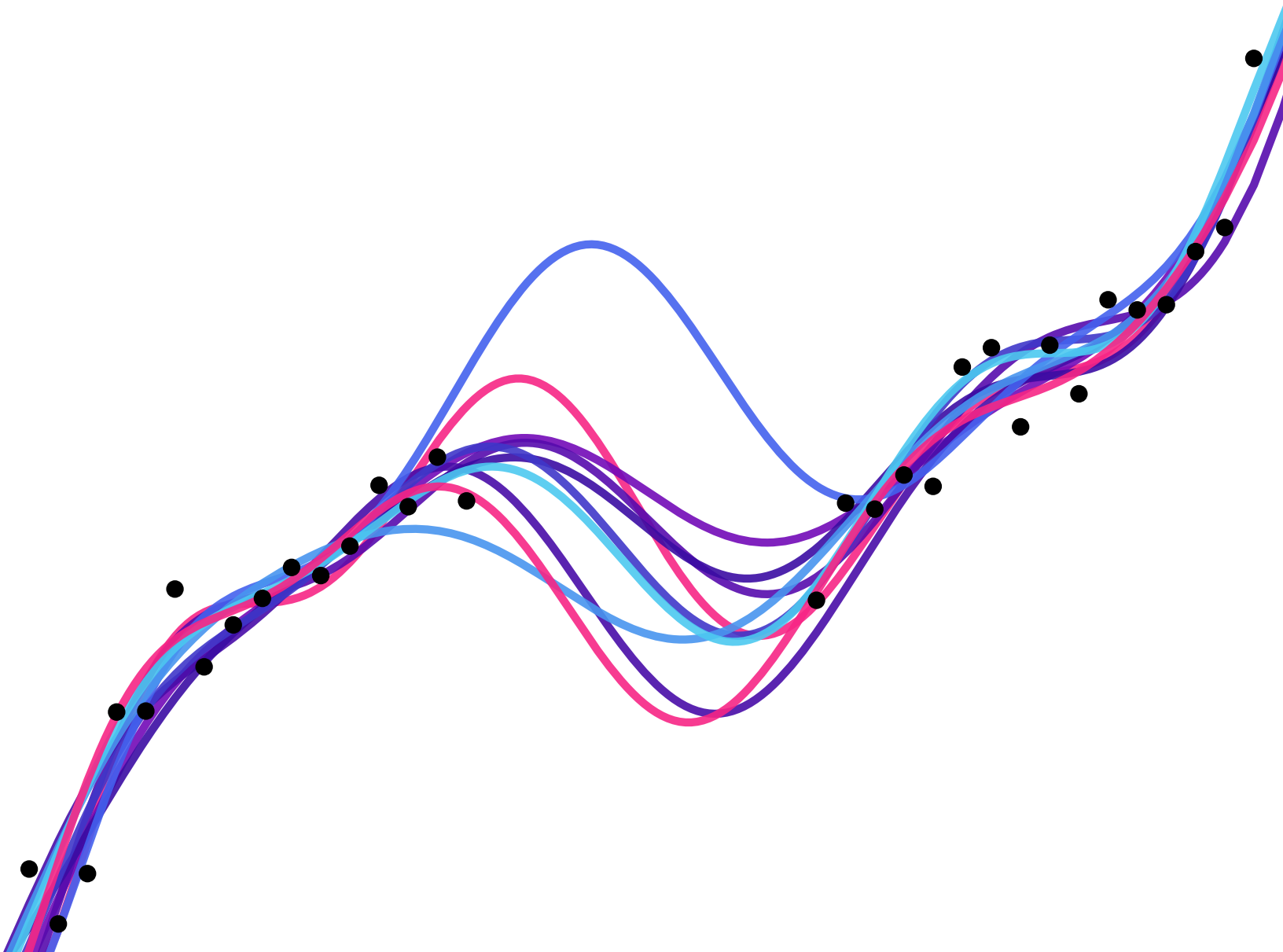
An Introduction to
Machine Learning

A textbook for Harvard's CS181



Thank you to William J. Deuschle, the original author of this textbook, and to course staffs and students for maintaining this work across the years.

Last updated February 2022.

"Nobody phrases it this way, but I think that artificial intelligence is almost a humanities discipline. It's really an attempt to understand human intelligence and human cognition."

- Sebastian Thrun

Contents

Notation

The machine learning community is notorious for using widly varying notations. The following standard is not meant to provide a standard lookup for all machine learning literature, but specifically for this textbook. The fastest way to become fluent in understanding notation is by reading machine learning papers.

Set theory

$\mathcal{A}$	a set
$\mathbb{N}$	the set of all natural numbers
$\mathbb{N}^*$	the set of all natural numbers excluding 0
$\mathbb{Z}$	the set of all integers
$\mathbb{R}$	the set of all real numbers
$\mathbb{R}^D$	the set of all real-valued D-dimensional vectors
$\mathbb{R}^{N \times D}$	the set of all real-valued $N \times D$ matrices

Numerical objects

$x \in \mathbb{R}$	a scalar
$\boldsymbol{x} \in \mathbb{R}^D$	a vector
$\boldsymbol{X} \in \mathbb{R}^{N \times D}$	a matrix
$\boldsymbol{I}_N \in \mathbb{R}^{N \times N}$	the identify matrix

Functions and operators

$a := b$ or $b =: a$	a is defined to be b
$f : \mathcal{A} \to \mathcal{B}$	a function f that maps from domain $\mathcal{A}$ to codomain $\mathcal{B}$
$f \circ g$	the composite function $f(g(\cdot))$
$f(x;\, \theta)$	a function of x parametrized by θ
$\log(\cdot)$	the natural logarithm
$\log_b(\cdot)$	the base b logarithm
$\|\|\mathbf{x}\|\|_p$	the L^p norm of $\mathbf{x}$
$\|\|\mathbf{x}\|\|$	the L^2 norm of $\mathbf{x}$
$\mathbf{1}(\cdot)$	the indicator function, evaluates to 1 if argument is true else 0

Probability theory

x	$\mathbb{R}$	a scalar-valued r.v.
$\mathbf{x}$	$\mathbb{R}^D$	a vector-valued r.v.
$\mathbf{X}$	$\mathbb{R}^{N\times D}$	a matrix-valued r.v.
$\mathrm{x}\sim\mathrm{P}$		r.v. X is distributed as some distribution P
$\mathrm{x}\sim p(x)$		r.v. X is distributed as some distribution whose PDF is $p(x)$
$P(\mathrm{x}=x)$		the probability of the event X takes on value x
$\mathbb{E}[X]$		expectation of r.v. X
$\mathrm{Var}(X)$		variance of r.v X
$\mathrm{Cov}(X,Y)$		covariance of r.v.s X and Y
$\mathcal{N}(\mu,\sigma)$		the Guassian distribution with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$
$\mathcal{N}(x;\,\mu,\sigma)$		the PDF of the distribution $\mathcal{N}(\mu,\Sigma)$

Indexing

x_i, $[\mathbf{x}]_i$	$\mathbb{R}$	the i-th element of vector $\mathbf{x}$
$x_{i,j}$, $[\mathbf{X}]_{i,j}$	$\mathbb{R}$	the element of matrix $\mathbf{X}$ at row i and column j. Commas in the subscript can be ommitted if not ambiguous.
$\mathbf{X}_{:,j}$	$\mathbb{R}^N$	the j-th column of matrix $\mathbf{X}\in\mathbb{R}^{N\times D}$
$\mathbf{X}_{i,:}$	$\mathbb{R}^D$	the i-th row of matrix $\mathbf{X}\in\mathbb{R}^{N\times D}$

Object generation

$[a:b]$		the set of integers $\mathbb{Z}\cup[a,b]$ between a and b (inclusive)
$[a]$		the set of integers $[1:a]$
$\{x_i\}_{n=1}^N$		the set of N objects $x_1,\ldots,x_N$.
$[\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n]$	$\mathbb{R}^{d\times n}$	the matrix $\boldsymbol{X}$ such that $\boldsymbol{X}_{:,i}=x_i$ where $x_i\in\mathbb{R}^d$

Machine learning

N	$\mathbb{R}$	number of datapoints in our training dataset
D	$\mathbb{R}$	number of features in each datapoint; dimensionality of our input space
x	$\mathcal{X}$	An example. Here, $\mathcal{X}$ is our input space. Usually, $\mathcal{X}=\mathbb{R}^D$.
x^*	$\mathcal{X}$	A test example. This is refers to an example outside our training dataset whose label we do not know and are trying to predict.
y	$\mathcal{Y}$	A label. Here, $\mathcal{Y}$ is our output space. In linear regression, $\mathcal{Y}=\mathbb{R}$.
$\hat{y}^*$	$\mathcal{Y}$	A predicted label. This is the label our model predicts for some test example x^*.
w or θ	$\mathcal{W}$	The model weights
$\mathcal{L}:\mathcal{W}\to\mathbb{R}$		a loss or objective function
$f:\mathcal{X}\times\mathcal{W}\to\mathcal{Y}$		a model parametrized by possible weights $\mathcal{W}$.

