**Machine Learning**

Framing: Key ML Terminology

What is (supervised) machine learning? Concisely put, it is the following:

* ML systems learn how to combine input to produce useful predictions on never-before-seen data.

Let's explore fundamental machine learning terminology.

**Labels**

A **label** is the thing we're predicting—the y variable in simple linear regression. The label could be the future price of wheat, the kind of animal shown in a picture, the meaning of an audio clip, or just about anything.

**Features**

A **feature** is an input variable—the x variable in simple linear regression. A simple machine learning project might use a single feature, while a more sophisticated machine learning project could use millions of features, specified as:

x1,x2,...xn

In the spam detector example, the features could include the following:

* words in the email text
* sender's address
* time of day the email was sent
* email contains the phrase "one weird trick."

**Examples**

An **example** is a particular instance of data, **x**. (We put **x** in boldface to indicate that it is a vector.) We break examples into two categories:

* labeled examples
* unlabeled examples

A **labeled example** includes both feature(s) and the label. That is:

  labeled examples: {features, label}: (x, y)

Use labeled examples to **train** the model. In our spam detector example, the labeled examples would be individual emails that users have explicitly marked as "spam" or "not spam."

For example, the following table shows 5 labeled examples from a [data set](https://developers.google.com/machine-learning/crash-course/california-housing-data-description) containing information about housing prices in California:

|  |  |  |  |
| --- | --- | --- | --- |
| **housingMedianAge (feature)** | **totalRooms (feature)** | **totalBedrooms (feature)** | **medianHouseValue (label)** |
| 15 | 5612 | 1283 | 66900 |
| 19 | 7650 | 1901 | 80100 |
| 17 | 720 | 174 | 85700 |
| 14 | 1501 | 337 | 73400 |
| 20 | 1454 | 326 | 65500 |

An **unlabeled example** contains features but not the label. That is:

  unlabeled examples: {features, ?}: (x, ?)

Here are 3 unlabeled examples from the same housing dataset, which exclude medianHouseValue:

|  |  |  |
| --- | --- | --- |
| **housingMedianAge (feature)** | **totalRooms (feature)** | **totalBedrooms (feature)** |
| 42 | 1686 | 361 |
| 34 | 1226 | 180 |
| 33 | 1077 | 271 |

Once we've trained our model with labeled examples, we use that model to predict the label on unlabeled examples. In the spam detector, unlabeled examples are new emails that humans haven't yet labeled.

**Models**

A model defines the relationship between features and label. For example, a spam detection model might associate certain features strongly with "spam". Let's highlight two phases of a model's life:

* **Training** means creating or **learning** the model. That is, you show the model labeled examples and enable the model to gradually learn the relationships between features and label.
* **Inference** means applying the trained model to unlabeled examples. That is, you use the trained model to make useful predictions (y'). For example, during inference, you can predict medianHouseValue for new unlabeled examples.

**Regression vs. classification**

A **regression** model predicts continuous values. For example, regression models make predictions that answer questions like the following:

* What is the value of a house in California?
* What is the probability that a user will click on this ad?

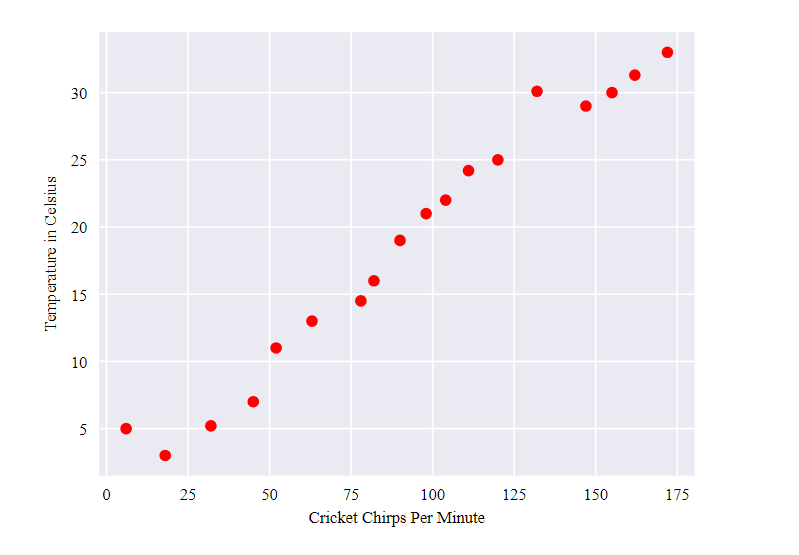
A **classification** model predicts discrete values. For example, classification models make predictions that answer questions like the following:

* Is a given email message spam or not spam?
* Is this an image of a dog, a cat, or a hamster?

Descending into ML: Linear Regression

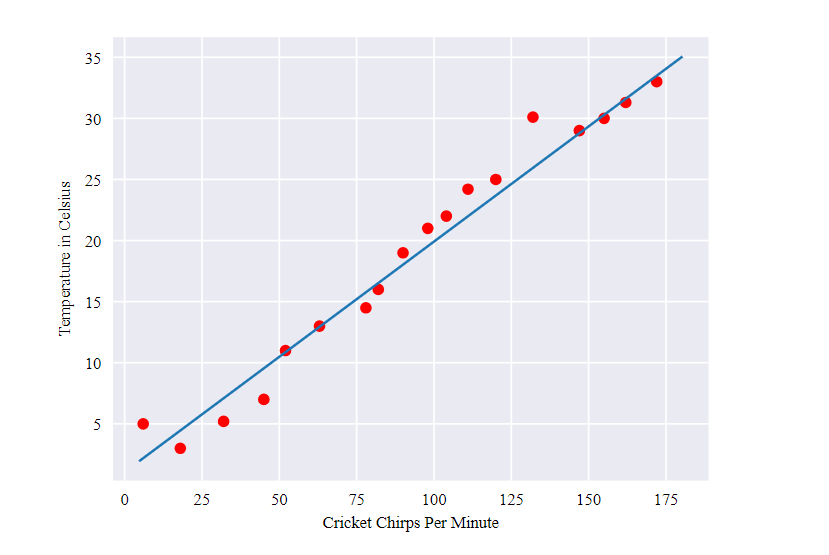
t has long been known that crickets (an insect species) chirp more frequently on hotter days than on cooler days. For decades, professional and amateur scientists have cataloged data on chirps-per-minute and temperature. As a birthday gift, your Aunt Ruth gives you her cricket database and asks you to learn a model to predict this relationship. Using this data, you want to explore this relationship.

First, examine your data by plotting it:



**Figure 1. Chirps per Minute vs. Temperature in Celsius.**

As expected, the plot shows the temperature rising with the number of chirps. Is this relationship between chirps and temperature linear? Yes, you could draw a single straight line like the following to approximate this relationship:



**Figure 2. A linear relationship.**

True, the line doesn't pass through every dot, but the line does clearly show the relationship between chirps and temperature. Using the equation for a line, you could write down this relationship as follows:

y=mx + b

where:

* y is the temperature in Celsius—the value we're trying to predict.
* m is the slope of the line.
* x is the number of chirps per minute—the value of our input feature.
* b is the y-intercept.

By convention in machine learning, you'll write the equation for a model slightly differently:

y′=b+w1x1

where:

* y′ is the predicted [label](https://developers.google.com/machine-learning/crash-course/framing/ml-terminology#labels) (a desired output).
* b is the bias (the y-intercept), sometimes referred to as �0.
* w1 is the weight of feature 1. Weight is the same concept as the "slope" � in the traditional equation of a line.
* x1 is a [feature](https://developers.google.com/machine-learning/crash-course/framing/ml-terminology#features) (a known input).

To **infer** (predict) the temperature y′ for a new chirps-per-minute value x1, just substitute the x1 value into this model.

Although this model uses only one feature, a more sophisticated model might rely on multiple features, each having a separate weight (w1, w2, etc.). For example, a model that relies on three features might look as follows:

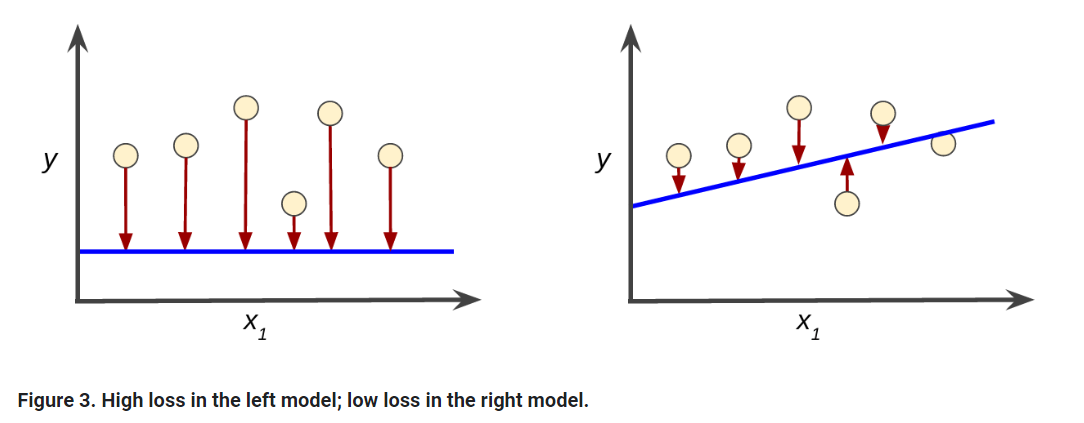
y′=b+w1x1+w2x2+w3x3

Descending into ML: Training and Loss

**Training** a model simply means learning (determining) good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called **empirical risk minimization**.

Loss is the penalty for a bad prediction. That is, **loss** is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. The goal of training a model is to find a set of weights and biases that have *low* loss, on average, across all examples. For example, Figure 3 shows a high loss model on the left and a low loss model on the right. Note the following about the figure:

* The arrows represent loss.
* The blue lines represent predictions.



Notice that the arrows in the left plot are much longer than their counterparts in the right plot. Clearly, the line in the right plot is a much better predictive model than the line in the left plot.

You might be wondering whether you could create a mathematical function—a loss function—that would aggregate the individual losses in a meaningful fashion.

### Squared loss: a popular loss function

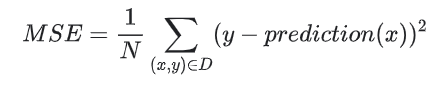
The linear regression models we'll examine here use a loss function called **squared loss** (also known as **L2 loss**). The squared loss for a single example is as follows:

= the square of the difference between the label and the prediction

= (observation - prediction(**x**))2

= (y - y')2

**Mean square error** (**MSE**) is the average squared loss per example over the whole dataset. To calculate MSE, sum up all the squared losses for individual examples and then divide by the number of examples:



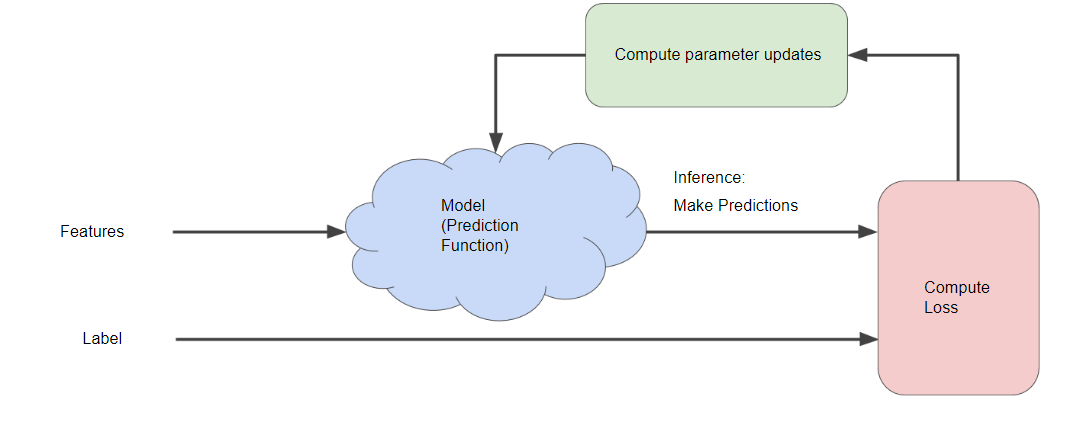


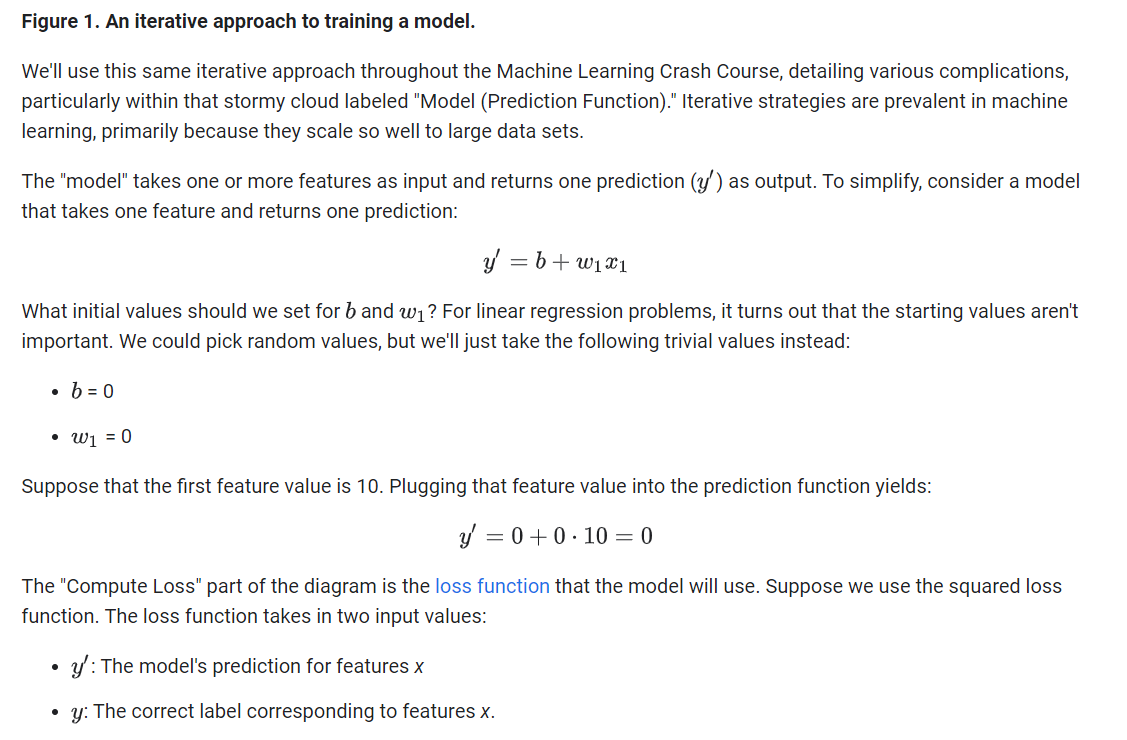
Reducing Loss: An Iterative Approach

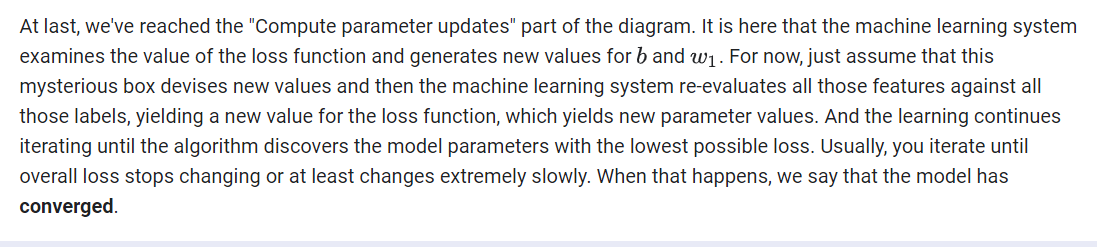
The [previous module](https://developers.google.com/machine-learning/crash-course/descending-into-ml) introduced the concept of loss. Here, in this module, you'll learn how a machine learning model iteratively reduces loss.

Iterative learning might remind you of the ["Hot and Cold"](http://www.howcast.com/videos/258352-how-to-play-hot-and-cold/) kid's game for finding a hidden object like a thimble. In this game, the "hidden object" is the best possible model. You'll start with a wild guess ("The value of w1 is 0.") and wait for the system to tell you what the loss is. Then, you'll try another guess ("The value of w1 is 0.5.") and see what the loss is. Aah, you're getting warmer. Actually, if you play this game right, you'll usually be getting warmer. The real trick to the game is trying to find the best possible model as efficiently as possible.

The following figure suggests the iterative trial-and-error process that machine learning algorithms use to train a model:







The iterative approach diagram ([Figure 1](https://developers.google.com/machine-learning/crash-course/reducing-loss/an-iterative-approach#ml-block-diagram)) contained a green hand-wavy box entitled "Compute parameter updates." We'll now replace that algorithmic fairy dust with something more substantial.

