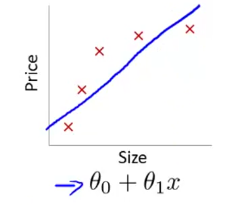
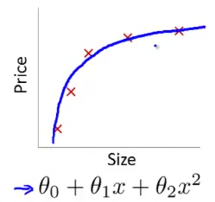
***Solving the Problem of Overfitting***

**I. THE PROBLEM OF OVERFITTING**

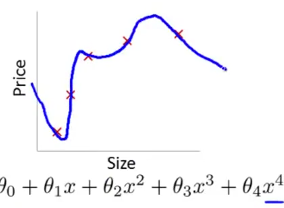
* When you apply linear + logistic regression to certain ML applications, they can run into a problem called **overfitting** that can cause them to perform very poorly.
* 1 thing we could do to our housing prices data set is fit a linear function to this data via linear regression and get a *straight*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



* This isn't a good model b/c as we see size increase, we see 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/evidence to the contrary* + it ends up a poor fit to the data.
* Model is too simple/uses too few features
* We could fit a quadratic function



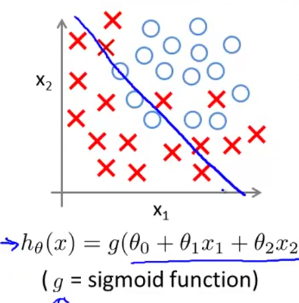
* This works well.
* We could fit a 4th-order other polynomial to the data (5 parameters)



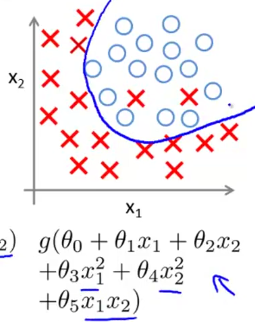
* This creates curve through all 5 training examples 🡪 seems to do a very good job fitting the training set, *but it's going all over the place + is not such a good model*
* **Overfitting** = this algorithm has **high variance**
* If fitting a high order polynomial, then it’s as if the hypothesis can fit can fit almost any function so the space of possible hypotheses is too large/variable + we don't have enough data to constrain it to give us a good hypothesis
* **Overfitting**: If we have too many features, the **learned hypothesis** may fit training data very well

)

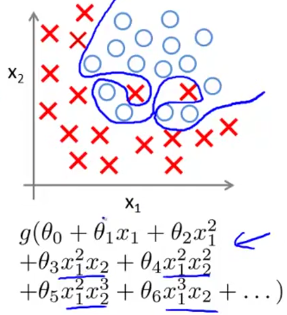
* But the function tries *too hard* to fit to the *training* set such that it fails to generalize to new examples/test data (to predict prices on new house size examples)
* **Generalize =** how well a hypothesis applies to new examples (data it has not seen in training set)
* Overfitting can apply to logistic regression as well.
* For a simple example with 2 features, x1 and x2, we could fit it w/ logistic regression w/ just a simple hypothesis + end up w/ a hypothesis with straight line to separate 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.



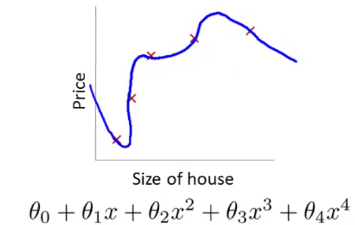
* that's a good fit to data
* If you fit a very high-order polynomial, logistical regression may contort itself to find a decision boundary that fits every single training example well.