Chapter 1

Introduction

Machine learning refers to the study of algorithms which improve automatically through experience or the use of data. These algorithms often build some model, or mathematical understanding of the world, based on data which we, the humans, provide. After learning from this data, a model can help us make predictions, decision, or conclusions that we never explicitly programmed or even knew beforehand. A bank, for example, may use some machine learning algorithm to inform their decisions on who they should give out loans to. With this algorithm, the bank hopes to achieve its original goal of making fewer risky investments.

Indeed, machine learning is indeed very useful in aiding human decision making. However, it is irresponsible of us to surrender all our decision-power onto machines. If the bank's machine learning algorithm learned from historical data, it may learn to distrust certain demographics who have been socio-economically disadvantaged in the past, thus perpetuating historical prejudices by refusing to recommend loans to the people of these communities. Machine learning has made it all too easy to simply delegate our actions onto whatever an algorithm tells us will "maximize profits" or "minimize cost". Too often are these algorithms are built thoughtlessly and used within some decision-system wrecklessly. The purpose of this book is to provide a framework with which to approach machine learning problems and understand how these tools we create ought to be situated in the real world.

1.1 Data representation

The idea behind any machine learning algorithm is to reveal patterns and extrapolate information from data. Consider the example data in Figure 1.1a. Each row in the table represents one *datapoint*, and each datapoint has the same number of values or *features* in them. Since some of these features are non-numerical, like "name", a data scientist might choose a numerical representation. With the aid of a *domain expert*, an expert in the field that this data comes from, a numerical representation is chosen which might look like Figure 1.1b. Some features, like the individual's name, may be left out not only for privacy but because that information probably does not lend insight on the task at hand. Mathematically, the table in Figure 1.1b is represented by the dataset $\{\boldsymbol{x}_i\}_{i=1}^{6}$ where $\boldsymbol{x}_i \in \mathbb{R}^3$. This would mean, for example, that $\boldsymbol{x}_2 = [34, 66.22, 177.80]^\top$. In machine learning convention, we denote the number of examples in a dataset with N and the number of features in each example with D.

Name	Age	Weight	Height
Juan	18	162lb	6ft 1in
Joe	34	146lb	5ft 10in
Mei	39	193lb	5ft 8in
Bella	21	124lb	5ft 3in
Abdul	57	141lb	5ft 7in
Hanna	40	134lb	5ft 4in

(a) Data in raw representation.

Age	Weight (kg)	Height (cm)
18	73.48	185.42
34	66.22	177.80
39	87.54	172.72
21	56.25	160.02
57	63.96	170.18
40	60.78	162.56

(b) Data in numerical representation.

Figure 1.1: Data representations; raw data must be given a chosen numerical representations in order to be operated on by an algorithm.

Definition 1.1.1 (datapoint): A datapoint or example $x \in \mathcal{X}$ is a numerical representation of an element from a real-world class of objects. If our data is tabular, the input space $\mathcal{X}$ is a subset of $\mathbb{R}^D$ where each dimension correspnds to a feature or attribute of x.

With our data represented numerically, we can use mathematics to extract information from the data. For example, we can choose to interpret some distance metric between any two datapoints as the similarity between those examples.

A common machine learning problem is trying to predict an unknown quality of interest. For example, given a person's age, weight, and height in Figure 1.1, can we predict their blood pressure? This additional, unknown feature is referred to as the *label*.

Definition 1.1.2 (label): The label $y \in \mathcal{Y}$ is an unknown feature of a datapoint x. We often want the power to predict the label y of any arbitrary x.

In the case of predicting blood pressure, our label is continuous and the output space $\mathcal{Y}$ takes on $\mathbb{R}$. However, labels can be discrete. We could instead by trying to classify people into either being at risk or not at risk for a disease. In this example, the output space is discrete and takes on $\{$"at risk", "not at risk"$\}$.

It is worth noting that not all data is tabular. Commonly natural language processing (NLP), we wish to process audio signals or written text which can be of varying, sequential length. In computer vision, our datapoints may be images which encode spatial information which would be lost if we collapsed everything into vectors. In such cases, we represent each datapoint as a matrix or tensor.

1.2 Models

Similarly to real-world data, we can also translate our understanding of real-world dynamics and relationships into mathematics. For example, imagine that we are placing text onto a background color. If that background color is dark, we intuitively know we should use white text. If the background color is bright, naturally we know we should use black text. Notice that with clever thinking, we can translate this understanding into math.

First we represent background colors as datapoints in $[0,1] \times [0,1] \times [0,1] \subset \mathbb{R}^3$ (red, green, and blue as values between 0 and 1). One standard way of calculating the brightness, or luma, of a color $\boldsymbol{x}$ is to take a convex combination of the red, green, and blue channels:

$$\text{luma}(\boldsymbol{x}) = 0.2x_1 + 0.7x_2 + 0.1x_3.$$

We can arbitrarily make up a model that when given a background color $\boldsymbol{x}^*$ predicts a text font $\hat{y}^* \in \{\text{“white”, “black”}\}$. Logically, one could construct such a model to be a function

$$f(\mathbf{x}) = \begin{cases} \text{black} & \text{if } \text{luma}(\mathbf{x}) > 0.5, \\ \text{white} & \text{otherwise.} \end{cases}$$

What makes things more interesting is that people have differing standards on how to calcualte the luma of a color. Some standards, for example, give more weight to the blue channel:

$$\text{luma}(\mathbf{x}) = 0.1x_1 + 0.5x_2 + 0.4x_3.$$

In fact, there is an entire family of functions that calculate luma indexed by these three coefficients. You can imagine these three coefficients (which we will vectorize as $\boldsymbol{w} \in \mathbb{R}^3$) as dials that we can toggle up and down freely to get models which behave differently. Consider the model parametrized by the settings, or *weights*, $\boldsymbol{w} = [1,0,0]^\top$; interpretting this model, we see that it equates the amount of red in the background color as the brighness, completely disregarding the blue and green channels.

Definition 1.2.1 (model): A model is a mathematical articulation of some real-world dynamic which we hope to make inference on or whatever this is hard.

A natural question arises: how do we compare the different models? That is to say, whose function will give us the best predictions? One could say a good model should make accurate predictions or conclusions, but what does this even mean? As we have done before, we will translate the idea of measuring “goodness” into maths. One way we can achieve this is to define a *loss function*.

Definition 1.2.2 (loss function): A loss function $\mathcal{L} : \mathcal{W} \to \mathbb{R}$ is a metric quantifying the badness of set of model parameters and satisfies the condition $\mathcal{L}(w) > 0 \forall w$. If $\mathcal{L}(\boldsymbol{w}_1) > \mathcal{L}(\boldsymbol{w}_2)$, then the model parametrized by weights $\boldsymbol{w}_1$ achieves less loss than that of $\boldsymbol{w}_2$.

With the colors example, one way to measure loss could be to simply count how many times our model makes an incorrect prediction. Let’s say we have a set of background colors paired with labels which a perfect model should predict: $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)\}_{n=1}^N$. We can define the loss, then, to be

$$\mathcal{L}(\boldsymbol{w}) = \sum_{n=1}^{N} \mathbf{1}(f(\boldsymbol{x}_n) \text{ is } y_n).$$

Now we can directly compare two models, but this is still not enough to find a good model from scratch. The size of the model family (that is, the number of all possible combinations of weights), is infinite in this case and therefore impossible to individually comb through. The way we search for a possible model is called *learning*.

Note: The loss function is implicitly a function of the labelled dataset $\mathcal{D}$. This means that for the same weights $\boldsymbol{w}$, this loss metric can produce different values depending the training dataset we use. This delema in selecting models reappears in our discussion of the bias-variance tradeoff.

1.3 The Cube

We will use the **Machine Learning Cube** in order to describe the domain of the problems we encounter. The hope is that the cube will be a useful way to delineate the techniques we will apply to different types of problems. Understanding the different facets of the cube will aid you in understanding machine learning as a whole, and can even give you intuition about techniques that you have never encountered before. Let's now describe the features of the cube.

1.3.1 Discrete vs continuous

Our cube has three axes. On the first axis we will put the output domain. The domain of our output can take on one of two forms: **discrete** or **continuous**. Discrete, or categorical data, is data that can only fall into one of a finite number of classes. For example, if we sample random flowers of a field of K many species, each flower will fall cleanly into one of the K species. Continuous data is that which falls on the real number line. For example, if we sample people from the street, their blood pressure will be a continuous, positive number.

1.3.2 Probabilistic vs non-probabilistic

The second axis of the cube is reserved for the statistical nature of the machine learning technique in question. Specifically, it will fall into one of two broad categories: **probabilistic** or **non-probabilistic** techniques. Probabilistic techniques are those for which we incorporate our data using some form of statistical distribution or summary. In general, we are then able to discard some or all of our data once we have finished tuning our probabilistic model.

