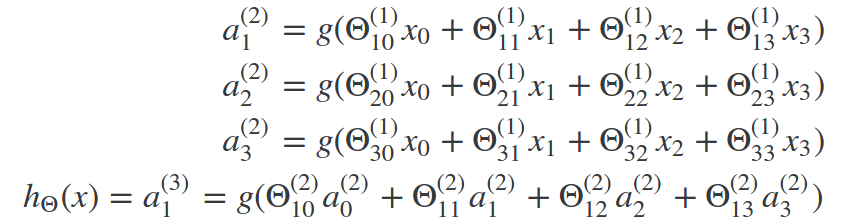
- We label these intermediate or “hidden” layer nodes and call them “activation units”:



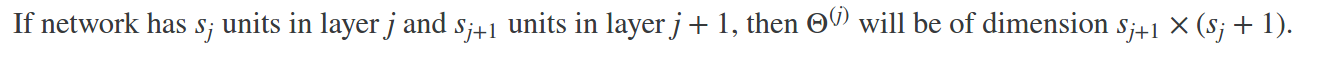
If we had one hidden layer, it would look visually something like:



=> The values for each of the “activation” node is obtained as follows:

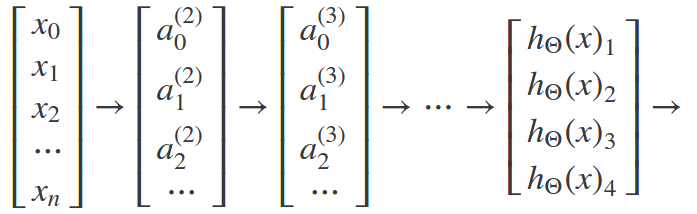


- Each layer gets its own matrix of weights, theta(j). The dimension of these matrices of weights is determined as follow:

=> The +1 comes from the addition in theta(j) of the “bias nodes”, x0 and theta0(j). In other words the output nodes will not include the bias node while the inputs will.

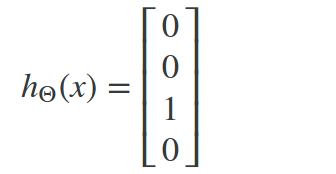
**2. Multiclass classification:**

**-** To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of your final resulting classes:



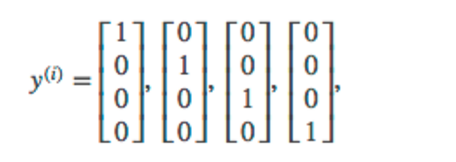
- Our final layer of nodes, when multiplied by its theta matrix, will result in another vector, on which we apply the g() logistic function to get a vector of hypothesis values.

- Our resulting hypothesis for one set of inputs may look like:



- In which case our resulting class is third one down.

=> We can define our set of resulting classes as y:



=> Our final value of our hypothesis for a set of inputs will be one of elements in y.

**Week 5: Neutral Networks: Learning**

**1. Cost Function:**

- Let’s first define a few variables that we need to use:

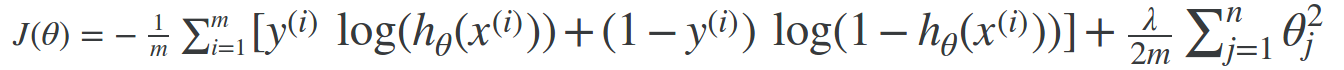
a) L = total number of layer in the networks

b) sl = number of units (not counting as bias unit) in layer I

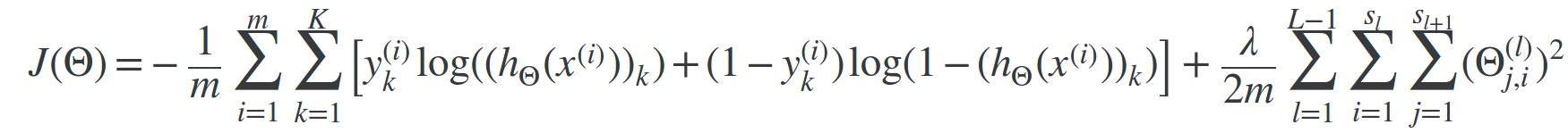
c) K = number of output units/classes

- Recall that in neural networks, we may have many output nodes. We denote htheta(x)k as being a hypothesis that results in the kth output. Our cost function for neutral networks is going to be a generalization of one we used for logistic regression.

- Recall that the cost function for regularized logistic regression was:



- For neural networks, it is going to be slightly more complicated:



=> We have added a few nested summations to account for our multiple output node. In the first part of the equation, between the square brackets, we have an additional nested summation that loops through the number of output nodes.

- In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

**Note:**

- The double sum simply adds up to the logistic regression costs calculated for each cell in the output layer.

- The tripple sum simple adds up the squares of all the individual theta in the entire network.

- The I in the triple sum does not refer to training examples I.

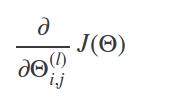
**2. Backpropagation Algorithm:**

- “Backpropagation” if neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression.

- Our goal is to compute: min J(theta)

=> That is, we want to minimize our cost function J using an optimal set of parameters in theta.

- In this section we’ll look at the equations we use to compute the partial derivative of J(theta):



- In back propagation we’ll going to compute for every node:

*δ*(*l*)*j =* “error” of node j in the layer I.

- Recall that *a*(*l*)*j* is activation node j in layer l.

- For the last layer, we can compute the vector of delta values with:



Where L is our total number of layers and a(L) is the vector of outputs of the activation units for the last year.

- To get the delta values of the layers before the last layer, we can use equation that steps us back right to left:

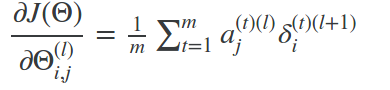


- The g-prime, which is derivative of the activation function g evaluated with the input values given by z(i).

- The g-prime derivative terms can also be written out as:



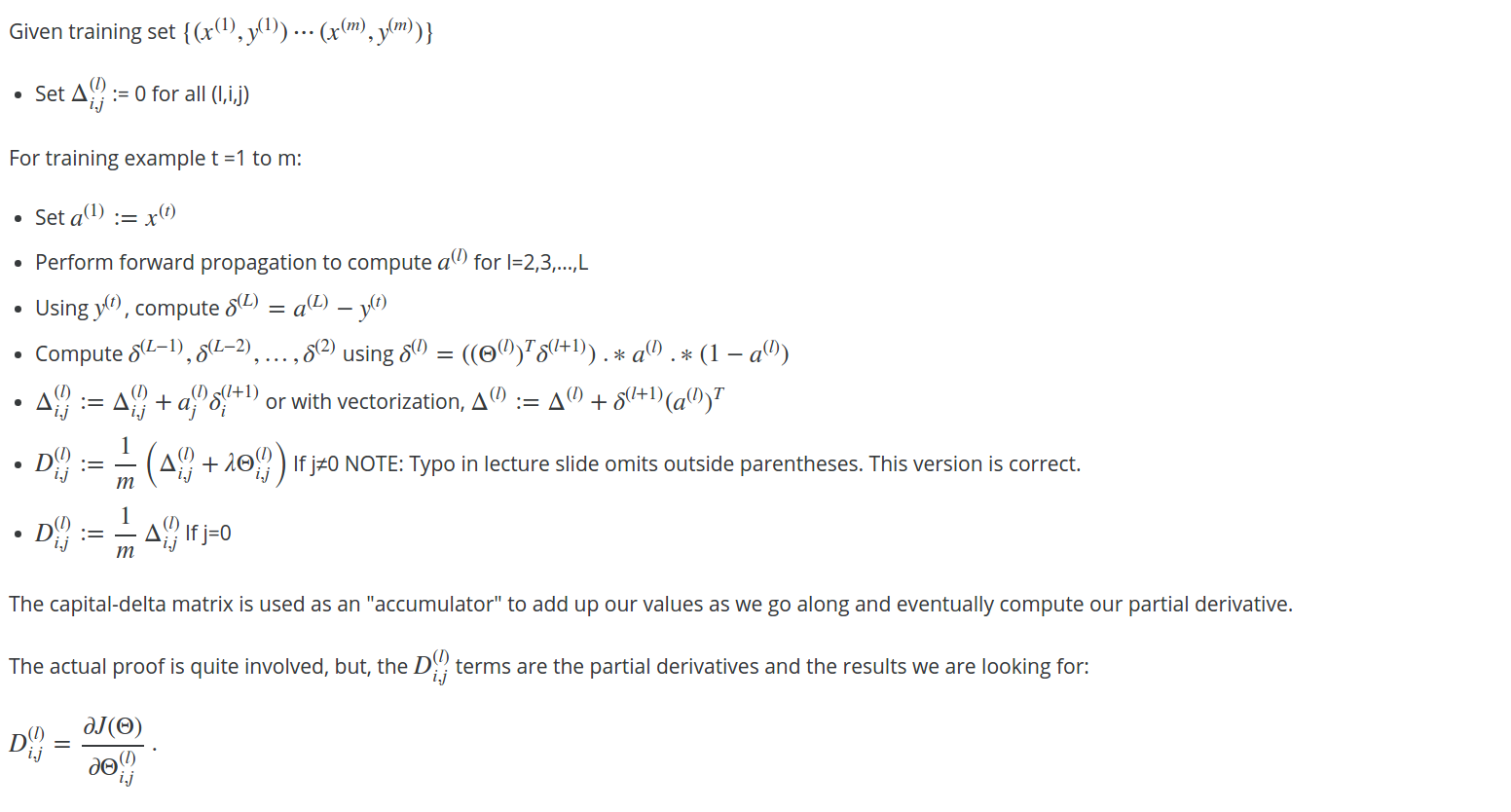
Then, we can compute the partial derivative terms by multiplying our activation values and our error values for each training example t:



=> This however ignores regularization, which we’ll deal with later.

