CHAPTER 2

Regression

Regression is an important machine-learning problem that provides a good starting point for diving deeply into the field.

"Regression," in common parlance, means moving backwards. But this is forward progress!

2.1 Problem formulation

A *hypothesis* h is employed as a model for solving the regression problem, in that it maps inputs x to outputs y,

$$x \rightarrow \boxed{h} \rightarrow y \quad ,$$

where $x \in \mathbb{R}^d$ (i.e., a length d column vector of real numbers), and $y \in \mathbb{R}$ (i.e., a real number). Real life rarely gives us vectors of real numbers; the x we really want to take as input is usually something like a song, image, or person. In that case, we'll have to define a function $\varphi(x)$, whose range is $\mathbb{R}^d$, where φ represents *features* of x, like a person's height or the amount of bass in a song, and then let the $h : \varphi(x) \rightarrow \mathbb{R}$. In much of the following, we'll omit explicit mention of φ and assume that the $x^{(i)}$ are in $\mathbb{R}^d$, but you should always have in mind that some additional process was almost surely required to go from the actual input examples to their feature representation, and we'll talk a lot more about features later in the course.

Regression is a *supervised learning* problem, in which we are given a training dataset of the form

$$\mathcal{D}_n = \left\{ \left(x^{(1)}, y^{(1)} \right), \ldots, \left(x^{(n)}, y^{(n)} \right) \right\} \quad ,$$

which gives examples of input values $x^{(i)}$ and the output values $y^{(i)}$ that should be associated with them. Because y values are real-valued, our hypotheses will have the form

$$h : \mathbb{R}^d \rightarrow \mathbb{R} \quad .$$

This is a good framework when we want to predict a numerical quantity, like height, stock value, etc., rather than to divide the inputs into discrete categories.

What makes a hypothesis useful? That it works well on *new* data; that is, that it makes good predictions on examples it hasn't seen. But we don't know exactly what data this hypothesis might be tested on when we use it in the real world. So, we have to *assume* a connection between the training data and testing data – typically, the assumption is that

My favorite analogy is to problem sets. We evaluate a student's ability to *generalize* by putting questions on the exam that were not on the homework (training set).

they (the training and testing data) are drawn independently from the same probability distribution.

To make this discussion more concrete, we have to provide a *loss function*, to say how unhappy we are when we guess an output g given an input x for which the desired output was a.

Given a training set $\mathcal{D}_n$ and a hypothesis h with parameters Θ, we can define the *training error* of h to be the average loss on the training data:

$$\mathcal{E}_n(h;\Theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(h(x^{(i)};\Theta), y^{(i)}) \ , \tag{2.1}$$

The training error of h gives us some idea of how well it characterizes the relationship between x and y values in our data, but it isn't the quantity that we *most* care about. What we most care about is *test error*:

$$\mathcal{E}(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'} \mathcal{L}(h(x^{(i)}), y^{(i)})$$

on n' new examples that were not used in the process of finding the hypothesis.

It might be worthwhile to stare at the two errors and think about what's the difference. For example, notice how Θ is no long a variable in the testing error? this is because in evaluating the testing error, the parameters will have been "picked"/"fixed" already.

For now, we will try to find a hypothesis with small training error (later, with some added criteria) and try to make some design choices so that it *generalizes well* to new data, meaning that it also has a small *test error*.

2.2 Regression as an optimization problem

Given data, a loss function, and a hypothesis class, we need a method for finding a good hypothesis in the class. One of the most general ways to approach this problem is by framing the machine learning problem as an optimization problem. One reason for taking this approach is that there is a rich area of math and algorithms studying and developing efficient methods for solving optimization problems, and lots of very good software implementations of these methods. So, if we can turn our problem into one of these problems, then there will be a lot of work already done for us!

We begin by writing down an *objective function* $J(\Theta)$, where Θ stands for *all* the parameters in our model (i.e., all possible choices over parameters). We often write $J(\Theta;\mathcal{D})$ to make clear the dependence on the data $\mathcal{D}$.

Don't be too perturbed by the semicolon where you expected to see a comma! It's a math way of saying that we are mostly interested in this as a function of the arguments before the ";", but we should remember that there's a dependence on the stuff after it, as well.

