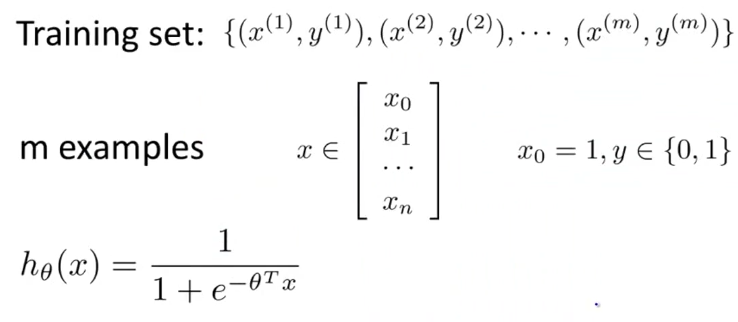
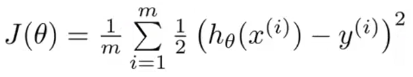
***Logistic Regression Model***

**I. COST FUNCTION**

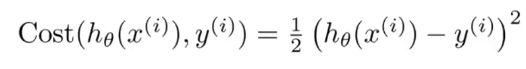
* Now we need to fit the parameters of θ for the logistic regression and define the optimization objective/cost function we'll use to fit the parameters.
* Here's a supervised learning problem of fitting logistic regression model

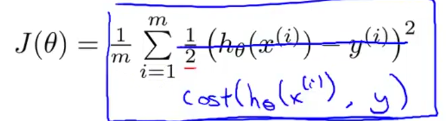


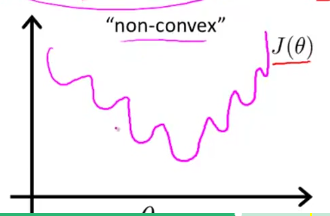
* We have a training set of m training examples + each example is represented by an n+1 dimensional vector w/ x(0) = 1 (1st feature/a zero feature is always equal to 1)
* B/c this is a classification problem, our training set has the property that every label y is either 0 or 1
* *How do we fit the parameter's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



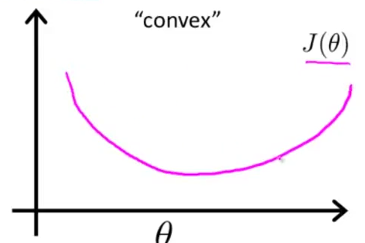
* Now we want to use an alternative way of writing this out by instead of writing out the square of errors, write in here costs(h(x),y)



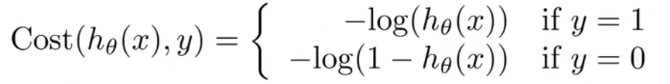
* We define this as one-half of the squared error.
* So now we see more clearly that the **cost function** is a *sum* over my *training* set 🡺 1/m \* the sum of my new training cost term.
* 
* This is the cost we want our learning algorithm to have to pay if its *output/prediction* = h(x) and the *actual label* = y.
* This cost function worked fine for linear regression, but here, we're interested in logistic regression.
* If we use this particular 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 h(x) function has a nonlinearity 🡪 **1 / (1 + E^-(θ(t)\*x)**, a pretty complicated nonlinear sigmoid function.
* If you take this sigmoid function + plug it into the Cost(h(x),y) and plug this cost() into J(θ), you find J(θ) can look like this w/ many local optima, **a non-convex function**



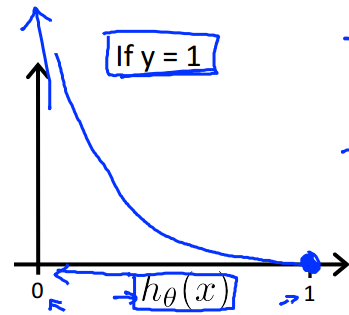
* If you were to run gradient descent on this sort of function, it is not guaranteed to converge to the global minimum.
* What we want is a cost function that is **convex** = a single bow-shaped function so we would be guaranteed that gradient descent would converge to the global minimum.



