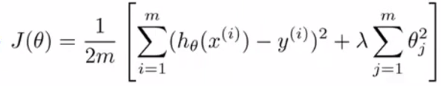
***Evaluating a Learning Algorithm***

**I. DECIDING WHAT TO TRY NEXT**

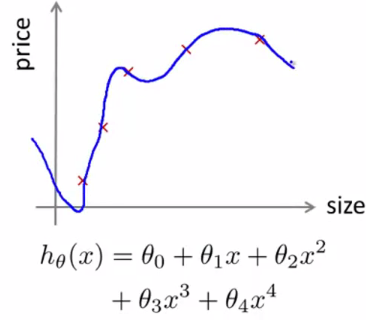
* Even among people that know a certain learning algorithm, there's often a huge difference between someone that knows how to powerfully + effectively apply it vs. someone who doesn't understand how to apply it + can end up wasting a lot of time trying things out that don't really make sense
* If developing ML systems, you should know how to choose the most promising avenues to spend time pursuing
* Suppose you are developing a ML system or trying to improve the performance of a ML system, how do you go about deciding what are the promising avenues to try next?
* To explain this, let's continue using our example of learning to predict housing prices.
* Let's say you've implement and ***regularized* linear regression**, thus minimizing the cost function J



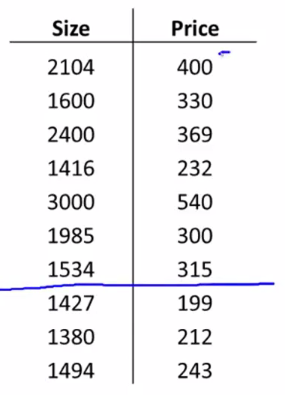
* Now suppose that after taking your learned parameters + testing your hypothesis on a new set of houses, you find huge errors in its predictions of the housing prices.
* The question is *what should you then try next in order to improve the learning algorithm*?
* There are many things that one could do to improve the performance of the learning algorithm.
* Get more training examples.
* The sad thing a lot of people spend a lot of time collecting more training examples, thinking if they have twice as much or ten times as much training data, it’s certainly going to help
* But sometimes getting more training data doesn't actually help
* Try a smaller set of features.
* Spend time carefully selecting some small subset of features to prevent **overfitting**.
* Try to get additional features.
* Maybe the current set of features aren't informative enough + you want to collect more data via getting more features.
* This is the sort of project that can scale way, way up
* It would be nice to know in advance if this is going to help before we spend a lot of time and $$$ doing something like this.
* Try adding polynomial features
* Try other things like increasing or decreasing lambda (the regularization parameter)
* Some of these options can easily scale up to 6 month or longer projects.
* Unfortunately, the most common method people use to pick one of these is to go by gut feeling.
* What many people will do is *randomly pick* 1 of these options
* So people spend 6 months doing 1 of these avenues chosen at random only to discover 6 months later that that really wasn't a promising avenue to pursue.
* Fortunately, there is a pretty simple technique that can let you very quickly rule out half the things on this list as not being potentially promising things to pursue.
* It can potentially save a lot of time pursuing something that's just is not going to work.
* First, we have to **evaluate learning algorithms** + then perform **machine learning diagnostics**.
* A **diagnostic** = a test you can run to get insight into what is or isn't working w/ an algorithm + gain guidance as to how best to improve its performance
* This will often give insight as to what are promising things to try to improve a learning algorithm's performance.
* Diagnostics can take time to implement + can sometimes take quite a lot of time to implement + understand, but doing so can be a very good use of time when developing learning algorithms
* They can often save you from spending many months pursuing an avenue that is not going to be fruitful by identifying this much earlier

