***Computing Parameters Analytically***

**I. FEATURES AND POLYNOMIAL REGRESSION**

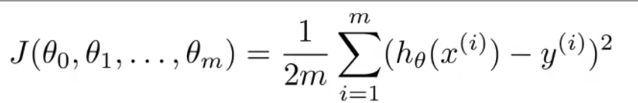
* **The Normal Equation 🡪** gives a much better way to solve for the optimal value of the parameters θ for *some* linear regression problems.
* So far, the algorithm we've been using for linear regression is **gradient descent**, where in order to minimize J(θ), we use an iterative algorithm (multiple iterations of gradient descent) to converge to a global minimum.
* In contrast, the **normal equation** gives a method to solve for θ *analytically*
* Rather than running an iterative algorithm, we just solve for the optimal value for θ all in 1 go
* Imagine a very simplified cost function J(θ) where θ is a real number 



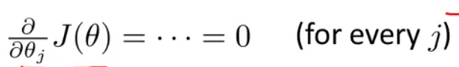
* For now, imagine θ is just a scalar value or a row value, rather than a vector.
* So we have a cost function = a quadratic function of a real-value parameter θ
* How do you minimize a quadratic function? *🡪 take derivatives + set them to 0*
* Take derivative of J w/ respect to parameters θ 🡪 set that derivative = 0 🡪 allows you to solve for the value of θ that minimizes J(θ)
* That was a simpler case of when θ was a real #.
* In the problem *we*’re interested in, θ is not a real #, but, instead, an n*+1-dimensional parameter vector* of values for θ1-θm



* A cost function J(θ) = a function of *this* vector = (θ 0,… θm)



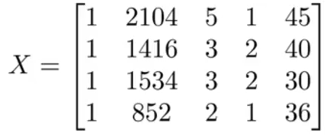
* How do we minimize *this* cost function?
* 1 way to do so is to take the **partial derivative** of J w/ respect to *every* parameter if J(θ) + to set all of those derivatives to 0.

I.E: 

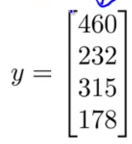
* Then we’d solve for θ0 up to θm + this gives the values of θ to minimize the cost function.
* If you actually work through the calculus + through the solution to the parameters θ0 through θm, the derivative ends up being somewhat long and involved.
* We only need to know a little in order to implement this algorithm + get it to work w/out heavy math
* Ex: m = 4 training examples.



* In order to implement the normal equation method, take the data set + add an extra column that corresponds to an extra feature, x0, that always takes on value = 1.
* Then construct a matrix, X , which contains all feature values from the training data



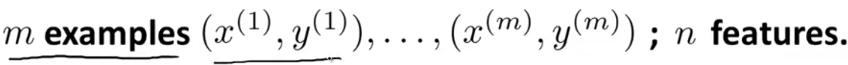
* Then we do the same for y's (values we’re trying to predict)



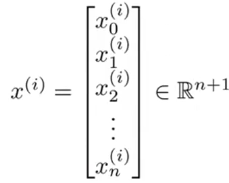
* X = an *m\*(n+1)*-*dimensional* matrix, Y = a *m-dimensional* vector, where m = training examples + n = original features (we have n+1 b/c of the extra feature x0)
* We then calculate the **inverse of (X(t)\*X)** \* **X(t)\*Y** = values of θ that minimizes the cost function.



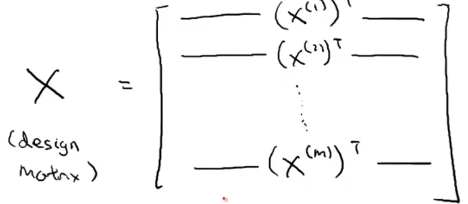
* θ ends up being a *n-dimensional vector*
* General case:



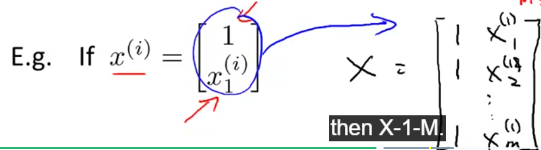
* M training examples + n features + each training example x(i) looks like an n+1 dimensional feature vector.