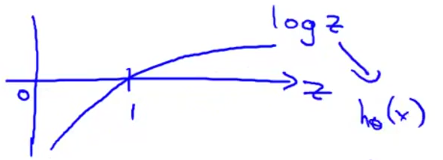
* The problem w/ using this squared cost function is that b/c of the very nonlinear sigmoid function in the place of h(x), J(θ) ends up being a nonconvex function
* Need to instead of come up w/ a different cost function that is convex so we can apply an algorithm, like gradient descent + be guaranteed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



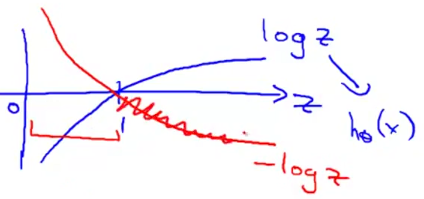
* We to say the cost/penalty the algorithm pays, if it outputs/predicts a value of h(x) (say 0.7) + the actual cost turns out to be the value of y.
* Cost = **-log(h(x))** if y = 1 and **-log(1- h(x))** if y = 0.



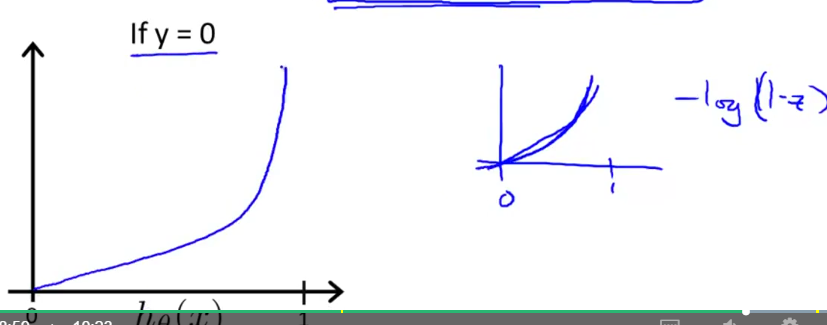
* Here, h(x), varies between 0-1.
* 1 way to see why the plot looks like this is b/c if you plot log z, + it approaches -infinity as z approaches 0



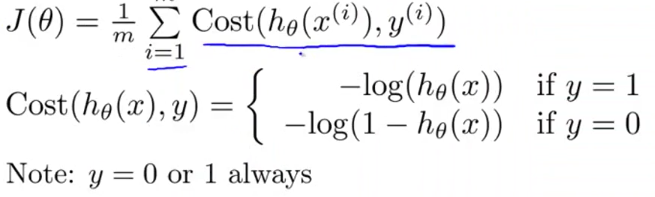
* z is playing the role of h(x)
* In flipping the sign to -log z between 0-1, we're left w/ the curve for the cost function when y = 1



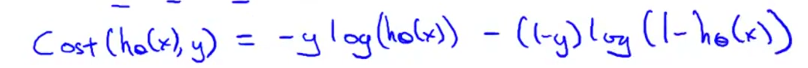
* This cost function has a few interesting and desirable properties.
* notice if y = 1 and h(x) = 1 (in other words, hypothesis exactly predicts the actual value), then cost = 0 🡪 where we'd like it to be b/c we correctly predict the output y so cost is 0.
* As h(x) approaches 0, the cost blows up + goes to infinity + captures the intuition that if a hypothesis = 0 🡪 the chance of y = 1 is equal to 0
* But if it turns out y *actually does* = 1 even though we said probability of it happening = 0, we penalize the learning algorithm by a very, very large cost.
* If y is equal to 0, then the cost looks like -log(1-z)



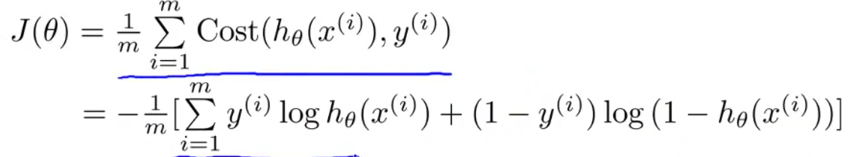
* Cost goes to infinity as h goes to 1 b/c if y turns out to be 0 but we predicted y = 1, we end up paying a very large cost.
* Conversely, if h = 0 and y = 0, the hypothesis accurately predicted y + the cost function = 0.
* So if h(x) = y, cost = 0 whether y = 0 or y = 1 since we correctly predicted y
* Also, regardless of whether y = 0 or 1, if h(x) = 0.5, then cost > 0
* Now for a slightly simpler way to write the cost function than we have been using so far