**Note:** The process produces a gradient term for every element in theta.

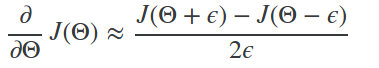
**Main algorithm:**



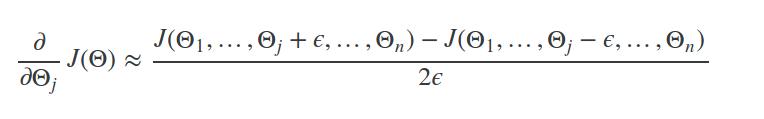
**3. Gradient checking:**

**-** Gradient checking will assure that our backpropagation works as intended.

- We can approximate the derivative of our cost function with:



- With multiple theta matrices, we can approximate the derivative with respect to theta(j) as follows:

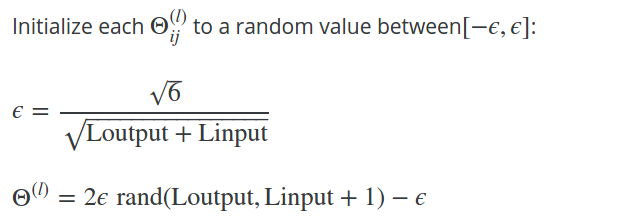
**-** A good small value for epsilon guarantees the math above to become true. If the value be much smaller, may we will end up with numerical problems. The professor Andrew usually uses with the value epsilon = 10e-4.

=> Once we’ve verified **once that your backpropagation algorithm is correct, then you don’t need to compute gradApprox again** (the code to compute gradApprox is very slow).

**4. Random Initialization:**

- Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly.

- Instead we can randomly initialize our weights:



**Note:** L(output) = number of outputs, L(input) = number of input.

**Week 6: Advice for Applying Machine Learning (Very very important)**

**1. Deciding what to try next:**

- Errors in your predictions can be troubleshooted by:

+ Getting more training examples

+ Trying smaller set of features

+ Increasing lambda

Or:

+ Trying additional features

+ Tying polynomial features

+ Decreasing lambda

=> **Do not jut pick one of these avenues at random.** We’ll explore techniques for choosing one of the above solutions in the following sections.

**2. Evaluating a Hypothesis:**

- A hypothesis may have low error for the training examples but still be inaccurate (because of overfitting).

- With a given dateset of training example, we can split up data into 2 sets: **A training set and a test set.**

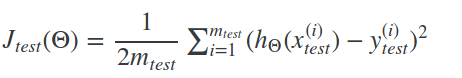
- The new procedure using the two sets is then:

1) Learning theta and minimize J(theta)train using the training set.

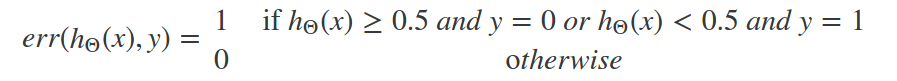
2) Compute the test set erro J(theta)test.

**The test set error:**

- For linear regression:

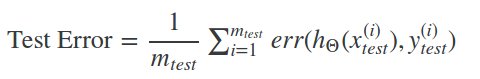


- For classification – misclassification error (aka 0/1 misclassification error):



=> this give use a binary 0 or 1 error result based on a misclassification.

- The average test error for the test set is:



=> This give us the proportion of the test data that was misclassified.

**Model Selection and Train/Validation/Test Sets:**

- Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis.

- The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than any other data set.

=> In order to choose the model of your hypothesis, you can test **degree of polynomial (d)** and look at the error result.

**Without the Validation Set:**

1) Optimize the parameters in theta using the training set for each polynomial degree.

2) Find the polynomial degree d with the least error using the test set.

3) Estimate the generation error also using the test set.

=> In this case, we have trained one variable, d, or the degree of polynomial, using the test set. This cause our error value to be greater of any other set of data.

**Use of the CV Set:**

- To solve this, we can introduce a third set, the **Cross Validation Set,** to serve as an intermediate set that we can train d with. Then our test set will give us an accurate, non-optimistic error.

- One example way to break down our dataset into the three sets is: Training/Cross validation/test set: 60/20/20.

=> We can now calculate three separate error values for the three different sets:

1) Optimize the parameters in theta using the training set for each polynomial degree.

2) Find the polynomial degree d with the least error using the cross validation set.

3) Estimate the generalization error using the test set.

=> The degree of the polynomial d has not been trained using the test set.

**Note:**

- This method presumes we do not use the CV set for the regularization.

- Be aware that using the CV set to select ‘d’ means that **we cannot use it for the validation curve process of setting lambda value).**

**3. Diagnosing Bias vs. Variance**

- In the section we examine the relationship between the degree of the polynomial d and the underfiting or overfitting of our hypothesis:

+ We need to distinguish whether bias or variance is the problem contributing to bad prediction.

+ High bias is underfitting and high variance is overfitting. We need to find a golden mean between these two.

- The training error will tend to decrease as increase the degree of the polynomial.

- At the same time, the cross validation error will tend to decrease as we increase d up to a point, and then it will increase as d increased, forming a convex curve.

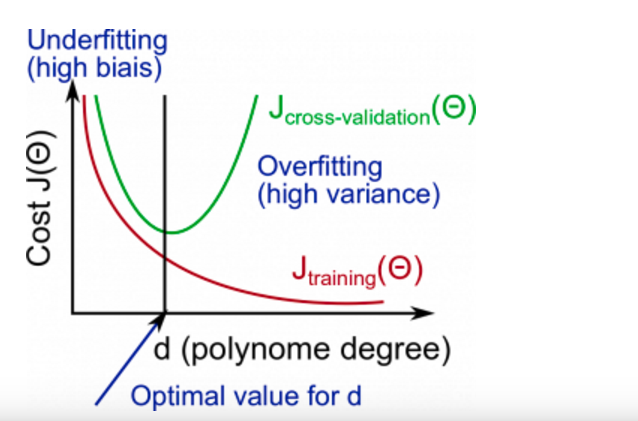
- **High bias (underfitting):**



**- High variance (overfitting):**

****

**=>** The is represented in the figure below:

****

**Regularization and Bias/Variance**

- Instead of looking at the degree d contributing to bias/variance, now we look at the regularization parameter lambda.

+ **Low lambda:**

****

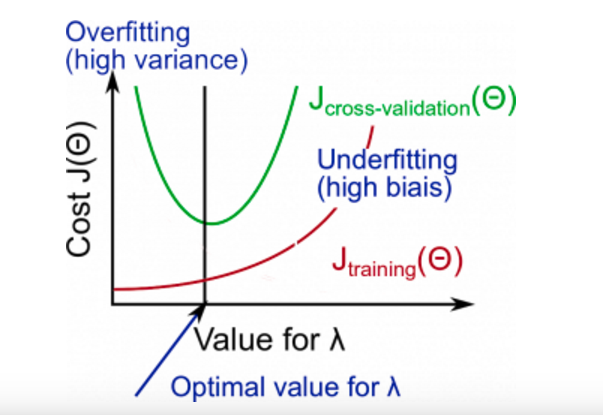
**+ Intermediate lambda:**

****

**+ Large Lambda:**

****

**=>** The figure below illustrates the relationship between lambda and the hypothesis:



=> In order to choose the model and the regulariztion lambda, we need:

1) Create a list of lambda (lambda = {0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10).

2) Select a lambda to compute.

3) Create a model set like degree of the polynomial or others.

4) Select a model to learn theta.

5) Learn the parameter theta for the model selected, using Jtrain(theta) with lambda selected.

6) Compute the train error using the learned theta (computed with lambda) on the Jtrain(theta) without regularization or lambda = 0.

7) Compute the cross validation error using the learned theta (computed with lambda) on the Jcv(theta) without regularization or lambda = 0.

8) Do this for the entire model set and lambdas, then select the best combo that produces the lowest error on the cross validation set.