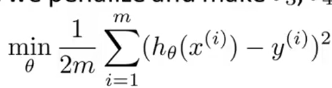
* This doesn't look like a very good hypothesis for making predictions as it tries too hard to fit every training example well
* Its **overfitting** = hypothesis has **high variance** = unlikely to generalize well to new examples.
* In previous examples, we had 1 or 2 dimensional data, so we could just plot hθ(x) + see what was going on + select the appropriate degree of polynomial.
* So plotting hθ(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 + it’s not just a matter of selecting what degree polynomial to use
* W/ too many features, it becomes much harder to plot data + to decide what features to keep
* *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
* 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 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.
* Model Selection *algorithm*
* **Regularization**
* Keep all features but reduce the **magnitude**/values of the parameters θ(j)
* This works well when we have a lot of features, each of which contributes a little bit to predicting the value of y

**II. COST FUNCTION**

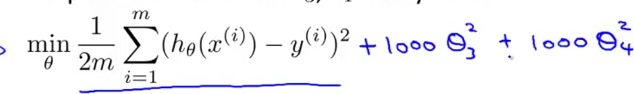
* So, remember that if we were to fit a quadratic function to data, we can get a pretty good fit
* Whereas, if fitting an overly-high-ordered polynomial, we end up w/ a curve that fits a training set very well, but overfits the data + does not generalize well.



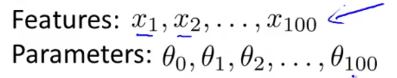
* Looking at the effect of penalizing 2 of the parameter values being large, suppose we penalize + make the parameters θ3 and θ4 really small.
* *\*\*\*\*\*Reduce the weight that some of the terms in our function carry by increasing their cost.*
* Our **optimization problem**, where we minimize our usual squared error cause function:



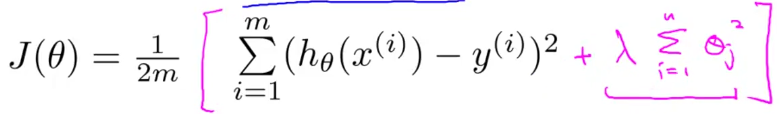
* Say we modify this objective + add to 1000\*θ3^2 + 1000\*θ4^2



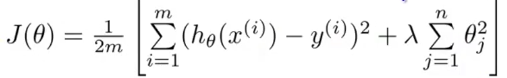
* The only way to make this *new* cost function small is if θ3 and θ4 are small (close to 0), as if we're getting rid of these 2 terms at the end of the polynomial.
* If θ3 + θ4 are close to 0, we’re left w/ a quadratic function plus tiny contributions from the very small terms θ3 + θ4 which gives a good fit to the data
* Idea behind **regularization:**
* *Having small values for parameters θ0-θn usually corresponds to having a “simpler” hypothesis*
* We penalize just θ3 + θ4 to make them close to 0 + wound up w/ a much simpler hθ(x), which was essentially a quadratic function.
* But more broadly, if we penalize ALL parameters usually, think of it as trying to give us a simpler hypothesis
* More generally, it’s possible to show that having smaller values of parameters corresponds to smoother, more simple functions (usually), which are therefore, less prone to overfitting.
* Ex: For housing price prediction, we may have 100 features +, unlike the polynomial example, we don't know that θ3 + θ4 are the high order polynomial terms.

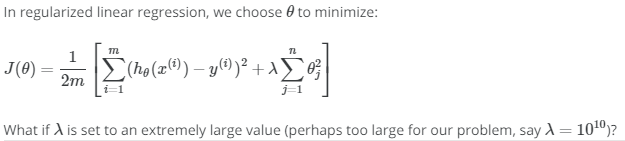


* So, if we have just a set of a 100 features, it's hard to pick in advance which are the ones less likely to be relevant so we don’t know which parameters to shrink.
* In regularization, we take our cost function for linear regression + modify it to shrink ALL parameters (b/c we don't know which to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



* W/ this extra regularization term, we shrink every single parameter from θ1- θn (not penalizing θ0 because the sum is from i = 1 to m)
* By convention, the summation for the regularization term starts from 1 so we’re don’t penalize θ0 being large.
* makes very little difference θ0 or not in the summation in practice,