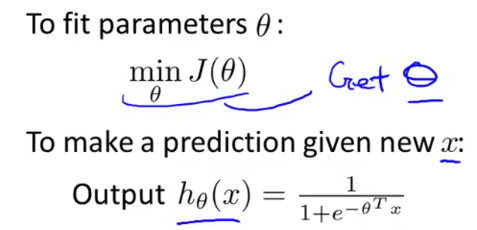
* Rather than writing out a cost function on 2 separate lines w/ 2 separate cases (y = 1 or 0), compress them into 1 equation so its more convenient to write a cost function + derive gradient descent

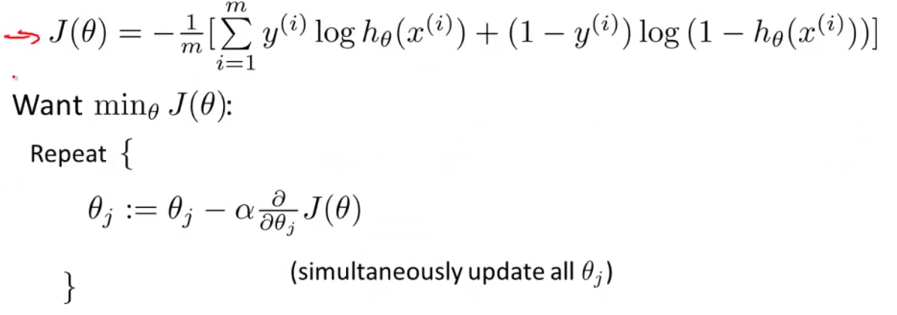


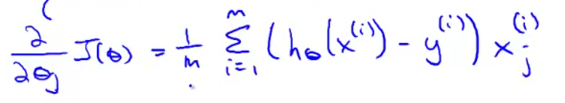
* We know there are only 2 possible cases, Y = 0 or 1, so suppose Y = 1, then the 2nd term is multiplied by 0 and goes away + we're left w/ only the 1st term, **-y\*log(h(x))**, the equation we have up above for if y = 1.
* The other case is y = 0, the 1st term = 0 and the cost function simplifies to 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:



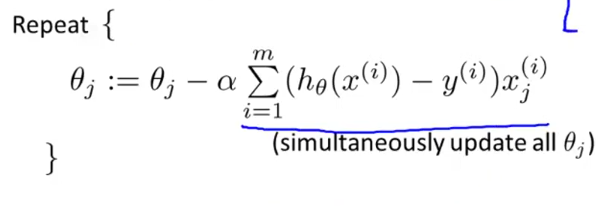
* This cost function can be derived from statistics using the **principle of maximum likelihood estimation**, an idea for how to efficiently find parameters θ for different models + it is convex.
* Given this cost function, in order to fit the parameters θ, try to find the ones that minimize J(θ).
* If given a new example w/ some set of features x, we can then take these θ’s that we fit to our training set + output our prediction



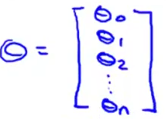
* **Output of h = probability y = 1 given the input x parameterized by θ.**
* All that remains to be done is to figure out how to actually minimize J(θ) as a function of θ so that we can actually fit the parameters to our training set.
* Again, we minimize the cost function using gradient descent.
* 
* To minimize J as a function of θ, again, repeatedly update each parameter by updating it as itself minus **learning rate alpha** times the **derivative term**.
* If you actually compute the derivative term, you get this equation:



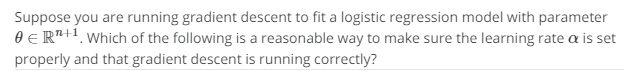
* We can take this partial derivative term and plug it back into the repeating steps + write out our gradient descent algorithm as follows:



* Now if you have n features, you would have a parameter vector θ w/ parameters θ0 to θn + use this update to simultaneously update all values of θ.