In contrast, non-probabilistic techniques are those that use the data directly to perform some action. A very common and general example of this is comparing how close a new data point is to other points in your existing data set. Non-probabilistic techniques potentially make fewer assumptions, but they do require that you keep around some or all of your data. They are also potentially slower techniques at runtime because they may require touching all of the data in your dataset to perform some action. These are a very broad set of guidelines for the distinction between probabilistic and non-probabilistic techniques - you should expect to see some exceptions and even some techniques that fit into both of these categories to some degree. Having a sense for their general benefits and drawbacks is useful, and you will gain more intuition about the distinction as we begin to explore different techniques.

1.3.3 Supervised vs unsupervised

The third and final axis of the cube describes the type of training we will use. There are two major classes of machine learning techniques: **supervised** and **unsupervised**. In fact, these two classes of techniques are so important to describing the field of machine learning that we will roughly divide this textbook into two halves dedicated to techniques found within each of these categories. A supervised technique is one for which we get to observe a data set of both the inputs and the outputs ahead of time, to be used for training. For example, we might be given a data set about weather

conditions and crop production over the years. Then, we could train a machine learning model that learns a relationship between the input data (weather) and output data (crop production). The implication here is that given new input data, we will be able to predict the unseen output data. An unsupervised technique is one for which we only get a data set of 'inputs' ahead of time. In fact, we don't even need to consider these as inputs anymore: we can just consider them to be a set of data points that we wish to summarize or describe. Unsupervised techniques revolve around clustering or otherwise describing our data.

We will see examples of all of these as we progress throughout the book, and you will gain an intuition for where different types of data and techniques fall in our cube. Eventually, given just the information in the cube for a new technique, you will have a solid idea of how that technique operates.

Chapter 2

Regression

As hinted to before, a major problem that machine learning hopes to solve is being able to predict some target value y given some explanatory inputs x. When this prediction is a continuous, real number, we call this prediction procedure *regression*.

We can think of a few real-world situations where it would be helpful to be able to predict continuous targets:

1. Predicting a person's height given the height of their parents.
2. Predicting the amount of time someone will take to pay back a loan given their credit history.
3. Predicting what time a package will arrive given current weather and traffic conditions.

In each of these examples, we are trying to estimate a single value after being given one or more input values. The assumption is that these inputs hopefully encode enough information about the desired target value. It seems reasonable to predict a person's height given their parents' heights. However, it seems foolish to try to predict stock prices given meteorlogical activity on Mars. An attempt to do either of these is indeed still regression. The term "regression" is general and encompasses any method that will predict a value by trying to model its relationship to some inputs.

Definition 2.0.1 (regression): A class of method that make predictions about unknown target y given observed inputs x.

2.1 Linear Regression

The simplest form of regression is *linear regression*, which predicts the target value by taking a linear combination of the input features.

Definition 2.1.1 (linear regression): Linear regression is the process of predicting the target value of some input features $\boldsymbol{x} \in \mathbb{R}^d$ through the linear function

$$f(\boldsymbol{x}; \boldsymbol{w}) := w_0 + w_1 x_1 + \ldots + w_d x_d. \tag{2.1}$$

A linear regression model $f(\boldsymbol{x}; \boldsymbol{w})$ is fully specified by its weights $\boldsymbol{w}$. Therefore, we refer to a model as simply the set of weights $\boldsymbol{w}$ which parametrize it.

Technically, Equation 2.1 is affine, not linear, because it includes a bias term. For convinience, we often use the "bias trick" to express the model as a linear function:

$$f(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}. \tag{2.2}$$

This is possible because we impose $x_0 = 1$ for all $\boldsymbol{x} \in \mathbb{R}^d$. More specifically, we apply the following transformation before feeding our inputs into our model:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \mapsto \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}.$$

With this trick, we recover our original definition Equation 2.1:

$$\begin{aligned} f(\boldsymbol{x}; \boldsymbol{w}) &= \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x} \\ &= w_0 x_0 + w_1 x_1 + \ldots + w_d x_d \\ &= w_0 \cdot 1 + w_1 x_1 + \ldots + w_d x_d \\ &= w_0 + w_1 x_1 + \ldots + w_d x_d. \end{aligned}$$

Note: Even though this adds a dimension to the input space, we still say that $\boldsymbol{x} \in \mathbb{R}^d$ for convinience-sake. This means we also say $\boldsymbol{w} \in \mathbb{R}^d$.

Example 2.1.1 (predicting height):
Assume we are working in a healthcare setting and the doctor gives us a linear regression model

$$\boldsymbol{w} = \begin{bmatrix} 34 \\ 0.39 \\ 0.33 \end{bmatrix}$$

predict any child's future height. The model assumes we represent any child $\boldsymbol{x}$ by two features: x_1 describing the mother's height (in cm) and x_2 describing the father's height (in cm). If the child's mother is 165cm tall and father 185cm tall, then

$$\boldsymbol{x}^* = \begin{bmatrix} 165 \\ 185 \end{bmatrix}$$

represents the child which we hope to make a prediction for. Using our model, we predict this child's future height is

$$\hat{y}^* := f(\boldsymbol{x}^*; \boldsymbol{w}) = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}^* = 34 + 0.39(165) + 0.33(185) = 159.4.$$

Notice the use of $\hat{y}^*$ to mean our model prediction.

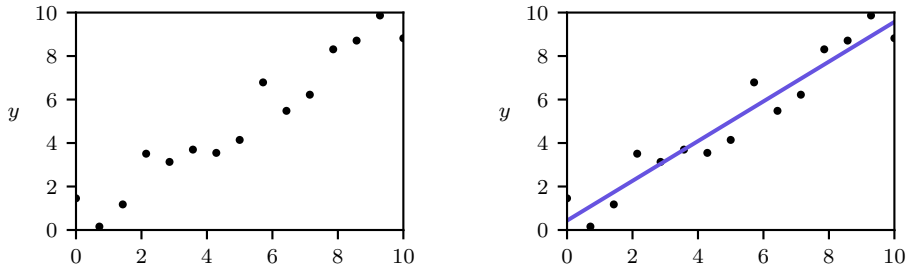


Figure 2.1: A dataset with a clear linear trend (left). A line which seems to fit the apparent linear relationship (right).

2.1.1 Visualizing a linear regression model

Let's try to build some intuition about how linear regression works. Our algorithm is given a labelled training dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ is the i-th input and $y_i \in \mathbb{R}$ is that input's corresponding target. Our goal is to find weights $\boldsymbol{w}$ such that given a new data point $\boldsymbol{x}^* \notin \mathcal{D}$, we can predict its target value y^*. Admittedly, it is not likely that we can exactly predict the true target y^* for every $\boldsymbol{x} \in \mathcal{X}$. (In fact, just the idea that there even exists a true y^* out there is imposed by us). Hence, we instead try to estimate y^* with a different but hopefully "close" value $\hat{y}^*$ (we can define what close means in many ways). This is visualizable in the simple case where we only have 1-dimensional inputs: $\mathcal{X} \subset \mathbb{R}$.

Our eyes are naturally able to detect the very clear trend in this data. If we were given a datapoint $\boldsymbol{x}^*$, not necessarily in our training dataset, how would we predict its target value y^*? We would first fit a line to our data, as in Figure 2.1 (right), and then we select $\hat{y}^*$ according to the point $(\boldsymbol{x}^*, \hat{y}^*)$ on the line.

That is the entirety of linear regression. It fits a line to our training data, and then uses that line to make predictions. In 1-dimensional input space, this manifests itself as the simple problem seen above,where we need only find a single bias term w_0 (which acts as the intercept of the line) and single weight w_1 (which acts as the slope of the line). However, the same principle applies to higher dimensional data as well. We're always fitting the hyperplane that best predicts the data.

> *Note:* Although our input data points $\boldsymbol{x}$ can take on multiple dimensions, our output data y is always a 1-dimensional real number when dealing with regression problems.

2.1.2 Least squares loss

Now that we've defined our model as a weighted combination of our input variables, we need some way to choose our value of $\boldsymbol{w}$. To do this, we need an objective or loss function.

Recall that the objective function measures the "goodness" of a model. We can optimize this function to identify the best possible model for our data. Note that in the case of linear regression, our model is entirely specified by our settings of parameters $\boldsymbol{w}$.

An objective function will sometimes be referred to as *loss*. Loss actually measures how bad a model is, and then our goal is to minimize it. It is common to think in terms of loss when discussing linear regression, and we incur loss when the hyperplane we fit is far away from our data.

So how do we compute the loss for a specific setting of $\boldsymbol{w}$? To do this, we often use *residuals*.

Definition 2.1.2 (residual): The residual is the difference between the target y and prediction $\hat{y} = f(\boldsymbol{x}; \boldsymbol{w})$ value that a model produces:

$$y - f(\boldsymbol{x}; \boldsymbol{w}) = y - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}.$$

Commonly, loss is a function of the residuals produced by a model. For example, you can imagine taking the absolute value of all of the residuals and adding those up to produce a measurement of loss. This is referred to as *L1 loss*. Or, you might square all of the residuals and then add those up to produce loss, which is called *L2 loss* or *least squares loss*. You might also use some combination of L1 and L2 loss. For the most part, these are the two most common forms of loss you will see when discussing linear regression.

