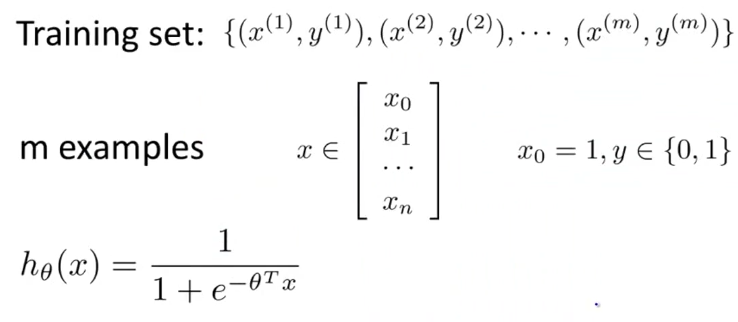
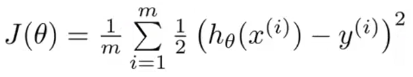
***Logistic Regression Model***

**I. COST FUNCTION**

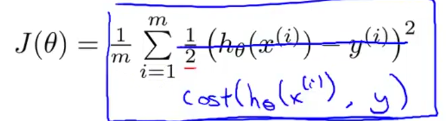
* Now we need to fit the parameters of θ for the logistic regression and define the optimization objective (the cost function) we'll use to fit the parameters.
* Here's a supervised learning problem of fitting logistic regression model

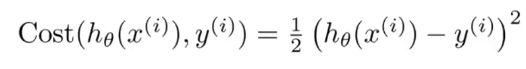


* We have a training set of m training examples + each example is represented by an n+1 dimensional feature vector w/ x0 = 1 (1st feature always equal to 1 for intercept)
* This is a classification problem, so the training set has the property that every label y is 0 or 1
* *How do we fit the parameter's θ?*
* Back when developing linear regression models, we used the following cost function:

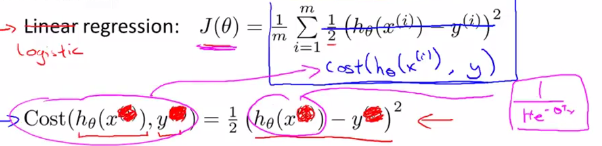


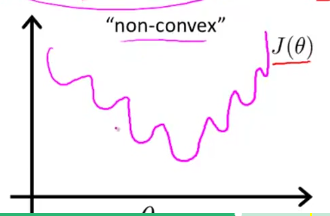
* Now we want to use an alternative way of writing this out by instead of writing out the square of errors, write it as **cost(hθ(x),y)**



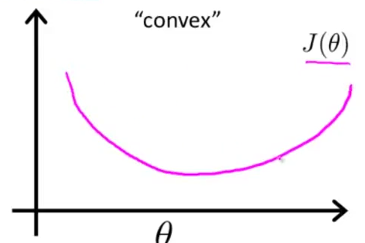


* We define this as one-half of the squared error.
* So now we see more clearly that the **cost function** is a *sum* over the *training* set \* 1/m
* This is the cost we want our learning algorithm to have to pay if its *output/prediction* = hθ(x(i)) and the *actual label* ended up being y(i).
* This cost function worked fine for linear regression, but here, we're interested in logistic regression.
* If we use the linear regression cost function, this would be a **non-convex function** of the parameter's θ.
* By “non-convex”, say we have some cost function J(θ), and for logistic regression, our hθ(x) function has a nonlinearity [**1 / (1 + E^-(θ(t)\*x)** , a pretty complicated nonlinear function]
* If we take the sigmoid function + plug it into the cost(h(θx),y) and plug that resulting cost into J(θ), you find J(θ) can have many local optima = is **a non-convex function**

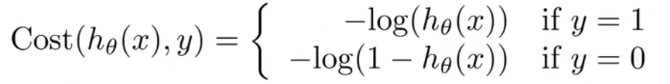




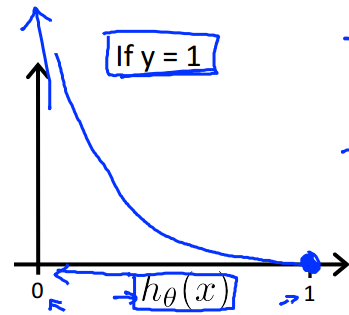
* If we ran gradient descent on this sort of function, it is *NOT* guaranteed to converge to the global minimum.
* What we want is a J(θ) that is **convex** = a single bow-shaped function so we’d be guaranteed that gradient descent would converge to the global minimum.