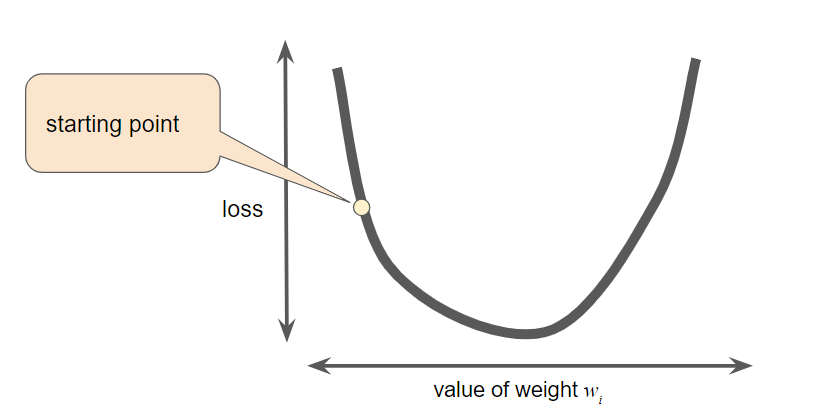
Suppose we had the time and the computing resources to calculate the loss for all possible values of w1. For the kind of regression problems we've been examining, the resulting plot of loss vs. w1 will always be convex. In other words, the plot will always be bowl-shaped, kind of like this:



Convex problems have only one minimum; that is, only one place where the slope is exactly 0. That minimum is where the loss function converges.

Calculating the loss function for every conceivable value of w1 over the entire data set would be an inefficient way of finding the convergence point. Let's examine a better mechanism—very popular in machine learning—called **gradient descent**.

The first stage in gradient descent is to pick a starting value (a starting point) for w1. The starting point doesn't matter much; therefore, many algorithms simply set w1 to 0 or pick a random value. The following figure shows that we've picked a starting point slightly greater than 0:



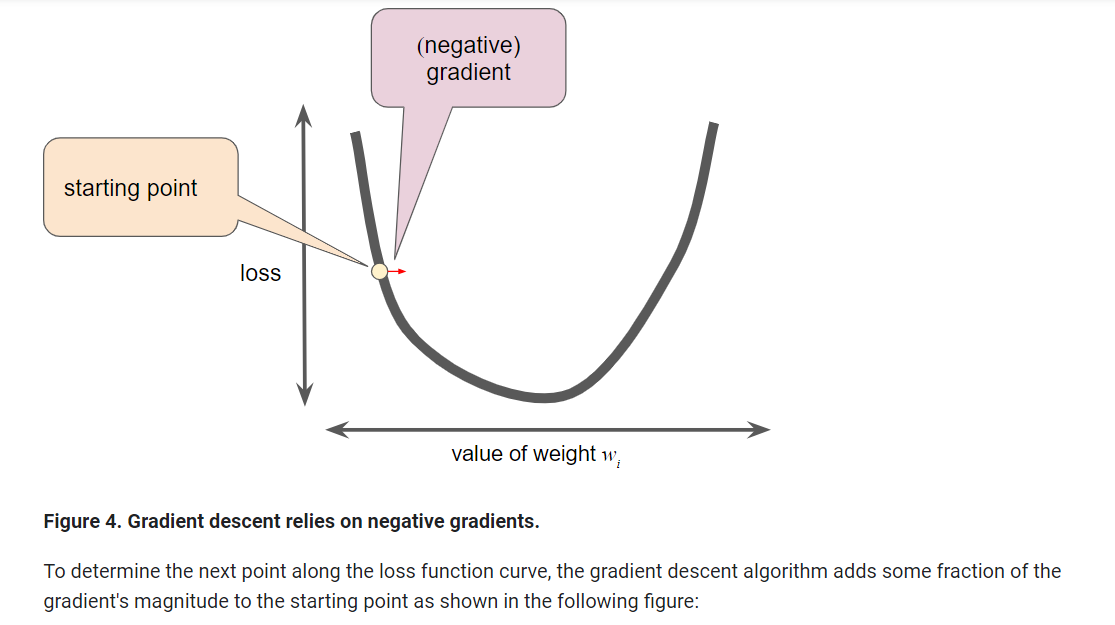
**Figure 3. A starting point for gradient descent.**

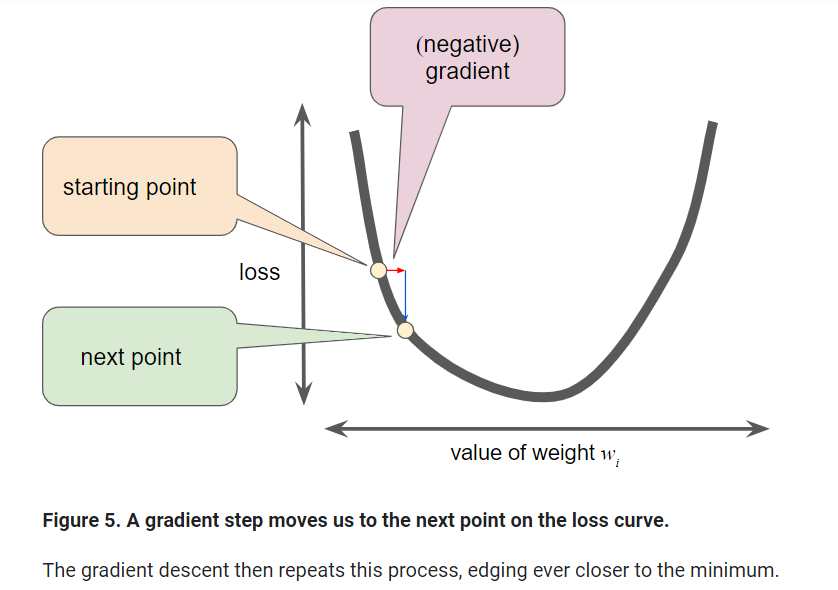
The gradient descent algorithm then calculates the gradient of the loss curve at the starting point. Here in Figure 3, the gradient of the loss is equal to the [derivative](https://wikipedia.org/wiki/Differential_calculus#The_derivative) (slope) of the curve, and tells you which way is "warmer" or "colder." When there are multiple weights, the **gradient** is a vector of partial derivatives with respect to the weights.

Note that a gradient is a vector, so it has both of the following characteristics:

* a direction
* a magnitude

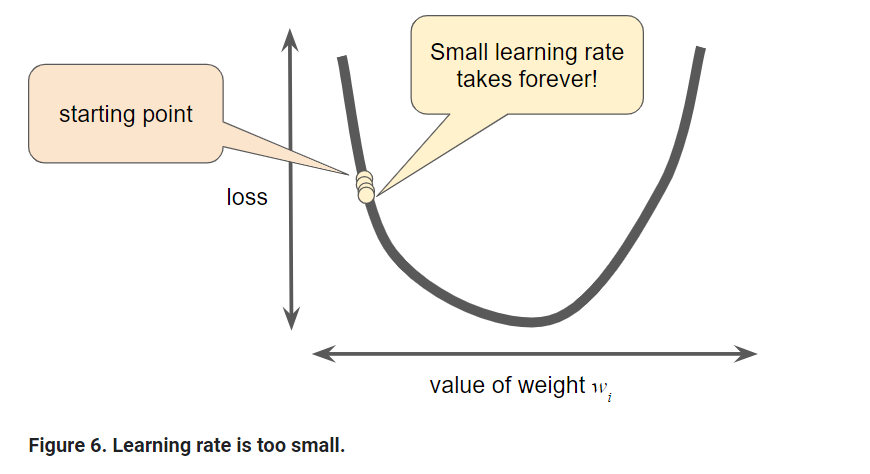
The gradient always points in the direction of steepest increase in the loss function. The gradient descent algorithm takes a step in the direction of the negative gradient in order to reduce loss as quickly as possible.



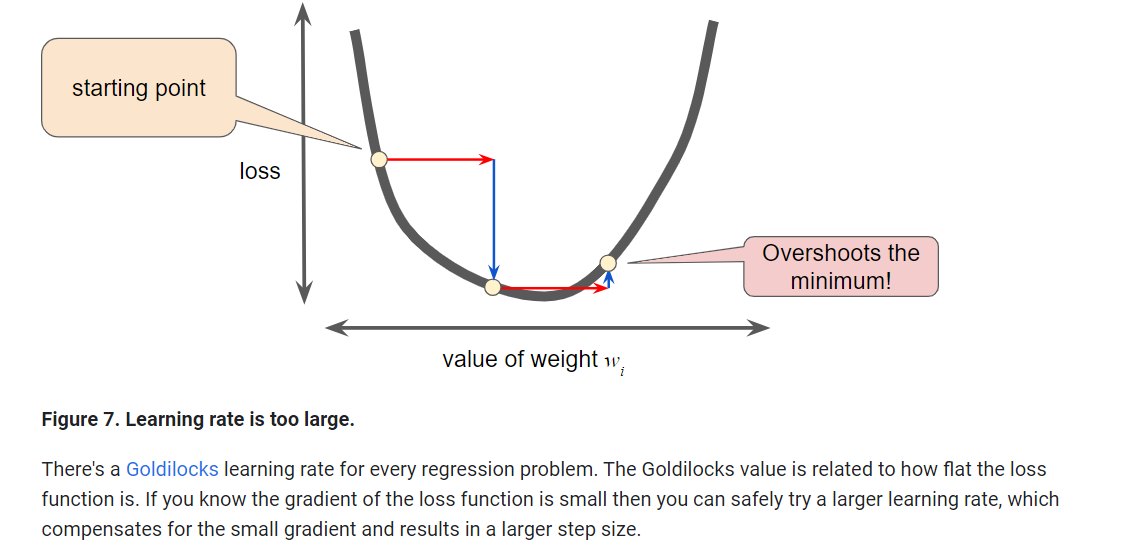


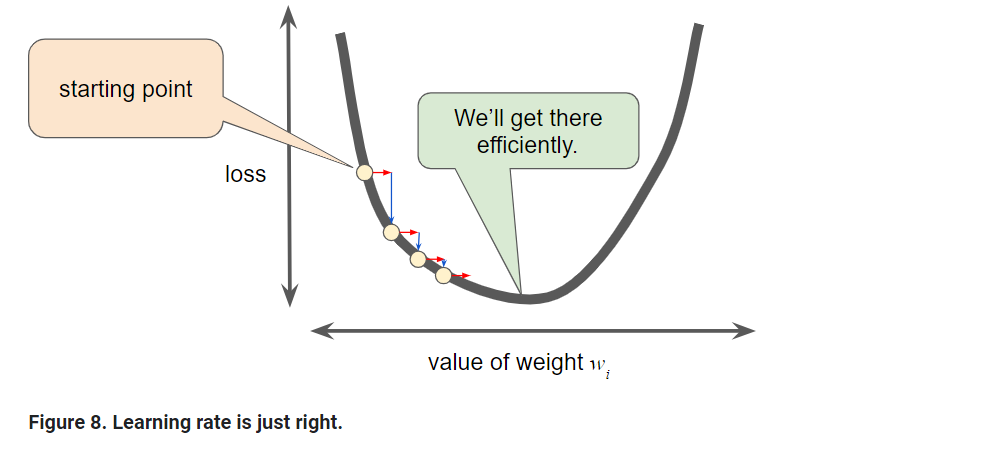
As noted, the gradient vector has both a direction and a magnitude. Gradient descent algorithms multiply the gradient by a scalar known as the **learning rate** (also sometimes called **step size**) to determine the next point. For example, if the gradient magnitude is 2.5 and the learning rate is 0.01, then the gradient descent algorithm will pick the next point 0.025 away from the previous point.

**Hyperparameters** are the knobs that programmers tweak in machine learning algorithms. Most machine learning programmers spend a fair amount of time tuning the learning rate. If you pick a learning rate that is too small, learning will take too long:



Conversely, if you specify a learning rate that is too large, the next point will perpetually bounce haphazardly across the bottom of the well like a quantum mechanics experiment gone horribly wrong:





Reducing Loss: Stochastic Gradient Descent

In gradient descent, a **batch** is the set of examples you use to calculate the gradient in a single training iteration. So far, we've assumed that the batch has been the entire data set. When working at Google scale, data sets often contain billions or even hundreds of billions of examples. Furthermore, Google data sets often contain huge numbers of features. Consequently, a batch can be enormous. A very large batch may cause even a single iteration to take a very long time to compute.

A large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows. Some redundancy can be useful to smooth out noisy gradients, but enormous batches tend not to carry much more predictive value than large batches.

What if we could get the right gradient on average for much less computation? By choosing examples at random from our data set, we could estimate (albeit, noisily) a big average from a much smaller one. **Stochastic gradient descent** (**SGD**) takes this idea to the extreme--it uses only a single example (a batch size of 1) per iteration. Given enough iterations, SGD works but is very noisy. The term "stochastic" indicates that the one example comprising each batch is chosen at random.

**Mini-batch stochastic gradient descent** (**mini-batch SGD**) is a compromise between full-batch iteration and SGD. A mini-batch is typically between 10 and 1,000 examples, chosen at random. Mini-batch SGD reduces the amount of noise in SGD but is still more efficient than full-batch.

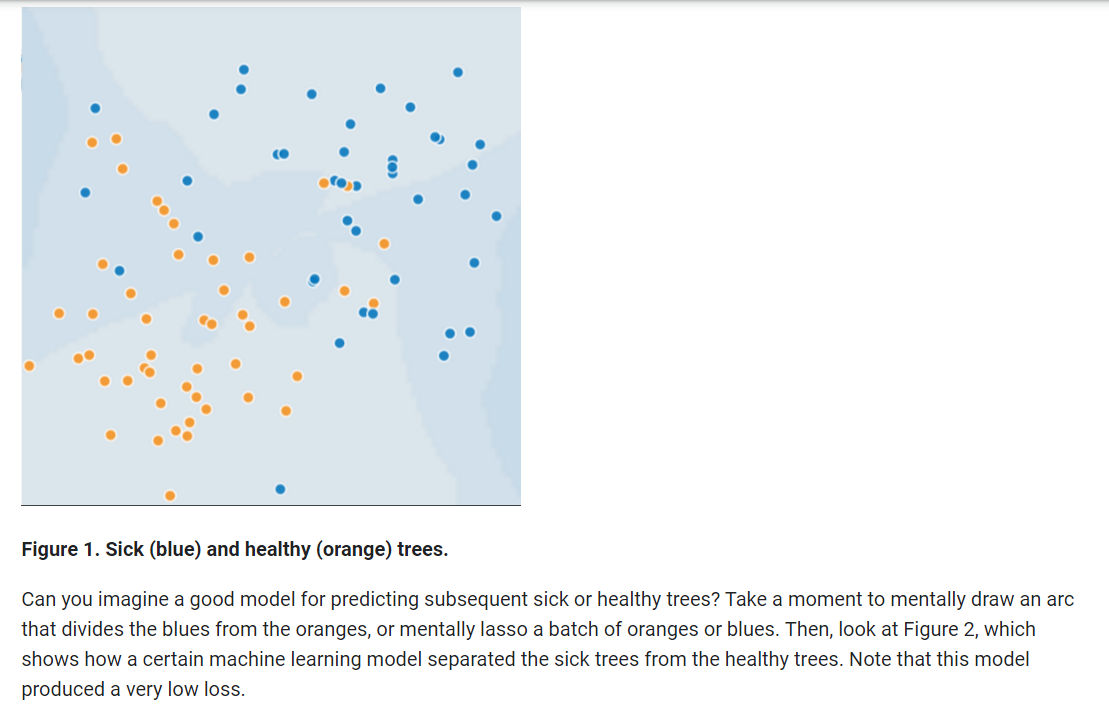
To simplify the explanation, we focused on gradient descent for a single feature. Rest assured that gradient descent also works on feature sets that contain multiple features.