Definition 2.1.3 (least squares loss): The least squares loss of a model $\boldsymbol{w}$ over a labelled dataset $\mathcal{D}$ is the sum of the squared residuals or "losses":

$$\mathcal{L}(\boldsymbol{w}; \mathcal{D}) := \frac{1}{2} \sum_{(\boldsymbol{x}, y) \in \mathcal{D}} (y - f(\boldsymbol{x}; \boldsymbol{w}))^2.$$

Note: Notice that our summation happens over the dataset: this means that for same model $\boldsymbol{w}$ can have very different losses depending on the training dataset collected. This is why we hope our training dataset is an accurate representation of the entire global dataset of possible inputs yet seen. We often will leave this out as implicit, simply using $\mathcal{L}(\boldsymbol{w})$.

When minimized, these distinct measurements of loss will produce solutions for $\boldsymbol{w}$ that have different properties. For example, L2 loss is not robust to outliers due to the fact that we are squaring residuals. Furthermore, L2 loss will produce only a single solution while L1 loss can potentially have many equivalent solutions. Finally, L1 loss produces unstable solutions, meaning that for small changes in our dataset, we may see large changes in our solution $\boldsymbol{w}$.

Loss is a concept that we will come back to very frequently in the context of supervised machine learning methods. Before exploring exactly how we use loss to fit a line, let's consider least squares loss in greater depth.

There is a satisfying statistical interpretation for using this loss function which we will explain later in this chapter, but for now it will suffice to discuss some of the properties of this loss function that make it desirable.

First, notice that it will always take on positive values. This is convenient because we can focus exclusively on minimizing our loss, and it also allows us to combine the loss incurred from different data points without worrying about them cancelling out.

A more subtle but enormously important property of this loss function is that it is strongly convex. This means we also know that there exists a unique global minimum where the derivative of the function is equal to 0, as seen in Figure 2.2. This is in part credit to the fact that strongly convex functions are continuously differentiable. In contrast, L1 loss is not continuously differentiable over the entirety of its domain.

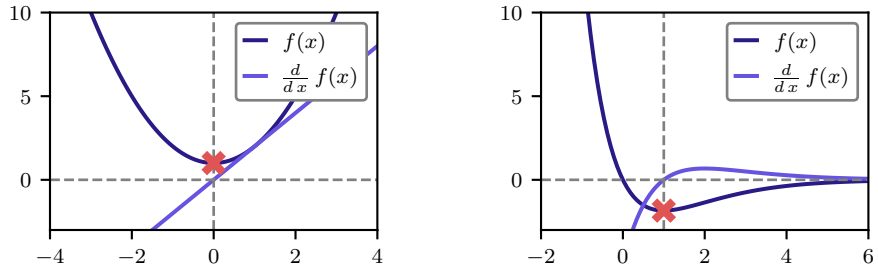


Figure 2.2: Strongly convex functions are minimized where their derivative 0.

2.1.3 Solving for Optimal Weights Analytically

Now that we have our least squares loss function, we can finally begin to fit a line to our data. Our goal is to find the optimal set of weights $\boldsymbol{w}^*$ with respect to a particular labelled training dataset $\mathcal{D}$:

$$\boldsymbol{w}^* := \operatorname*{argmin}_{w} \mathcal{L}(\boldsymbol{w}; \mathcal{D}).$$

For convinience, we can concatenate our dataset $\mathcal{D} := \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ as $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{y})$ where $\boldsymbol{X} := [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]^{\mathrm{T}} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{y} := [y_1, \ldots, y_n]^{\mathrm{T}}$. The matrix $\boldsymbol{X}$ is commonly referred to as the *design matrix*.

Derivation 2.1.1 (least squares optimal weights):
We find the optimal weights $\boldsymbol{w}^*$ as follows:
Start by taking the gradient of the loss with respect to our parameter $\boldsymbol{w}$:

$$\nabla \mathcal{L}(\boldsymbol{w}; \mathcal{D}) = \sum_{n=1}^{N} (y_n - \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n)(-\boldsymbol{x}_n).$$

Setting this gradient to 0 and multiplying both sides by -1, we get

$$\begin{aligned} 0 &= \sum_{n=1}^{N} y_n \boldsymbol{x}_n - \sum_{n=1}^{N} (\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n) \boldsymbol{x}_n \\ &= \sum_{n=1}^{N} y_n \boldsymbol{x}_n - \sum_{n=1}^{N} \boldsymbol{x}_n (\boldsymbol{x}_n^{\mathrm{T}} \boldsymbol{w}). \end{aligned}$$

We get this last line by noticing that $(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n)\boldsymbol{x}_n = (\boldsymbol{x}_n^{\mathrm{T}} \boldsymbol{w})\boldsymbol{x}_n = \boldsymbol{x}_n (\boldsymbol{x}_n^{\mathrm{T}} \boldsymbol{w})$. (This is because $a^{\mathrm{T}} = a$ and $a\boldsymbol{v} = \boldsymbol{v}a$ for any scalar a and vector $\boldsymbol{v}$).
At this point, it is convenient to rewrite these summations as matrix operations. Using the design matrix $\boldsymbol{X}$ and target values $\boldsymbol{y}$, we have

$$\boldsymbol{X}^{\mathrm{T}} \boldsymbol{y} = \sum_{n=1}^{N} y_n \boldsymbol{x}_n, \quad \boldsymbol{X}^{\mathrm{T}} \boldsymbol{X} \boldsymbol{w} = \sum_{n=1}^{N} \boldsymbol{x}_n (\boldsymbol{x}_n^{\mathrm{T}} \boldsymbol{w}).$$

After substituting,

$$0 = \boldsymbol{X}^{\mathrm{T}} \boldsymbol{y} - \boldsymbol{X}^{\mathrm{T}} \boldsymbol{X} \boldsymbol{w},$$

we can solve for $\boldsymbol{w}^*$:

$$\boldsymbol{w}^* = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}. \tag{2.3}$$

For this to be well defined we need $\boldsymbol{X}$ to have full column rank (features are not colinear) so that $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ is positive definite and the inverse exists.

The quantity $(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}$ in Derivation 2.1.1 has a special name: the *Moore-Penrose pseudoinverse*. You can think of it as the generalization of matrix inversion for non-square matrices.

2.1.4 Solving for Optimal Weights Geometrically

Another common interpretation of linear regression is that of a projection of our targets, $\boldsymbol{y}$, onto the column space of our inputs $\boldsymbol{X}$. This can be useful for building intuition.

We showed above that the quantity $(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}$ can be thought of as the pseudoinverse for our inputs $\boldsymbol{X}$. Let's now consider the case where $\boldsymbol{X}$ is square and the pseudoinverse is equal to the true inverse: $\boldsymbol{X}^{-1} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}$. With this assumption,

$$\begin{aligned}\boldsymbol{w}^* &= (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y} \\ &= \boldsymbol{X}^{-1}\boldsymbol{y}.\end{aligned}$$

We can recover our target values $\boldsymbol{y}$ by multiplying either side by $\boldsymbol{X}$:

$$\begin{aligned}\boldsymbol{X}\boldsymbol{w}^* &= \boldsymbol{X}\boldsymbol{X}^{-1}\boldsymbol{y} \\ \boldsymbol{X}\boldsymbol{w}^* &= \boldsymbol{y}\end{aligned}$$

We were able to recover our targets $\boldsymbol{y}$ exactly because $\boldsymbol{X}$ is an invertible tranformation. However, in the general case where $\boldsymbol{X}$ is not invertible and we have to use the approximate pseudoinverse $(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}$, we instead recover $\hat{\boldsymbol{y}}$:

$$\begin{aligned}\boldsymbol{X}\boldsymbol{w}^* &= \boldsymbol{X}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y} \\ \boldsymbol{X}\boldsymbol{w}^* &= \hat{\boldsymbol{y}}\end{aligned}$$

where $\hat{\boldsymbol{y}}$ can be thought of as the closest projection of $\boldsymbol{y}$ onto the column space of $\boldsymbol{X}$.

This motivates the intuition that $\boldsymbol{w}^*$ is the set of coefficients that best transforms our input space $\boldsymbol{X}$ into our target values $\boldsymbol{y}$.

2.2 Probabalistic Linear Regression

We've thus far been discussing linear regression exclusively in terms of a loss function that helps us fit a set of weights to our data. In particular, we have been working with least squares, which has nice properties that make it a reasonable loss function.

In a very satisfying fashion, least squares also has a statistical foundation. In fact, you can recover the least squares loss function purely from a statistical derivation that we present here.