9) Now if you need visualize to help you understand your decision, you can plot to the figure like above with: (lambda x CostJtrain(theta)) and (lambda x CostJcv(theta)).

10) Now, unsing the best combo theta and lambda, apply it on the Jtest(theta) to see if it has a good generalization of the problem.

11) To help decide the best polynomial degree and lambda to use, we can diagnose with the learning curves, that is the next subject.

**4. Learning curve**

**-** Training 3 examples will easily have 0 errors because we can always find a quadratic curve that exactly touches 3 points:

+ As the training set gets larger, the error for a quadratic function increases.

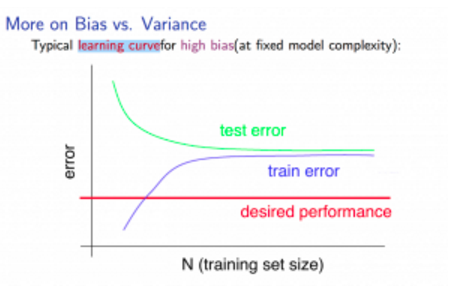
+ The error value will plateau out after a certain m, or training set size.

**With high bias:**

**- Low training set size:** Cause Jtrain(theta) to be low and Jcv(theta) to be high.

- **Large training set size:** Causes both Jtrain and Jcv(theta) to be high with Jtrain(theta) ~= Jcv(theta).

=> If a learning algorithm is suffering from high bias, **getting more training data will not (by itself) help much.**

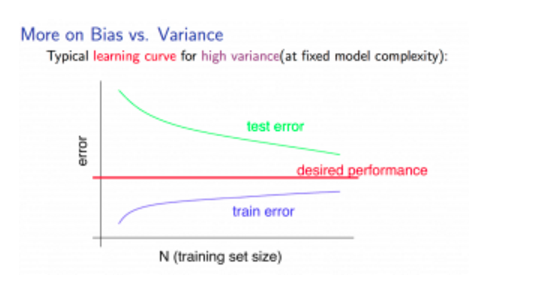


**With high variance:**

**- Low training set size:** Jtrain(theta) will be low and Jcv(theta) will be high.

- **Large training set size:** Jtrain(theta) increases with training set size and Jcv(theta) continues to decrease without leveling off. Also, Jtrain(theta) < Jcv(theta) but the difference between them remains significant.

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

****

**5. Deciding what to do next revisted**

**- Fixes high variance:**

+ Getting more training examples.

+ Trying smaller sets of features.

+ Increasing lambda.

- **Fix high bias:**

+ Adding features.

**+** Adding polynomial features.

+ Decreasing lambda.

**Diagnosing Neural Networks**

**-** A neural network with fewer parameters is prone to **underfitting.** It is also **computationally cheaper.**

**-** A large neural network with the parameters is prone to **overfititng**. It is also **computational expensive**. In this case you can use regularization (increase lambda) to address the overfitting.

=> Using a single hidden layer is a good starting default. You can train your neural network on a number of hidden layers using your cross validation set.

**6. Model Selection**

**-** Choosing M the order of polynomial. How can we tell which parameters theta to leave in the model (known as “model selection”)?

+ Get more data (very difficult).

+ Choose the model with best fits the data without overfitting (very difficult).

**+ Reduce the opportunity to overfitting through regularization.**

**Bias: Approximation error (difference between expected value and optimal value)**

**-** High Bias = UnderFitting (BU).

- Jtrain(theta) and Jcv(theta) both will be high and Jtrain(theta) ~= Jcv(theta).

**Variance: estimation error due to finite data**

- High Variance = OverFitting (VO).

- Jtrain(theta) is low and Jcv(theta) >> Jtrain(theta).

**Intuition for the bias-variance trade-off:**

- Complex model => sensitive to data => much affected changes in X => high variance, low bias.

- Simple model => more rigid => does not change as much with changes in X => low variance, high bias.

=> **One of the most important goals in learning:** Finding a model that is just right in the bias-variance trade-off.

**Regularization effects:**

**-** Small values of lambda allow model to become finely tuned to noise leading to large variance => overffiting.

- Large values of lambda pull weight parameters to zero leading to large bias => underfititng.

**Model Complexity Effects:**

**-** Lower-order polynomial (low model complexity) have high bias and low variance. In this case, the model fits poorly consistently.

- Higher-order polynomial (high model complexity) fit the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.

=> In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

**7. Machine Learning System Design**

**Prioritizing What to Work On**

- Different ways we can approach a machine learning problems:

+ Collect lots of data.

+ Develop sophisticated features.

+ Develop algorithms to process your input in different ways.

=> It’s difficult to tell which of the options will be helpful.

**Error Analysis**

**-** The recommended approach to solving machine learning problems is:

**+ Start with a simple algorithm, implement it quickly, and test it early.**

**+ Plot learning curves to decide if more data, more features, etc. will help.**

**+ Error analysis: manually examine the errors on the examples in the cross validation set and try to spot a trend.**

=> It’s important to get error results as a single, numerical value. Otherwise, it is difficult to assess your algorithm’s performance.

- You may need to process your input before it is useful. For example, if your input is a set of words, you may want to treat the same word with different forms (fail/failing/failed) as one word, so must use “stemming software” to recognize them all as one.

**8. Error Metrics for Skewed Classes**

**-** It is sometimes difficult to tell whether a reduction in error is actually an improvement of the algorithm.

- For example: In predicting a cancer diagnoses where 0.5% of the examples have cancer, we find our learning algorithm has a 1% error. However, if we were to simply classify every single example as a 0, then our error would reduce to 0.5% even though we did not improve the algorithm.

=> This usually happens with **skewed classes**; that is, when our class is very rare in the entire data set.

- For this we can use Precision/Recall:

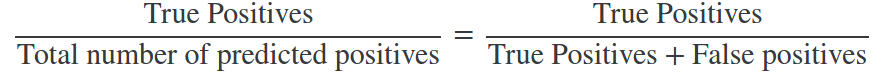
+ Predict :1, Actual: 1 => True positive.

+ Predict :1, Actual: 0 => False positive.

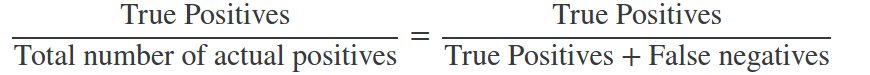
+ Predict :0, Actual: 0 => True negative.

+ Predict :1, Actual: 1 => False negative.

**Precision:** All of patients are predicted where y = 1, what fraction actually has cancer?



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



=> Those two matrices give us a better sense of how our classifier is doing. We want both precision and recall to be high.

- In the example at the beginning of the section, if we classify all patients as 0, recall will be 0 => we can quickly see it has worse recall.

**Trading Off Precision and Recall**

**-** We might want to confident prediction of two classes using logistic regression. One way is to increase our threshold:

+ Predict 1 if h(theta) >= 0.7

+ Predict 0 if h(theta) < 0.7

=> This way, we only predict cancer if the patient has a 70% chance. Doing this, we will have **higher precision but lower recall** (refer to definitions in the previous section).

- In the opposite example, we can lower our threshold (0.3) and get a very safe prediction. This will cause **higher recall but lower precision.**

=> In oder to turn these two metrics into one single number, we can take the **F score**:



=> In order for the F score to be large, both precision and recall must be large.

- We want to train precision and recall on the **cross validation set** so as not to bias our test set.

**8. Data for Machine Learning**

- How much data should we train on? **In certain cases, an “inferior algorithm”, if given enough data, can outperform a superior algorithm with less data.**

- We must choose our features to have **enough** information. A useful test is: Given input x, would a human expert be able to confidently predict y?

**Rationale for large data:** If we have a low bias algorithm (many features or hidden units making a very complex function), then the larger training set we use, the less we will have overfitting (and more accurate the algorithm will be on the test set).