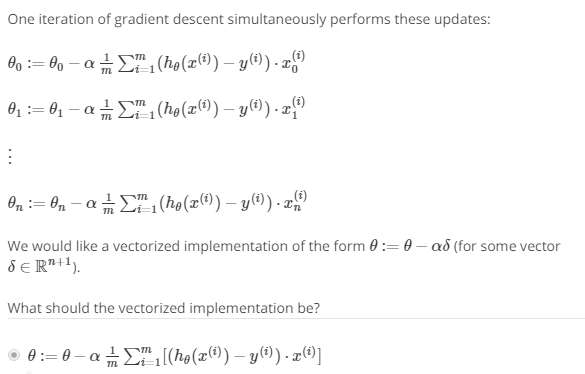


* If comparing this update rule to what we were doing for linear regression, this equation is the same
* *Are linear regression and logistic regression different algorithms or not?*
* For logistic regression, what has changed is the definition for the hypothesis
* For linear regression, we had h(x) = θ(t)\*X, for logistic regression, we have 1 / (1 + E^-( θ(t)\*X).
* So even though the update rule looks identical, b/c the definition of h(x) changed, this is NOT the same thing as gradient descent for linear regression.
* We usually apply the same method from linear regression to logistic regression to monitor gradient descent to make sure it is converging correctly.

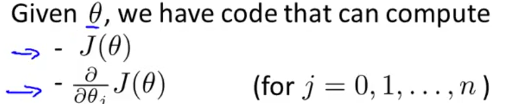




* When implementing logistic regression w/ gradient descent, we have all these different parameter values (θ0 to θn) we need to update
* 1 thing we could do is have a FOR loop 🡪 FOR i = 0:n + 1, update each parameter value in turn.
* Rather than using a FOR loop, ideally we’d also use a vector-wise implementation,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



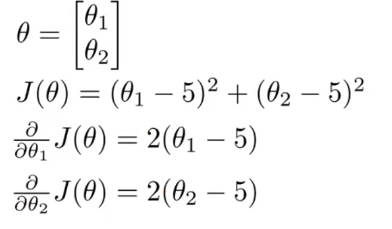
* Now for some advanced optimization algorithms + concepts to get logistic regression to run much more quickly than w/ gradient descent + also lets the algorithms scale much better to very large ML problems, such as if we had a very large # of features.
* What gradient descent is doing 🡪 have some cost function J + want to minimize it 🡪 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.



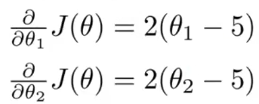
* Given code that can do these 2 things, gradient descent repeatedly performs the update to the parameters θ



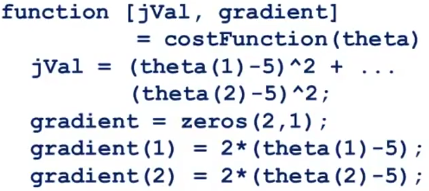
* We need to supply code to compute J(θ) + derivative terms + these get plugged into gradient descents, which try to minimize the cost function for us.
* Gradient descent isn't the only algorithm we can use 🡪 there are other more advanced + sophisticated ones that, if we provide them a way to compute these J(θ) and the derivative terms, they use different approaches to optimize the cost function
* **Conjugate gradient, BFGS** + **L-BFGS** are 3 examples of more sophisticated optimization algorithms than gradient descent to minimize the cost function.
* These 3 algorithms have a number of advantages.
* 1 – W/ any of these algorithms, you usually don’t need to manually pick the learning rate alpha.
* 1 way to think of these algorithms is, given a way to compute the derivative + cost function, these algorithms have a clever inter-loop called a **line search algorithm** that automatically tries out different values for alpha + automatically picks a good alpha
* Can even pick a different learning rate for each iteration.
* 2 – Do more sophisticated things than just pick a good learning rate + often end up converging much faster than gradient descent.
* 3 - is entirely possible to use these algorithms successfully and apply to lots of different learning problems w/out actually understanding what these algorithms do.
* If these algorithms have a disadvantage, the main disadvantage is they're quite a lot more complex than gradient descent.
* probably shouldn’t implement these algorithms unless you're an expert in numerical computing.
* for these algorithms, just use a software library link in MATLAB or Octave to implement some of these advanced optimization algorithms.
* There is a difference between good and bad implementations 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.
* There is a difference in performance between a good implementation vs. a bad implementation
* Assume you have a problem w/ 2 parameters, θ0 and θ1 + your cost function = (θ1 – 5)^2 + (θ2 -5)^2