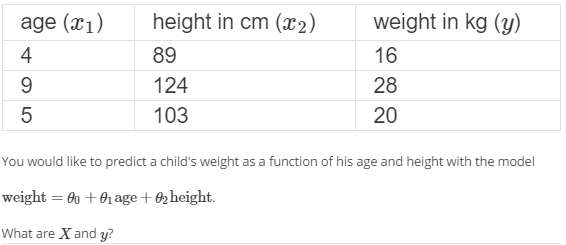
* Each training example gives an n+1 dimensional vector of features values, like above
* Then, to construct the matrix "X" (**the design matrix**) take the transpose of the 1st training example vector (x(1)) + make it 1st first row of the design matrix]
* Then repeat until the last training example.



* *That makes an M by N+1 dimensional matrix*
* As a concrete example, let's say we have only 1 feature other than x0 (always = 1).



* End up w/ an M\*2-dimensional matrix.
* The Y-vector is created by taking all y-values from the training set + stacking them into an m-dimensional vector



* \*\*\*\*\*\*If using this normal equation method, then *feature scaling isn't actually necessary*\*\*\*\*\*\*\*\*
* If using gradient descent, *features scaling is still important*.
* Advantages of gradient descent
* Works pretty well even when n (# of features) is large
* Disadvantage of gradient descent
* Need to choose the learning rate **α**.
* Often, this means running gradient descent w/ different **α**‘s to see what works best
* It needs many more iterations.
* Advantages of normal equation:
* *Don't* need to choose learning rate **α** 🡪 simple to implement, just run it + it usually works
* *Don't* need to iterate (don't need to plot J(θ), check convergence, or take those extra steps)
* Disadvantages of the normal equation:
* In order to solve for the parameter θ, we need to solve for **(X(t)\*X)^-1** (n\*n matrix)
* For most computed implementations, the cost of inverting a matrix grows roughly as much as the cube of the dimension of the matrix 🡪 
* So, computing the inverse costs roughly Order\*n^3 time (Sometimes slightly faster)
* If # of features *n* is very large, computing that matrix can be slow + the normal equation method will be much slower 🡪 may use gradient descent
* *What does small and large mean?*
* If n is on the Order of 100, then inverting a 100x100 matrix is no problem, by computing standards.
* If n = 1k, we can still use the normal equation method b/c Inverting a 1000x1000 matrix is actually really fast on a modern CPU.
* If n = 10k, inverting a 10kx10k matrix starts to get kind of slow 🡪 *might* start to lean in the direction of gradient descent
* But if it gets much bigger than that, probably use gradient descent.
* If n = 10^6 (1M features), inverting a 1M x 1M matrix = very expensive = favors gradient descent
* Hard to give a strict # of how large set of features has to be before converting to gradient descent
* *So long as n is not too large, the normal equation gives us a great alternative method to solve for the parameter θ.*

**II. Normal Equation Noninvertability**

* There's a phenomenon called **noninvertability** that you may run into that may be somewhat useful to understand. Remember the normal equation method



* **Issue:** *What if the matrix X(t)\*X is* ***noninvertible?***
* Some matrices do not have an inverse 🡪 **singular** or **degenerate matrices**.
* The issue of X(t)\*X being non-invertible should only happen rarely
* If X(t)\*X is non-invertible, there are usually 2 common causes for this.
* 1: Somehow in your learning problem, you have *redundant features*
* If trying to predict housing prices + if x1 = size of the house in square feet + x2 = size of the house in square *meters*
* 1 meter = 3.28 ft., so the 2 features will always satisfy the constraint that x1 = 3.28^2\*x2
* You can actually show that if your 2 features are related via the above linear equation, the matrix is noninvertible
* 2: If in training + trying to run the learning algorithm w/ too many features (*if m <= n*)
* Imagine m = 10 training example and n = 100 features
* So we’re trying to fit a parameter vector θ (n+1 = 101-dimensional) of 101 parameter values from just 10 training examples.
* This turns out to work *sometimes*, but it’s not always be a good idea.
* We might not have enough data if you only have 10 examples to fit 101 parameters to
* Commonly, if m <= n, we can see if we can either delete some features or use a technique called **regularization** that lets you fit a lot of parameters/use a lot features, even if you have a relatively small training set.
* To summarize, if ever you find that X(t)\*X is singular/it’s non-invertible, 1st look at your features + see if you have redundant features (you're being **linearly dependent**), just delete 1 of these features
* If so, keep deleting redundant features until they're no longer redundant
* If your features are NOT redundant, check if we may have too many features + if so, either delete some features (if we can bear to use fewer features) or else consider using regularization
* But, as mentioned, X(t)\*X being non-invertible should happen pretty rarely + should not be a problem for most implementations of linear regression