Consider our dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, where $\boldsymbol{x}_n \in \mathbb{R}^d$ and $y \in \mathbb{R}$. Let's imagine that every label y_i was generated from a probability distribution determined by $\boldsymbol{x}_i$ in some way. Namely, imagine y_i is a realization of the random variable

$$\mathrm{y}_i \sim \mathcal{N}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n, \sigma^2).$$

The PDF of the r.v. y_i is hence given by:

$$P(\mathrm{y}_i = y_i) = p(y_i; \boldsymbol{x}_i, \boldsymbol{w}) = \mathcal{N}(y_i; \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_i, \sigma^2) \tag{2.4}$$

The interpretation of the story we are imposing here is that each datapoint's label is drawn randomly. This story makes sense: if a doctor measures a patient's blood pressure, there is randomness associated with the imperfections of the instruments with which the measurements were taken. By assuming each label y_i is distributed as Gaussian with mean $\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_i$, we are assuming that it is unlikely to see y_i far from $\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_i$.

Also notice that σ^2 is fixed. Namely, $p(y_i; \boldsymbol{x}_i, \boldsymbol{w})$ is not parametrized by σ^2. It absolutely can be. In fact, we can assume a different variance for every single training datapoint. But, assuming they all have the same variance is a common and safe assumption (and we will see why).

Note: Notice the notational difference between the r.v. y (normal case) and the realization y (italics). Also notice the difference between $\mathcal{N}(\mu, \sigma^2)$, the Gaussian distribution with mean μ and variance σ^2, and its the PDF

$$\mathcal{N}(x; \mu, \sigma^2) := \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x-\mu)^2}{\sigma^2}\right).$$

2.2.1 Maximum likelihood estimation

As before, how do we solve for the optimal weights $\boldsymbol{w}$? One approach we can take is to choose the weights $\boldsymbol{w}$ which maximize the likelihood of observing our labels $\boldsymbol{y}$. This technique is known as *maximum likelihood estimation*.

The likelihood of some weights $\boldsymbol{w}$ is the probability of observing our $\mathcal{D} := (\boldsymbol{X}, \boldsymbol{y})$ given those weights:

$$\ell(\boldsymbol{w}; \mathcal{D}) := \prod_{i=1}^{n} p(y_i; \boldsymbol{x}_i, \boldsymbol{w}).$$

Since we assume the training datapoints are independent, we simply multiplied the probabilities $P(\mathrm{y}_i = y_i)$ together. Our goal is to find the optimal model $\boldsymbol{w}^*$ which maximizes the likelihood of our training data:

$$\boldsymbol{w}^* := \operatorname*{argmax}_{\boldsymbol{w}} \ell(\boldsymbol{w}; \mathcal{D}).$$

Derivation 2.2.1 (MLE optimal weights):
The likelihood of our model $\boldsymbol{w}$ given a dataset $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{y})$ and a fixed variance σ^2 is given by

$$p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n, \beta^{-1})$$

We then take the logarithm of the likelihood, and since the logarithm is a strictly increasing, continuous function, this will not change our optimal weights $\boldsymbol{w}$:

$$\ln p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \beta) = \sum_{n=1}^{N} \ln \mathcal{N}(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n, \beta^{-1})$$

Using the density function of a univariate Gaussian:

$$\begin{aligned}\ln p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}, \beta) &= \sum_{n=1}^{N} \ln \frac{1}{\sqrt{2\pi\beta^{-1}}} e^{-(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)^2/2\beta^{-1}} \\ &= \frac{N}{2}\ln\beta - \frac{N}{2}\ln(2\pi) - \frac{\beta}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)^2\end{aligned}$$

Notice that this is a quadratic function in $\boldsymbol{w}$, which means that we can solve for it by taking the derivative with respect to $\boldsymbol{w}$, setting that expression to 0, and solving for $\boldsymbol{w}$:

$$\begin{aligned}\frac{\partial \ln p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \beta)}{\partial \boldsymbol{w}} &= -\beta \sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)(-\boldsymbol{x}_n) \\ \Leftrightarrow \quad \sum_{n=1}^{N} y_n \boldsymbol{x}_n - \sum_{n=1}^{N}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)\boldsymbol{x}_n &= 0.\end{aligned}$$

Notice that this is exactly the same form as Equation **??**. Solving for $\boldsymbol{w}$ as before, we have:

$$\boldsymbol{w}^* = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y} \tag{2.5}$$

Notice that our final solution is exactly the same form as the solution in Equation 2.3, which we solved for by minimizing the least squares loss! The takeaway here is that minimizing a least squares loss function is equivalent to maximizing the probability under the assumption of a linear model with Gaussian noise.

2.3 Kernel Regression

Occasionally, linear regression will fail to recover a good solution for a dataset. While this may be because our data doesn't actually have predictive power, it might also just indicate that our data is provided in a format unsuitable for linear regression. This section explores that problem, in particular focusing on how we can manipulate the flexibility of our models to make them perform better.

2.3.1 Basis Functions

There are some situations where linear regression is not the best choice of model for our input data $\boldsymbol{X}$. Because linear regression only scales and combines input variables, it is unable to apply more complex transformations to our data such as a *sin* or square root function. In those situations where we need to transform our input variable somehow prior to performing linear regression (which is known as moving our data into a new *basis*), we can apply a **basis function**.

Definition 2.3.1 (basis function): Typically denoted by the symbol $\phi(\cdot)$, a basis function is a transformation applied to an input data point $\boldsymbol{x}$ to move our data into a different *input basis*, which is another phrase for *input domain*.

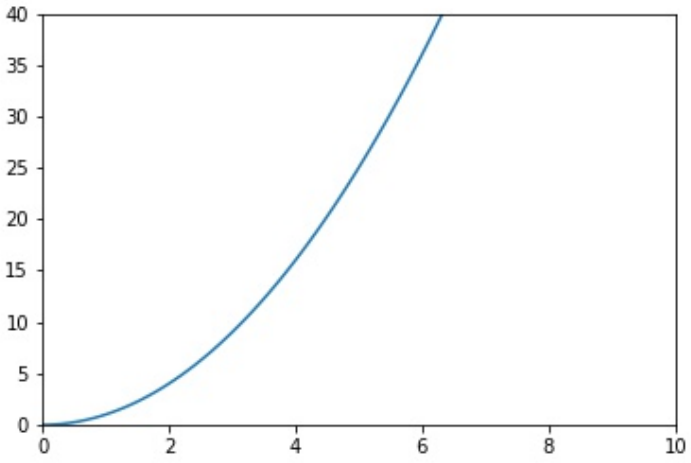


Figure 2.3: Data with no basis function applied.

For example, consider our original data point:

$$\boldsymbol{x} = (x^{(1)}, x^{(2)})'$$

We may choose our basis function $\phi(\boldsymbol{x})$ such that our transformed data point in its new basis is:

$$\phi(\boldsymbol{x}) = (x^{(1)}, {x^{(1)}}^2, x^{(2)}, \sin(x^{(2)}))'$$

Using a basis function is so common that we will sometimes describe our input data points as $\boldsymbol{\phi} = (\phi^{(1)}, \phi^{(2)}, ..., \phi^{(D)})'$.

Note: The notation $\boldsymbol{x} = (x^{(1)}, x^{(2)})'$ is a way to describe the dimensions of a single data point $\boldsymbol{x}$. The term $\boldsymbol{x}^{(1)}$ is the first dimension of a data point $\boldsymbol{x}$, while $\boldsymbol{x}_1$ is the first data point in a dataset.

Basis functions are very general - they could specify that we just keep our input data the same. As a result, it's common to rewrite the least squares loss function from Equation **??** for linear regression in terms of the basis function applied to our input data:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}_n)^2 \tag{2.6}$$

To motivate why we might need basis functions for performing linear regression, let's consider this graph of 1-dimensional inputs $\boldsymbol{X}$ along with their target outputs $\boldsymbol{y}$, presented in Figure 2.3.

As we can see, we're not going to be able to fit a good line to this data. The best we can hope to do is something like that of Figure 2.4.

However, if we just apply a simple basis function to our data, in this case the square root function,

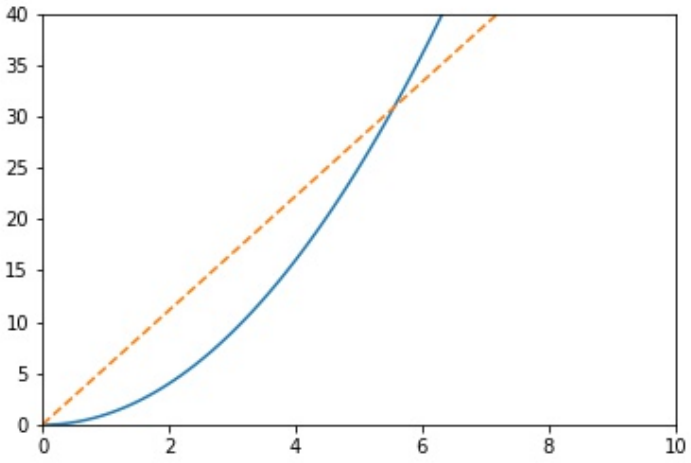


Figure 2.4: Data with no basis function applied, attempt to fit a line.

$\phi(\boldsymbol{x}) = (\sqrt{x_1})'$, we then have the red line in Figure 2.5. We now see that we can fit a very good line to our data, thanks to basis functions.