**Week 7: Support Vector Machine (SVM)**

**1. Optimization Object**

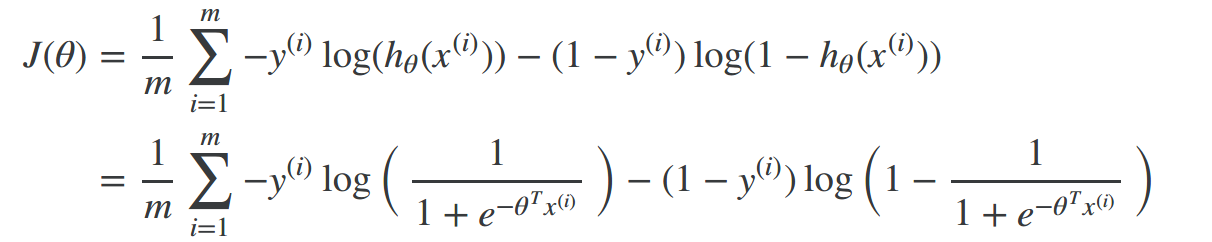
- The Support Vector Machine (SVM) is yet another type of supervised machine learning. **It is sometimes cleaner and more powerful.**

- Recall that in logistic regression, we use the following rules:

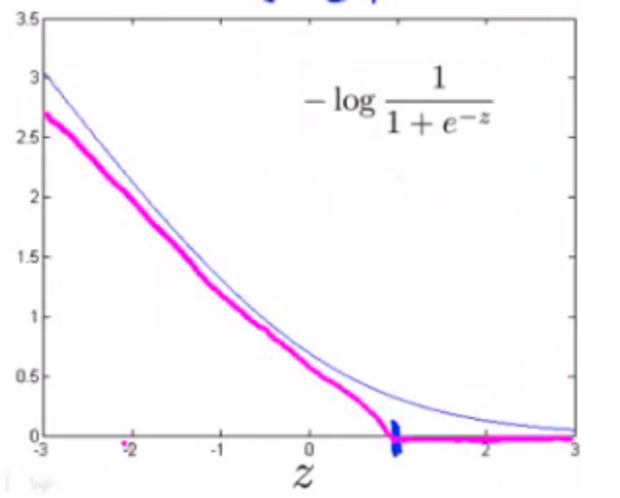
+ If y = 1, then h(theta)x ~= 1 and theta\*x >> 0.

+ If y = 0, then h(theta)x ~= 0 and theta\*x << 0.

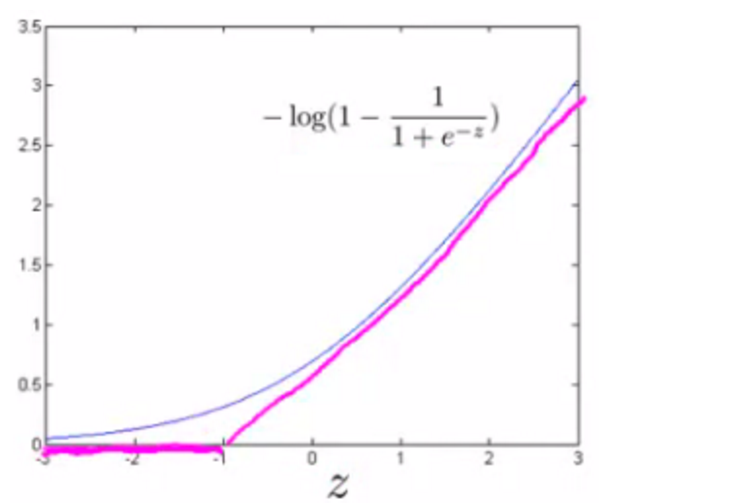
=> Recall the cost function for (unregularized) logistic regression:



- To make support vector machine, we will modify the first term of the cost function, so that when theta\*x (from now, we shall refer to this as z) is greater than 1, it outputs 0. Furthermore, for value of z less than 1, we shall use a straight deceasing line instead of sigmoid curve (in the literature, this is call a hinge loss function).



Similarly, we modify the second term of the cost function. So that when z is less than -1, it outputs 0. We also modify it so that for values of z greater than -1, we use a straight line instead of the sigmoid curve.



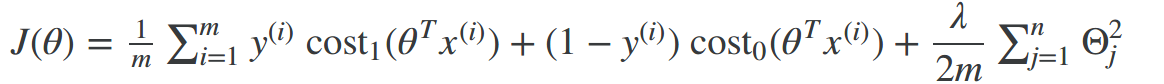
=> We shall denotes these as cost1(z) and cost0(z) (respectively, not that cost1(z) is the cost for classifying when y = 1, and cost0(z) is the cost for classifying when y = 0), and we may define them as follows (where k is an arbitrary constant defining the magnitude of the slope of the line):

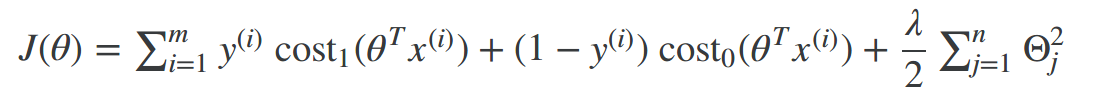
z = theta\*x

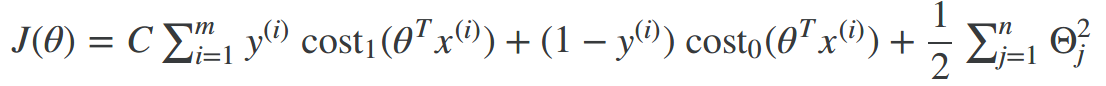
cost0(z) = max(0, k(1+z))

cost1(z) = max(0, k(1-z))

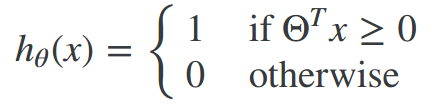
=> Recall the full cost function from (regularized) logistic regression:

- We can optimize this a bit by multiplying this by m (thus removing the m factor in the denominators). Note that this does not affect our optimization:

- Furthermore, convention dictates that we **regularize using a factor C, instead of lambda**, like so:

=> This is equivalent to multiplying the equation by C = 1/lambda, and thus results in the same values when optimized. Now, when we wish to regularize more (that is, reduce overfititng), we **decrease** C, and when we wish to regularize less (that is, reduce underfitting), we **increase** C.

- Finally, note that the hypothesis of the Support Vector Machine is not interpreted as the probability of y being 1 or 0 (as it is for the hypothesis of logistic function). Instead, it outputs either 1 or 0 (in technical terms, it is discriminant function).



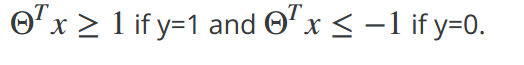
**2. Large Margin Intuition**

- A useful way to think about Support Vector Machines is to think of them as Large Margin Classifiers:

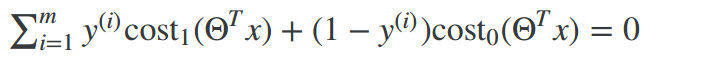
+ If y = 1, we want theta\*x >= 1 (not just >= 0)

+ if y = 0, we want theta\*x <= -1 (not just < 0)

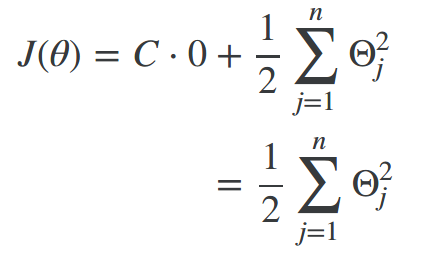
- Now when we set our constant C to a very large value, our optimizing function will constrain theta such that the equation A (the summation of the cost for each example) equals 0. We impose the following constrain on theta:



=> If C is very large, we must choose theta parameters such that A = 0:



=> This reduces our cost function to:



- Recall the decision boundary from logistic regression (the line separating the positive and negative examples). In SVMs, the decision boundary has the special property that as far away as possible from both the positive and negative examples.

- The distance of the decision boundary to the nearest example is called **margin.** Since SVMs maximize this margin, it is often called a **Large Margin Classifier.**

- The SVM will separate the negative and positive examples by a **large margin.** This large margin is only achieved when **C is very large.**

- Data is linearly separable when a straight line can separate the positive and negative examples. If we have **outlier** examples that we don’t want to affect the decision boundary, then we can **reduce** C.

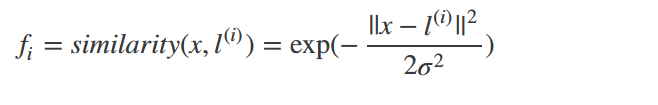
=> Increasing and deceasing C is similar to respectively decreasing and increasing lambda, and can simplify our decision boundary.

**3. Kernel:**

- Kernel allows us to make complex, non-linear classifiers using Support Vector Machines.

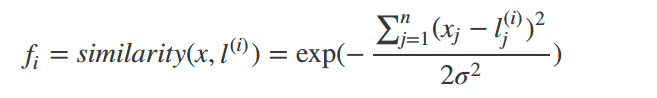
- Given x, compute new features depending on proximity to landmarks l(1), l(2), l(3).

- To do this, we find the “similarity” of x and some landmark l(i):

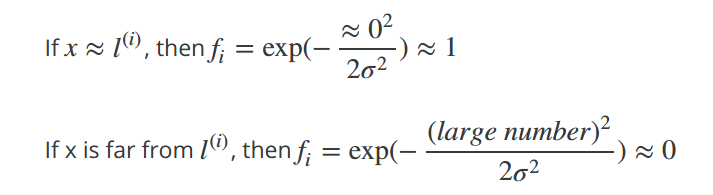


=> This similarity function is called a Gaussian Kernel. It is a specific example of a kernel.