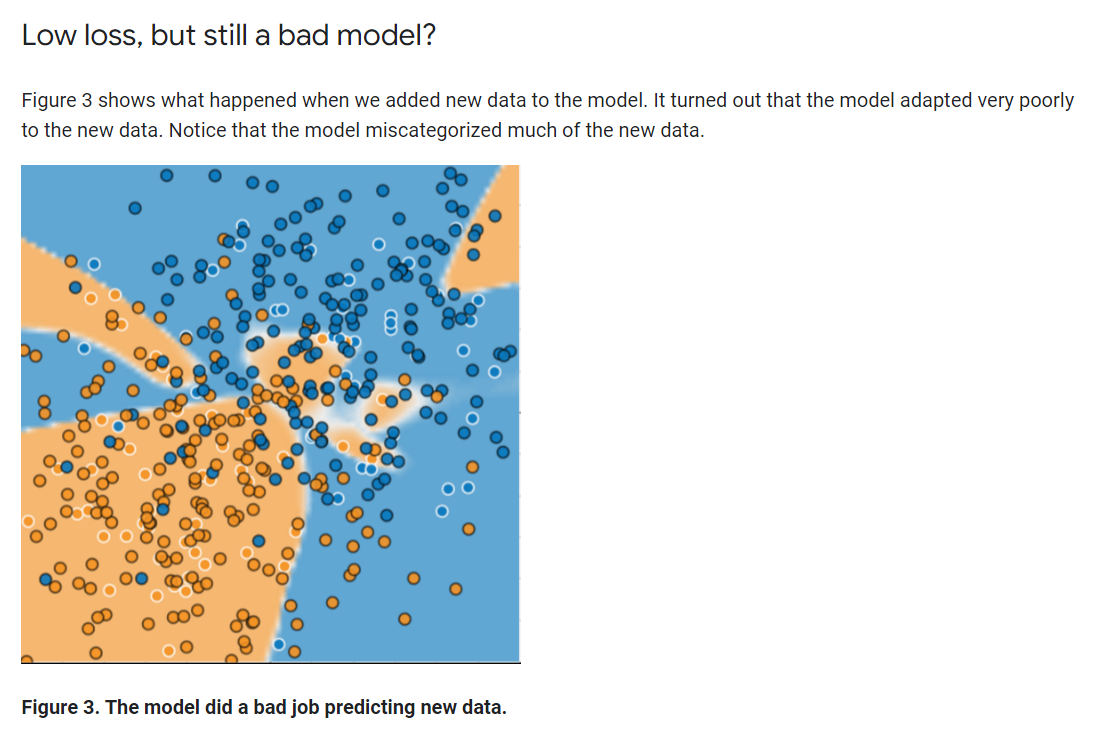
# Generalization: Peril of Overfitting

This module focuses on generalization. In order to develop some intuition about this concept, you're going to look at three figures. Assume that each dot in these figures represents a tree's position in a forest. The two colors have the following meanings:

* The blue dots represent sick trees.
* The orange dots represent healthy trees.

With that in mind, take a look at Figure 1.





The model shown in Figures 2 and 3 **overfits** the peculiarities of the data it trained on. An overfit model gets a low loss during training but does a poor job predicting new data. If a model fits the current sample well, how can we trust that it will make good predictions on new data? As you'll see [later on](https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/l2-regularization), overfitting is caused by making a model more complex than necessary. The fundamental tension of machine learning is between fitting our data well, but also fitting the data as simply as possible.

Machine learning's goal is to predict well on new data drawn from a (hidden) true probability distribution. Unfortunately, the model can't see the whole truth; the model can only sample from a training data set. If a model fits the current examples well, how can you trust the model will also make good predictions on never-before-seen examples?

William of Ockham, a 14th century friar and philosopher, loved simplicity. He believed that scientists should prefer simpler formulas or theories over more complex ones. To put Ockham's razor in machine learning terms:

The less complex an ML model, the more likely that a good empirical result is not just due to the peculiarities of the sample.

In modern times, we've formalized Ockham's razor into the fields of **statistical learning theory** and **computational learning theory**. These fields have developed **generalization bounds**--a statistical description of a model's ability to generalize to new data based on factors such as:

* the complexity of the model
* the model's performance on training data

While the theoretical analysis provides formal guarantees under idealized assumptions, they can be difficult to apply in practice. Machine Learning Crash Course focuses instead on empirical evaluation to judge a model's ability to generalize to new data.

A machine learning model aims to make good predictions on new, previously unseen data. But if you are building a model from your data set, how would you get the previously unseen data? Well, one way is to divide your data set into two subsets:

* **training set**—a subset to train a model.
* **test set**—a subset to test the model.

Good performance on the test set is a useful indicator of good performance on the new data in general, assuming that:

* The test set is large enough.
* You don't cheat by using the same test set over and over.

## The ML fine print

The following three basic assumptions guide generalization:

* We draw examples **independently and identically** (**i.i.d**) at random from the distribution. In other words, examples don't influence each other. (An alternate explanation: i.i.d. is a way of referring to the randomness of variables.)
* The distribution is **stationary**; that is the distribution doesn't change within the data set.
* We draw examples from partitions from the **same distribution.**

In practice, we sometimes violate these assumptions. For example:

* Consider a model that chooses ads to display. The i.i.d. assumption would be violated if the model bases its choice of ads, in part, on what ads the user has previously seen.
* Consider a data set that contains retail sales information for a year. User's purchases change seasonally, which would violate stationarity.

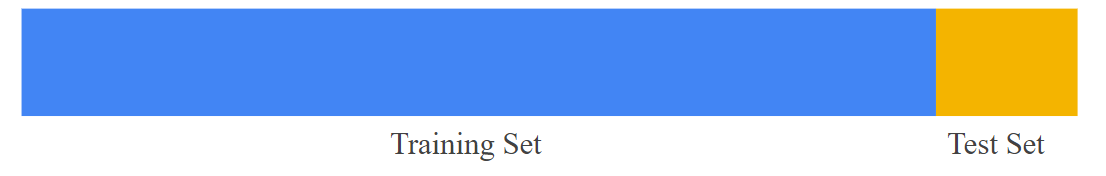
When we know that any of the preceding three basic assumptions are violated, we must pay careful attention to metrics.

# Training and Test Sets: Splitting Data

The previous module introduced the idea of dividing your data set into two subsets:

* **training set**—a subset to train a model.
* **test set**—a subset to test the trained model.

You could imagine slicing the single data set as follows:

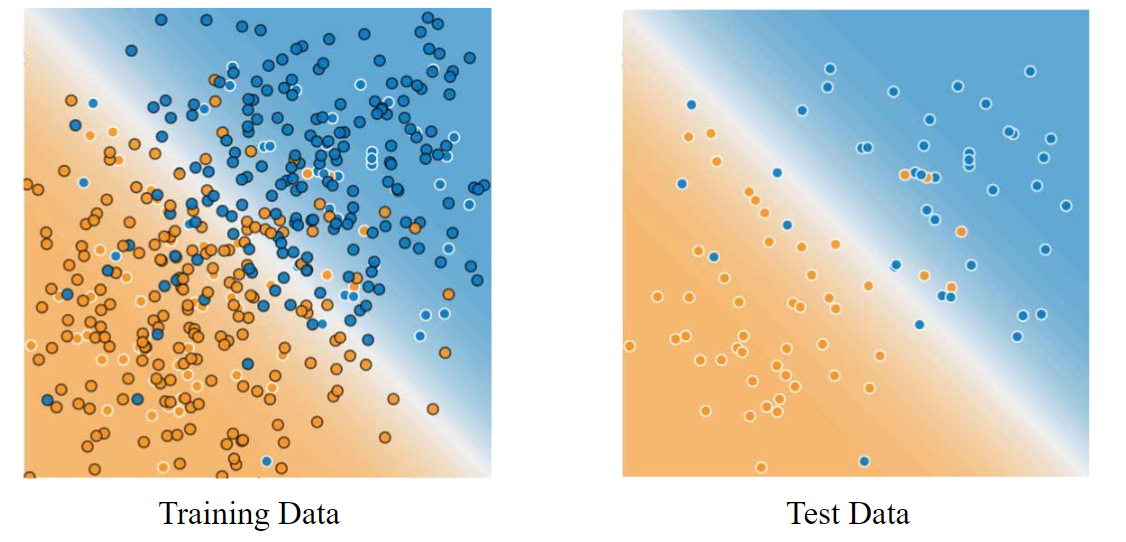


**Figure 1. Slicing a single data set into a training set and test set.**

Make sure that your test set meets the following two conditions:

* Is large enough to yield statistically meaningful results.
* Is representative of the data set as a whole. In other words, don't pick a test set with different characteristics than the training set.

Assuming that your test set meets the preceding two conditions, your goal is to create a model that generalizes well to new data. Our test set serves as a proxy for new data. For example, consider the following figure. Notice that the model learned for the training data is very simple. This model doesn't do a perfect job—a few predictions are wrong. However, this model does about as well on the test data as it does on the training data. In other words, this simple model does not overfit the training data.



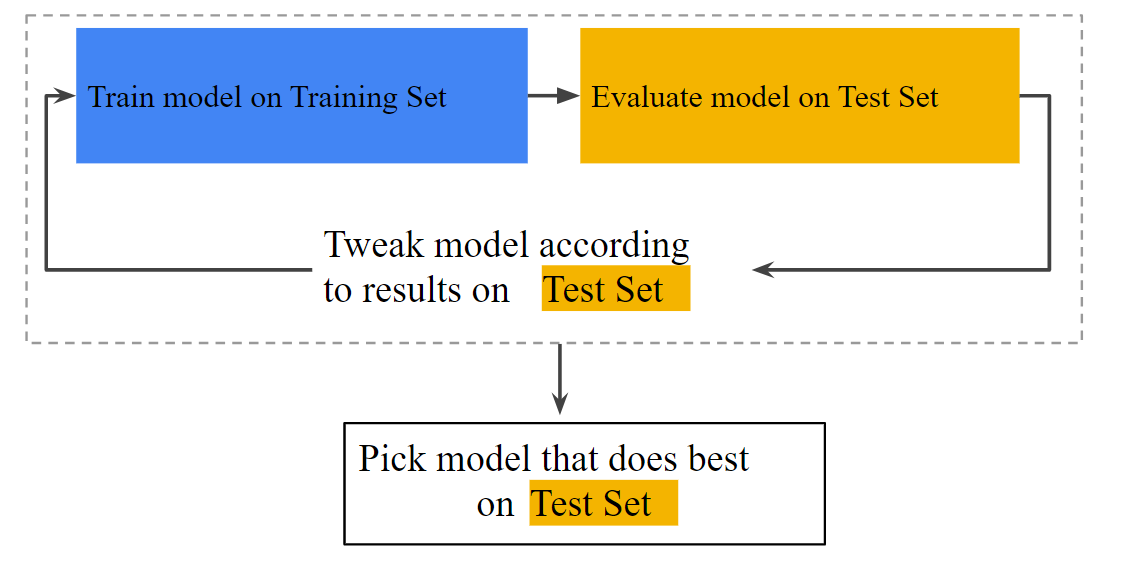
**Figure 2. Validating the trained model against test data.**

**Never train on test data.** If you are seeing surprisingly good results on your evaluation metrics, it might be a sign that you are accidentally training on the test set. For example, high accuracy might indicate that test data has leaked into the training set.

For example, consider a model that predicts whether an email is spam, using the subject line, email body, and sender's email address as features. We apportion the data into training and test sets, with an 80-20 split. After training, the model achieves 99% precision on both the training set and the test set. We'd expect a lower precision on the test set, so we take another look at the data and discover that many of the examples in the test set are duplicates of examples in the training set (we neglected to scrub duplicate entries for the same spam email from our input database before splitting the data). We've inadvertently trained on some of our test data, and as a result, we're no longer accurately measuring how well our model generalizes to new data.

Validation Set: Another Partition

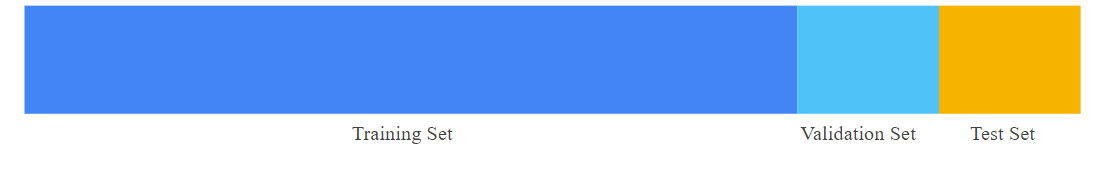
The [previous module](https://developers.google.com/machine-learning/crash-course/training-and-test-sets/video-lecture) introduced partitioning a data set into a training set and a test set. This partitioning enabled you to train on one set of examples and then to test the model against a different set of examples. With two partitions, the workflow could look as follows:

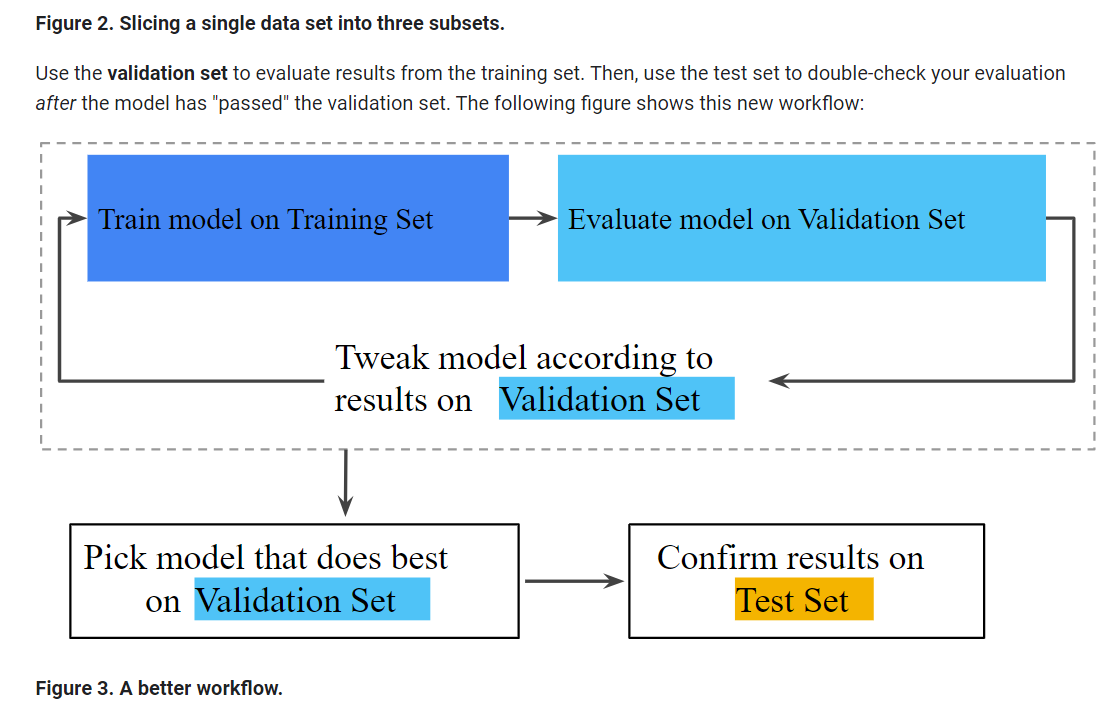


**Figure 1. A possible workflow?**

In the figure, "Tweak model" means adjusting anything about the model you can dream up—from changing the learning rate, to adding or removing features, to designing a completely new model from scratch. At the end of this workflow, you pick the model that does best on the *test set*.

Dividing the data set into two sets is a good idea, but not a panacea. You can greatly reduce your chances of overfitting by partitioning the data set into the three subsets shown in the following figure:





In this improved workflow:

1. Pick the model that does best on the validation set.
2. Double-check that model against the test set.

This is a better workflow because it creates fewer exposures to the test set.

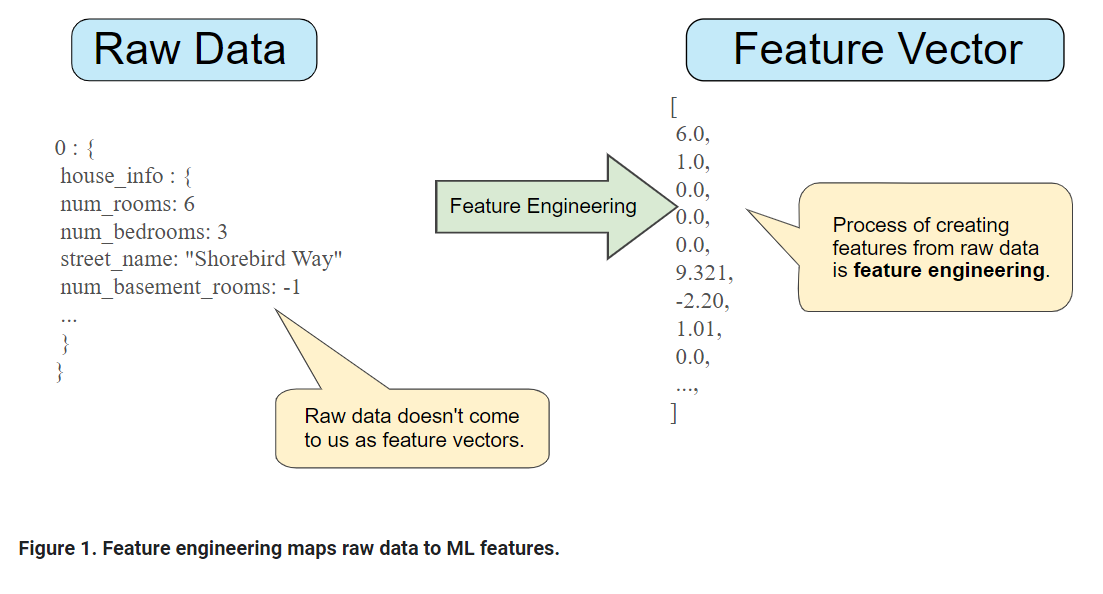
# Representation: Feature Engineering

In traditional programming, the focus is on code. In machine learning projects, the focus shifts to representation. That is, one way developers hone a model is by adding and improving its features.