Still, the logical question remains: how can I choose the appropriate basis function? This toy example had a very obviously good basis function, but in general with high-dimensional, messy input data, how do we choose the basis function we need?

The answer is that this is not an easy problem to solve. Often, you may have some domain specific knowledge that tells you to try a certain basis, such as if you're working with chemical data and know that an important equation involves a certain function of one of your inputs. However, more often than not we won't have this expert knowledge either. Later, in the chapter on neural networks, we will discuss methods for discovering the best basis functions for our data automatically.

2.3.2 Regularization

When we introduced the idea of basis functions above, you might have wondered why we didn't just try adding many basis transformations to our input data to find a good transformation. For example, we might use this large basis function on a D-dimensional data point $\boldsymbol{z}$:

$$\phi(\boldsymbol{z}) = (z^{(1)}, {z^{(1)}}^2, ..., {z^{(1)}}^{100}, z^{(2)}, {z^{(2)}}^2, ..., {z^{(2)}}^{100}, ..., z^{(D)}, {z^{(D)}}^2, ..., {z^{(D)}}^{100})'$$

where you can see that we expand the dimensions of the data point to be 100 times its original size.

Let's say we have an input data point $\boldsymbol{x}$ that is 1-dimensional, and we apply the basis function described above, so that after the transformation each data point is represented by 100 values. Say we have 100 data points on which to perform linear regression, and because our transformed input space has 100 values, we have 100 parameters to fit. In this case, with one parameter per data point, it's possible for us to fit our regression line perfectly to our data so that we have no loss! But is this a desirable outcome? The answer is no, and we'll provide a visual example to illustrate that.

Imagine Figure 2.6 is our dataset. There is a very clear trend in this data, and you would likely draw a line that looks something like that of Figure 2.7 to fit it.

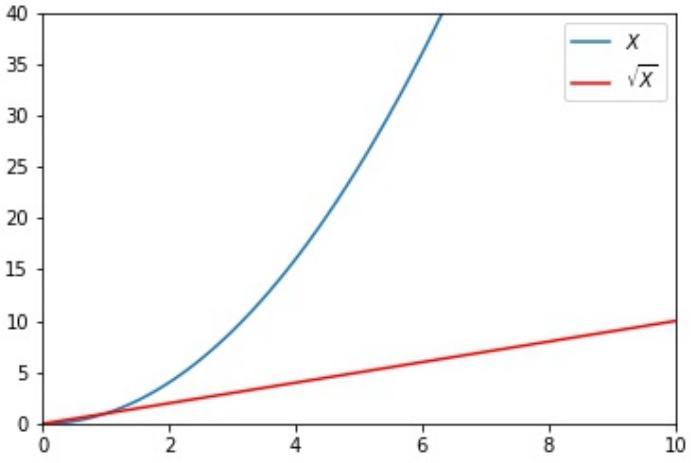


Figure 2.5: Data with square root basis function applied.

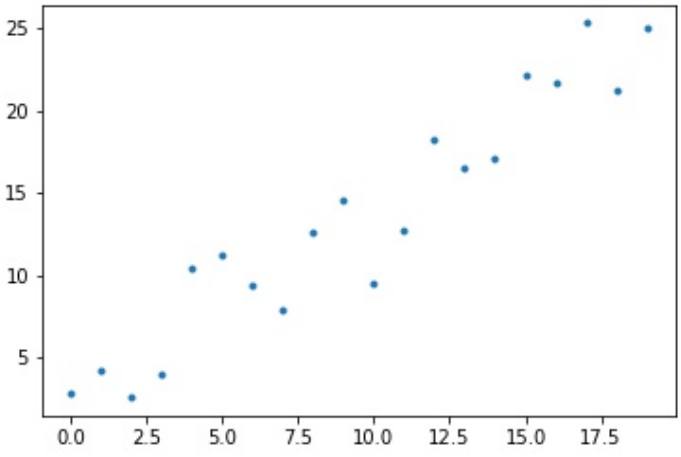


Figure 2.6: Dataset with a clear trend and Gaussian noise.

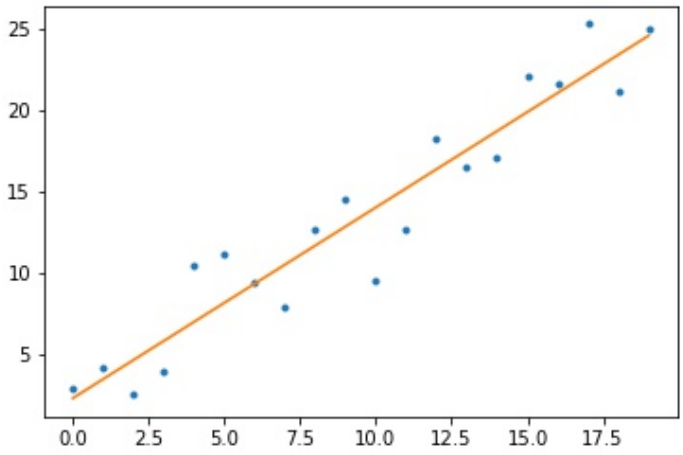


Figure 2.7: Natural fit for this dataset.

However, imagine we performed a large basis transformation like the one described above. If we do that, it's possible for us to fit our line perfectly, threading every data point, like that in Figure 2.8.

Let's see how both of these would perform on new data points. With our first regression line, if we have a new data point $\boldsymbol{x} = (10)'$, we would predict a target value of $\mathbf{14.1}$, which most people would agree is a pretty good measurement. However, with the second regression line, we would predict a value of $\mathbf{9.5}$, which most people would agree does not describe the general trend in the data. So how can we handle this problem elegantly?

Examining our loss function, we see that right now we're only penalizing predictions that are not correct in training. However, what we ultimately care about is doing well on new data points, not just our training set. This leads us to the idea of **generalization**.

Definition 2.3.2 (generalization): Generalization is the ability of a model to perform well on new data points outside of the training set.

A convoluted line that matches the noise of our training set exactly isn't going to generalize well to new data points that don't look exactly like those found in our training set. If wish to avoid recovering a convoluted line as our solution, we should also penalize the total size of our weights $\boldsymbol{w}$. The effect of this is to discourage many complex weight values that produce a messy regression line. By penalizing large weights, we favor simple regression lines like the one in Figure 2.7 that take advantage of only the most important basis functions.

The concept that we are introducing, penalizing large weights, is an example of what's known as **regularization**, and it's one that we will see come up often in different machine learning methods.

Definition 2.3.3 (regularization): Applying penalties to parameters of a model.

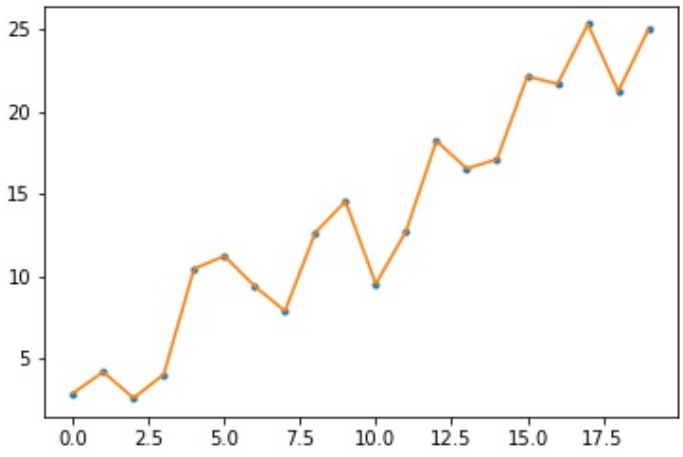


Figure 2.8: Unnatural fit for this dataset.

There is obviously a tradeoff between how aggressively we regularize our weights and how tightly our solution fits to our data, and we will formalize this tradeoff in the next section. However, for now, we will simply introduce a regularization parameter λ to our least squares loss function:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}_n)^2 + \frac{\lambda}{2}\boldsymbol{w}^{\top}\boldsymbol{w} \tag{2.7}$$

The effect of λ is to penalize large weight parameters. The larger λ is, the more we will favor simple solutions. In the limit $\lim_{\lambda\to\infty}\mathcal{L}(\boldsymbol{w})$, we will drive all weights to 0, while with a nonexistant $\lambda = 0$ we will apply no regularization at all. Notice that we're squaring our weight parameters - this is known as *L2 norm regularization* or **ridge regression**. While L2 norm regularization is very common, it is just one example of many ways we can perform regularization.

To build some intuition about the effect of this regularization parameter, examine Figure 2.9. Notice how larger values of λ produce less complex lines, which is the result of applying more regularization. This is very nice for the problem we started with - needing a way to choose which basis functions we wanted to use. With regularization, we can select many basis functions, and then allow regularization to 'prune' the ones that aren't meaningful (by driving their weight parameters to 0). While this doesn't mean that we should use as many basis transformations as possible (there will be computational overhead for doing this), it does allow us to create a much more flexible linear regression model without creating a convoluted regression line.