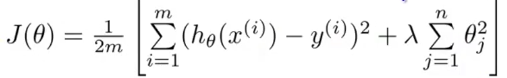
* **Lambda (λ)** = the **regularization parameter** 🡪 controls the trade-off between 2 different goals.
* 1 🡪 Would like to fit training data well (captured by 1st term in the objective/original objective)
* 2 🡪 Want to keep parameters small (captured by the 2nd term = the regularization objective)
* **λ**'s controlled trade-off between them 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 a wiggly overfit + gives a much better hypothesis for this data.
* 
* 
* In **regularized linear regression**, if the **regularization parameter** **λ** is set to be *very large*, we end up penalizing parameters θ1-θn very highly + they end up close to 0
* This is basically getting rid of all terms in hθ(x), we're just left w/ a hθ(x) = θ0
* This is akin to fitting a *flat horizontal straight line* to the data = **underfitting**



* Another way of saying this is that this hθ(x) has *too strong a preconception* or *too high bias* that housing prices are equal to θ0, despite clear data to the contrary
* For regularization to work well, some care should be taken to choose a good **λ**

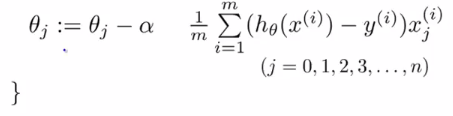
**III. REGULARIZED LINEAR REGRESSION**

* For linear regression, we have previously worked out 2 learning algorithms: 1 based on **gradient descent**, 1 based on the **normal equation**
* Optimization objective for regularized linear regression.

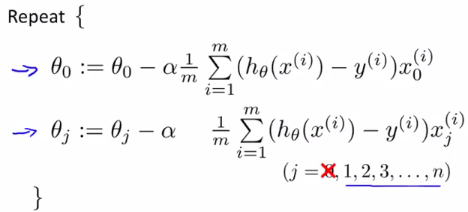


* 1st part = usual objective for linear regression, 2nd part = additional regularization term w/ regularization parameter **λ**
* We want to find parameters θ that minimizes this regularized cost function J(θ).
* Previously, we were using gradient descent for the original cost function *w/out* a regularization term

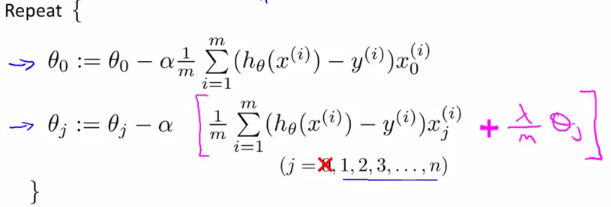




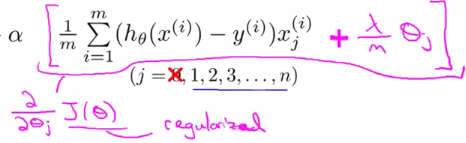
* We would repeatedly update the parameters θj for j0-j(n)
* Write the case for θ0 separately.



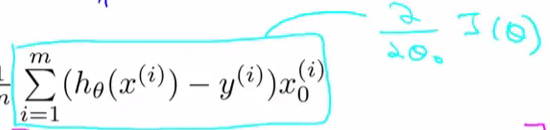
* We haven't changed anything yet, right.
* Remember that for regularized linear regression, *we don't penalize θ0.*
* So, when we modify this algorithm for gradient descent for regularized linear regression, we're going to end up treating θ0 slightly differently.
* Concretely, if we want to modify this algorithm to use the regularized objective, all we need to do is add λ/m times θj to the bottom term



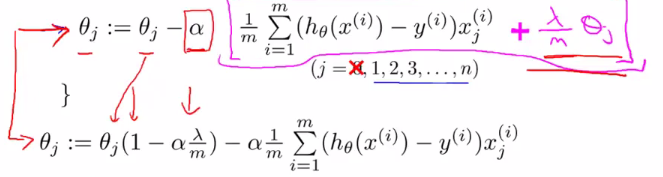
* If you implement this, then you have gradient descent for trying to minimize the regularized J(θ).
* W/in the pink square brackets is the partial derivative w/ respect to J(θ) using the *new* definition of J(θ) w/ the regularization term.