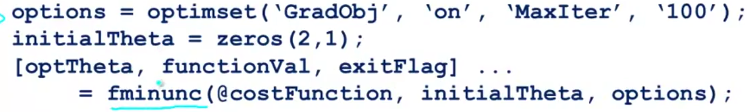
* To minimize J(θ) as a function of θ, values that minimize it are θ 1 = 5, θ 2 = 5
* The derivatives of the cost function J w/ respect to θ1 and θ2 turn out to be



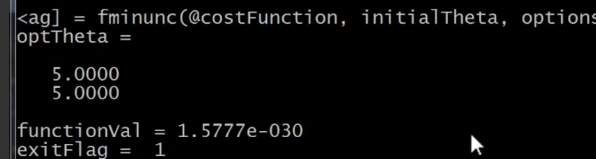
* To apply 1 of the advanced optimization algorithms to minimize cost function J (if we didn't know the minimum was at 5, 5), implement an Octave cost function like this



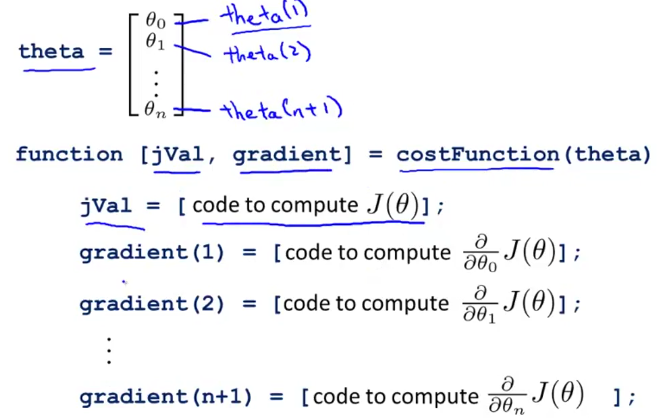
* This returns 2 arguments, **J-val** = cost function value, and **gradient**, a 2x1 vector, whose 2 elements of correspond to the 2 partial derivative terms
* Having implemented this cost function, you can now call an advanced optimization function **fminunc** (**function minimization unconstrained**)



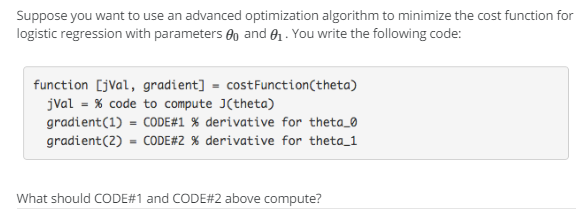
* We have **options** as a data structure that stores the options you want.
* **GradObj** sets the gradient objective parameter to ‘ON’ 🡪 just means you are indeed going to provide a gradient to this algorithm
* set the maximum number of iterations to 100
* Give it an initial guess for θ in a 2x1 vector + 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 learning rate alpha + then attempting to use advanced optimization algorithms (like gradient descent on steroids) to find the optimal value of θ.

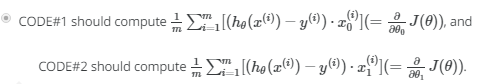


* It found the optimal value of θ is θ1 = 5, θ2 = 5, exactly as we were hoping.
* Cost function value at the optimum is = 1.5^-30, essentially 0, which is also what we were hoping for
* **Exit flag =** 1 🡪 shows what the convergence status of this 🡪 verifies if this algorithm has converged.
* That's 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 🡪 θ 0 is written θ 1 in Octave
* Then write a cost function for logistic regression which returns J-val + the gradients from values we give it (as args)
* gradient 1 = some code to compute the partial derivative in respect to θ 0, gradient 2 = code to compute partial derivative respect to θ 1 and so on.

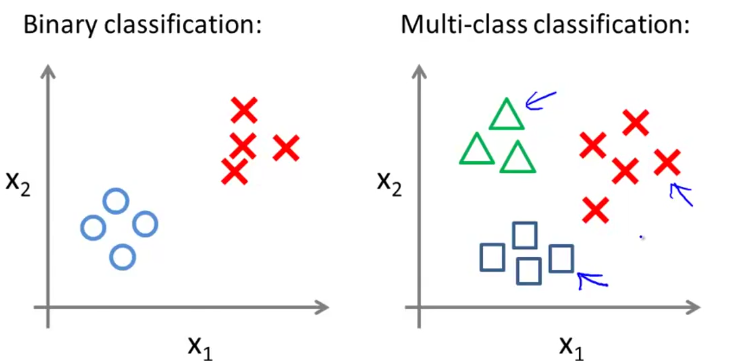


* In order to apply this to logistic regression (or even linear regression), we need to do plug in the appropriate code to compute these things 13:15