2.3.3 Generalizing Regularization

We've thus far only discussed one form of regularization: ridge regression. Remember that under ridge regression, the loss function takes the form:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}_n)^2 + \frac{\lambda}{2}\boldsymbol{w}^{\top}\boldsymbol{w},$$

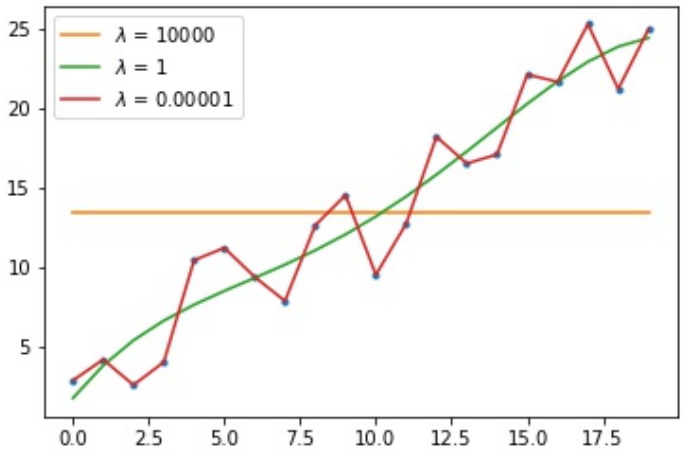


Figure 2.9: Effect of different regularization parameter values on final regression solution.

where the $(\lambda/2)\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}$ term is for the regularization. We can generalize our type of regularization by writing it as:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}_n)^2 + \frac{\lambda}{2}||\boldsymbol{w}||_h^h$$

where h determines the type of regularization we are using and thus the form of the optimal solution that we recover. For example, if $h = 2$ then we add $\lambda/2$ times the square of the L2 norm. The three most commonly used forms of regularization are lasso, ridge, and elastic net.

Ridge Regression
This is the case of $h = 2$, which we've already discussed, but what type of solutions does it tend to recover? Ridge regression prevents any individual weight from growing too large, providing us with solutions that are generally moderate.

Lasso Regression
Lasso regression is the case of $h = 1$. Unlike ridge regression, lasso regression will drive some parameters w_i to zero if they aren't informative for our final solution. Thus, lasso regression is good if you wish to recover a sparse solution that will allow you to throw out some of your basis functions. You can see the forms of ridge and lasso regression functions in Figure 2.10. If you think about how Lasso is L1 Norm (absolute value) and Ridge is L2 Norm (squared distance), you can think of those shapes as being the set of points (w_1, w_2) for which the norm takes on a constant value.

Elastic Net
Elastic net is a middle ground between ridge and lasso regression, which it achieves by using a linear combination of the previous two regularization terms. Depending on how heavily each regulariza-

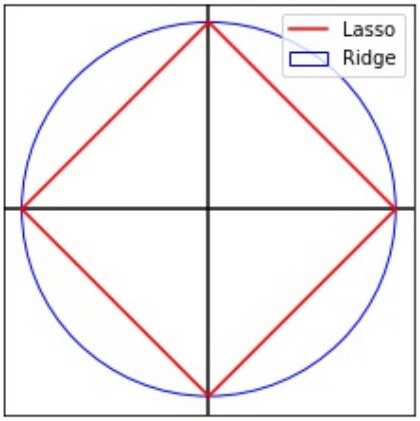


Figure 2.10: Form of the ridge (blue) and lasso (red) regression functions.

tion term is weighted, this can produce results on a spectrum between lasso and ridge regression.

2.3.4 Bayesian Regularization

We've seen regularization in the context of loss functions, where the goal is to penalize large weight values. How does the concept of regularization apply to Bayesian linear regression?

The answer is that we can interpret regularizing our weight parameters as adding a prior distribution over $\boldsymbol{w}$. Note that this is a different conception of regularization than we saw in the previous section. In the Bayesian framework, we are averaging over different models specified by different values of $\boldsymbol{w}$. Therefore, in this context regularization entails weighting models with smaller values of $\boldsymbol{w}$ more heavily.

Derivation 2.3.1 (Bayesian Regularization Derivation):
Because we wish to shrink our weight values toward 0 (which is exactly what regularization does), we will select a Normal prior with mean 0 and variance $\boldsymbol{S}_0^{-1}$:

$$\boldsymbol{w} \sim \mathcal{N}(0, \boldsymbol{S}_0^{-1}\boldsymbol{I})$$

Remember from Equation **??** that the distribution over our observed data is Normal as well, written here in terms of our entire dataset:

$$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \beta) = \mathcal{N}(\boldsymbol{X}\boldsymbol{w}, \beta^{-1}\boldsymbol{I})$$

$$\underbrace{p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}, \beta)}_{\text{posterior}} \propto \underbrace{p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \beta)}_{\text{likelihood}} \underbrace{p(\boldsymbol{w})}_{\text{prior}}$$

We now wish to find the value of $\boldsymbol{w}$ that maximizes the posterior distribution. We can maximize the log of the posterior with respect to $\boldsymbol{w}$, which simplifies the problem slightly:

$$\ln p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}, \beta) \propto \ln p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \beta) + \ln p(\boldsymbol{w})$$

Let's handle $\ln p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \beta)$ first:

$$\begin{aligned}
\ln p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \beta) &= \ln \prod_{n=1}^{N} \mathcal{N}(y_n|\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n, \beta^{-1}) \\
&= \ln \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\beta^{-1}}} \exp\left\{ -\frac{\beta}{2}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)^2 \right\} \\
&= \boldsymbol{C} - \frac{\beta}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)^2 + \frac{1}{\sqrt{2\pi\beta^{-1}}}
\end{aligned}$$

where $\boldsymbol{C}$ collects the constant terms that don't depend on $\boldsymbol{w}$. Let's now handle $\ln p(\boldsymbol{w})$:

$$\begin{aligned}
\ln p(\boldsymbol{w}) &= \ln \mathcal{N}(0, \boldsymbol{S}_0^{-1}\boldsymbol{I}) \\
&= \ln \frac{1}{(|2\pi\boldsymbol{S}_0^{-1}\boldsymbol{I}|)^{\frac{1}{2}}} \exp\left\{ -\frac{\boldsymbol{S}_0}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w} \right\} \\
&= \boldsymbol{C} - \frac{\boldsymbol{S}_0}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}
\end{aligned}$$

combining the terms for $\ln p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \beta)$ and $\ln p(\boldsymbol{w})$:

$$\ln p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}, \beta) = -\frac{\beta}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)^2 - \frac{\boldsymbol{S}_0}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}$$

dividing by a positive constant β:

$$\ln p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}, \beta) = -\frac{1}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)^2 - \frac{\boldsymbol{S}_0}{\beta}\frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}$$

Notice that maximizing the posterior probability is equivalent to minimizing the sum of squared errors $(y_n - \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n)^2$ and the regularization term $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}$.

The interpretation of this is that adding a prior over the distribution of our weight parameters w and then maximizing the resulting posterior distribution is equivalent to adding a regularization term where $\lambda = \frac{\boldsymbol{S}_0}{\beta}$

2.4 Model Selection

2.4.1 Bias-Variance Tradeoff and Decomposition

Now that you know about regularization, you might have some intuition for why we need to find a balance between complex and simple regression solutions. A complex solution, while it might fit all of our training data, may not generalize well to future data points. On the other hand, a line that is too simple might not vary enough to provide good predictions at all. This phenomenon is not unique to linear regression- it's actually a very fundamental concept in machine learning that's known as the **bias-variance tradeoff**.

Definition 2.4.1 (Bias-Variance Tradeoff): When constructing machine learning models, we have a choice somewhere on a spectrum between two extremes: fitting exactly to our training data or not varying in response to our training data at all. The first extreme, fitting all of our training data, is a situation of high *variance*, because our output changes heavily in reponse to our input data (see the red line in Figure 2.9). At the other extreme, a solution that doesn't change in response to our training data at all is a situation of high *bias* (see the yellow line in Figure 2.9). This means our model heavily favors a specific form regardless of the training data, so our target outputs don't fluctuate between distinct training sets.

Obviously a good solution will fall somewhere in between these two extremes of high variance and high bias. Indeed, we have techniques like regularization to help us balance the two extremes (improving generalization), and we have other techniques like *cross-validation* that help us determine when we have found a good balance (measuring generalization).

Note: In case you are not familiar with the terms *bias* and *variance*, we provide their statistical definitions here:

$$\text{bias}(\theta) = \text{E}[\theta] - \theta$$

$$\text{variance}(\theta) = \text{E}[(\theta - \text{E}[\theta])^2]$$

Before we discuss how to effectively mediate between these opposing forces of error in our models, we will first show that the bias-variance tradeoff is not only conceptual but also has probabilistic underpinnings. Specifically, any loss that we incur over our training set using a given model can be described in terms of bias and variance, as we will demonstrate now.