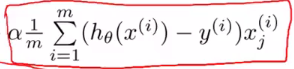
* Similarly, up on top we have the partial derivative of J(θ) w/ respect of θ0



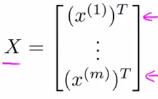
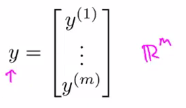
* If you look at the update for θj, it gets updated as θj – **α**\*this other term that depends on θj.
* So if you group all the terms that depend on θj together, you can show that this update can be written equivalently as:



* The term **(1 – α\*(λ/m))** is a pretty interesting term + has a pretty interesting effect.
* This term is going to be a number that is usually a little bit less than 1, because **α\*(λ/m)** is going to be positive, + if **α** is small + if m is large, it’s usually pretty small.
* Think of it as a number like 0.99
* Now say that θj gets replaced by θj\*0.99 🡪 this has the effect of *shrinking θj* a little bit towards 0.
* More formally, this makes the **squared norm** of θj a little bit smaller.
* After that, the second term is the same as the original gradient descent update we had before we added all the regularization stuff.



* So in *regularized* linear regression, on every iteration we're multiplying θj by a number that's a little bit < 1 to shrink the parameter a little bit + *then* performing a similar update as before.
* Of course, that's just the intuition behind what this particular update is doing.
* Mathematically what its doing is *exactly* gradient descent on the regularization cost function of J(θ) + the regularization term.
* Gradient descent was just 1 of 2 algorithms for fitting a linear regression model.
* The 2nd algorithm was based on the **normal equation**, where we had the **design matrix** X (n+1 dimensional where each row corresponded to the feature values to a separate training example) and an m-dimensional vector y that contained the actual labels/features from the training set.

* In order to minimize the cost function J(θ), we found 1 way to do so is to set



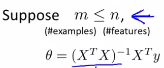
* This value for θ minimizes the cost when we are *NOT* using regularization.
* Now that we ARE using regularization (reducing weights to trade-off high costs), if you were to derive what the minimum of J(θ) is (by taking partial derivatives w/ respect to each parameter, set it to 0, + then do a bunch of calculus + show a formula above that minimizes the cost function)

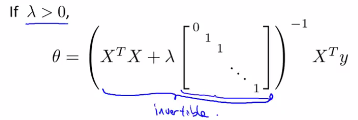
 🡺 

* Concretely, if using regularization, this formula changes as follows.



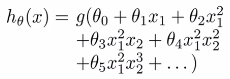
* This adds an “almost-identity” matrix, but w/ a 0 in the upper left-most entry.
* More generally, this matrix is an *(n+1) by (n+1) dimensional matrix*.
* If n = 2 🡺 
* It’s possible to prove that if using the new definition of J(θ) w/ the regularization objective, this new formula for θ is the global minimum of J(θ).
* **Non-invertibility**:
* Consider a setting where m <= n
* If you have fewer examples than features, X(t)\*X will be non-invertible, or **singular**/**degenerate**.
* Fortunately, regularization also takes care of this for us
* Concretely, so long as the regularization parameter **λ** is *strictly greater than 0*, it’s actually possible to prove that X(t)\*X + λ\*the almost-identity matrix, it’s possible to prove that this matrix will NOT be singular + WILL be invertible.
* So using regularization also takes care of any non-invertibility issues of the X(t)\*X matrix as well.