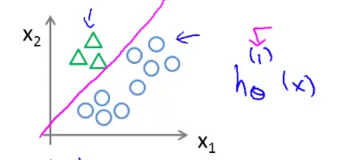




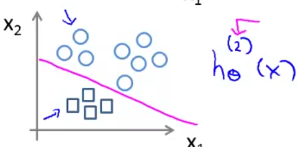
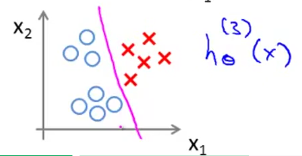
* Logistic regression can work for **multiclass classification** problems as well, such as w/ an algorithm called **one-versus-all classification.**
* Multiclass classification problems:
* Say you want a learning algorithm to automatically tag emails (work, friends, family, etc.)
* 4 classes to which we might assign emails 🡪 classes y = 1, y = 2, y = 3, and y = 4. A
* For medical diagnosis, if a patient comes in w/ a stuffy nose, possible diagnosis could be they're not ill (y = 1), have a cold (y = 2), or have a flu (y = 3)
* Classify weather as sunny, cloudy, rainy, or snow,
* In all of these examples, y can take on a # of values
* Whereas for a binary classification problem, our data sets looks like the left, for a multi-class classification problem our data sets may look like the right



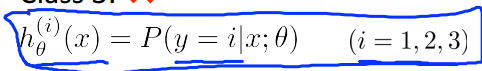
* The question is, given the data set w/ 3 classes, how do we get a learning algorithm to work here?
* We already know how to do binary classification using a regression, maybe fit a straight line to set for positive + negative classes = an idea called **one-vs-all/one-vs-rest classification**.
* We can then take this + make it work w/ multi-class classification as well.
* For 3 classes, take the training set + turn it into 3 *separate binary/2-class classification problems*



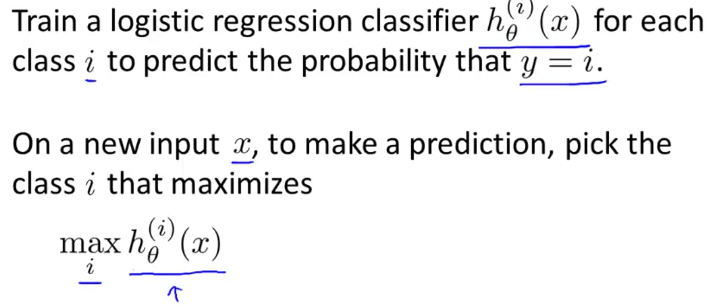
* Class 1 = the triangle 🡪 essentially create a new fake training set where classes 2 and 3 get assigned to the negative class + class 1 gets assigned to the positive class + then fit a **classifier**, h(x)(1)
* Think of the triangles being assigned the value = 1 and the circles assigned the value = 0.
* Then train a standard logistic regression classifier to give us a position boundary
* Then do the same thing for class 2 and 3

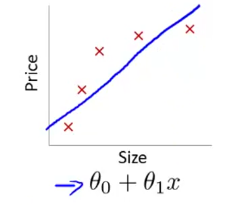
* We've fit 3 classifiers, trying to estimate the probability y = class i, given an x, parametrized by θ



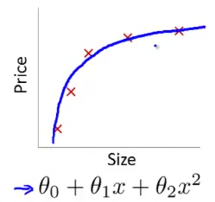
* In the 1st, the classifier was learning to recognize the triangles/thinking of the triangles as a positive clause 🡪 h(x)(1) is trying to estimate the probability y = one, given x is parametrized by θ.
* Then we treat the square class as a positive class + estimate probability y = 2 and so on.
* So we have 3 classifiers, each trained to recognize 1 of the 3 classes.
* To summarize, we want to train a logistic regression classifier h(x) for each class i to predict the probability y = i.
* Finally to make a prediction, when given a new input x to make a prediction, we run all 3 classifiers on the input x + pick the class i that maximizes the 3 = pick the classifier that is most confident
* Whichever value of i gives us the highest probability, we predict y to be that value.