The objective function describes how we feel about possible hypotheses Θ: we will generally look for values for parameters Θ that minimize the objective function:

$$\Theta^* = \arg\min_{\Theta} J(\Theta) \ .$$

A very common form for a machine-learning objective is

$$J(\Theta) = \left(\frac{1}{n}\sum_{i=1}^{n} \underbrace{\mathcal{L}(h(x^{(i)};\Theta), y^{(i)})}_{\text{loss}}\right) + \underbrace{\lambda}_{\text{non-negative constant}} R(\Theta). \tag{2.2}$$

The *loss* tells us how unhappy we are about the prediction $h(x^{(i)};\Theta)$ that Θ makes for $(x^{(i)}, y^{(i)})$. Minimizing this loss makes the prediction better. The *regularizer* is an additional term that encourages the prediction to remain general, and the constant λ adjusts the balance between reproducing seen examples, and being able to generalize to unseen examples. We will return to discuss this balance, and more about the idea of regularization, in Section 2.6.

You can think about Θ^* here as "the theta that minimizes J": $\arg\min_x f(x)$ means the value of x for which $f(x)$ is the smallest. Sometimes we write $\arg\min_{x\in\mathcal{X}} f(x)$ when we want to explicitly specify the set $\mathcal{X}$ of values of x over which we want to minimize.

2.3 Linear regression

To make this discussion more concrete, we have to provide a hypothesis class and a loss function.

We will begin by picking a class of hypotheses $\mathcal{H}$ that we think might provide a good set of possible models of the relationship between x and y in our data. We will start with a very simple class of *linear* hypotheses for regression. It is both simple to study and very powerful, and will serve as the basis for many other important techniques (even neural networks!).

In linear regression, the set $\mathcal{H}$ of hypotheses has the form

$$h(x; \theta, \theta_0) = \theta^T x + \theta_0 \quad , \tag{2.3}$$

with model parameters $\Theta = (\theta, \theta_0)$. In one dimension ($d = 1$) this has the same familiar slope-intercept form as $y = mx+b$; in higher dimensions, this model describes the so-called hyperplanes.

We define a *loss function* to describe how to evaluate the quality of the predictions our hypothesis is making, when compared to the "target" y values in the data set. The choice of loss function is part of modeling your domain. In the absence of additional information about a regression problem, we typically use *squared loss*:

$$\mathcal{L}(g, a) = (g - a)^2 \quad .$$

where $g = h(x)$ is our "guess" from the hypothesis, and a is the "actual" observation (in other words, here a is being used equivalently as y). With this choice of squared loss, the average loss as generally defined in 2.1 will become the so-called *mean squared error (MSE)*, which we'll study closely very soon.

The squared loss penalizes guesses that are too high the same amount as it penalizes guesses that are too low, and has a good mathematical justification in the case that your data are generated from an underlying linear hypothesis with the so-called Gaussian-distributed noise added to the y values. But there are applications in which other losses would be better, and much of the framework we discuss can be applied to different loss functions, although this one has a form that also makes it particularly computationally convenient.

We won't get into the details of Gaussian distribution in our class; but it's one of the most important distributions and well-worth studying closely at some point. One obvious fact about Gaussian is that it's symmetric; this is in fact one of the reasons squared loss works well under Gaussian settings, as the loss is also symmetric.

Our objective in linear regression will be to find a hyperplane that goes as close as possible, on average, to all of our training data.

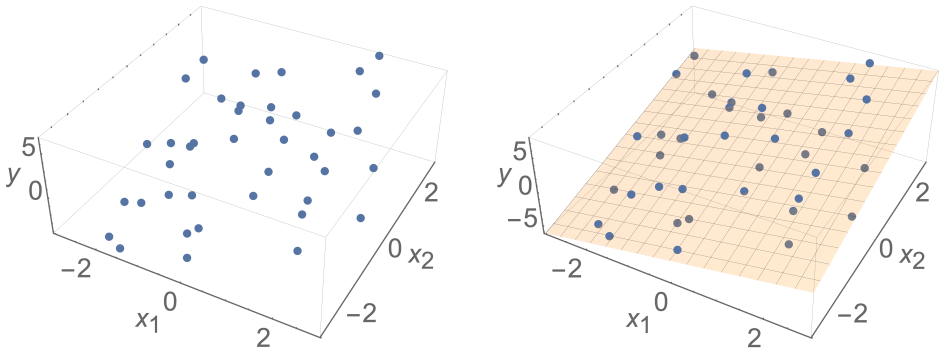
Applying the general optimization framework to the linear regression hypothesis class of Eq. 2.3 with squared loss and no regularization, our objective is to find values for $\Theta = (\theta, \theta_0)$ that minimize the MSE:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left(\theta^T x^{(i)} + \theta_0 - y^{(i)} \right)^2 \quad , \tag{2.4}$$