**II. EVALUATING A HYPOTHESIS LEARNED BY AN ALGORITHM**

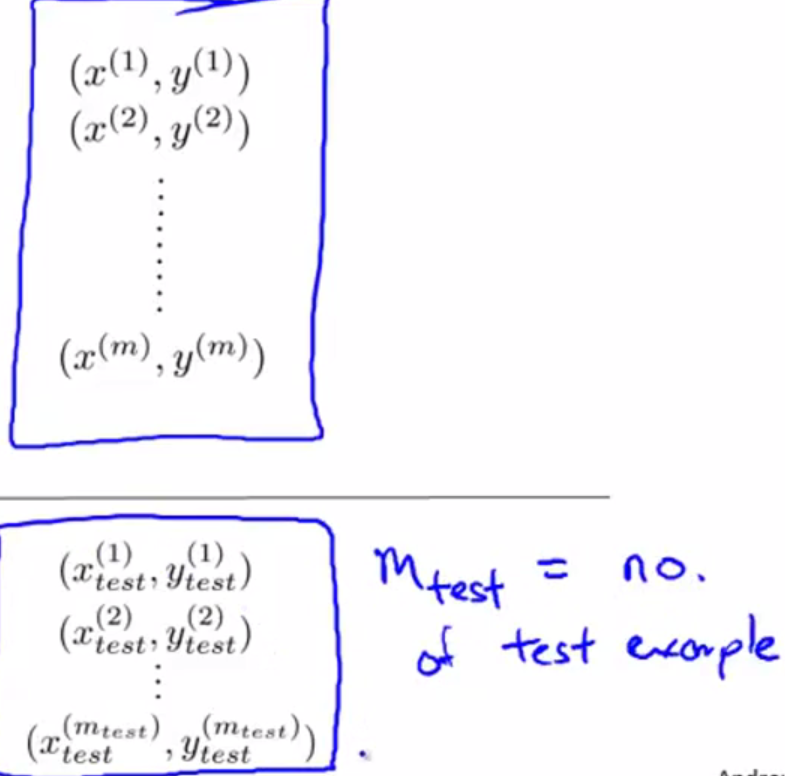
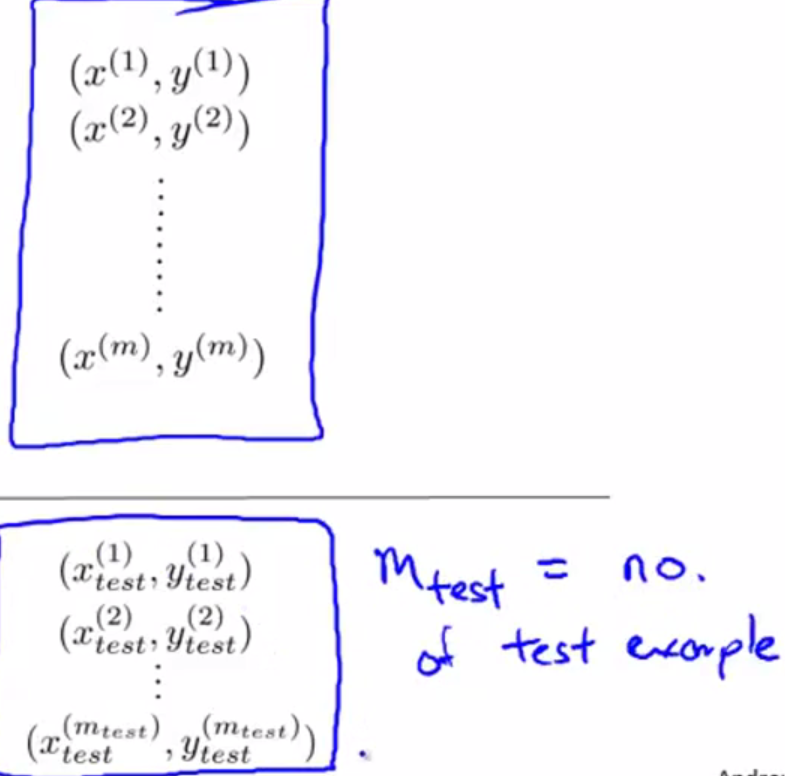
* When we fit the parameters of a learning algorithm, we think about choosing parameters to minimize the training error.
* One might think getting a really low value might be a good thing, but we’ve already seen that just b/c a hypothesis has low training error, it doesn't mean it’s necessarily a good hypothesis (model)