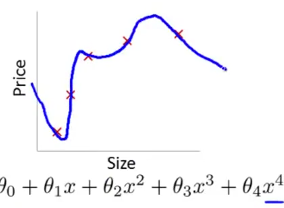
* When you apply linear + logistic regression to certain ML applications, they can run into a problem called **overfitting** that can cause them to perform very poorly.
* 1 thing we could do to our housing prices data set via linear regression is fit a linear function to this data + get a *straight* line fit to the data.



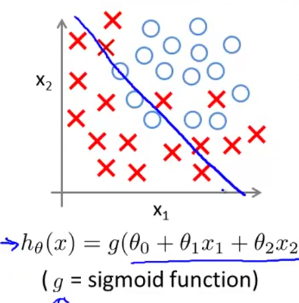
* This isn't a good model b/c as size increases, prices plateau 🡪 this algorithm does not fit well = **underfitting** = this algorithm has **high bias**.
* It's as if the algorithm has a very strong preconception/bias that prices are going to vary linearly w/ size, despite data to the contrary.
* Despite evidence of the contrary, it’s still biased + ends up a poor fit to the data.
* Fit a quadratic function



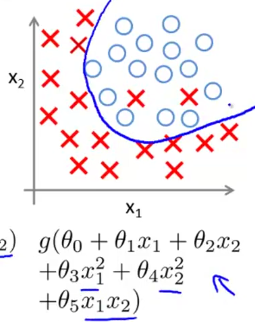
* Works well.
* Fit a 4th-order other polynomial to the data (5 parameters)



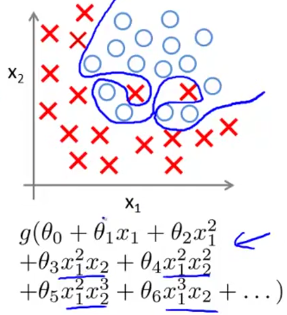
* Creates curve through all 5 training examples 🡪 seems to do a very good job fitting the training set, but it's going all over the place + is not such a good model
* **Overfitting** = this algorithm has **high variance**
* If fitting a high order polynomial, the hypothesis can fit can fit almost any function + we don't have enough data to constrain it to give us a good hypothesis
* Overfitting comes from if we have too many features, the learned hypothesis may fit training data very well (cost function ~= 0 ).
* But the function tries too hard to fit to the training set such that it fails to generalize to new examples and/or to predict prices on new examples
* **Generalized =** how well a hypothesis applies to new examples (data it has not seen in training set)
* A similar thing can apply to logistic regression as well.
* 1 thing we could do, is fit logistic regression w/ just a simple hypothesis + end up w/ a straight line to separate the positive and the negative class.



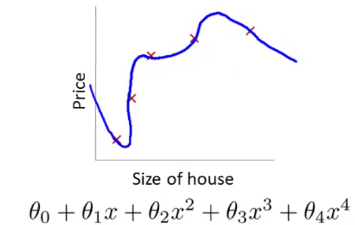
* Not very good fit = underfitting = hypothesis has high bias.
* Could add quadratic terms to your features these + get a decision boundary that's a good fit to data.



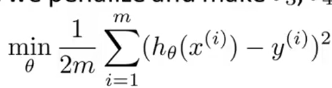
* If you fit a very high-order polynomial, logistical regression may contort itself to find a decision boundary that fits every single training example well.



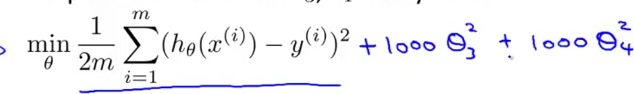
* This doesn't look like a very good hypothesis for making predictions = overfitting = hypothesis has high variance = unlikely to generalize well to new examples.
* In previous examples, we had 1 or 2 dimensional data so we could just plot h(x) + see what was going on + select the appropriate degree of polynomial.
* So plotting h(x) could be 1 way to try to decide what degree polynomial to use, but that doesn't always work.
* More often than not, we have a lot of features + its not just a matter of selecting what degree polynomial to use
* When we have so many features, it also becomes much harder to plot data + to decide what features to keep or not.
* If we have a lot of features + very little training data, overfitting can become a problem.
* To address over fitting, there are 2 main options
* 1 🡪 try to reduce the # of features
* manually look through list of features + decide which are more important + which to ignore
* But, by throwing away features, we also throw away some info we have about the problem.
* Maybe, ALL features are indeed useful for predicting housing prices so, we don't want to throw some of our info/features away.
* 2 🡪 **Regularization** = keep all features but reduce the **magnitude**/values of the parameters 
* This works well when we have a lot of features, each of which contributes a little bit to predicting the value of Y
* So, if we were to fit a quadratic function to data, we can get a pretty good fit
* Whereas, if fitting an overly high ordered polynomial, we end up w/ a curve that may fit training set very well, but overfit the data + not generalize well.