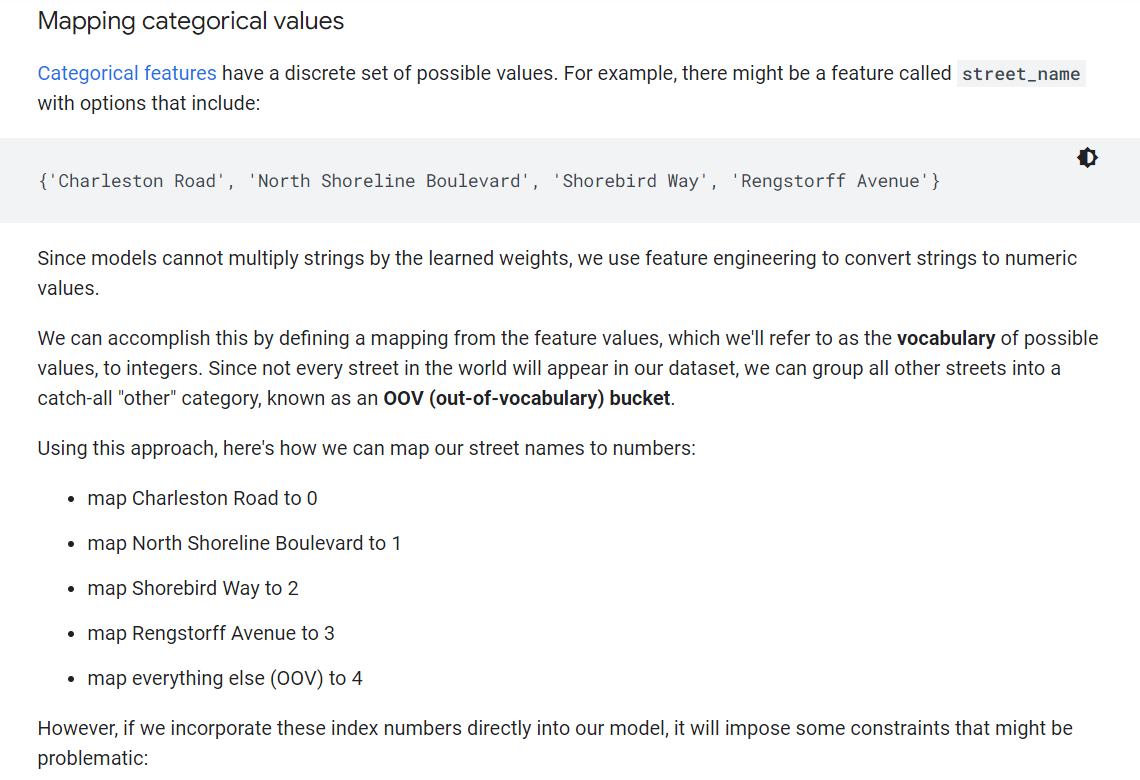
## Mapping Raw Data to Features

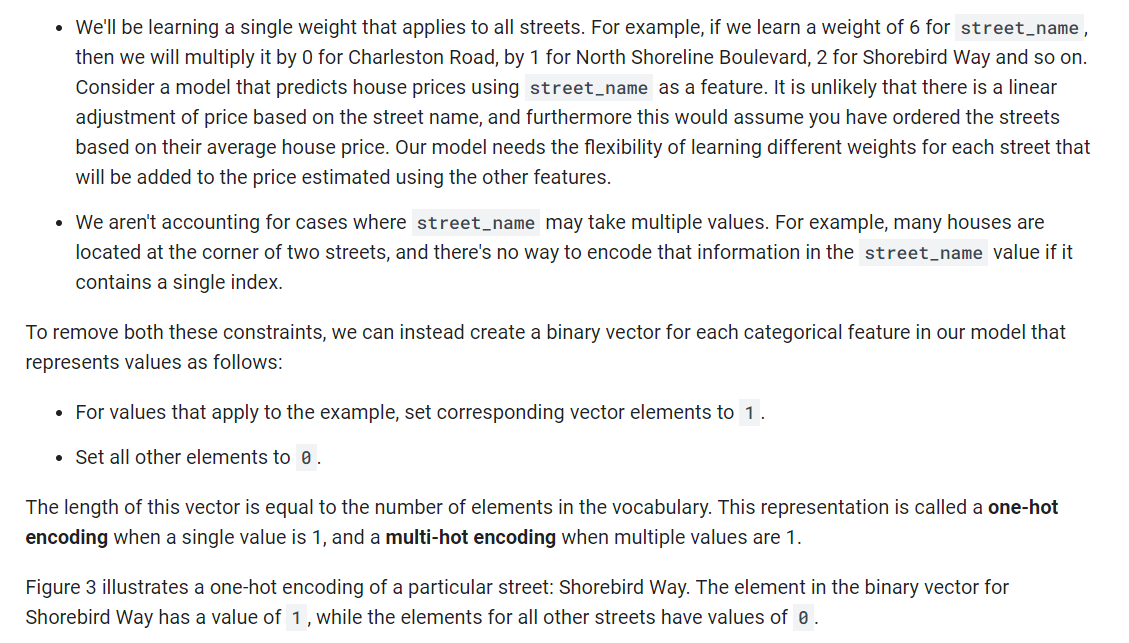
The left side of Figure 1 illustrates raw data from an input data source; the right side illustrates a **feature vector**, which is the set of floating-point values comprising the examples in your data set. **Feature engineering** means transforming raw data into a feature vector. Expect to spend significant time doing feature engineering.

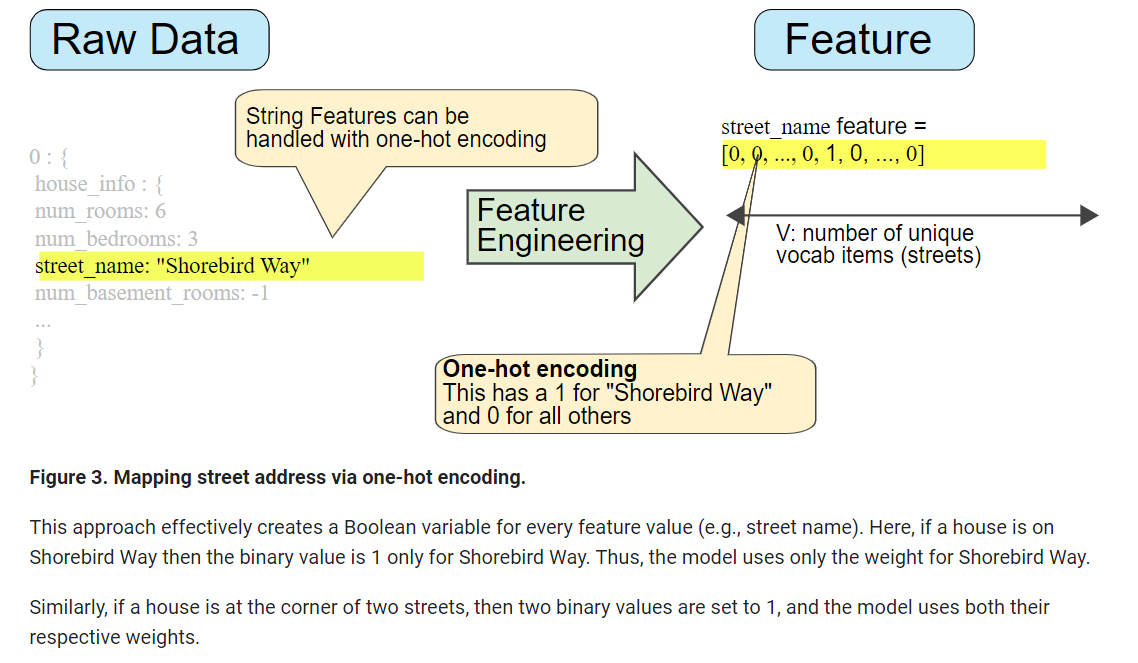
Many machine learning models must represent the features as real-numbered vectors since the feature values must be multiplied by the model weights.











### Sparse Representation

Suppose that you had 1,000,000 different street names in your data set that you wanted to include as values for street\_name. Explicitly creating a binary vector of 1,000,000 elements where only 1 or 2 elements are true is a very inefficient representation in terms of both storage and computation time when processing these vectors. In this situation, a common approach is to use a [sparse representation](https://developers.google.com/machine-learning/glossary#sparse_representation) in which only nonzero values are stored. In sparse representations, an independent model weight is still learned for each feature value, as described above.

Representation: Qualities of Good Features

We've explored ways to map raw data into suitable feature vectors, but that's only part of the work. We must now explore what kinds of values actually make good features within those feature vectors.

### Avoid rarely used discrete feature values

Good feature values should appear more than 5 or so times in a data set. Doing so enables a model to learn how this feature value relates to the label. That is, having many examples with the same discrete value gives the model a chance to see the feature in different settings, and in turn, determine when it's a good predictor for the label. For example, a house\_type feature would likely contain many examples in which its value was victorian:

✔house\_type: victorian

Conversely, if a feature's value appears only once or very rarely, the model can't make predictions based on that feature. For example, unique\_house\_id is a bad feature because each value would be used only once, so the model couldn't learn anything from it:

✘unique\_house\_id: 8SK982ZZ1242Z

### Prefer clear and obvious meanings

Each feature should have a clear and obvious meaning to anyone on the project. For example, the following good feature is clearly named and the value makes sense with respect to the name:

✔house\_age\_years: 27

Conversely, the meaning of the following feature value is pretty much indecipherable to anyone but the engineer who created it:

✘house\_age: 851472000

In some cases, noisy data (rather than bad engineering choices) causes unclear values. For example, the following user\_age\_years came from a source that didn't check for appropriate values:

✘user\_age\_years: 277

### Don't mix "magic" values with actual data

Good floating-point features don't contain peculiar out-of-range discontinuities or "magic" values. For example, suppose a feature holds a floating-point value between 0 and 1. So, values like the following are fine:

✔quality\_rating: 0.82

quality\_rating: 0.37

However, if a user didn't enter a quality\_rating, perhaps the data set represented its absence with a magic value like the following:

✘quality\_rating: -1

To explicitly mark magic values, create a Boolean feature that indicates whether or not a quality\_rating was supplied. Give this Boolean feature a name like is\_quality\_rating\_defined.

In the original feature, replace the magic values as follows:

* For variables that take a finite set of values (discrete variables), add a new value to the set and use it to signify that the feature value is missing.
* For continuous variables, ensure missing values do not affect the model by using the mean value of the feature's data.

### Account for upstream instability

The definition of a feature shouldn't change over time. For example, the following value is useful because the city name probably won't change. (Note that we'll still need to convert a string like "br/sao\_paulo" to a one-hot vector.)

✔city\_id: "br/sao\_paulo"

But gathering a value inferred by another model carries additional costs. Perhaps the value "219" currently represents Sao Paulo, but that representation could easily change on a future run of the other model:

✘inferred\_city\_cluster: "219"

# Representation: Cleaning Data

Apple trees produce some mixture of great fruit and wormy messes. Yet the apples in high-end grocery stores display 100% perfect fruit. Between orchard and grocery, someone spends significant time removing the bad apples or throwing a little wax on the salvageable ones. As an ML engineer, you'll spend enormous amounts of your time tossing out bad examples and cleaning up the salvageable ones. Even a few "bad apples" can spoil a large data set.

### Scaling feature values

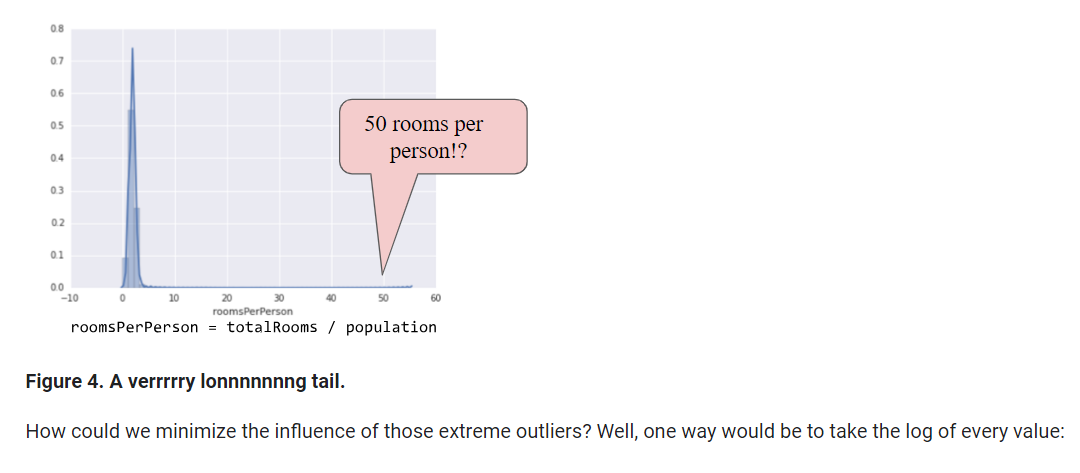
**Scaling** means converting floating-point feature values from their natural range (for example, 100 to 900) into a standard range (for example, 0 to 1 or -1 to +1). If a feature set consists of only a single feature, then scaling provides little to no practical benefit. If, however, a feature set consists of multiple features, then feature scaling provides the following benefits:

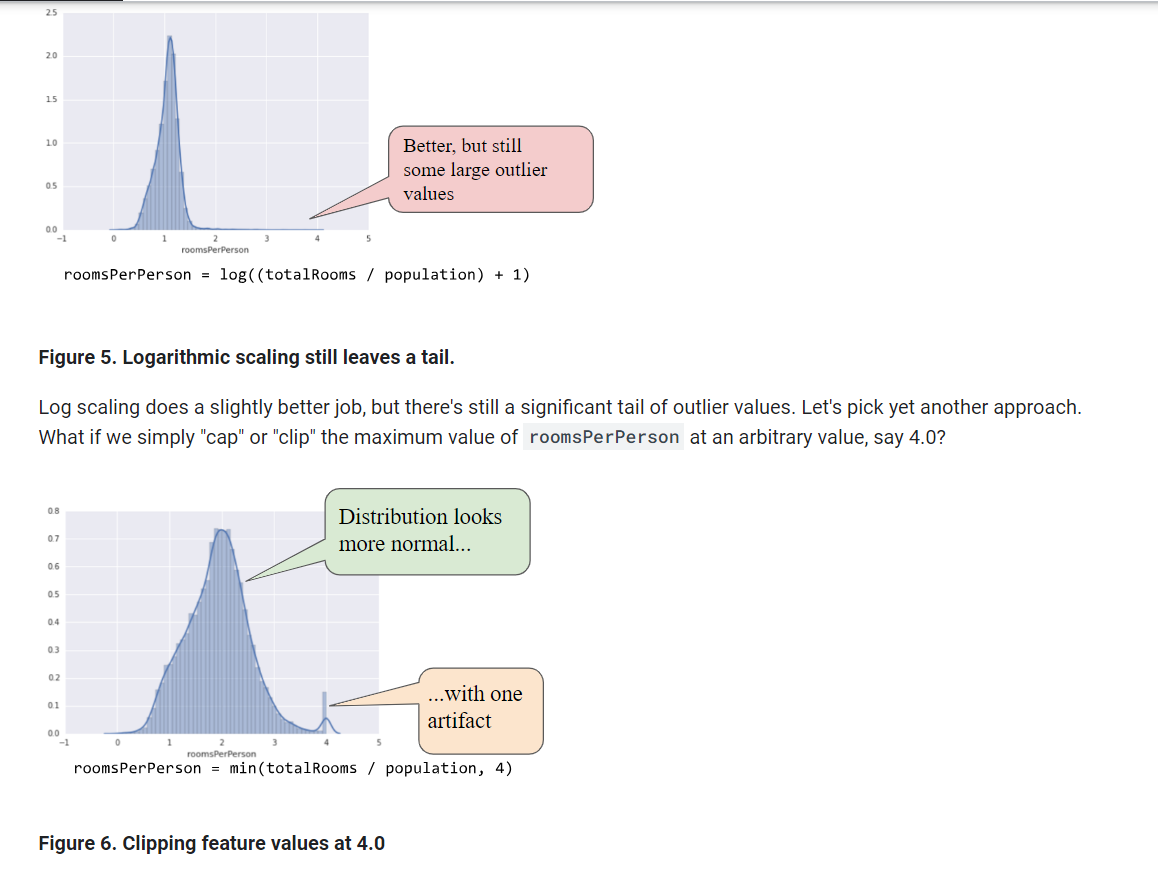
* Helps gradient descent converge more quickly.
* Helps avoid the "NaN trap," in which one number in the model becomes a [NaN](https://wikipedia.org/wiki/NaN) (e.g., when a value exceeds the floating-point precision limit during training), and—due to math operations—every other number in the model also eventually becomes a NaN.
* Helps the model learn appropriate weights for each feature. Without feature scaling, the model will pay too much attention to the features having a wider range.

You don't have to give every floating-point feature exactly the same scale. Nothing terrible will happen if Feature A is scaled from -1 to +1 while Feature B is scaled from -3 to +3. However, your model will react poorly if Feature B is scaled from 5000 to 100000.