resulting in the solution:

$$\theta^*, \theta_0^* = \arg\min_{\theta, \theta_0} J(\theta, \theta_0) \quad . \tag{2.5}$$

For one-dimensional data ($d = 1$), this becomes the familiar problem of fitting a line to data. For $d > 1$, this hypothesis may be visualized as a d-dimensional hyperplane embedded in a $(d + 1)$-dimensional space (that consists of the input dimension and the y dimension). For example, in the left plot below, we can see data points with labels y and input dimensions x_1 and x_2. In the right plot below, we see the result of fitting these points with a two-dimensional plane that resides in three dimensions. We interpret the plane as representing a function that provides a y value for any input (x_1, x_2).



A richer class of hypotheses can be obtained by performing a non-linear feature transformation before doing the regression, as we will later see (in Chapter 5), but it will still end up that we have to solve a linear regression problem.

2.4 A gloriously simple linear regression algorithm

Okay! Given the objective in Eq. 2.4, how can we find good values of θ and θ_0? We'll study several general-purpose, efficient, interesting algorithms. But before we do that, let's start with the simplest one we can think of: *guess a whole bunch (*k*) of different values of* θ *and* θ_0, see which one has the smallest error on the training set, and return it.

RANDOM-REGRESSION$(\mathcal{D}, k)$

1 For i in $1 \ldots k$: Randomly generate hypothesis $\theta^{(i)}, \theta_0^{(i)}$
2 Let $i = \arg\min_i J(\theta^{(i)}, \theta_0^{(i)}; \mathcal{D})$
3 Return $\theta^{(i)}, \theta_0^{(i)}$

This seems kind of silly, but it's a learning algorithm, and it's not completely useless.

Study Question: If your data set has n data points, and the dimension of the x values is d, what is the size of an individual $\theta^{(i)}$?

Study Question: How do you think increasing the number of guesses k will change the training error of the resulting hypothesis?

2.5 Analytical solution: ordinary least squares

One very interesting aspect of the problem of finding a linear hypothesis that minimizes mean squared error is that we can find a closed-form formula for the answer! This general problem is often called the *ordinary least squares* (OLS)

Everything is easier to deal with if we assume that all of the the $x^{(i)}$ have been augmented with an extra input dimension (feature) that always has value 1, so that they are in $d+1$ dimensions, and rather than having an explicit θ_0, we let it be the last element of our θ vector, so that we have, simply,

$$y = \theta^T x \quad .$$

What does "closed form" mean? Generally, that it involves direct evaluation of a mathematical expression using a fixed number of "typical" operations (like arithmetic operations, trig functions, powers, etc.). So equation 2.5 is not in closed form, because it's not at all clear what operations one needs to perform to find the solution.

In this case, the objective becomes

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \quad . \tag{2.6}$$

Study Question: Stop and prove to yourself that adding that extra feature with value 1 to every input vector and getting rid of the θ_0 parameter is equivalent to our original model.

We approach this just like a minimization problem from calculus homework: take the derivative of J with respect to θ, set it to zero, and solve for θ. There are additional steps required, to check that the resulting θ is a minimum (rather than a maximum or an inflection point) but we won't work through that here. It is possible to approach this problem by:

- Finding $\partial J/\partial\theta_k$ for k in $1, \ldots, d$,
- Constructing a set of k equations of the form $\partial J/\partial\theta_k = 0$, and
- Solving the system for values of θ_k.

We will use d here for the total number of features in each $x^{(i)}$, including the added 1.