- The similarity function can also be written as follows:

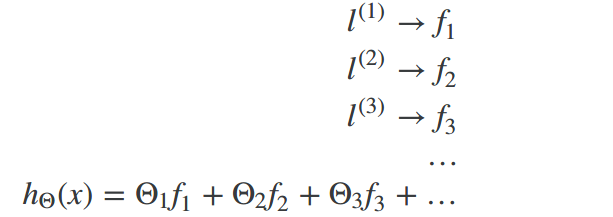


- There are a couples properties of the similarity function:



=> In other words, if x and the landmark are close, then the similarity will be close to 1, and if x and the landmark are far away from each other, the similarity will be close to 0.

- Each landmark give us the features in our hypothesis:

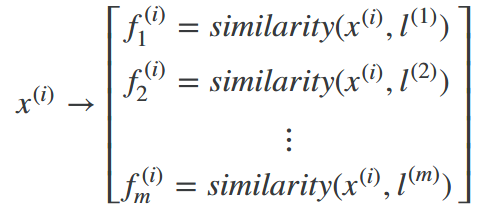


=> Sigma^2 is a parameter of the Gaussian Kernel, and it can be modified to increase the **drop-off of our feature fi.** Combined with looking at the values inside theta, we can choose these landmarks to get the general shape of the decision boundary.

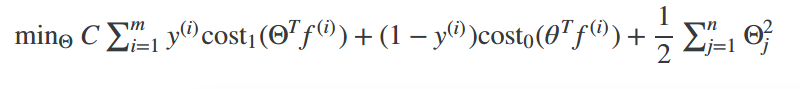
- One way to get the landmark is to put them in the **exact same** locations as all the training examples. This give us m landmarks, with one landmark per training example.

- Given example x:

=> This give us a “feature vector”, f(i) of all our features for example x(i). We may also get f(0) = 1 to correspond with theta0. Thus given training example x(i):



- Now to get the parameters theta we can use SVM minimization algorithm but with f(i) substituted in for x(i):

=> Using kernels to generate f(i) is not exclusive to SVMs and may also be applied to logistic regression. **However, because of computetional optimizations** on SVMs, kernel combined with SVMs is much faster then with other algorithms, so kernel are almost always found combined only with SVMs.

**Choosing SVM Parameters**

- Choosing C (= 1/lambda):

+ If C is large, then we get higher variance/lower bias.

+ If C is small, then we get lower variance/higher bias.

- The other parameter we must choose is sigma^2 from the Gaussian Kernel function:

+ With a large sigma^2, the features fi vary more smoothly, causing higher bias and lower variance.

+ With a small sigma^2, the features fi vary less smoothly, causing lower bias and higher variance.

**4. Using an SVM**

- There are lots of good SVM libraries already written. A. Ng often uses “liblinear” and “libsvm”. In practical application, you should use one of these libraries rather than rewrite the functions.

- In practical application, the choices you do need to make are:

+ Choice of parameter C

+ Choice of kernel (similarity function)

+ NO kernel (“linear” kernel) – gives standard linear classifer

+ Choose when n is large and when m is small

+ Gaussian Kernel (above) – need to choose sigma^2.

+ Choose when n is small and m is large.

- The library may ask you to provide kernel function.

**NOTE:**

**-** Do **perform feature scaling** before using the Gaussian Kernel.

- Not **all the similarity functions are valid kernels.** They mush satisfy “Mercer’s Theorem” which guarantees that the SVM package’s optimizations run correctly and do not diverge.

- You want to train C and the parameters for the kernel function using the training and cross-validation datasets.

**Multi-class Classification**

**-** Many SVM libraries have multi-class classification built-in.

- You can use one-vs-all method just like we did for logistic regression, where y = 1,2,3,...K for theta(1), theta(2),…, theta(K). We pick class I with the largest theta(i)\*x.

**Logistic Regression vs. SVMs**

**-** If **n is large (relative to m)**, then use **logistic regression, or SVM** without a kernel (the “linear kernel”).

- If **n is small and m is intermediate,** then use **SVM with a Gaussian Kernel.**

- If **n is small and m is large,** then **manually create/add more features, then use logistic regression or SVM without a kernel.**

=> In the first case, we don’t have enough examples to need a complicated polynomial hypothesis. In the second example, we have enough examples that we may need a complex non-linear hypothesis. In the last case, we want to increase our features so that logistic regression becomes applicable.

**Note:** A neural network is likely to work well for any of these situations, but may slower to train.

**Week 8: Clustering**

**1. Unsupervised Learning: Introduction**

- Unsupervised learning is contrasted from supervised learning because it uses an unlabeled training set rather than labeled one.

- In other words, we don’t have the vector y of excepted results, we only have a dataset of features where we can find structure.

- Clustering is good for:

+ Market segmentation.

+ Social network analysis

+ Organizing computer clusters

+ Astronomical data analysis

**2. K-Means Algorithm**

- The K-Means Algorithm is the most popular and widely used algorithm for automatically grouping data into coherent subsets.

1) Random initialize two points in the data set called the cluster centroids.

2) Cluster assignment: assign all examples into one of two groups base on which cluster centroid the example is closest to.

3) Move centroid: Compute the averages for all the points inside each of two cluster centroid groups, then move the cluster centroid points to those averages.

4) Re-run (2) and (3) until we have found our clusters.

- Our main variables are:

+ K (number of clusters)

+ Training set x(1), x(2),...x(m)

+ Where x(i) belongs to Rn

**Note** that we **will not use the x0 = 1 convention.**

**The algorithm**

1) Random initialize K cluster centroids mu(1), mu(2), … , mu(K)

2) Repeat:

for I = t to m:

c(i):= index (from 1 to K) of cluster centroid closest to x(i)

for k = 1 to K:

mu(k):= average (mean) of points assigned to cluster k

- **The first for-loop is the “Cluster Assignment” step**. We make a vector c where c(i) represents the centroid assigned to example x(i).

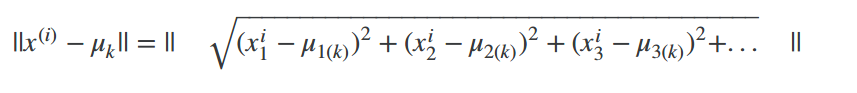
+ We can rewrite the operation of the Cluster Assignment step more mathematically as follow:

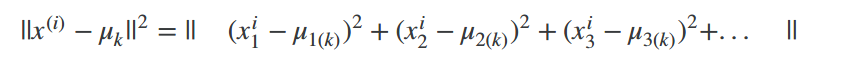


=> That is, each c(i) contains the index of the centroid that has minimal distance to x(i).

+ By convention, we square the right-hand-side, which make the function we are trying to minimize more sharply increasing. It is mostly just a convention. But a convention that helps reduce the computation load because the Euclidean distance requires a square root but it is canceled.

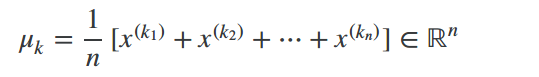
+ Without the square:

+With the square:

...so the square convention severs two purposes, minimize more sharply and less computation.

- The **second for-loop is the “Move Centroid”** where we move each centroid to the average of its group.

+ Move formally, the equation for this loop is as follows:



where each x(k) are training examples assign to group mu(k).

+ If you have a cluster centroid with 0 points assigned to it, you can randomly re-initialize that centroid to a new point. You can also simply eliminate that cluster group.

- After a number of iterations the algorithm will converge, where new iterations do not affect the clusters.

- **Note on non-separated cluster:** Some datasets have no real inner separation or nature structure. K-means can still evenly segment your data into K subsets, so can still be useful in this case.

**3. Optimization object**

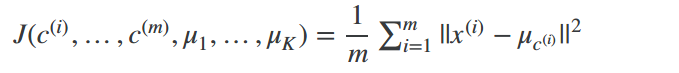
- Recall some of the parameters we used in our algorithm:

+ c(i) = index of cluster (1,2,…,K) to which examples x(i) is currently assigned.

+ u(k) = cluster centroid k (u(k) belong to Rn).

+ uc(i) = cluster centroid of cluster to which example x(i) has been assigned.

=> Using these variables we can define our cost function:



- Our **optimization objective** is to minimize all our parameters using the above cost function:



=> That is, we are finding all the values in set c, representing all our clusters, and u, representing all our centroid, that will **minimize the average of the distances** of every training example to its corresponding cluster centroid.

- The above cost function is often called the **distortion** of the training examples.

- In the **cluster assignment step,** our goal is to minimize J(…) with c(1), … ,c(m) (holding u1, … , uk fixed).

- In the **move centroid step,** our goal is to minimize J(…) with u1, … , uk.

=> With k-means, **it is not possible for the cost function to sometimes increase.** It should always descend.

**3. Random Initialization**

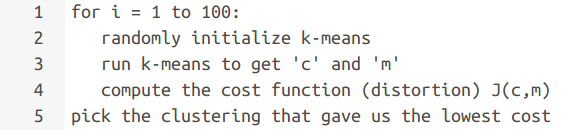
- There’s one particular recommended method for randomly initializing your cluster centroid.