### Handling extreme outliers

The following plot represents a feature called roomsPerPerson from the [California Housing data set](https://developers.google.com/machine-learning/crash-course/california-housing-data-description). The value of roomsPerPerson was calculated by dividing the total number of rooms for an area by the population for that area. The plot shows that the vast majority of areas in California have one or two rooms per person. But take a look along the x-axis.

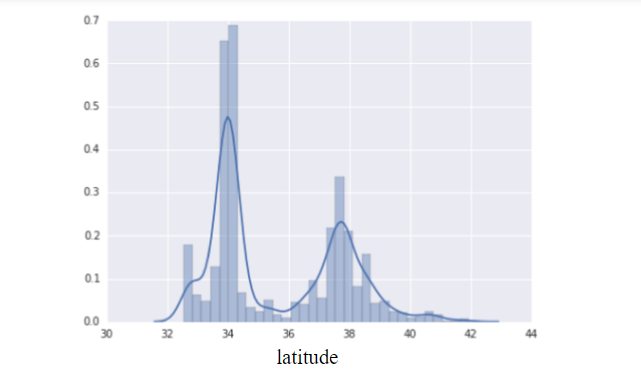


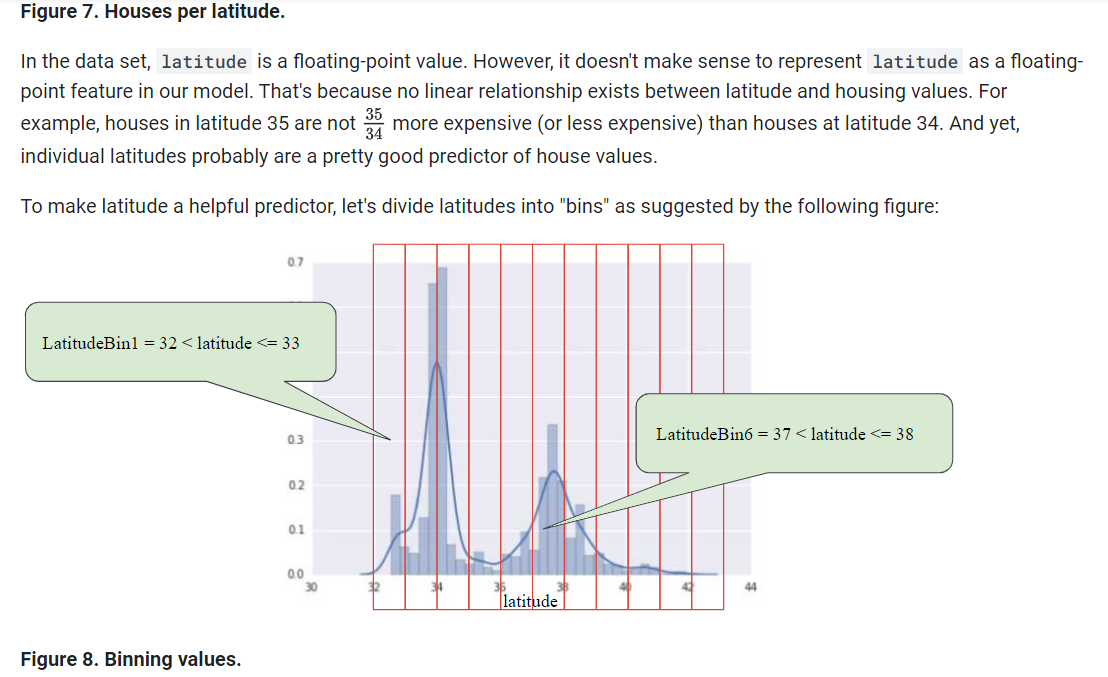


Clipping the feature value at 4.0 doesn't mean that we ignore all values greater than 4.0. Rather, it means that all values that were greater than 4.0 now become 4.0. This explains the funny hill at 4.0. Despite that hill, the scaled feature set is now more useful than the original data.

### Binning

The following plot shows the relative prevalence of houses at different latitudes in California. Notice the clustering—Los Angeles is about at latitude 34 and San Francisco is roughly at latitude 38.





**Figure 8. Binning values.**

Instead of having one floating-point feature, we now have 11 distinct boolean features (LatitudeBin1, LatitudeBin2, ..., LatitudeBin11). Having 11 separate features is somewhat inelegant, so let's unite them into a single 11-element vector. Doing so will enable us to represent latitude 37.4 as follows:

[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

Thanks to binning, our model can now learn completely different weights for each latitude.

### Scrubbing

Until now, we've assumed that all the data used for training and testing was trustworthy. In real-life, many examples in data sets are unreliable due to one or more of the following:

* **Omitted values.** For instance, a person forgot to enter a value for a house's age.
* **Duplicate examples.** For example, a server mistakenly uploaded the same logs twice.
* **Bad labels.** For instance, a person mislabeled a picture of an oak tree as a maple.
* **Bad feature values.** For example, someone typed in an extra digit, or a thermometer was left out in the sun.

Once detected, you typically "fix" bad examples by removing them from the data set. To detect omitted values or duplicated examples, you can write a simple program. Detecting bad feature values or labels can be far trickier.

In addition to detecting bad individual examples, you must also detect bad data in the aggregate. Histograms are a great mechanism for visualizing your data in the aggregate. In addition, getting statistics like the following can help:

* Maximum and minimum
* Mean and median
* Standard deviation

Consider generating lists of the most common values for discrete features. For example, do the number of examples with country:uk match the number you expect. Should language:jp really be the most common language in your data set?

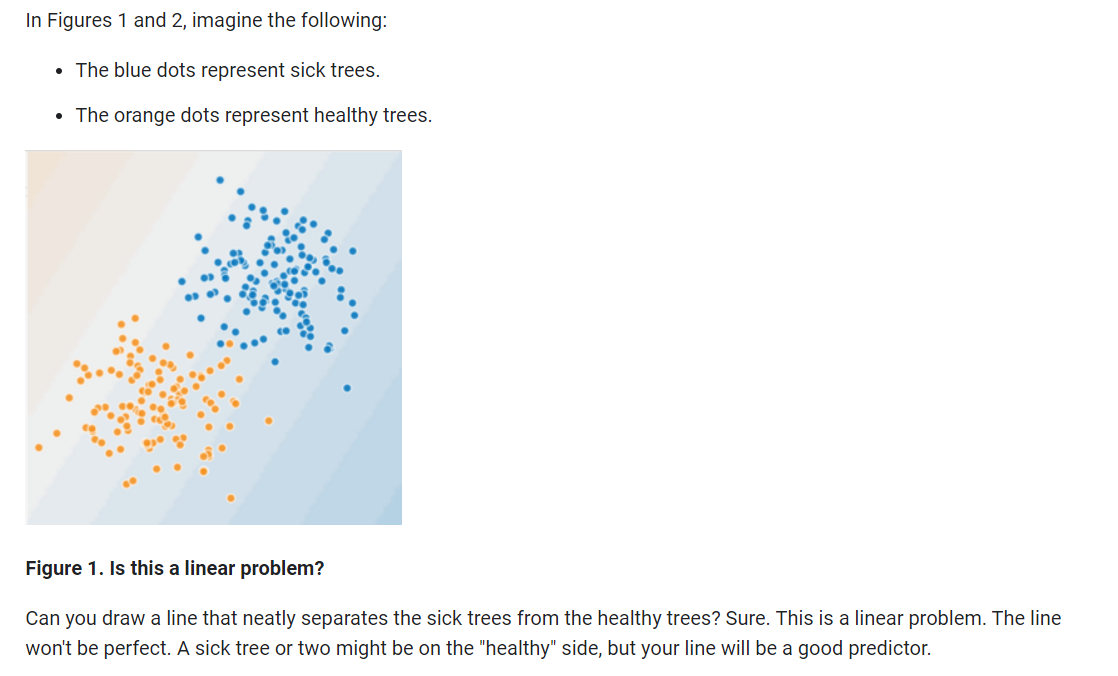
### Know your data

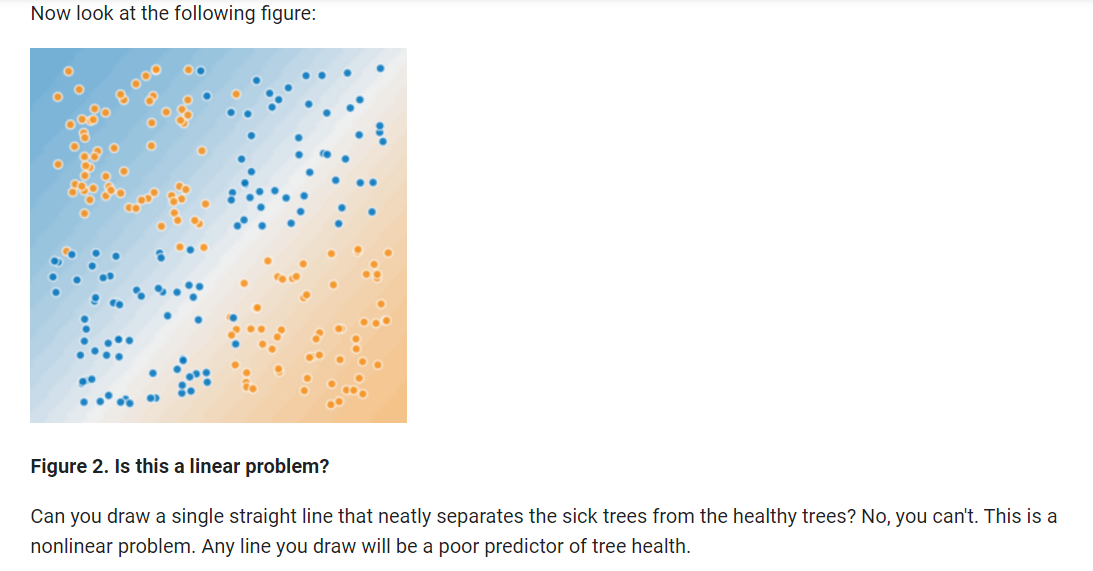
Follow these rules:

* Keep in mind what you think your data should look like.
* Verify that the data meets these expectations (or that you can explain why it doesn’t).
* Double-check that the training data agrees with other sources (for example, dashboards).

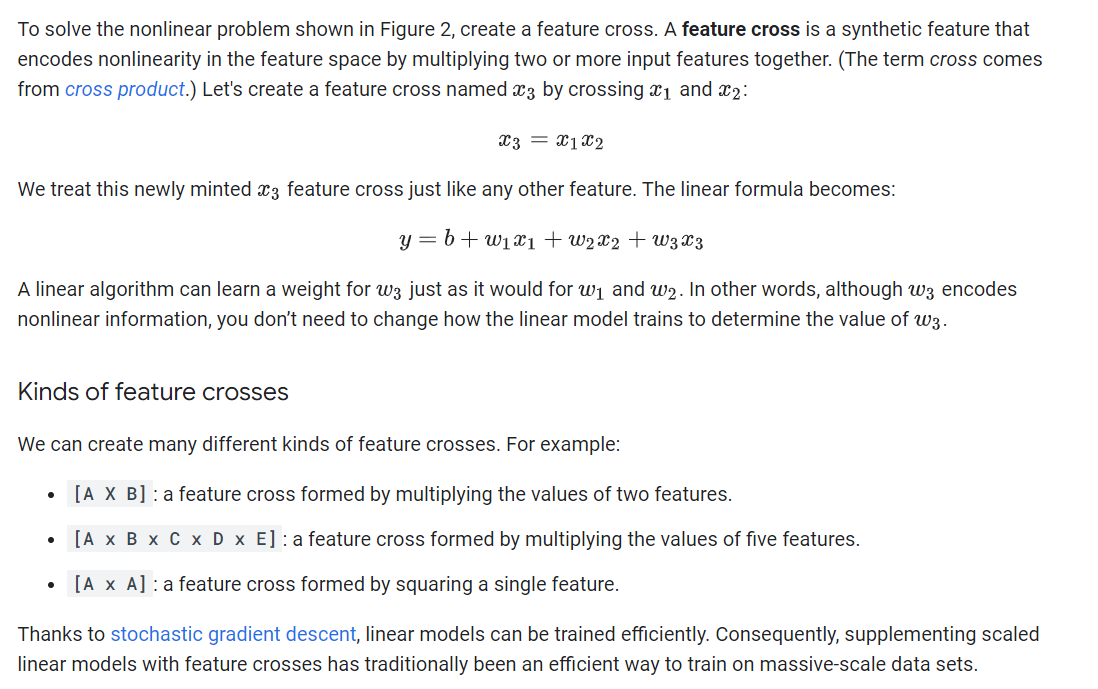
Treat your data with all the care that you would treat any mission-critical code. Good ML relies on good data.

Feature Crosses: Encoding Nonlinearity









Feature Crosses: Crossing One-Hot Vectors

So far, we've focused on feature-crossing two individual floating-point features. In practice, machine learning models seldom cross continuous features. However, machine learning models do frequently cross one-hot feature vectors. Think of feature crosses of one-hot feature vectors as logical conjunctions. For example, suppose we have two features: country and language. A one-hot encoding of each generates vectors with binary features that can be interpreted as country=USA, country=France or language=English, language=Spanish. Then, if you do a feature cross of these one-hot encodings, you get binary features that can be interpreted as logical conjunctions, such as:

  country:usa AND language:spanish

As another example, suppose you bin latitude and longitude, producing separate one-hot five-element feature vectors. For instance, a given latitude and longitude could be represented as follows:

binned\_latitude = [0, 0, 0, 1, 0]

binned\_longitude = [0, 1, 0, 0, 0]

Suppose you create a feature cross of these two feature vectors:

binned\_latitude X binned\_longitude

This feature cross is a 25-element one-hot vector (24 zeroes and 1 one). The single 1 in the cross identifies a particular conjunction of latitude and longitude. Your model can then learn particular associations about that conjunction.

Suppose we bin latitude and longitude much more coarsely, as follows:

binned\_latitude(lat) = [

0 < lat <= 10

10 < lat <= 20

20 < lat <= 30

]

binned\_longitude(lon) = [

0 < lon <= 15

15 < lon <= 30

]

Creating a feature cross of those coarse bins leads to synthetic feature having the following meanings:

binned\_latitude\_X\_longitude(lat, lon) = [

0 < lat <= 10 AND 0 < lon <= 15

0 < lat <= 10 AND 15 < lon <= 30

10 < lat <= 20 AND 0 < lon <= 15

10 < lat <= 20 AND 15 < lon <= 30

20 < lat <= 30 AND 0 < lon <= 15

20 < lat <= 30 AND 15 < lon <= 30

]

Now suppose our model needs to predict how satisfied dog owners will be with dogs based on two features:

* Behavior type (barking, crying, snuggling, etc.)
* Time of day

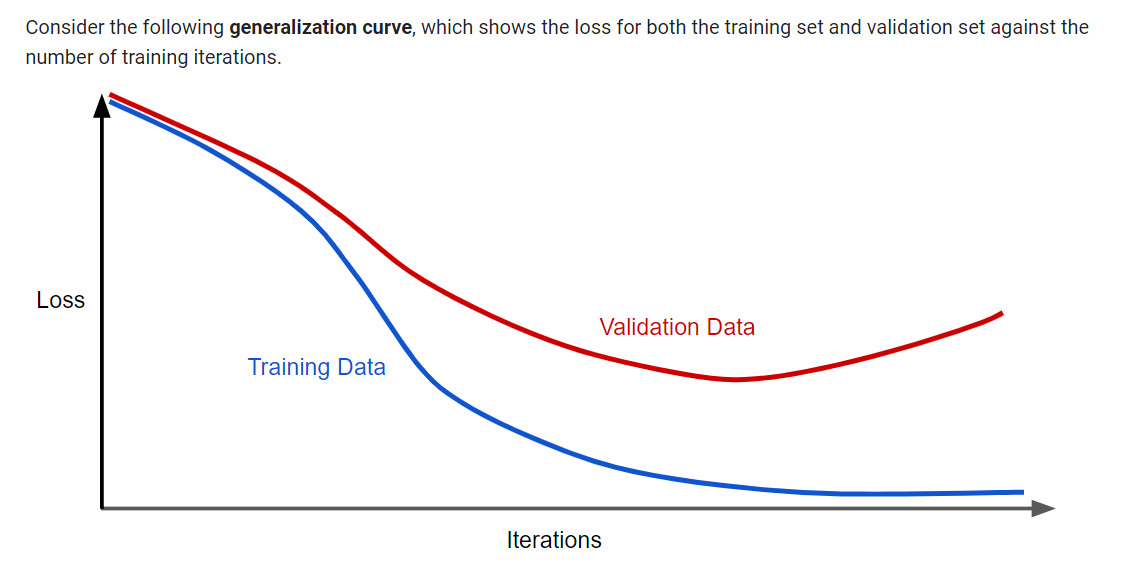
If we build a feature cross from both these features:

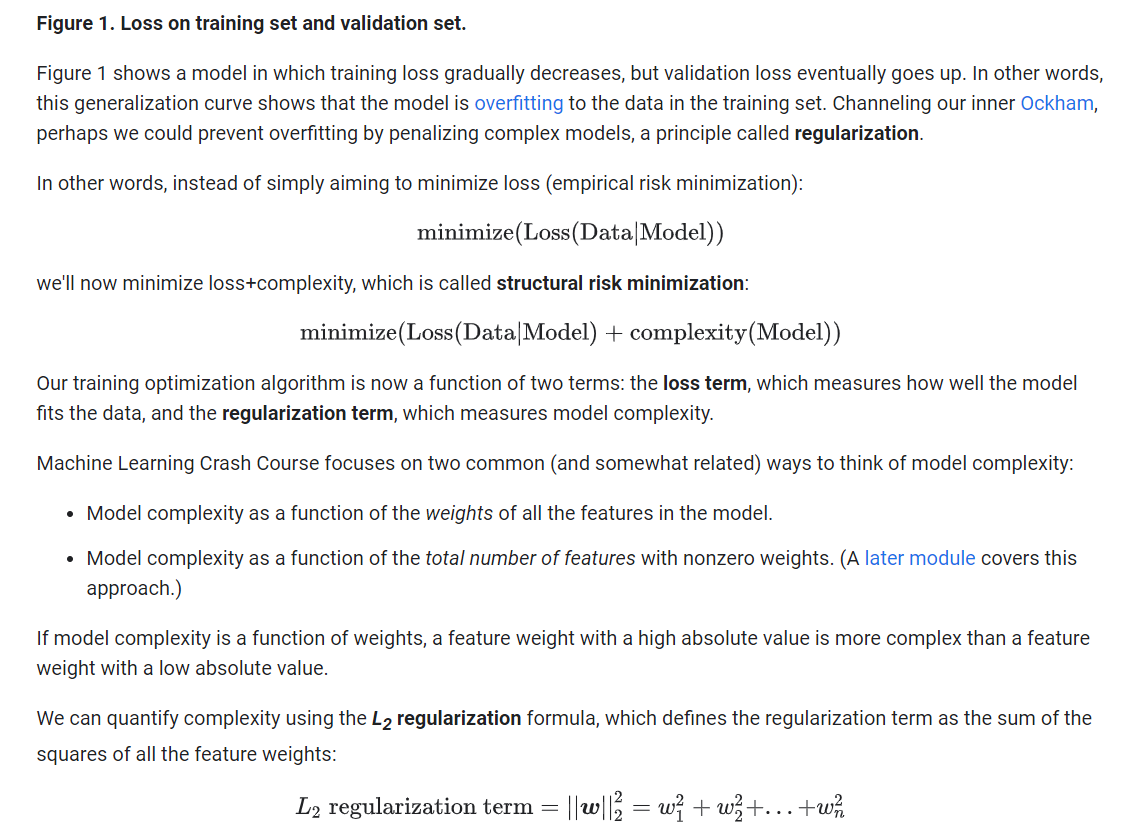
[behavior type X time of day]

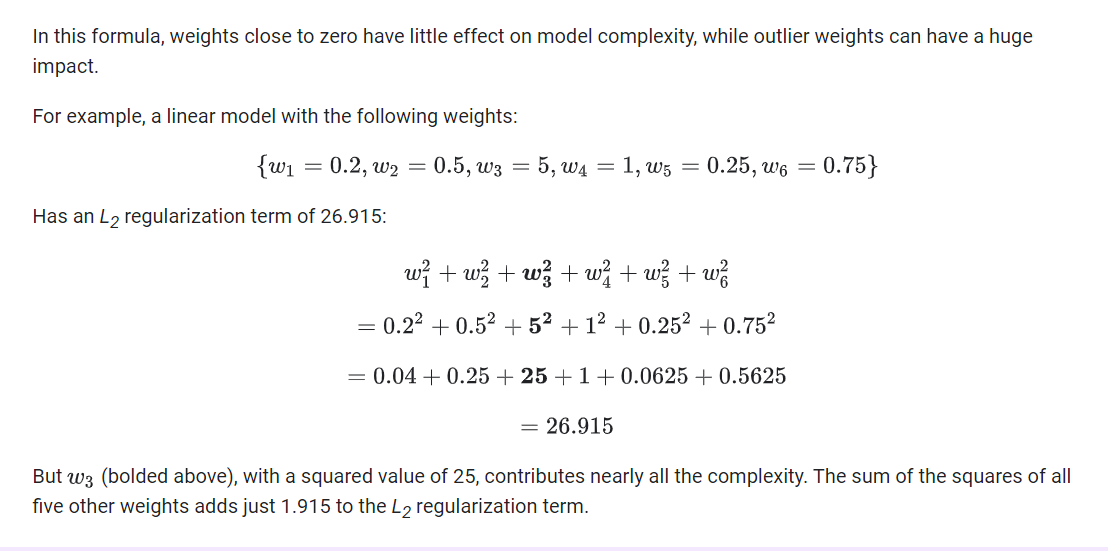
then we'll end up with vastly more predictive ability than either feature on its own. For example, if a dog cries (happily) at 5:00 pm when the owner returns from work will likely be a great positive predictor of owner satisfaction. Crying (miserably, perhaps) at 3:00 am when the owner was sleeping soundly will likely be a strong negative predictor of owner satisfaction.

Linear learners scale well to massive data. Using feature crosses on massive data sets is one efficient strategy for learning highly complex models. [Neural networks](https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks) provide another strategy.

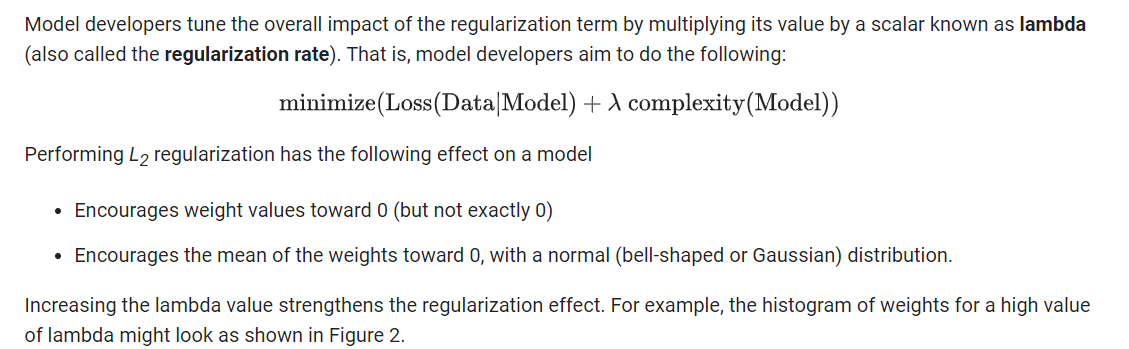
Regularization for Simplicity: L₂ Regularization

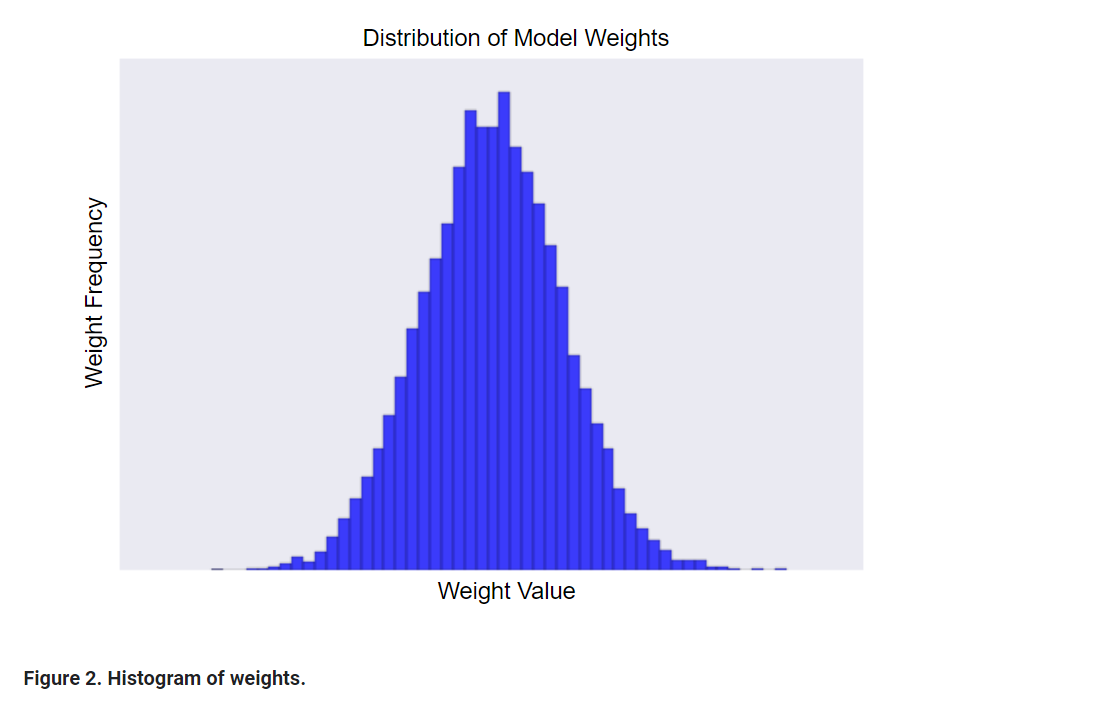


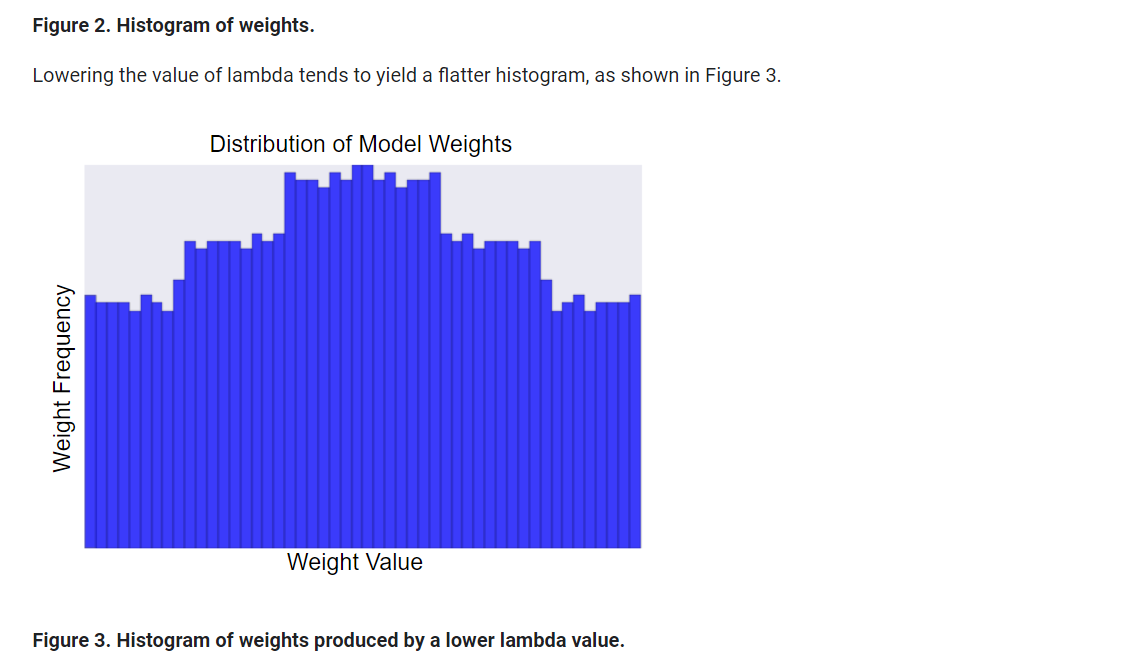




Regularization for Simplicity: Lambda







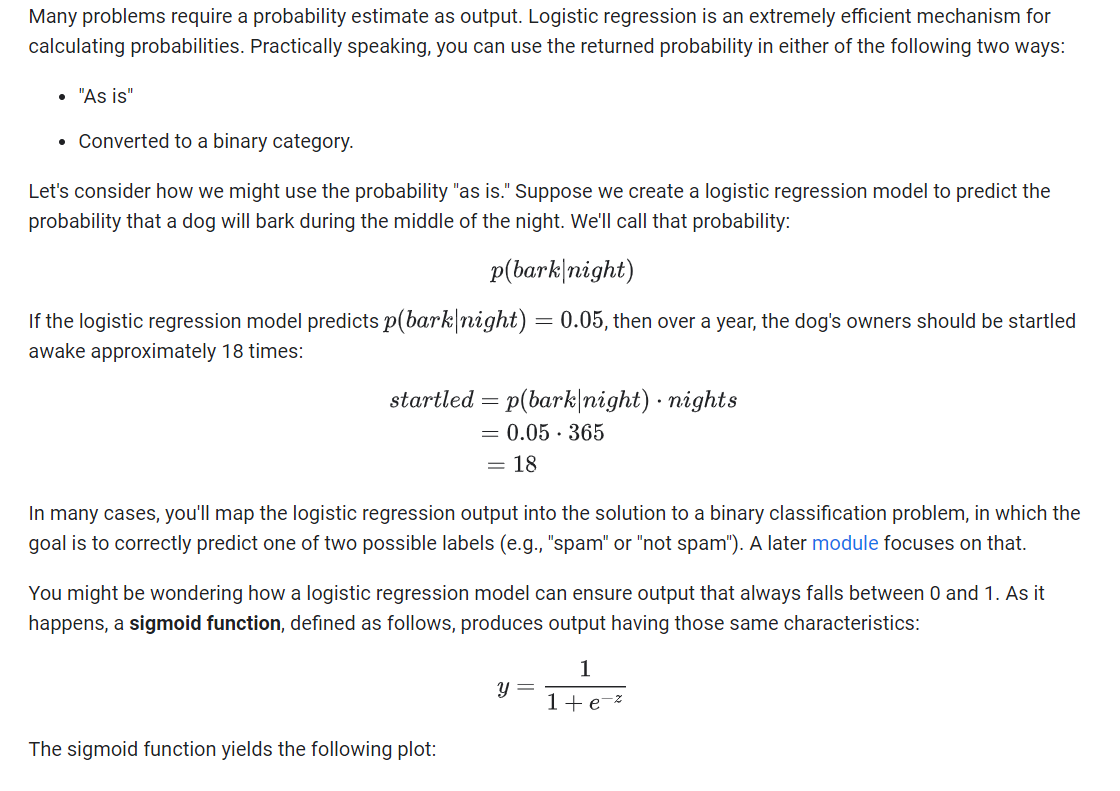
When choosing a lambda value, the goal is to strike the right balance between simplicity and training-data fit:

* If your lambda value is too high, your model will be simple, but you run the risk of *underfitting* your data. Your model won't learn enough about the training data to make useful predictions.
* If your lambda value is too low, your model will be more complex, and you run the risk of *overfitting* your data. Your model will learn too much about the particularities of the training data, and won't be able to generalize to new data.

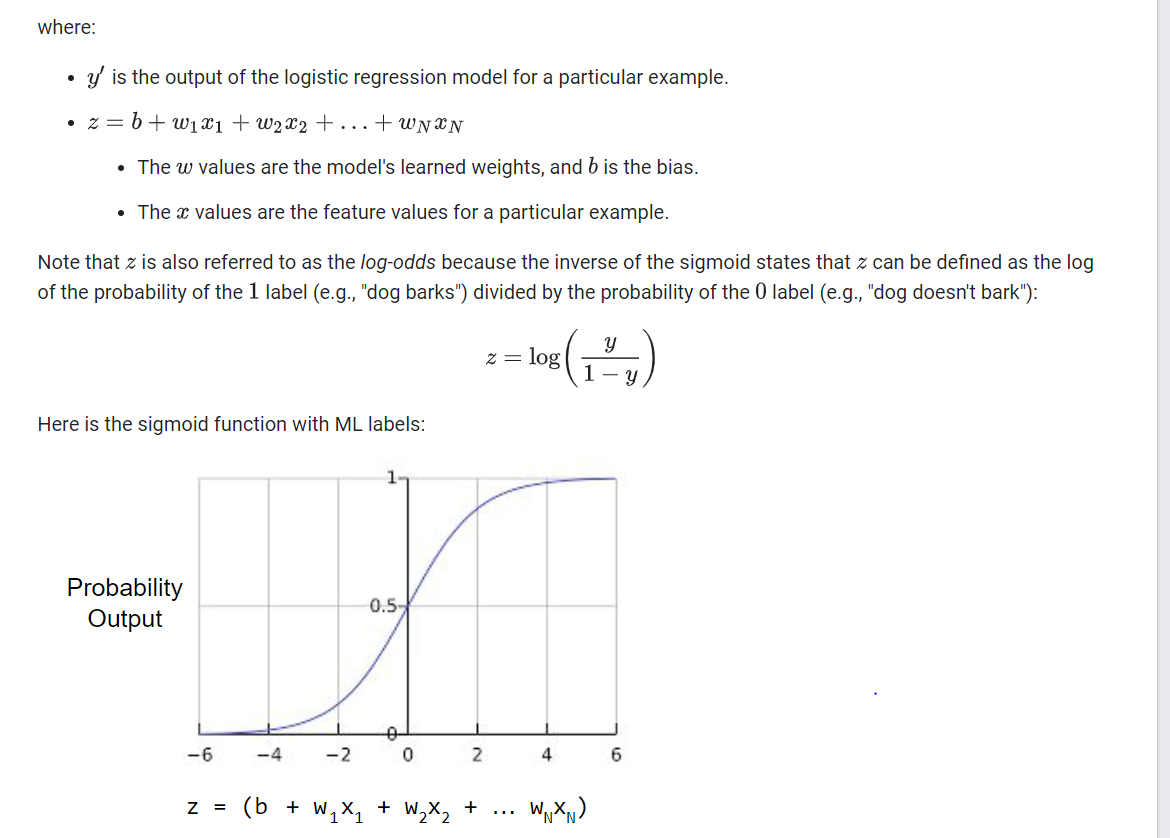
**Note:** Setting lambda to zero removes regularization completely. In this case, training focuses exclusively on minimizing loss, which poses the highest possible overfitting risk.