That works just fine. To get practice for applying techniques like this to more complex problems, we will work through a more compact (and cool!) matrix view. Along the way, it will be helpful to collect all of the derivatives in one vector. In particular, the gradient of J with respect to θ is following column vector of length d:

$$\nabla_\theta J = \begin{bmatrix} \partial J/\partial\theta_1 \\ \vdots \\ \partial J/\partial\theta_d \end{bmatrix} .$$

Study Question: Work through the next steps and check your answer against ours below.

We can think of our training data in terms of matrices X and Y, where each column of X is an example, and each "column" of Y is the corresponding target output value:

$$X = \begin{bmatrix} x_1^{(1)} & \ldots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \ldots & x_d^{(n)} \end{bmatrix} \quad Y = \begin{bmatrix} y^{(1)} & \ldots & y^{(n)} \end{bmatrix} \quad .$$

Study Question: What are the dimensions of X and Y?

In most textbooks, they think of an individual example $x^{(i)}$ as a row, rather than a column. So that we get an answer that will be recognizable to you, we are going to define a new matrix and vector, $\tilde{X}$ and $\tilde{Y}$, which are just transposes of our X and Y, and then work with them:

$$\tilde{X} = X^T = \begin{bmatrix} x_1^{(1)} & \ldots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(n)} & \ldots & x_d^{(n)} \end{bmatrix} \quad \tilde{Y} = Y^T = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix} \quad .$$

Study Question: What are the dimensions of $\tilde{X}$ and $\tilde{Y}$?

Now we can write

$$J(\theta) = \frac{1}{n} \underbrace{(\tilde{X}\theta - \tilde{Y})^T}_{1\times n} \underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1} = \frac{1}{n} \sum_{i=1}^{n} \left(\left(\sum_{j=1}^{d} \tilde{X}_{ij}\theta_j \right) - \tilde{Y}_i \right)^2$$

and using facts about matrix/vector calculus, we get

$$\nabla_\theta J = \frac{2}{n} \underbrace{\tilde{X}^T}_{d\times n} \underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1} \quad . \tag{2.7}$$

You should be able to verify this by doing a simple (say, 2×2) example by hand.

See Appendix A for a nice way to think about finding this derivative.

Setting $\nabla_\theta J$ to 0 and solving, we get:

$$\begin{aligned} \frac{2}{n}\tilde{X}^T(\tilde{X}\theta - \tilde{Y}) &= 0 \\ \tilde{X}^T\tilde{X}\theta - \tilde{X}^T\tilde{Y} &= 0 \\ \tilde{X}^T\tilde{X}\theta &= \tilde{X}^T\tilde{Y} \\ \theta &= (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T\tilde{Y} \end{aligned}$$

And the dimensions work out!

$$\theta = \underbrace{(\tilde{X}^T\tilde{X})^{-1}}_{d\times d} \underbrace{\tilde{X}^T}_{d\times n} \underbrace{\tilde{Y}}_{n\times 1}$$

So, given our data, we can directly compute the linear regression that minimizes mean squared error. That's pretty awesome!

2.6 Regularization

The objective function of Eq. 2.2 balances (training-data) memorization, induced by the *loss* term, with generalization, induced by the *regularization* term. Here, we address the need for regularization specifically for linear regression, and show how this can be realized using one popular regularization technique called *ridge regression*.

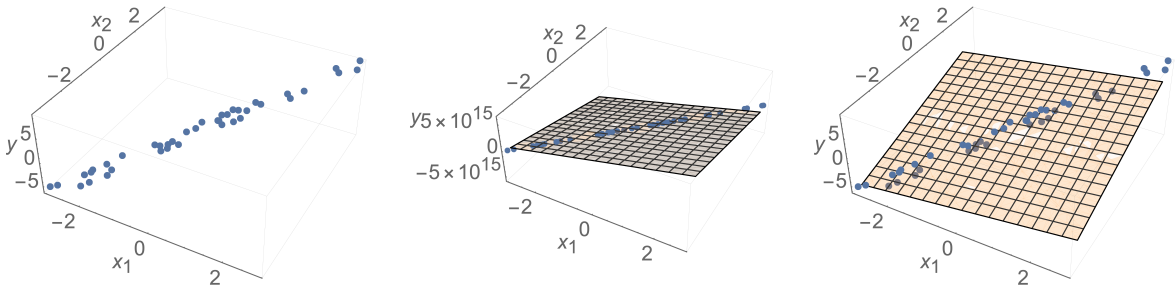
2.6.1 Regularization and linear regression