1. Have K << m. That is, make sure the number of your cluster is less than the number of your training examples.

2. Randomly pick K training examples (not mention in the lecture, but also be sure the selected examples are unique).

3. Set u1, …. , uk equal to these K examples.

- K-means **can get stuck in local optima.** To decrease the chance of this happening, you can run the algorithm on many different random initializations. In cases where K<10 it is strongly recommended to run a loop of random initializations.



**4. Choosing the Number of Clusters**

**-** Choosing K can be quite arbitrary and ambiguous.

- **The elbow method:** Plot the cost J and the number of clusters K. The cost function should reduce as we increase the number of cluster, and then flatten out. Choose K as the point where the cost function starts to flatten out. However, fairly often, the curve is **very gradual,** so there’s no clear elbow.

**Note:** J will **always** decrease as K is increased. The one exception is if k-means get stuck at bad local optimum.

- Another way to choose K is to observe how well k-mean performs on a downstream purpose. In other words, you choose K the proves to be most useful for some goal you’re trying to achieve from using these clusters.

**5. Dimensionality Reduction**

**Motivation I: Data Compression**

- We want to reduce the dimension of our features if we have a lot of redundant data.

- To do this, we find two highly correlated features, plot them, and make a new line that seems to describe both features accurately. We place all the new features in the single line.

=> Doing dimensionality reduction will reduce the total data we have to store in computer memory and will speed up our learning algorithm.

**Note:** In dimensionality reduction, we are reducing our features than our number of examples. Our variable m will stay the same size; n – the number of features each example from x(1) to x(m) carries, will be reduced.

**Motivation II: Visualization**

- It is not easy to visualize data that is more than three dimensions. We can reduce the dimensions of our data to 3 or less in order to plot it.

- We need to find new features, z1, z2 (and perhaps z3) that can effectively summarize all the other features.

**Example:** Hundreds of features related to country’s economic system may all be combined into one feature that you call “Economic Activity”.

**Principal Component Analysis Problem Formulation**

- The most popular dimensionality reduction algorithm is **Principal Component Analysis (PCA).**

**Problem formulation:**

- Given two features, x1 and x2, we want to find a single line that effectively describes both features at once. We then map our old features onto this new line to get a new single feature.

- The same can be done with three features, where we map them to a plane.

- The **goal of PCA is to reduce** the average of all the distances of every feature to the projection line. This is the projection error.

- Reduce from 2d to 1d: Find a direction (a vector u(1) belongs to Rn) onto which to project the data so as to minimize the projection error.

- The more general case is as follows: Reduce from n-dimension to k-dimension: Find k vectors u(1), u(2), … , u(k) onto which project the data so as to minimize the projection error.

**PCA is not linear regression**

- In linear regression, we are minimizing the squared error from every point to our predictor line. These are vertical distance.

- In PCA, we are minimizing the shortest distance, or shortest orthogonal distance, to our data points.

=> More generally, in linear regression we are taking all our examples in x and applying the parameters in theta to predict y. In PCA, we are taking a number of features x1, x2, … , xn and finding a closest common dataset among them. We are not trying to predict any theta weights to the features.

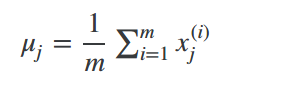
**Principal Component Analysis Algorithm**

- Before we can apply PCA, there is a data pre-processing step we must perform:

**Data preprocessing**

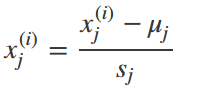
- Given training set: x(1), x(2), … , x(m)

- Preprocess (feature scaling / mean normalization):



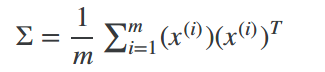
- Replace each xj(i) with x(i)j – uj

- If different features on different scales, scale features to have comparable range of values. Above, we first subtract the mean of each feature from the original feature. Then we scale all of the features:



=> We can define specifically what it means to reduce from 2d to 1d data as follows:

**1) Compute “covariance matrix”:**

****

=> This can be vectorized in Octave as: sigma = (1/m) \* X’ \*X

- We denote the covariance matrix with a capital sigma (which happens to be the same symbol for summation, confusingly – they present entirely different things).

- Note that x(i) has n\*1 vector, (x(i)^T is an 1\*n vector and X is a m\*n matrix (row-wise stored examples). The product of those will be n\*n matrix, which are the dimensions of sigma.

**2) Compute “eigenvectors” of covariance matrix sigma**

[U, S, V] = svd(Sigma);

- svd() is the “singular value decomposition”, a built-in Octave function.

- What we actually want out of svd() is the “U” matrix of the covariance matrix: U belongs to R(n\*n). U containts u(1), … , u(n) which is exactly what we want.

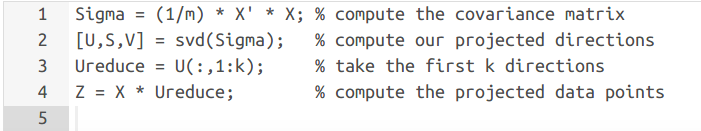
**3) Take the first k columns of the U matrix and compute z**

- We’ll assign the first k columns of U to a variable called “Ureduce”. This will be an n\*k matrix. We compute z with:



- UreduceZ(T) will have dimensions k\*n while x(i) will have dimension n\*1. The product will have dimension k\*1.

=> To summarize, the whole algorithm in octave is roughly:



**Reconstruction from Compressed Representation**

- If we use PCA to compress our data, how can we uncompress our data, or go back to our original number of features?

- To go from 1-dimension back to 2d we do:



- Note that we can only get approximation of our original data.

- It turns out that U matrix has the special property that it is a Unitary Matrix. One of the special properties of a Unitary Matrix is:



where the “\*” means “conjugate transpose”. Since we are dealing with real number here, this is equivalent to:



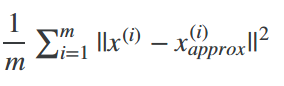
So we could compute the inverse and use that, but it would be waste of energy and compute cycles.

**Choosing the Number of Principal Components**

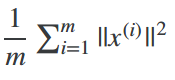
**-** How do we choose k, also called the number of principal components? Recall that k is the dimension we are reducing to.

- One way to choose k is by using the following formula:

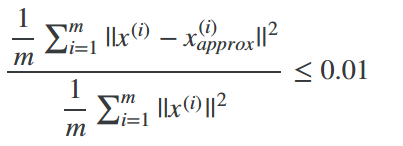
+ Given the average square project error:



+ Also given the total variation in the data:



+ Choose k to be the smallest value such that:



- In other words, the squared projection error divided by the total variation should be less than 1%, so that **99% of the variance is retained.**

**Advice for Applying PCA:**

**- The most common use of PCA is to speed up supervised learning.**

- Given a training set with a larger number of features, we can use PCA to reduce the number of features in each example of the training set.

- Application:

+ Compressions: Reduce space of data, speed up algorithm.

+ Visualization of data: Choose k = 2 or k = 3.

**- Bad use of PCA: Trying to prevent overfitting.** We might think that reducing the features with PCA would be an effective way to address overfitting. It might work, but is not recommended because it does not consider the values of our results. Using just regularization will be at least as effective.

**-** Do not assume you need to do PCA. Try your full machine learning algorithm without PCA first. Then use PCA if you find that you need.

**Week 9: Anomaly Detection**

**1. Problem Motivation**

- Just like in other learning problems, we are given a dataset x(1), x(2), … , x(n).

- We are then given a new example, xtest, and we want to know whether this new example is abnormal/anomalous.

- We define a “model” p(x) that tells us the probability the example is not anomalous. We also use a threshold epsilon as a dividing line so we can say which examples are anomalous and which are not.

- A very common application of anomaly detection is detecting fraud:

+ x(i) = features of user i’s activities

+ Model p(x) from the data

+ Identify unusual users by checking which have p(x) < epsilon

=> If our anomaly detector if flagging **too many** anomalous examples, then we need to **decrease** our threshold epsilon.

**Gaussian Distribution**

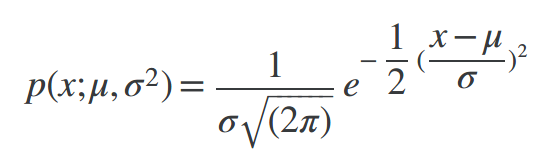
**-** The Gaussian Distribution is a familiar bell-shaped curve that can be described by a function N(u, sigma^2).

- Let x belongs to R. If the probability of x is Gaussian with mean u, variance sigma^2, then:

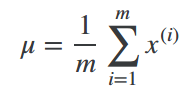


=> The Gaussian Distribution is parameterized by a mean and a variance.Mu, or u, describes the center of the curve, called the man. The width of the curve is described by sigma, called standard deviation.