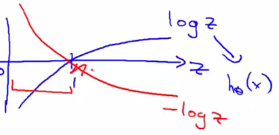
* So, the problem w/ using linear regression’s squared cost function is b/c we have the very nonlinear sigmoid function in the place of hθ(x), J(θ) ends up being a nonconvex function
* Need to come up w/ a different cost function that *IS* convex so we can apply an algorithm like gradient descent + be guaranteed to find the global minimum.
* **Logistic regression Cost Function**



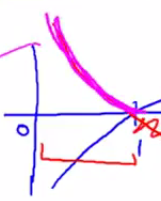
* \*\*\*The cost/penalty the algorithm pays if it *outputs/predicts* a value of hθ(x) (say 0.7) + the *actual* cost turns out to be the value of y.



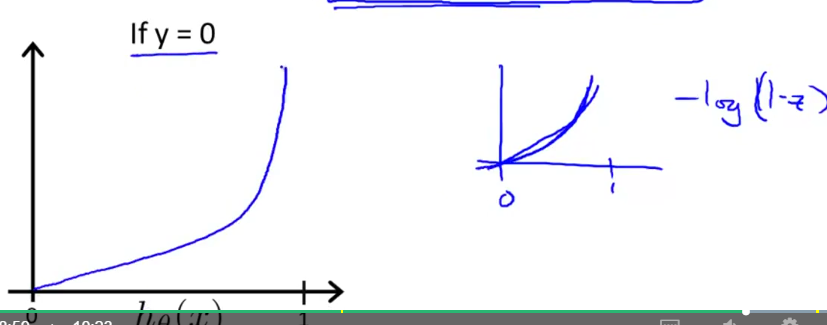
* If y = 1, then **cost(hθ(x),y)** = –log(hθ(x)), where hθ(x) varies between 0-1 b/c its logistic regression
* 1 way to see why the plot looks like this is b/c if you plot log(z), as z approaches 0, log(z) approaches -Infinity,

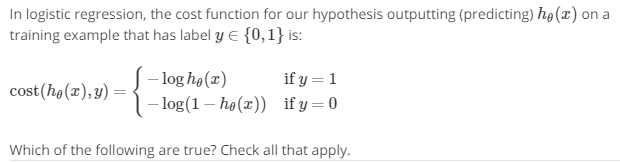


* z is playing the role of hθ(x), so we’re only interested in the values of x (and –log(z)) between 0-1



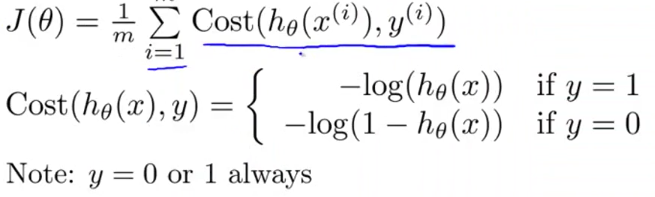
* This cost function has a few interesting and desirable properties.
* If y = 1 and hθ(x) = 1 (exactly predict actual value), then cost = 0
* This is where we'd like it to be b/c if we correctly predict the output y, our cost is 0.
* As hθ(x) approaches 0, the cost blows up + goes to Infinity.
* This captures the intuition that if hθ(x) = 0 (chance of y = 1 🡪 0 or 0%) + it turns out y *actually* = 1, we penalize the learning algorithm by a very, very large cost.
* If y is equal to 0, then the cost is similar to **-log(1 - z)**



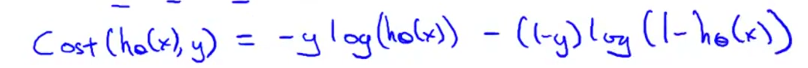
* Now our cost goes to infinity as hθ(x) goes to 1 b/c if y turns out to be 0 but we predicted y = 1, we end up paying a very large cost.
* Conversely, if hθ(x) = 0 and y = 0, the hypothesis accurately predicted y so the cost = 0.
* 
* 
* 
* 

**II. SIMPLIFIED COST FUNCTION AND GRADIENT DESCENT**

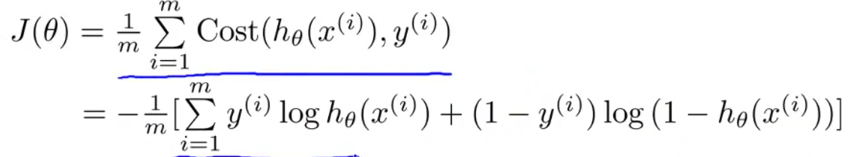
* Now for a slightly simpler way to write the logistic regression cost function we’ve been using so far
* Original:



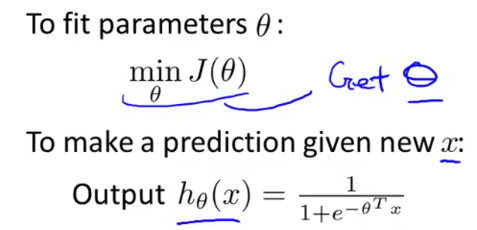
* Rather than writing out a cost function on 2 separate lines w/ 2 separate cases (y = 1 or 0), compress them into 1 equation so it’s more convenient to write a cost function + then derive gradient descent



* We know there are only 2 possible cases, y = 0 or 1, so suppose y = 1:
* *Then the 2nd term is multiplied by 0, goes away, + we're left w/ only the 1st term,* ***-y\*log(h(x))****, the equation we have up above for if y = 1.*
* The other case is y = 0:
* *The 1st term = 0 and the cost function simplifies to just the 2nd term,* ***-(1-y)log (1- h(x))****, which is exactly what we had for when y = 0.*
* We can therefore write all our cost functions for logistic regression as follows:



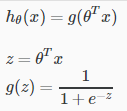
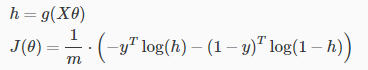
* This cost function can be derived from statistics using the **Principle of Maximum Likelihood Estimation**, an idea for how to efficiently find parameters θ for different models + it is convex.
* *“a method of*[*estimating*](https://en.wikipedia.org/wiki/Estimator)*the*[*parameters*](https://en.wikipedia.org/wiki/Statistical_parameter)*of a*[*statistical model*](https://en.wikipedia.org/wiki/Statistical_model)*given observations*”
* Given this cost function, in order to fit the parameters θ, we try to find the ones that minimize J(θ)
* Then given a new example w/ some set of features x, we can then take those θ’s we fit to our training set + output our prediction



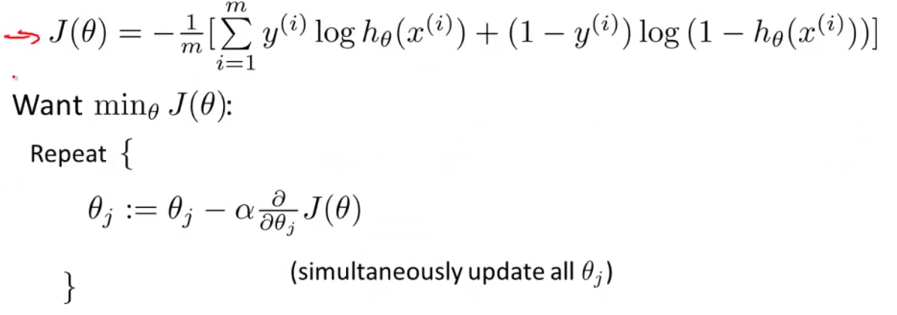
* **Output of hθ(x) 🡪 probability y = 1 *given the input x and parameterized by θ*.**



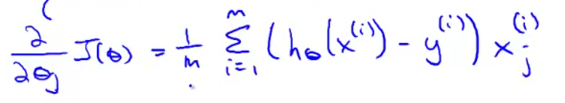
* Vectorized Implementation

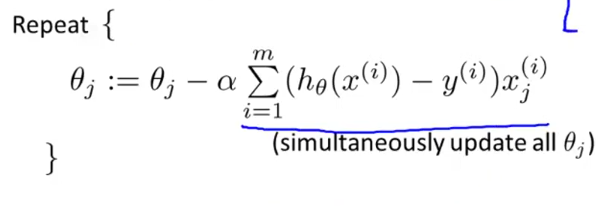
* All that remains to be done is to figure out how to *actually minimize J(θ)* as a function of θ so that we can actually fit the resulting parameters to our training set.
* Again, we minimize the cost function using **gradient descent**.