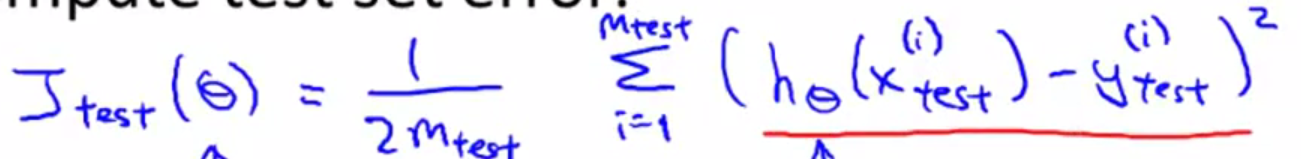
* i.e. A hypothesis can overfit + therefore *fail to generalize* to new examples (not in the training set)
* So how do you tell if the hypothesis might be overfitting?
* In general, for problems w/ > 1 feature, it becomes hard/maybe impossible to plot what a hypothesis looks like, so we need some other way to evaluate our hypothesis.
* The standard way to evaluate a learned hypothesis is as follows:
* Suppose we have a data set (10 training examples below, but we may have dozens, hundreds, or maybe thousands of training examples)
* In order to make sure we can evaluate our hypothesis, we **split** the data we have into 2 portions.



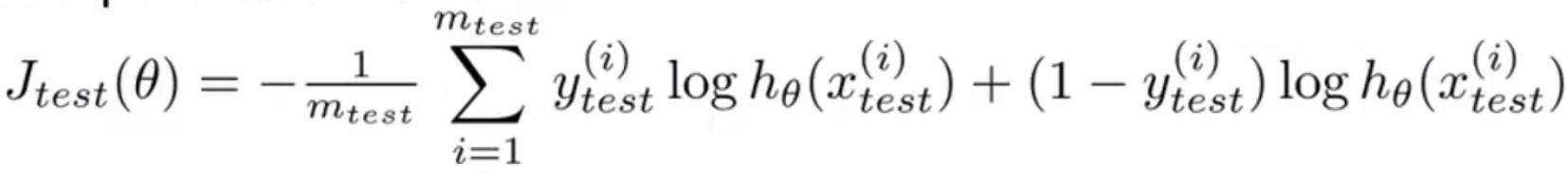
* The 1st portion is our usual training set + the 2nd is our test set
* A pretty typical split is around say a 70/30 split w/ more data going into the training set

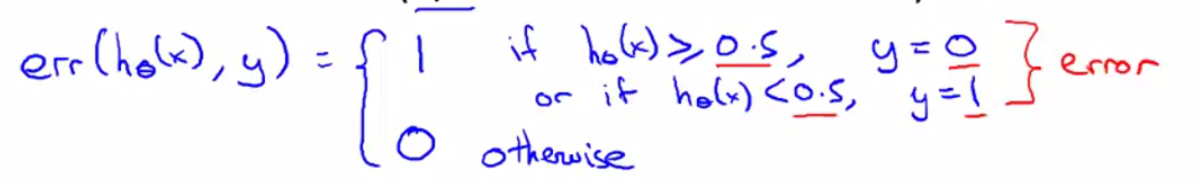
* {x\_test(1), y\_test(1)} = my 1st test example
* If there is any sort of ordering to the data, it’s better to set a random 70% of data to training + a random 30% of data to test.
* Here is a fairly typical procedure for how to train + test a learning algorithm
* 1) Learn the parameters ϴ from the training set to complete the usual objective of minimizing the training error = J(ϴ)/cost function, where J(ϴ) was defined using 70% of all data you have.
* 2) Compute the test error, **J\_test(ϴ)**
* Take your parameters ϴ you learned from the training set, plug it in here, + compute the test set error.