If all we cared about was finding a hypothesis with small loss on the training data, we would have no need for regularization, and could simply omit the second term in the objective. But remember that our ultimate goal is to *perform well on input values that we haven't trained on!* It may seem that this is an impossible task, but humans and machine-learning methods do this successfully all the time. What allows *generalization* to new input values is a belief that there is an underlying regularity that governs both the training and testing data. One way to describe an assumption about such a regularity is by choosing a limited class of possible hypotheses. Another way to do this is to provide smoother guidance, saying that, within a hypothesis class, we prefer some hypotheses to others. The regularizer articulates this preference and the constant λ says how much we are willing to trade off loss on the training data versus preference over hypotheses.

For example, consider what happens when $d = 2$, and x_2 is highly correlated with x_1, meaning that the data look like a line, as shown in the left panel of the figure below. Thus, there isn't a unique best hyperplane. Such correlations happen often in real-life

Sometimes there's technically a unique best hyperplane, but just because of noise.

data, because of underlying common causes; for example, across a population, the height of people may depend on both age and amount of food intake in the same way. This is especially the case when there are many feature dimensions used in the regression. Mathematically, this leads to $\tilde{X}^T\tilde{X}$ close to singularity, such that $(\tilde{X}^T\tilde{X})^{-1}$ is undefined or has huge values, resulting in unstable models (see the middle panel of figure and note the range of the y values—the slope is huge!):



A common strategy for specifying a *regularizer* is to use the form

$$R(\Theta) = \left\|\Theta - \Theta_{prior}\right\|^2$$

when we have some idea in advance that Θ ought to be near some value Θ_{prior}.

Here, the notion of distance is quantified by squaring the l_2 *norm* of the parameter vector: for any d-dimensional vector $v \in \mathbb{R}^d$, the l_2 *norm* of v is defined as,

$$\|v\| = \sqrt{\sum_{i=1}^{d} |v_i|^2} \ .$$

Learn about Bayesian methods in machine learning to see the theory behind this and cool results!

In the absence of such knowledge a default is to *regularize toward zero*:

$$R(\Theta) = \|\Theta\|^2 \ .$$

When this is done in the example depicted above, the regression model becomes stable, producing the result shown in the right-hand panel in the figure. Now the slope is much more sensible.

2.6.2 Ridge regression

There are some kinds of trouble we can get into in regression problems. What if $(\tilde{X}^T\tilde{X})$ is not invertible?

Study Question: Consider, for example, a situation where the data-set is just the same point repeated twice: $x^{(1)} = x^{(2)} = [1 \ \ 2]^T$. What is $\tilde{X}$ in this case? What is $\tilde{X}^T\tilde{X}$? What is $(\tilde{X}^T\tilde{X})^{-1}$?

Another kind of problem is *overfitting*: we have formulated an objective that is just about fitting the data as well as possible, but we might also want to *regularize* to keep the hypothesis from getting *too* attached to the data.

We address both the problem of not being able to invert $(\tilde{X}^T\tilde{X})^{-1}$ and the problem of overfitting using a mechanism called *ridge regression*. We add a regularization term $\|\theta\|^2$ to the OLS objective, with a non-negative scalar value λ to control the tradeoff between the training error and the regularization term.

2.7.1 Evaluating hypotheses

The performance of a given hypothesis h may be evaluated by measuring *test error* on data that was not used to train it. Given a training set $\mathcal{D}_n$, a regression hypothesis h, and if we choose squared loss, we can define the OLS *training error* of h to be the mean square error between its predictions and the expected outputs:

$$\mathcal{E}_n(h) = \frac{1}{n}\sum_{i=1}^{n}\left[h(x^{(i)}) - y^{(i)}\right]^2 \quad .$$

It's a bit funny to interpret the analytical formulas given above for θ as "training," but later when we employ more statistical methods "training" will be a meaningful concept.