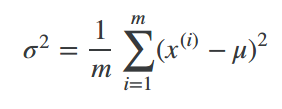
- The full function is as follows:



=> We can estimate the parameter u from a given dataset by simply taking the average of all the examples:



And we can estimate the other parameter, sigma^2 with our familiar squared error formula:



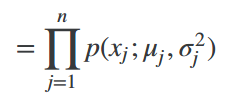
**2. Algorithm**

- Given a training set of examples, {x(1), … , x(m)} where each example is a vector, x belongs to Rn.



- In statistic, this is called an “independence assumption” on the values of the features inside training example x.

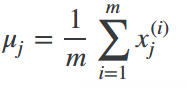
- More compactly, the above expression can be written as follows:

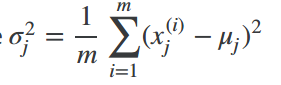


**The algorithm:**

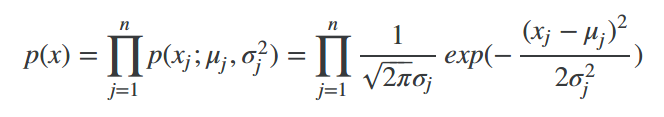
**-** Choose the features xi that you think might be indicative of anomalous examples.

- Fit parameters u1, … , un, sigma(1)^2, … , sigma(n)^2

- Calculate:

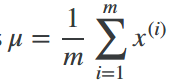
and:

- Given a new example x, compute p(x):



=> Anomaly if p(x) < epsilon.

- A vectorized version of the calculation for u is:



=> You can vectorize sigma^2 similarly.

**2. Developing and Evaluating an Anomaly Detection System**

**-** To evaluate our learning algorithm, we take some labeled data, categorized into anomalous and non-anomalous examples (y = 0 if normal, y = 1 if anomalous).

- Among that data, take a large proportion of good, non- anomalous data for the training set on which to train p(x).

- Then, take a smaller proportion of mixed anomalous and non-anomalous- examples (you will usually have many more non- anomalous examples) for your cross-validation and test sets.

- For example, we may have set where 0.2% of the data is anomalous. We take 60% of those examples, all of which are good (y = 0) for the training set. We then take 20% of the examples for the cross-validation set (with 0.1% of the anomalous examples) and another 20% from the test set (with another 0.1% of the anomalous).

- In other words, we split the data 60/20/20 training/CV/test and then split the anomalous examples 50/50 between the CV and test sets.

**Algorithm evaluation:**

- Fit model p(x) on the training set {x(1), … , x(m)}

- On a cross/text example x, predict:

+ If p(x) < epsilon (anomaly), then y = 1

+ If p(x) >= epsilon (normal), then y = 0

=> Possible evaluation metrics:

+ True positive, false positive, false negative, true negative.

+ Precision/recall

+ F1 score

**Note that we use the cross-validation set to choose parameter epsilon.**

**3. Anomaly Detection vs. Supervised Learning**

- When do we use anomaly detection and when do we use supervised learning?

- Use **anomaly detection** when:

+ We have a very small number of positive examples (y = 1 … 0-20 examples is common) and a large number of negative (y = 0) examples.

+ We have many different “types” of anomalies and it is hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we’ve seen so far.

- Use **supervised learning** when:

+ We have large number of both positive and negative examples. In other words, the training set is more evenly divided into classes.

+ We have enough positive examples for the algorithm to get a sense what new positives examples look like. The future positive examples are likely similar to ones in the training set.

**Choosing What Features to Use**

- The features will greatly affect how well your anomaly detection algorithm works.

- We can check that our features are **gaussian** by plotting a histogram of our data and checking for the bell-shaped curve.

- Some **transforms** we can try on an example features x that does not have the bell-shaped curve are:

+ log(x)

+ log(x + 1)

+ log(x + c) for some constant

+ sqrt(x)

+ x^(1/3)

=> We can play each of these to try and achieve the gaussian shape in our data.

- There is an **error analysis procedure** for anomaly detection that is very similar to the one in supervised learning.

- Our goal is for p(x) to be large for normal examples and small for anomalous examples.

- One common problem is when p(x) is similar for both type of examples. In this case, you need to examine the anomalous examples that are giving high probability in detail and try to figure out new features that will be distinguish the data.

- In general, **choose features that might take on unusually large on small values in the event of an anomaly.**

**4. Recommender System**

**Problem Formulation**

- Recommendation is currently a very popular application of machine learning.

- Say we are trying to recommend movies to customers. We can use the following definition:

+ n(u) = number of user

+ n(m) = number of movies

+ r(i,j) = 1 if user j has rated movie I

+ y(i,j) = rating given by user j to movie I (defined only if r(i,j) = 1)

**Content Base Recommendations**

- We can introduce two features, x1 and x2 which represents how much romance or how much action a movie may have (on a scale of 0-1).

- One approach is that we could do linear regression for every single user. For each user j, learn the parameter theta(j) belongs to R(3). Predict user j as rating movies I with (theta(j))^T\*x(i) stars.

+ theta(j) = parameter vector for user j

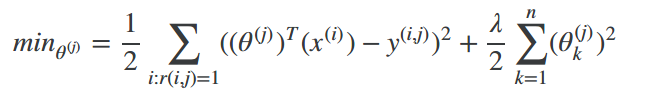
+ x(i) = feature vector for movie I

- For user j, movie I, predicted rating:



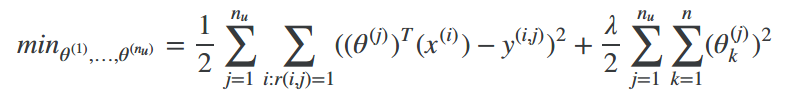
+ m(j) = number of movie rated by user j.

- To learn theta(j), we do the following:



=> This is our familiar linear regression. The base of the first summation is choosing all I such that r(i,j) = 1.

- To get the parameters for all our users, we do the following:



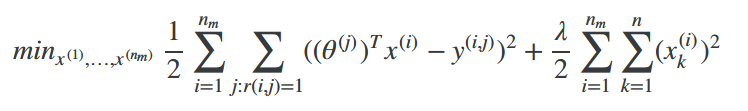
=> We can apply our linear regression gradient descent update using the above cost function. The only real difference is that we eliminate the constant 1/m.

**Collaborative Filtering**

- It can be very difficult to find features such “amount of romance” in a movie. To figure this out, we can use use feature finders.

- We can let the user tell us how much they like the different genres, providing their parameter vector immediately for us.

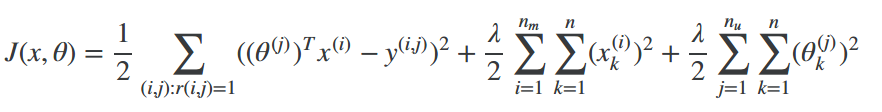
- To infer the features from given parameters, we use the squared error function with regularization over all the user:



=> You can also randomly guess the values for theta to guess the features repeatedly. You will actually converge to a good set of features.

**Collaborative Filtering Algorithm**

- To speed things up, we can simultaneously minimize our features and our paramers:



=> It looks very complicated, but we’ve only combined the cost function for theta and the cost function for x.

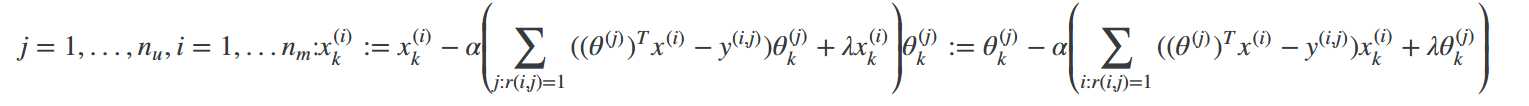
- Because the algorithm can learn them itself, the bias unit where x0 = 1 has been removed, therefore x and theta belong to Rn.

- There are the steps in the algorithm:

1) Initialize x and theta to small random values. This serves to break symmetry and ensures that the algorithm learns features x are different from each other.

2) Minimize J function, using gradient descent (or advanced optimization algorithm).

For every

3) For a use with parameters theta and a movie with (learned) feature x, predict a star rating of (theta)^T\*x.

**Vectorization: Low Rank Matrix Factorization**

**-** Given matrices X (each row containing features of a particular movie) and theta (each row containing the weights for those features for a given user), then the full matrix Y of all predicted rating of all movies by all users is given simply by:



- Predicting how similar two movies I and j can be done using the distance between their respective feature x. Specifically, we are looking a small value of



**Implementation Detail Mean Normalization**

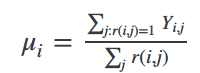
**-** If the ranking system for movies is used from the previous lectures, then new users (who have watched no movies), will be assigned new movies incorrectly. Specifically, they will assigned theta with all components equal to zero due to minimization of regularization term. That is, we assume that the new user will rank at movies 0, which does not seem intuitively correct.

- We rectify this problem by normalizing the data relative to the mean. First, we use a matrix Y to store data from previous ratings, where the ith row of Y is the ratings for the I movie and the jth column correspond to the ratings for the j user.