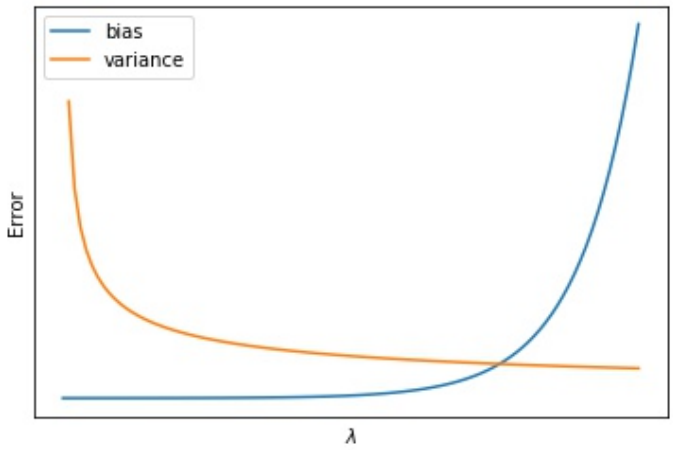


Figure 2.11: Bias and variance both contribute to the overall error of our model.

The key takeaway of the bias-variance decomposition is that the controllable error in our model is given by the squared bias and variance. Holding our error constant, to decrease bias requires increasing the variance in our model, and vice-versa. In general, a graph of the source of error in our model might look something like Figure 2.11.

For a moment, consider what happens on the far left side of this graph. Our variance is very high, and our bias is very low. In effect, we're fitting perfectly to all of the data in our dataset. This is exactly why we introduced the idea of regularization from before - we're fitting a very convoluted line that is able to pass through all of our data but which doesn't generalize well to new data points. There is a name for this: $\boldsymbol{overfitting}$.

Definition 2.4.2 (overfitting): A phenomenon where we construct a convoluted model that is able to predict every point in our dataset perfectly but which doesn't generalize well to new data points.

The opposite idea, $\boldsymbol{underfitting}$, is what happens at the far right of the graph: we have high bias and aren't responding to the variation in our dataset at all.

Definition 2.4.3 (underfitting): A phenomenon where we construct a model that doesn't respond to variation in our data.

So you can hopefully now see that the bias-variance tradeoff is important to managing the problem of overfitting and underfitting. Too much variance in our model and we'll overfit to our dataset. Too much bias and we won't account for the trends in our dataset at all.

In general, we would like to find a sweet spot of moderate bias and variance that produces minimal error. In the next section, we will explore how we find this sweet spot.

2.4.2 Cross-Validation

We've seen that in choosing a model, we incur error that can be described in terms of bias and variance. We've also seen that we can regulate the source of error through regularization, where heavier regularization increases the bias of our model. A natural question then is how do we know how much regularization to apply to achieve a good balance of bias and variance?

Another way to look at this is that we've traded the question of finding the optimal number of basis functions for finding the optimal value of the regularization parameter λ, which is often an easier problem in most contexts.

One very general technique for finding the sweet spot of our regularization parameter, other hyperparameters, or even for choosing among entirely different models is known as $\boldsymbol{cross-validation}$.

Definition 2.4.4 (cross-validation): A subsampling procedure used over a dataset to tune hyperparameters and avoid over-fitting. Some portion of a dataset (10-20% is common) is set aside, and training is performed on the remaining, larger portion of data. When training is complete, the smaller portion of data left out of training is used for testing. The larger portion of data is sometimes referred to as the *training set*, and the smaller portion is sometimes referred to as the *validation set*.

Cross-validation is often performed more than once for a given setting of hyperparameters to avoid a skewed set of validation data being selected by chance. In $\boldsymbol{K-Foldscross-validation}$, you perform cross-validation $\boldsymbol{K}$ times, allocating $\frac{1}{\boldsymbol{K}}$ of your data for the validation set at each iteration.

Let's tie this back into finding a good regularization parameter. For a given value of λ, we will incur a certain amount of error in our model. We can measure this error using cross-validation, where we train our model on the training set and compute the final error using the validation set. To find the optimal value for λ, we perform cross-validation using different values of λ, eventually settling on the value that produces the lowest final error. This will effectively trade off bias and variance, finding the value of λ that minimizes the total error.

You might wonder why we need to perform cross-validation at all - why can't we train on the entire dataset and then compute the error over the entire dataset as well?

The answer is again overfitting. If we train over the entire dataset and then validate our results on the exact same dataset, we are likely to choose a regularization parameter that encourages our model to conform to the exact variation in our dataset instead of finding the generalizable trends. By training on one set of data, and then validating on a completely different set of data, we force our model to find good generalizations in our dataset. This ultimately allows us to pick the regularization term λ that finds the sweet spot between bias and variance, overfitting and underfitting.

2.4.3 Making a Model Choice

Now that we're aware of overfitting, underfitting, and how those concepts relate to the bias-variance tradeoff, we still need to come back to the question of how we actually select a model. Intuitively, we are trying to find the middle ground between bias and variance: picking a model that fits our data but that is also general enough to perform well on yet unseen data. Furthermore, there is no such thing as the 'right' model choice. Instead, there are only model options that are either better or worse than others. To that end, it can be best to rely on the techniques presented above, specifically cross-validation, to make your model selection. Then, although you will not be able to

make any sort of guarantee about your selection being the 'best' of all possible models, you can at least have confidence your model achieved the best generalizability that could be proven through cross-validation.

2.4.4 Bayesian Model Averaging

We can also handle model selection using a Bayesian approach. This means we account for our uncertainty about the true model by averaging over the possible candidate models, weighting each model by our prior certainty that it is the one producing our data. If we have M models indexed by $m = 1, ..., M$, we can write the likelihood of observing our dataset $\boldsymbol{X}$ as follows:

$$p(\boldsymbol{X}) = \sum_{m=1}^{M} p(\boldsymbol{X}|m)p(m)$$

where $p(m)$ is our prior certainty for a given model and $p(\boldsymbol{X}|m)$ is the likelihood of our dataset given that model. The elegance of this approach is that we don't have to pick any particular model, instead choosing to marginalize out our uncertainty.

2.5 Linear Regression Extras

With most of linear regression under our belt at this point, it's useful to drill down on a few concepts to come to a deeper understanding of how we can use them in the context of linear regression and beyond.

2.5.1 Predictive Distribution

Remaining in the setting of Bayesian Linear Regression, we may wish to get a distribution over our weights $\boldsymbol{w}$ instead of a point estimator for it using maximum likelihood. As we saw in Section 2.3.4, we can introduce a prior distribution over $\boldsymbol{w}$, then together with our observed data, we can produce a posterior distribution over $\boldsymbol{w}$ as desired.

Derivation 2.5.1 (posterior predictive derivation):
For the sake of simplicity and ease of use, we will select our prior over $\boldsymbol{w}$ to be a Normal distribution with mean $\boldsymbol{\mu}_0$ and variance $\boldsymbol{S}_0^{-1}$:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{S}_0^{-1})$$

Remembering that the observed data is normally distributed, and accounting for Normal-Normal conjugacy, our posterior distribution will be Normal as well:

$$p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}, \beta) = \mathcal{N}(\boldsymbol{\mu}_n, \boldsymbol{S}_n^{-1})$$

where

$$\boldsymbol{S}_n = (\boldsymbol{S}_0^{-1} + \beta \boldsymbol{X}^{\mathrm{T}} \boldsymbol{X})^{-1}$$
$$\boldsymbol{\mu}_n = \boldsymbol{S}_n(\boldsymbol{S}_0^{-1}\boldsymbol{\mu}_0 + \beta \boldsymbol{X}\boldsymbol{y})$$

We now have a posterior distribution over $\boldsymbol{w}$. However, usually this distribution is not what we care about. We're actually interested in making a point prediction for the target y^* given a new

input $\boldsymbol{x}^*$. How do we go from a posterior distribution over $\boldsymbol{w}$ to this prediction?

The answer is using what's known as the **posterior predictive** over y^* given by:

$$\begin{aligned} p(y^*|\boldsymbol{x}^*, \boldsymbol{X}, \boldsymbol{y}) &= \int_{\boldsymbol{w}} p(y^*|\boldsymbol{x}^*, \boldsymbol{w}) p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) d\boldsymbol{w} \\ &= \int_{\boldsymbol{w}} \mathcal{N}(y^*|\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}^*, \beta^{-1}) \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}_n, \boldsymbol{S}_n^{-1}) d\boldsymbol{w} \end{aligned} \tag{2.8}$$

The idea here is to average the probability of y^* over all the possible setting of w, weighting the probabilities by how likely each setting of w is according to its posterior distribution.

2.6 Conclusion

In this chapter, we looked at a specific tool for handling regression problems known as linear regression. We've seen linear regression described in terms of loss functions, probabilistic expressions, and geometric projections, which reflects the deep body of knowledge that we have around this very common technique.

We've also discussed many concepts in this chapter that will prove useful in other areas of machine learning, particularly for other supervised techniques: loss functions, regularization, bias and variance, over and underfitting, posterior distributions, maximum likelihood estimation, and cross-validation among others. Spending time to develop an understanding of these concepts now will pay off going forward.

It may or may not be obvious at this point that we are missing a technique for a very large class of problems: those where the solution is not just a continuous, real number. How do we handle situations where we need to make a choice between different discrete options? This is the question we will turn to in the next chapter.