Test error captures the performance of h on unseen data, and is the mean square error on the test set, with a nearly identical expression as that above, differing only in the range of index i:

$$\mathcal{E}(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'}\left[h(x^{(i)}) - y^{(i)}\right]^2$$

on n' new examples that were not used in the process of constructing h.

In machine learning in general, not just regression, it is useful to distinguish two ways in which a hypothesis $h \in \mathcal{H}$ might contribute to test error. Two are:

Structural error: This is error that arises because there is no hypothesis $h \in \mathcal{H}$ that will perform well on the data, for example because the data was really generated by a sine wave but we are trying to fit it with a line.

Estimation error: This is error that arises because we do not have enough data (or the data are in some way unhelpful) to allow us to choose a good $h \in \mathcal{H}$, or because we didn't solve the optimization problem well enough to find the best h given the data that we had.

When we increase λ, we tend to increase structural error but decrease estimation error, and vice versa.

There are technical definitions of these concepts that are studied in more advanced treatments of machine learning. Structural error is referred to as *bias* and estimation error is referred to as *variance*.

2.7.2 Evaluating learning algorithms

Note that this section is relevant to learning algorithms generally—we are just introducing the topic here since we now have an algorithm that can be evaluated!

A *learning algorithm* is a procedure that takes a data set $\mathcal{D}_n$ as input and returns an hypothesis h from a hypothesis class $\mathcal{H}$; it looks like

$$\mathcal{D}_n \longrightarrow \boxed{\text{learning alg } (\mathcal{H})} \longrightarrow h$$

Keep in mind that h has parameters. The learning algorithm itself may have its own parameters, and such parameters are often called *hyperparameters*. The analytical solutions presented above for linear regression, e.g., Eq. 2.8, may be thought of as learning algorithms, where λ is a hyperparameter that governs how the learning algorithm works and can strongly affect its performance.

How should we evaluate the performance of a learning algorithm? This can be tricky. There are many potential sources of variability in the possible result of computing test error on a learned hypothesis h:

- Which particular *training examples* occurred in $\mathcal{D}_n$
- Which particular *testing examples* occurred in $\mathcal{D}_{n'}$
- Randomization inside the learning *algorithm* itself

2.7.2.1 Validation

Generally, to evaluate how well a learning *algorithm* works, given an unlimited data source, we would like to execute the following process multiple times:

- Train on a new training set (subset of our big data source)
- Evaluate resulting h on a *validation set* that does not overlap the training set (but is still a subset of our same big data source)

Running the algorithm multiple times controls for possible poor choices of training set or unfortunate randomization inside the algorithm itself.

2.7.2.2 Cross validation

One concern is that we might need a lot of data to do this, and in many applications data is expensive or difficult to acquire. We can re-use data with *cross validation* (but it's harder to do theoretical analysis).

CROSS-VALIDATE($\mathcal{D}, k$)

1 divide $\mathcal{D}$ into k chunks $\mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_k$ (of roughly equal size)
2 **for** $i = 1$ **to** k
3 train h_i on $\mathcal{D} \setminus \mathcal{D}_i$ (withholding chunk $\mathcal{D}_i$ as the validation set)
4 compute "test" error $\mathcal{E}_i(h_i)$ on withheld data $\mathcal{D}_i$
5 **return** $\frac{1}{k} \sum_{i=1}^{k} \mathcal{E}_i(h_i)$

It's very important to understand that (cross-)validation neither delivers nor evaluates a single particular hypothesis h. It evaluates the *learning algorithm* that produces hypotheses.

2.7.2.3 Hyperparameter tuning

The hyper-parameters of a learning algorithm affect how the algorithm *works* but they are not part of the resulting hypothesis. So, for example, λ in ridge regression affects *which* hypothesis will be returned, but λ itself doesn't show up in the hypothesis (the hypothesis is specified using parameters θ and θ_0).

You can think about each different setting of a hyper-parameter as specifying a different learning algorithm.

In order to pick a good value of the hyper-parameter, we often end up just trying a lot of values and seeing which one works best via validation or cross-validation.

Study Question: How could you use cross-validation to decide whether to use analytic ridge regression or our random-regression algorithm and to pick K for random regression or λ for ridge regression?