* This is basically the **average squared error** as *measured on your test set*.
* So if we run every test example through the hypothesis w/ parameter ϴ + measure the squared error hϴ(x)has on your *m\_test examples*.
* NOTE: this is the definition of the test set error if we’re using linear regression + using the squared error metric.
* How about if we were doing a classification problem + using logistic regression instead?
* In that case, the procedure for training and testing say logistic regression is pretty similar
* 1) Learn the parameters from the training data (1st 70% of the data)
* 2) Compute the test error w/ the same objective function we always use for logistic regression, except now we define it using our m\_test examples.



* While this definition of the test set error J\_test(ϴ) is perfectly reasonable, sometimes we can use an alternative test set metric that might be easier to interpret, the **misclassification error (0/1 misclassification error)**
* Let me define the error of a prediction given the label y **err(hϴ(x), y)**
* Then set this to 1 if hϴ(x)outputs a value >= 0.5 + y = 0 OR if hϴ(x)outputs a value < 0.5 + y = 1
* So both of these cases respond to if your hϴ(x)mislabeled an example, assuming a threshold = 0.5
* Either hϴ(x)thought a value was more likely to be 1, but it was actually 0, or thought it was more likely to be 0, but the label was actually 1.
* Otherwise, we define this error function to be 0 if hϴ(x)classified an example’s y correctly.



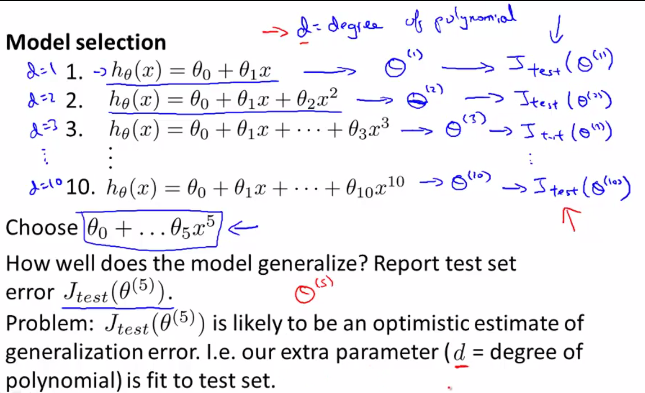
* We could then define the **test error using the misclassification error metric**, to be:



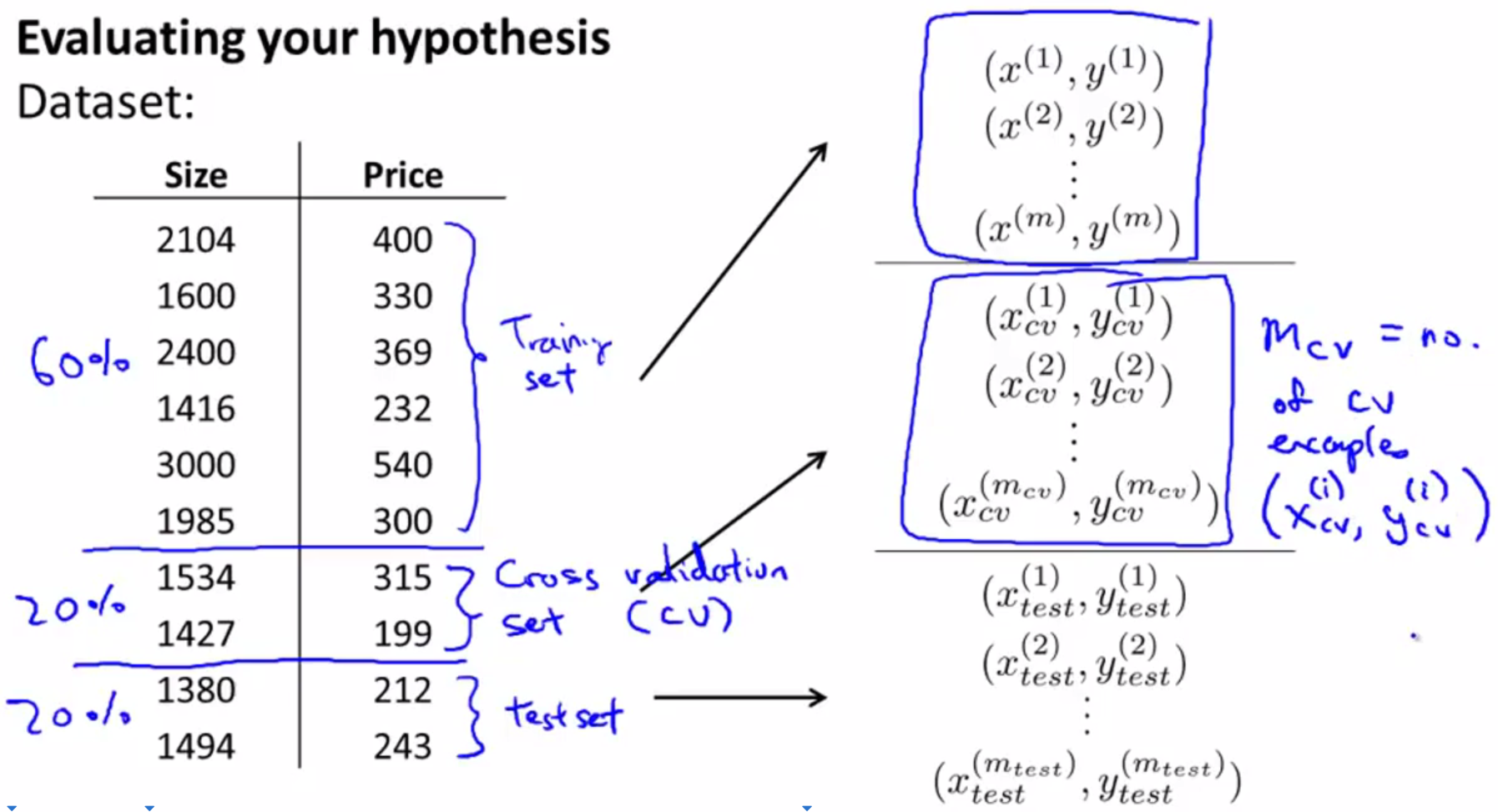
* This is exactly the fraction of examples in my test set that hϴ(x) has mislabeled