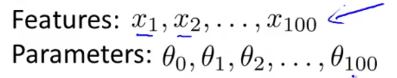
* Suppose we penalize, and, make the parameters theta 3 and theta 4 really small.
* Here's what I mean, here is our, or here is our optimization problem, where we minimize our usual squared error cause function. Let's
* Say we take the optimization objective (where we minimize our squared error cost function ) + modify it



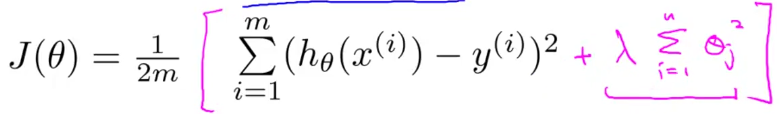
* Add to 1000\*theta3^2 + 1000\*theta4^2



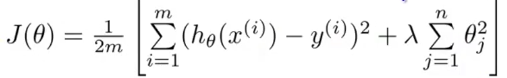
* The only way to make this new cost function small is if theta3 and 4 are small (close to 0), as if we're getting rid of these 2 terms at the end of the polynomial.
* If we do, w/ theta3 + 4 close to 0, we are left w/ a quadratic function plus tiny contributions from small terms, theta3 + theta4, + a good fit to the data
* We looked at the effect of penalizing 2 of the parameter values being large
* More generally, here is the idea behind **regularization =** Having small values for the parameters will somehow usually correspond to having a simpler hypothesis.
* If we penalize just theta 3 + theta 4 🡪 both were close to zero = wound up w/ a much simpler h(x) that was essentially a quadratic function.
* But more broadly, if we penalize ALL the parameters usually, we can think of that as trying to give us a simpler hypothesis
* More generally, it is possible to show that having smaller values of parameters corresponds to smoother, more simple functions (usually), which are therefore, less prone to overfitting.
* Specific example 🡪 For housing price prediction, we may 100 features +, unlike the polynomial example, we don't know that theta3 + theta4 are the high order polynomial terms.



* So, if we have just a set of a 100 features, it's hard to pick in advance which are the ones less likely to be relevant to try to shrink.
* In regularization, we take our cost function + modify it to shrink ALL parameters (b/c we don't know which try to shrink) via a new regularization term at the end.



* W/ this extra regularization term, we shrink every single parameter
* NOTE: By convention, the summation for the regularization term starts from 1 so we are not going penalize theta0 being large.
* makes very little difference theta0 or not in the summation in practice,
* But by convention, we usually regularize only theta1 through theta100.



* **lambda** = the **regularization parameter** 🡪 controls a trade-off between 2 different goals.
* 1 🡪 would like to fit training data well (captured by 1st term w/in the []’s)
* 2 🡪 want to keep parameters small (captured by the 2nd term = the regularization objective)
* Lambda/the regularization parameter controls the trade-off between these 2 goals + keeps the hypothesis relatively simple to avoid overfitting.
* Using the regularized objective w/ a high-ordered polynomial, you can get a curve that isn't quite a quadratic function, but is much smoother + simpler than the wiggly overfit + gives a much better hypothesis for this data.
* In regularized linear regression, if the regularization parameter monitor is set to be very large, we end up penalizing parameters theta1, theta2, theta3, theta4 very highly + they end up close to 0
* This is basically getting rid of those terms in h(x), we're just left w/ a h(x) = theta0 🡪 akin to fitting a *flat horizontal straight line* to the data = **underfitting**



* Another way of saying this is that this hypothesis has *too strong a preconception* or *too high bias* that housing prices are equal to theta0, despite clear data to the contrary,
* For regularization to work well, some care should be taken to choose a good regularization parameter lambda.