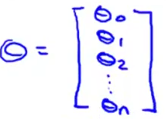
* To minimize J as a function of θ, repeatedly update each parameter by updating it as itself minus **learning rate α** times the **derivative term**.
* If you actually compute the derivative term, you get this equation:



* We can take this partial derivative term and plug it back into the repeating steps + write out our gradient descent algorithm as follows:



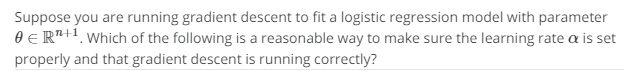
* Now if you have n features, you’d have a parameter vector θ w/ parameters θ0 to θn, which we use this new update formula on to simultaneously update all values of θ.



* If comparing this new update rule to the one for linear regression, *it’s exactly the same*
* For *linear* regression, we had h(x) = θ(t)\*X
* For *logistic* regression, the definition for the hypothesis changed

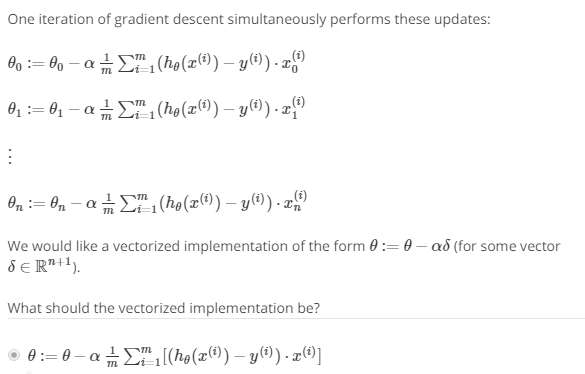


* So even though the *update rule looks identical*, b/c the definition of hθ(x) changed, this is NOT the same thing as gradient descent for linear regression.
* We usually apply the same method for monitoring gradient descent in linear regression to logistic regression to make sure it is converging correctly.



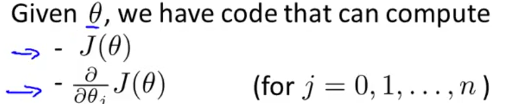


* When implementing logistic regression w/ gradient descent, we have all these different parameter values (θ0 to θn) that we need to update
* 1 thing we could do is have a FOR loop 🡪 FOR i = (0:(n + 1)), update each parameter value in turn.
* Rather than using a FOR loop, ideally we’d also use a **vector-wise implementation**, which can update all n+1 parameters all in 1 fell swoop.



**III. ADVANCED OPTIMIZATION**

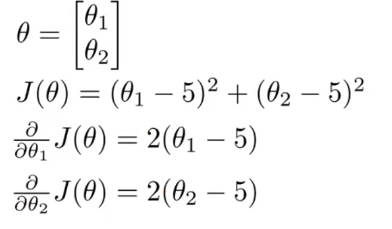
* Now for some advanced optimization algorithms + concepts to get logistic regression to run much more quickly than w/ gradient descent + also lets the algorithms scale much better to very large ML problems, such as if we had a very large # of features.
* What gradient descent is doing
* We want to minimize some cost function J(θ) 🡪 need to write code that can take the parameters θ as input + compute 2 things: J(θ) + the partial derivative terms for J = 0 up to J = N:



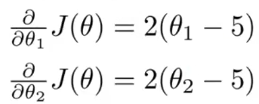
* Given the code that can do these 2 things, gradient descent repeatedly performs the update to the parameters θ with the partial derivatives from our code



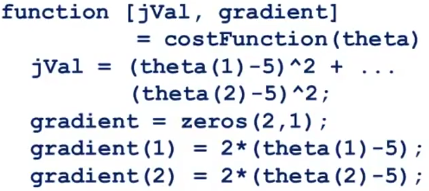
* i.e. We need to supply code to compute J(θ) + derivative terms, which get plugged into gradient descent, which tries to minimize J(θ) (don’t *really* need to calculate J(θ), just the derivatives)
* Gradient descent isn't the only algorithm we can use
* There more advanced + sophisticated ones that, if we provide them a way to compute J(θ) + the derivative terms, they use different approaches to optimize the cost function J(θ)
* **Conjugate gradient, BFGS** + **L-BFGS** are 3 examples of more sophisticated optimization algorithms than gradient descent to minimize J(θ).
* These 3 algorithms have a number of advantages.
* W/ any of these algorithms, you usually don’t need to manually pick the learning rate.
* Given a way to compute J(θ) + its derivatives, these algorithms have a clever **inter-loop** called a **line search algorithm** that automatically tries out different values for **α** + automatically picks a good one
* Can even pick a *different learning rate* for *each iteration* so we don’t need to choose it
* Often end up converging much faster than gradient descent.
* B/c they do more sophisticated things than just pick a good **α**
* *\*\*\*\*It’s entirely possible to use these algorithms successfully + apply them to lots of different learning problems w/out actually understanding what these algorithms do.*
* If these algorithms have a disadvantage, the main one would be they're quite a lot more complex than gradient descent.
* Probably shouldn’t implement these algorithms on you’re own unless you're an expert in numerical computing.
* Just use a software library link in MATLAB or Octave to implement some of these advanced optimization algorithms.
* There’s a difference between good + bad implementations (in performance) of these algorithms in different languages
* Might want to try out a couple of different libraries to make sure you find a good library for implementing these algorithms.
* Assume you have a problem w/ 2 parameters, θ0 + θ1 and J(θ) = (θ1 – 5)^2 + (θ2 - 5)^2