- We can now define a vector:

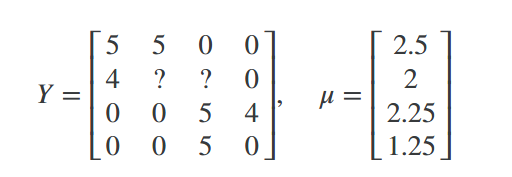


such that:

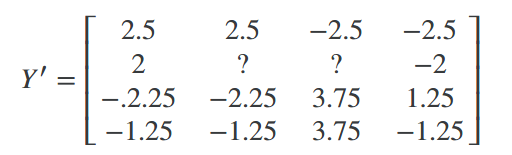


which is effectively the mean of the previous ratings for the I movie (where only movies that have been watched by user are counted). We now can normalize the data by subtracting u, the mean rating, from the actual ratings for each user (column in matrix Y):

+ As an example, consider the following matrix Y and the mean rating u:



=> The resulting Y’ vector is:



+ Now we must slightly modify the linear regression prediction to include the mean normalization term:



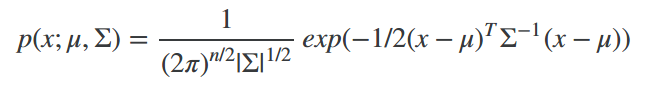
=> For a new user, the initial predict values will be equals to the u term instead of simply being initialized to zero, which is more accurate.

**5) Multivariate Gaussian Distribution**

- The multivariate gaussian distribution is an extension of anomaly detection and may (or may not) catch more anomalies.

- Instead of modeling p(x) separately, we will model p(x) all in one go. Our parameters will be:



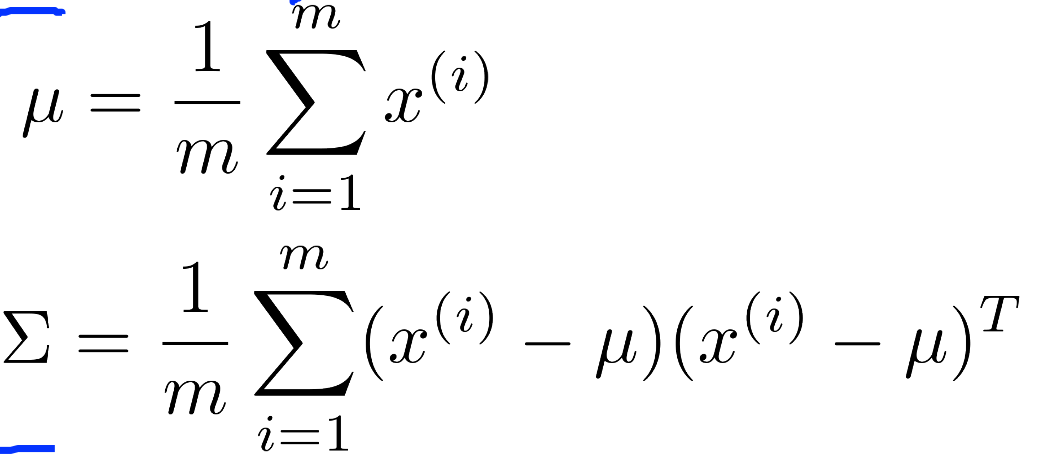


- The important effect is that we can model oblong gaussian contours, allowing us to better fit data that might not fit into the normal circular.

- Varying sigma changes the shape, width and orientation of the contours. Changing u will move the center of the distribution.

**Anomaly Detection using the Multivariate Gaussian Distribution**

- When doing anomaly detection with multivariate gaussian distribution, we compute u and sigma normally. We then compute p(x) using the new formula in the previous section and flag an anomaly if p(x) < epsilon.



- The original model for p(x) corresponds to a multivariate Gaussian where the contours of p(x;u,sigma) are axis-aligned.

- The muiltivariate Gaussian model can **automacially capture correlations between different features of x**.

- However, the original model maintains some advantages: it is computationally chaper (no matrix to invert, which is costly for large number of features) and it performs well even with small training set size (in multivariate Gaussian model, it should be greater than the number of features for sigma to be invertible).

**Week 10: Learning with Large Datasets**

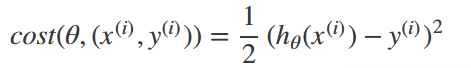
- We mainly benefit from a very large dataset when our algorithm has high variance when m is small. Recall that if our algorithm has high bias, more data will not have any benefit.

- Datasets can often approach such sizes as m = 1e8. In this case, our gradient descent step will have to make a summation over all one hundred million examples. We will want to try to avoid this – the approaches for doing so are described below.

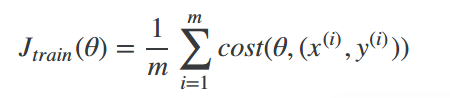
**1. Stochastic Gradient Descent**

**-** Stochastic gradient descent is an alternative to classic (or batch) gradient descent and is more efficient and scalable to large data sets.

- Stochastic gradient descent is written out in a different but similar way:



- The only difference in the above cost function is the elimination of the m constant within ½.



=> Jtrain is now just the average of the cost applied to all of our training examples.

- The algorithm is as follows:

1) Randomly “shuffle” the datasets

2) for I = 1...m:



=> This algorithm will only try to fit one training example at a time. This way we can make progress in gradient descent without having to scan all m training examples first. Stochastic gradient will be unlikely to converge at the global minimum and will instead wander around it randomly, but usually yeilds a result that is close enough. Stochastic gradient descent will usually take 1-10 passes through your data set to get the global minimum.

**2. Mini-Batch Gradient Descent**

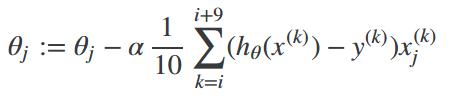
- Mini-batch gradient descent can sometimes be even faster than stochastic gradient descent. Instead of using all m examples as in batch gradient descent, and instead of using only 1 example as in stochastic gradient descent, we will use some in-between number of examples b.

- Typical values for b range from 2-100 or so.

- For example, with b = 10 and m = 1000:

Repeat:

for I = 1, 11, 21, 31, … , 991



=> We’re simply summing over ten examples at a time. The advantage of computing more than one example at a time is that we can use vectorized implementations over the b examples.

**Stochastic Gradient Descent Convergence**

**-** How do we choose the learning rate alpha for stochastic gradient descent? Also, how do we debug stochastic gradient descent to make sure it is getting as close as possible to the global optimum?

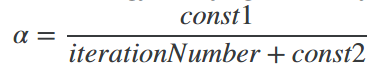
- One strategy is to plot the average cost of the hypothesis applied to every 1000 or so training examples. We can compute and save the costs during the gradient descent iterations.

- **With a smaller learning rate, it is possible that you may get a slightly better solution** with stochastic gradient descent. That is because stochastic gradient descent will oscillate and jump around global minimum, and it will make smaller random jumps with a smaller learning rate.

- If you increase the number of examples you average over to plot the performance of your algorithm, the plot’s line will become smoother.

- With a very small number of examples for the average, the line will be too noisy and it will be difficult to find the trend.

- One strategy for trying to actually converge at the global minimum is to **slowly decrease alpha over time.** For example:



=> However, this is not often done because people don’t want to have fiddle with even more parameters.

**3. Online learning**

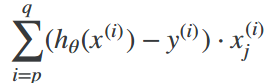
- With a continuous stream of users to a website, we can run an endless loop that gets (x,y), where we collect some user actions for the features in x to predict some behavior y.

- You can update theta for each individual (x,y) pair as you collect them. This way, you can adapt to new pools of users, since you are continuously updating theta.

**4. Map Reduce and Data Parallelism**

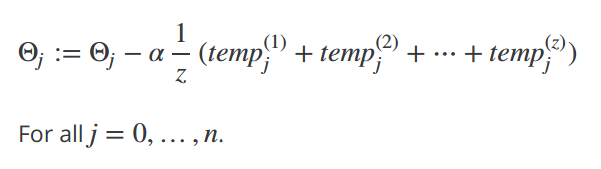
- We can divide up batch gradient descent and dispatch the cost function for a subset of the data to many different machines so that we can train our algorithm in parallel.

- You can split your training set into z subsets corresponding to the number of machines you have. On each of those machines calculate:



where we’ve split the data starting at p and ending at q.

- MapReduce will take all these dispatched (or mapped) job and ‘reduce’ them by calculating:



=> This is simply taking the computed cost from all the machines, calculating their average, multiplying by the learning rate, and updating theta.

- Your learning algorithm is MapReduceable if it can **expressed as computing sums of functions over the training set.** Linear regression and logistic regression are easily parallelizable.

- For neural networks, you can compute forward propagation and back propagation on subsets of your data on many machines. Those machines can report their derivatives back to a “master” server that will combine them.