**III. MODEL SELECTION AND TRAIN/VALIDATION/TEST SETS**

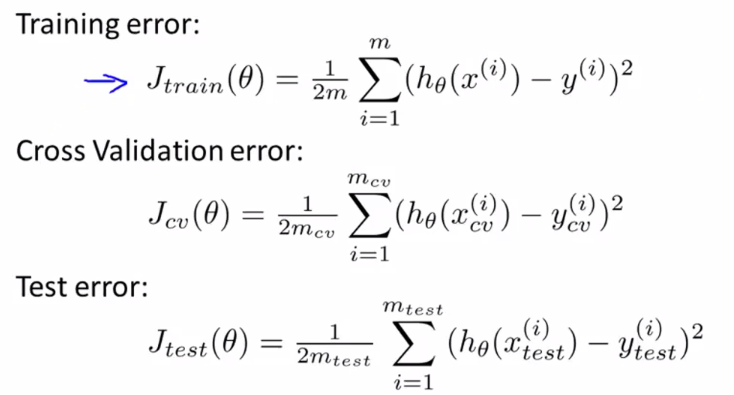
* Suppose you'd like to decide what degree of polynomial to fit to a data set, what features to include in a learning algorithm, or what the regularization parameter lambda for a learning algorithm is.
* These are **model selection problems**
* We've already now the problem of overfitting + why training set error is not a good predictor for how well hϴ(x) will do on new examples.
* A more general principle is that once parameters are fit to some set of data (maybe the training set, maybe something else), the error of that hϴ(x) as measured on that same data set is unlikely to be a good estimate of the **actual generalization error** (how well hϴ(x) will generalize to new examples)
* Let's say you're trying to choose what degree polynomial to fit to data (linear, quadratic, cubic, or a 10th-order polynomial.
* In addition to the ϴ parameters, there's 1 more parameter, **d**, you're trying to determine using your data set.
* The 1st option is d = 1 for a linear function, and then d = 2 up to d = 10
* Let's say that you want to choose a model/degree of polynomial, fit that model, + get some estimate of how well your fitted hypothesis will generalize to new examples.
* One thing you could do is take the 1st model (d = 1), minimize the training error to get some parameter vector ϴ1
* Then take your 2nd model (d = 2, a quadratic function), fit *that* to your training set, this give you some other parameter vector ϴ2, and continue on up to d = 10
* Then can compute J\_test(ϴ1) – J\_test(ϴ10) 🡪 take each of my hypotheses w/ their corresponding parameters + measure the performance on the test set.
* Then, to select one of these models, see which model has the *lowest test set error,* for example, the 5th-order polynomial.
* Now let's say I want to take this hypothesis + see how well this model generalizes
* 1 thing I could do is look at how well the 5th -order polynomial hϴ(x) performs on the test set.
* But the *problem* is this will NOT be a fair estimate of how well hϴ(x) generalizes
* This is b/c we've fit this extra parameter, **d**, using the test set
* Namely, we *chose* the value of d that gave us the best possible test set performance *(Chose d depending on the test set)*
* So, the performance of my parameter vector ϴ5 on the test set is likely to be an overly-optimistic estimate of generalization error.
* B/c we fit this parameter d to my test set, it’s no longer fair to evaluate my hϴ(x) on this test set, b/c I *fit my parameters to this test set*, *I've chosen the degree d of polynomial using the test set*.
* So my hϴ(x) is likely to do better on *this* test set than on new examples it hasn't seen before



* Just to reiterate, if we fit some set of parameters, ϴ0-ϴ10, to some training set, then the performance of the fitted model on the training set is *NOT predictive* of how well the hϴ(x) will generalize to new examples.
* This is b/c these parameters were fit to THAT training set, so they're likely to do well on the training set, even if the parameters don't do well on other examples.
* The SAME THING happens when we fit a parameter **d** to the *test set*.
* By fitting d to the test set, the performance of hϴ(x) on THAT test set may not be a fair estimate of how well hϴ(x) is likely to do on examples we haven't seen before.
* To address this problem, in a model selection setting, if we want to evaluate a hϴ(x), we usually do the following instead:
* Given a data set, instead of just splitting into a training + test set, split it into 3 pieces = training, **cross-validation**, + test
* A typical ratio at which to split these things will be to send 60% = training, 20% = validation, + 20% = test set.
* Our validation set, will have some number of examples. I'm going to denote that m\_cv

\

* Now that we've defined the sets, we can also define the different errors



* So when faced with a model selection problem, instead of using the *test* set to select a model, we're instead going to use the validation set,
* Concretely, we're going to take our 1st hypothesis/model + minimize the cost function, which gives some parameter vector ϴ for the linear model.
* Then continue on for the rest of the model options
* Now, instead of testing these hypotheses on the test set, *test them on the cross validation set* + measure J\_cv(ϴn) to see how well each of these hypotheses do on my cross-validation set.
* Then we pick the hϴ(x) w/ the lowest CV error.
* This means is that that degree of polynomial parameter ***d*** *was fit using the CV set* + is no longer fit to the test set
* We've saved away the test set + can use it to measure/estimate the generalization error of the model selected by the CV set
* Now we you take data, split it into a training, validation, + test sets, + then use your CV data to select a model + evaluate it on the test set.
* In ML machine learning, as it is practiced today, people are selecting models using a test set + using the same test set to report the error as though it were a good estimate of generalization error.
* Unfortunately, many people are doing this practice
* If you have a massive, massive test set, this isn’t maybe not *such* a terrible thing to do, but most ML practitioners tend to advise against this
* It's considered better practice to have validation and test sets.