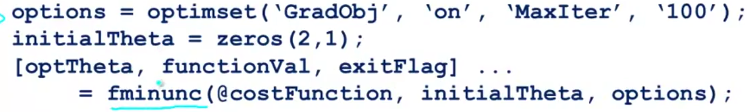
* To minimize J(θ) as a function of θ, the values that minimize it are θ1 = 5, θ2 = 5
* The derivatives of the cost function J w/ respect to θ1 and θ2 turn out to be



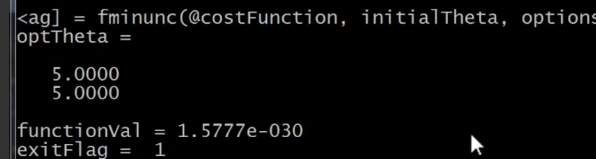
* To apply 1 of the advanced optimization algorithms to minimize J(θ) (i.e. if we didn't know the minimum was at 5, 5), we could implement an Octave cost function like this:



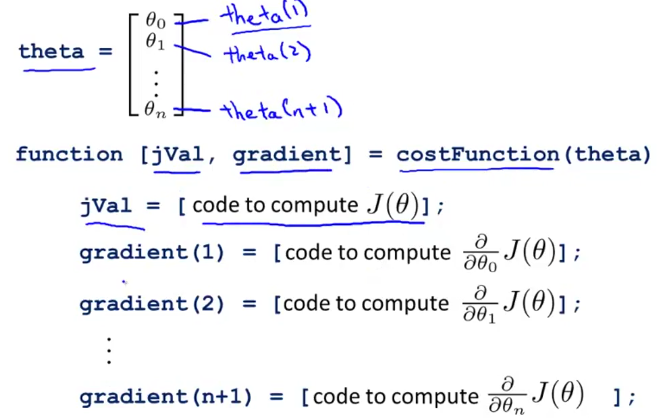
* This returns 2 arguments, **jVal** = cost function value, and **gradient**, a 2x1 vector, whose 2 elements of correspond to the 2 partial derivative terms
* Having implemented this cost function, you can now call an advanced optimization function **fminunc** (**function minimization unconstrained**)



* We have **options** as a data structure that stores the options you want.
* **GradObj** sets the gradient objective parameter to ‘ON’ = i.e. you’re going to provide a gradient to this algorithm
* Set the maximum number of iterations to 100
* Give it an initial guess for θ in a 2x1 vector + use the command to call fminunc
* @ symbol presents a pointer to the cost function just defined above.
* If you call this, it uses 1 of the more advanced optimization algorithms
* Think of it just like gradient descent but automatically choosing **α** + then attempting to use advanced optimization algorithms (like gradient descent on steroids) to find the optimal value of θ.



* It found the optimal value of θ is θ1 = 5, θ2 = 5, exactly as we were hoping.
* Cost function value at the optimum is = 1.5^-30, essentially 0, which is also what we were hoping for
* **Exit flag =** 1 🡪 shows what the convergence status 🡪 verifies if this algorithm has converged.
* That's how we optimize our trial example of this simple, quick-driving cost function.
* In logistic regression, we have a parameter vector θ comprised of parameters θ0 through θ(n+1)
* θ(n) in Octave b/c Octave indexes vectors from 1 🡪 θ0 is written θ1 in Octave
* Then we write a cost function for logistic regression which returns jVal + the gradients from values we give it (as args)
* *Gradient 1* = some code to compute the partial derivative in respect to θ0, *Gradient 2* = code to compute partial derivative respect to θ1 and so on.



* In order to apply this to logistic regression (or even linear regression), we need to do plug in the appropriate code to compute jVal and the gradient values