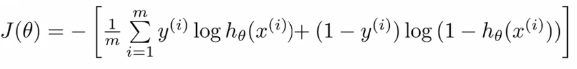


**IV. REGULARIZED LOGISTIC REGRESSION**

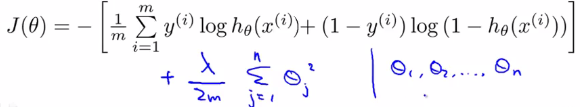
* For *logistic* regression, we previously talked about 2 types of optimization algorithms, gradient descent + other advanced optimization methods that require you to provide a way to compute your cost function J(θ) + a way to compute the derivatives.
* You can adapt both of those techniques in order to have them work for regularized logistic regression.
* 0:35
* We saw earlier that Logistic Regression can also be prone to overfitting if you fit it w/ a very high-order polynomial



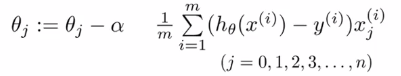
* Where g = the **sigmoid function** and in
* You’d end up w/ a hypothesis whose decision bound is an overly-complex + extremely-contorted function that really isn't such a great hypothesis for this training set
* More generally, if you have logistic regression w/ *a lot of features* (not necessarily polynomial ones), you can end up with overfitting.
* Here was our cost function for logistic regression:



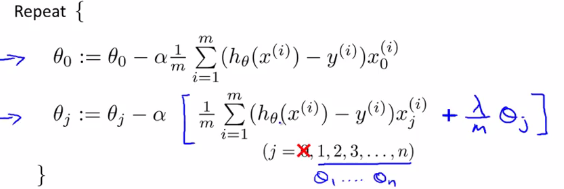
* To use regularization, add to it the following term:



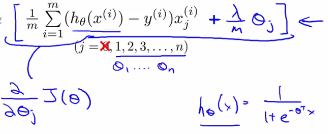
* This penalizes the parameters θ1- θn + keeps them from being too large.
* Even if fitting a very high-order polynomial w/ a lot of parameters, so long as you apply regularization to keep parameters small, you're more likely to get a more reasonable decision boundary.
* To implement this, look at the original gradient descent update algorithm

* Where hθ(x) is a different function (sigmoid)
* We want to now treat θ0 separately, + in order to modify this algorithm, to use a regularized cost function, just modify the 2nd update rule as before with the regularization term:

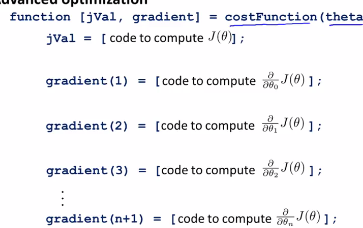


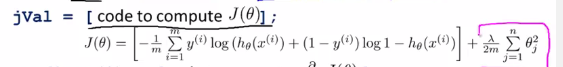
* Again, this looks identical to linear regression, our hθ(x) is different.
* Also, notice the term that **α** is multiplied by is actually the new partial derivative of J(θ) from earlier w/ respect to θ



* That was gradient descent for regularized linear regression.
* 



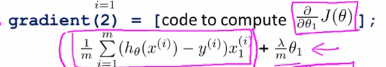
* Now let's talk about how to get regularized linear regression to work using the more advanced optimization methods.
* 
* For these methods, we needed to define the cost function that takes as input a parameter vector θ (θ0- θ(n+1)
* We use **fminunc**() to take the cost function and minimize it for us.
* The 2 main things the cost function needed to return were **jVval** via code to compute the new regularized logistic regression J(θ) w/ the additional regularization term at the end



* The other thing this cost function needs to provide with a gradient
* So gradient 1 needs to be set to the partial derivative of J(θ) w/ respect to θ0, gradient 2 w/ respect to θ1, and so on.
* The derivative for θ0 doesn't change compared to the non-regularized version



* The other terms DO change.



* So if we implement this cost function + pass it into **fminunc** or another advanced optimization techniques, we will minimize the new regularized cost function J(θ) + the parameters we get out will be the ones that correspond to logistic regression with regularization.
* In Silicon Valley, there are a lot of engineers that are frankly, making a ton of money for their companies using ML algorithms.
* If you *understand* linear regression, advanced optimization algorithms, + regularization, then by now, frankly, you probably know quite a lot more ML than many (certainly not all) engineers out there having very successful careers, making tons of $$$ for companies or building products.