The ideal value of lambda produces a model that generalizes well to new, previously unseen data. Unfortunately, that ideal value of lambda is data-dependent, so you'll need to do some tuning.

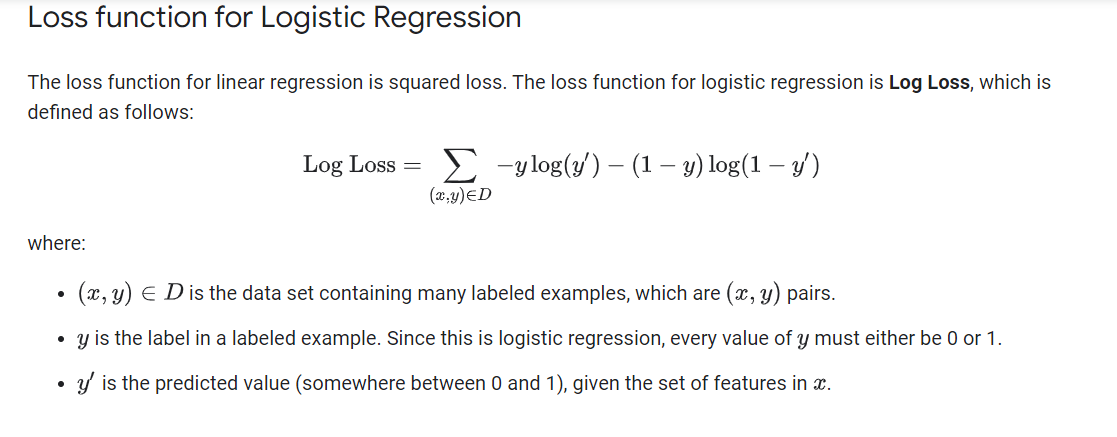
Logistic Regression: Calculating a Probability







Logistic Regression: Loss and Regularization



## Regularization in Logistic Regression

[Regularization](https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/video-lecture) is extremely important in logistic regression modeling. Without regularization, the asymptotic nature of logistic regression would keep driving loss towards 0 in high dimensions. Consequently, most logistic regression models use one of the following two strategies to dampen model complexity:

* L2 regularization.
* Early stopping, that is, limiting the number of training steps or the learning rate.

(We'll discuss a third strategy—L1 regularization—in a [later module](https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/video-lecture).)

Imagine that you assign a unique id to each example, and map each id to its own feature. If you don't specify a regularization function, the model will become completely overfit. That's because the model would try to drive loss to zero on all examples and never get there, driving the weights for each indicator feature to +infinity or -infinity. This can happen in high dimensional data with feature crosses, when there’s a huge mass of rare crosses that happen only on one example each.

Fortunately, using L2 or early stopping will prevent this problem.

Classification: Thresholding

Logistic regression returns a probability. You can use the returned probability "as is" (for example, the probability that the user will click on this ad is 0.00023) or convert the returned probability to a binary value (for example, this email is spam).

A logistic regression model that returns 0.9995 for a particular email message is predicting that it is very likely to be spam. Conversely, another email message with a prediction score of 0.0003 on that same logistic regression model is very likely not spam. However, what about an email message with a prediction score of 0.6? In order to map a logistic regression value to a binary category, you must define a **classification threshold** (also called the **decision threshold**). A value above that threshold indicates "spam"; a value below indicates "not spam." It is tempting to assume that the classification threshold should always be 0.5, but thresholds are problem-dependent, and are therefore values that you must tune.

The following sections take a closer look at metrics you can use to evaluate a classification model's predictions, as well as the impact of changing the classification threshold on these predictions.

Classification: True vs. False and Positive vs. Negative

In this section, we'll define the primary building blocks of the metrics we'll use to evaluate classification models. But first, a fable:

**An Aesop's Fable: The Boy Who Cried Wolf (compressed)**

A shepherd boy gets bored tending the town's flock. To have some fun, he cries out, "Wolf!" even though no wolf is in sight. The villagers run to protect the flock, but then get really mad when they realize the boy was playing a joke on them.

[Iterate previous paragraph N times.]

One night, the shepherd boy sees a real wolf approaching the flock and calls out, "Wolf!" The villagers refuse to be fooled again and stay in their houses. The hungry wolf turns the flock into lamb chops. The town goes hungry. Panic ensues.

Let's make the following definitions:

* "Wolf" is a **positive class**.
* "No wolf" is a **negative class**.

We can summarize our "wolf-prediction" model using a 2x2 [confusion matrix](https://developers.google.com/machine-learning/glossary#confusion_matrix) that depicts all four possible outcomes:

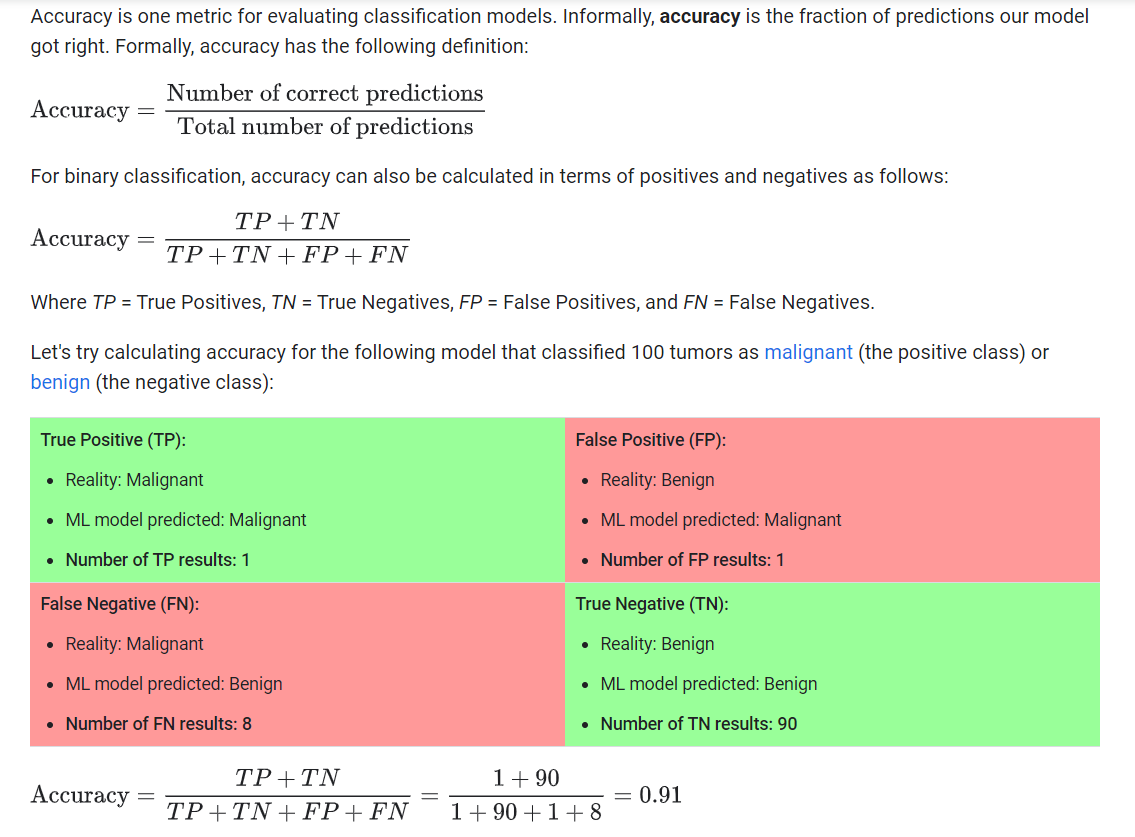
|  |  |
| --- | --- |
| True Positive (TP):   * Reality: A wolf threatened. * Shepherd said: "Wolf." * Outcome: Shepherd is a hero. | False Positive (FP):   * Reality: No wolf threatened. * Shepherd said: "Wolf." * Outcome: Villagers are angry at shepherd for waking them up. |
| False Negative (FN):   * Reality: A wolf threatened. * Shepherd said: "No wolf." * Outcome: The wolf ate all the sheep. | True Negative (TN):   * Reality: No wolf threatened. * Shepherd said: "No wolf." * Outcome: Everyone is fine. |

A **true positive** is an outcome where the model correctly predicts the positive class. Similarly, a **true negative** is an outcome where the model correctly predicts the negative class.

A **false positive** is an outcome where the model incorrectly predicts the positive class. And a **false negative** is an outcome where the model incorrectly predicts the negative class.

In the following sections, we'll look at how to evaluate classification models using metrics derived from these four outcomes.

Classification: Accuracy



Accuracy comes out to 0.91, or 91% (91 correct predictions out of 100 total examples). That means our tumor classifier is doing a great job of identifying malignancies, right?

Actually, let's do a closer analysis of positives and negatives to gain more insight into our model's performance.

Of the 100 tumor examples, 91 are benign (90 TNs and 1 FP) and 9 are malignant (1 TP and 8 FNs).

Of the 91 benign tumors, the model correctly identifies 90 as benign. That's good. However, of the 9 malignant tumors, the model only correctly identifies 1 as malignant—a terrible outcome, as 8 out of 9 malignancies go undiagnosed!

While 91% accuracy may seem good at first glance, another tumor-classifier model that always predicts benign would achieve the exact same accuracy (91/100 correct predictions) on our examples. In other words, our model is no better than one that has zero predictive ability to distinguish malignant tumors from benign tumors.

Accuracy alone doesn't tell the full story when you're working with a **class-imbalanced data set**, like this one, where there is a significant disparity between the number of positive and negative labels.

In the next section, we'll look at two better metrics for evaluating class-imbalanced problems: precision and recall.

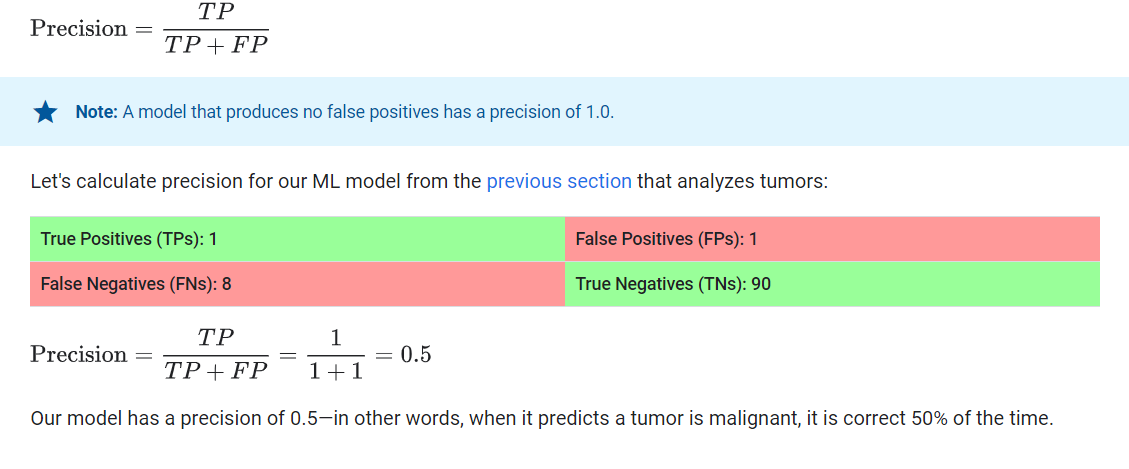
# **Classification: Precision and Recall**

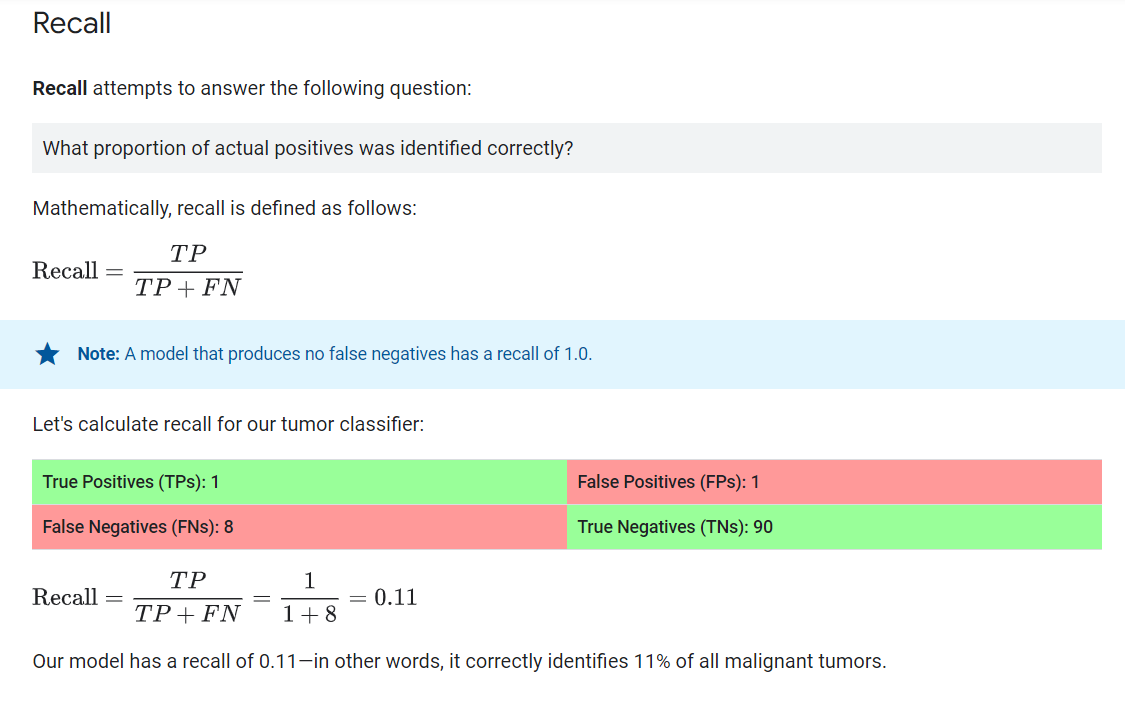
## Precision

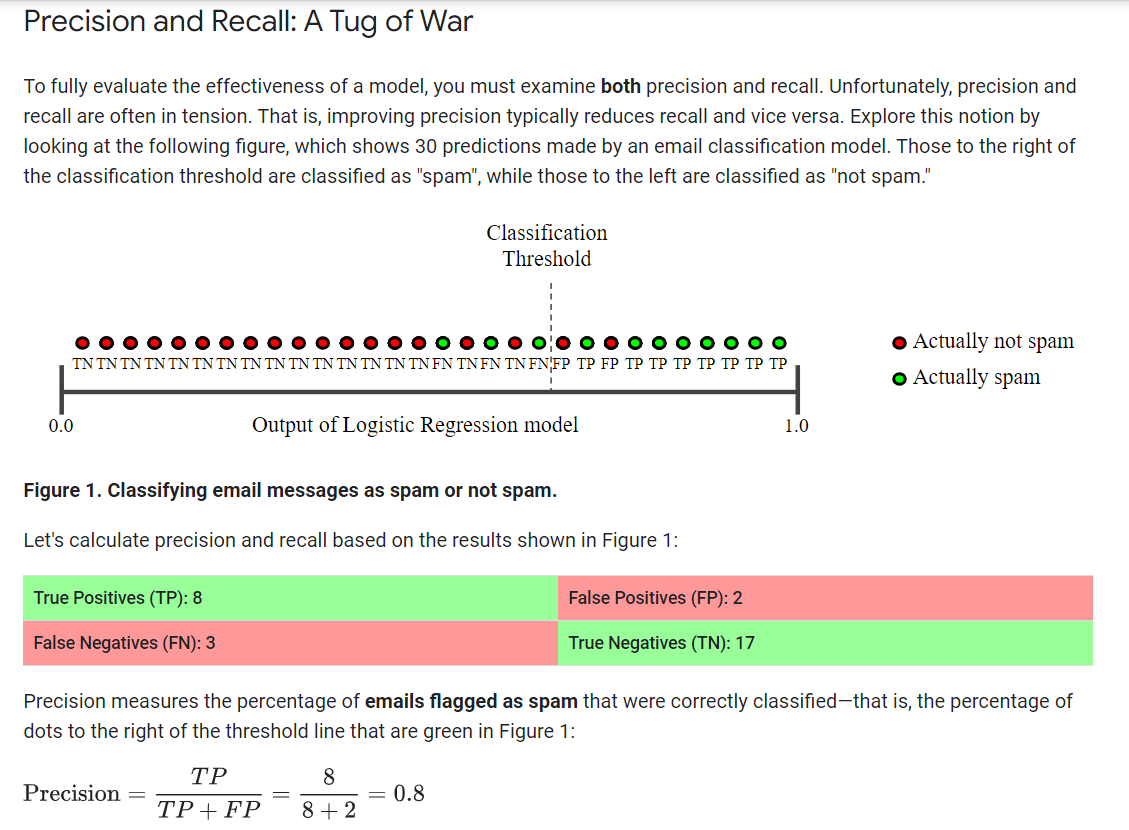
**Precision** attempts to answer the following question:

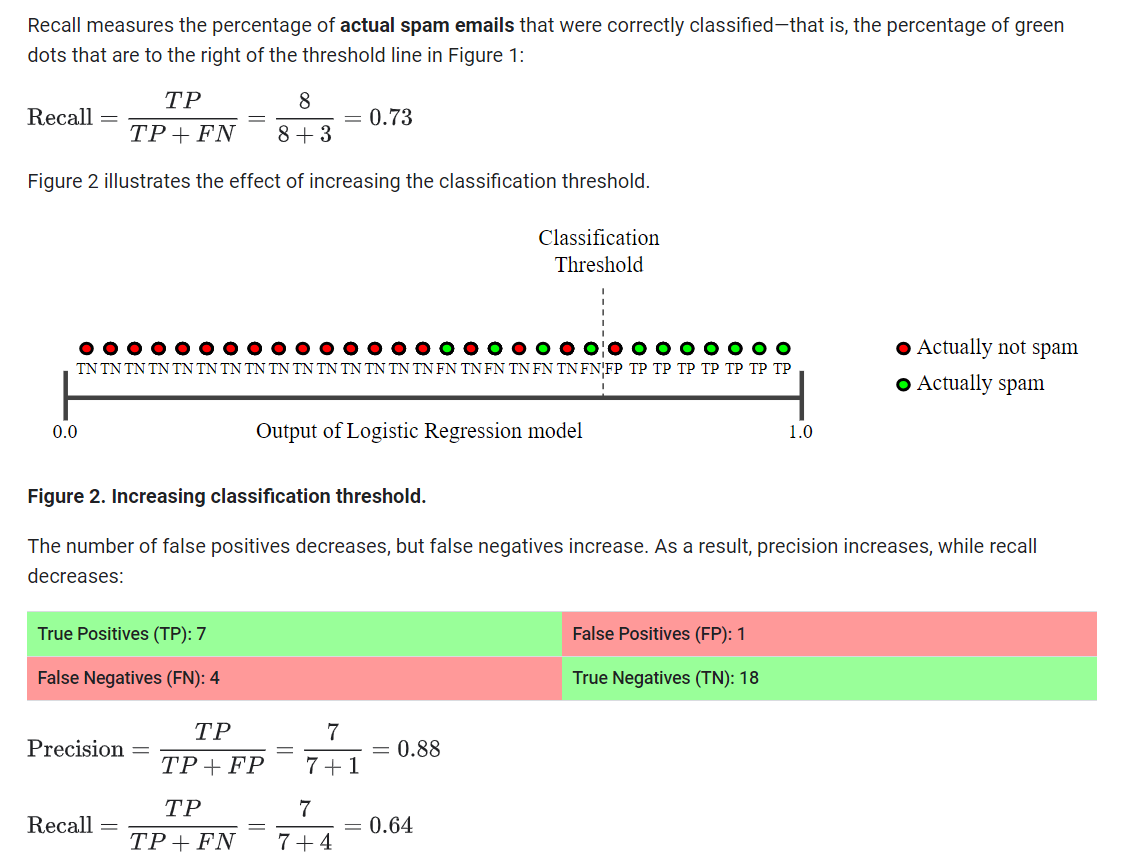
What proportion of positive identifications was actually correct?

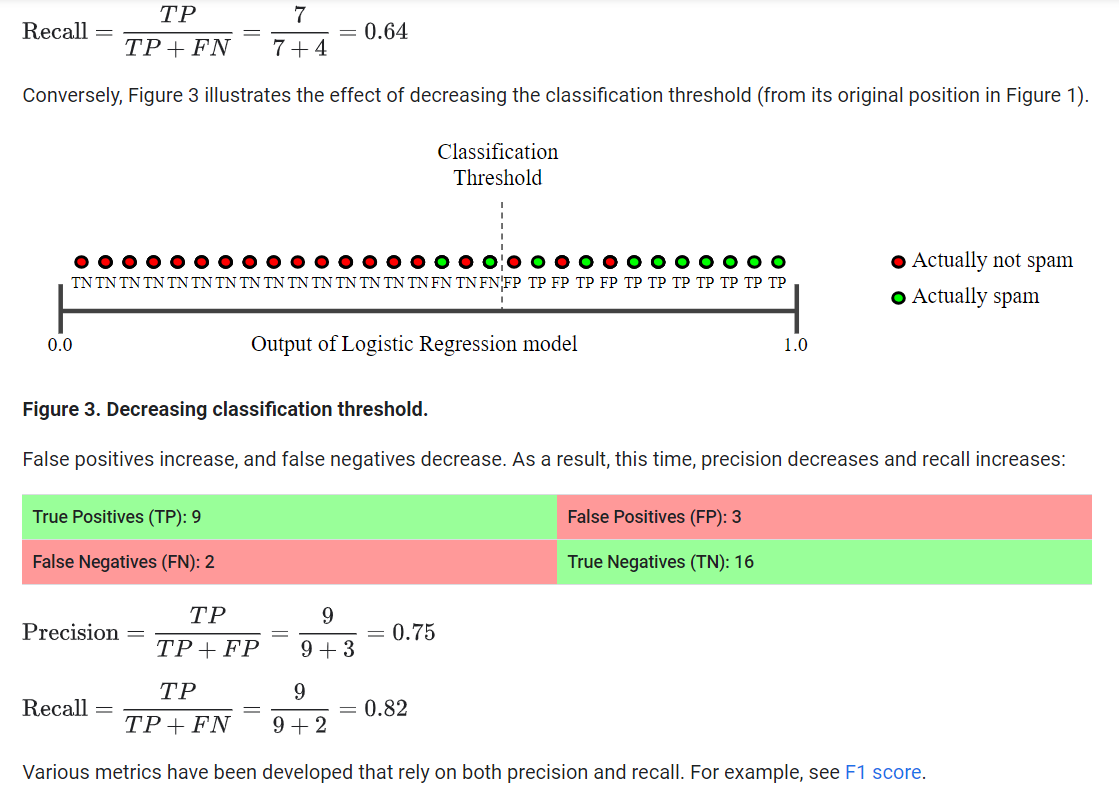
Precision is defined as follows:











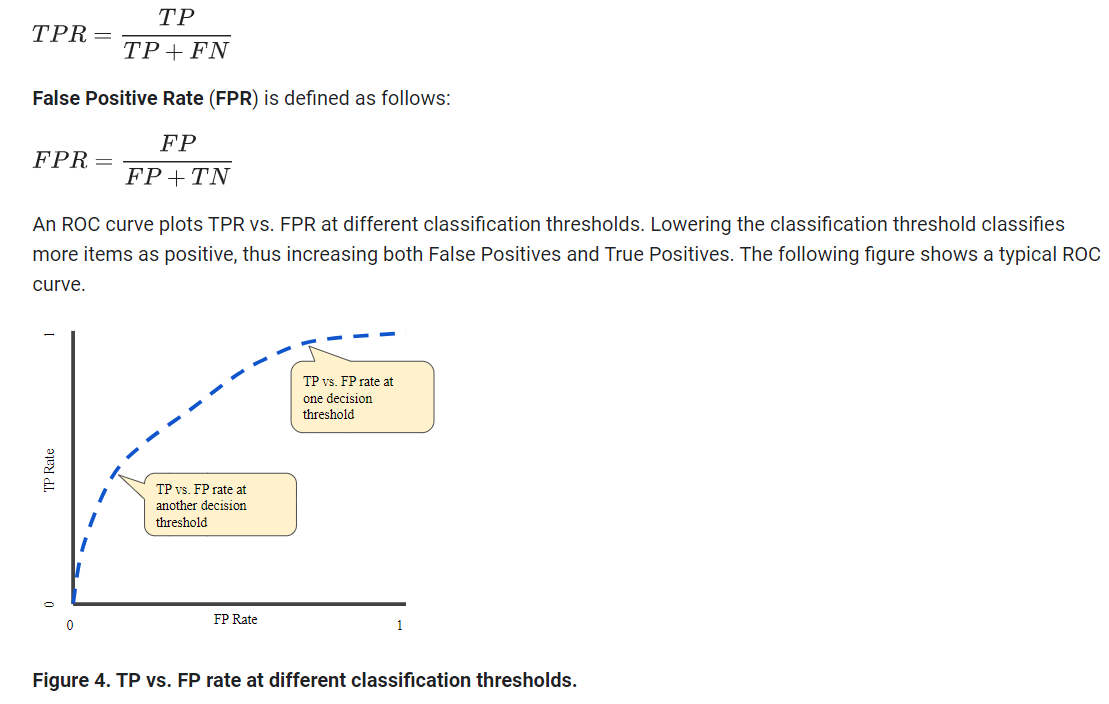
Classification: ROC Curve and AUC

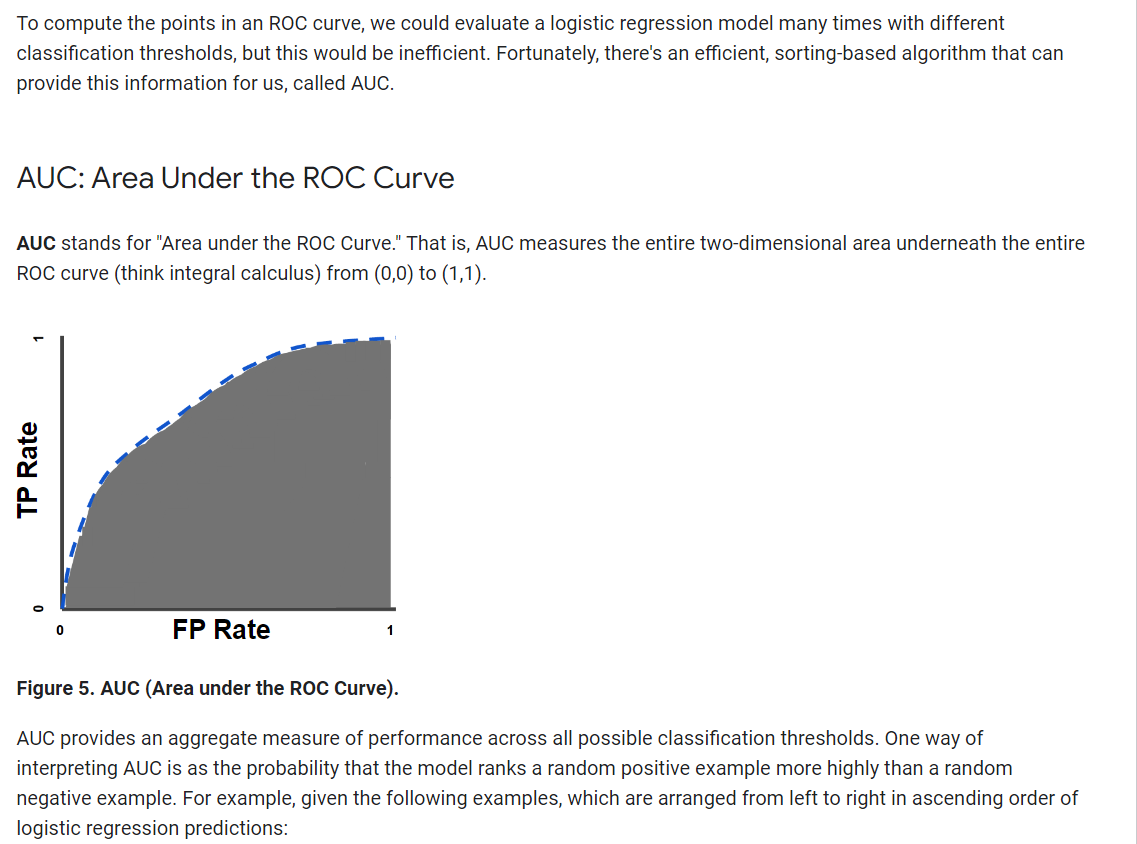
**ROC curve**

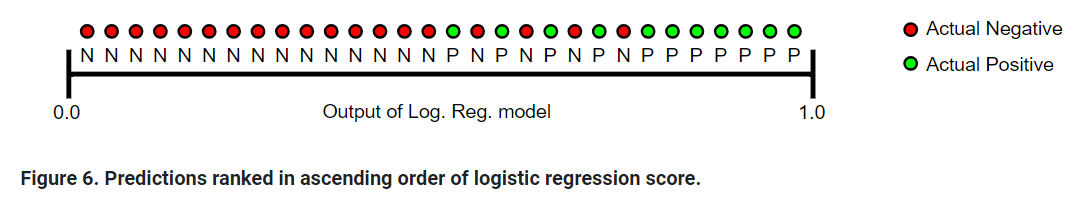
An **ROC curve** (**receiver operating characteristic curve**) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

* True Positive Rate
* False Positive Rate

**True Positive Rate** (**TPR**) is a synonym for recall and is therefore defined as follows:







AUC represents the probability that a random positive (green) example is positioned to the right of a random negative (red) example.

AUC ranges in value from 0 to 1. A model whose predictions are 100% wrong has an AUC of 0.0; one whose predictions are 100% correct has an AUC of 1.0.

AUC is desirable for the following two reasons:

* AUC is **scale-invariant**. It measures how well predictions are ranked, rather than their absolute values.
* AUC is **classification-threshold-invariant**. It measures the quality of the model's predictions irrespective of what classification threshold is chosen.

However, both these reasons come with caveats, which may limit the usefulness of AUC in certain use cases:

* **Scale invariance is not always desirable.** For example, sometimes we really do need well calibrated probability outputs, and AUC won’t tell us about that.
* **Classification-threshold invariance is not always desirable.** In cases where there are wide disparities in the cost of false negatives vs. false positives, it may be critical to minimize one type of classification error. For example, when doing email spam detection, you likely want to prioritize minimizing false positives (even if that results in a significant increase of false negatives). AUC isn't a useful metric for this type of optimization.

Classification: Prediction Bias

Logistic regression predictions should be unbiased. That is:

"average of predictions" should ≈ "average of observations"

**Prediction bias** is a quantity that measures how far apart those two averages are. That is:

prediction bias=average of predictions−average of labels in data set

**Note:** "Prediction bias" is a different quantity than [bias](https://developers.google.com/machine-learning/crash-course/descending-into-ml) (the b in wx + b).

A significant nonzero prediction bias tells you there is a bug somewhere in your model, as it indicates that the model is wrong about how frequently positive labels occur.

For example, let's say we know that on average, 1% of all emails are spam. If we don't know anything at all about a given email, we should predict that it's 1% likely to be spam. Similarly, a good spam model should predict on average that emails are 1% likely to be spam. (In other words, if we average the predicted likelihoods of each individual email being spam, the result should be 1%.) If instead, the model's average prediction is 20% likelihood of being spam, we can conclude that it exhibits prediction bias.

Possible root causes of prediction bias are:

* Incomplete feature set
* Noisy data set
* Buggy pipeline
* Biased training sample
* Overly strong regularization

You might be tempted to correct prediction bias by post-processing the learned model—that is, by adding a **calibration layer** that adjusts your model's output to reduce the prediction bias. For example, if your model has +3% bias, you could add a calibration layer that lowers the mean prediction by 3%. However, adding a calibration layer is a bad idea for the following reasons:

* You're fixing the symptom rather than the cause.
* You've built a more brittle system that you must now keep up to date.

If possible, avoid calibration layers. Projects that use calibration layers tend to become reliant on them—using calibration layers to fix all their model's sins. Ultimately, maintaining the calibration layers can become a nightmare.

**Note:** A good model will usually have near-zero bias. That said, a low prediction bias does not prove that your model is good. A really terrible model could have a zero prediction bias. For example, a model that just predicts the mean value for all examples would be a bad model, despite having zero bias.

## Bucketing and Prediction Bias

Logistic regression predicts a value between 0 and 1. However, all labeled examples are either exactly 0 (meaning, for example, "not spam") or exactly 1 (meaning, for example, "spam"). Therefore, when examining prediction bias, you cannot accurately determine the prediction bias based on only one example; you must examine the prediction bias on a "bucket" of examples. That is, prediction bias for logistic regression only makes sense when grouping enough examples together to be able to compare a predicted value (for example, 0.392) to observed values (for example, 0.394).

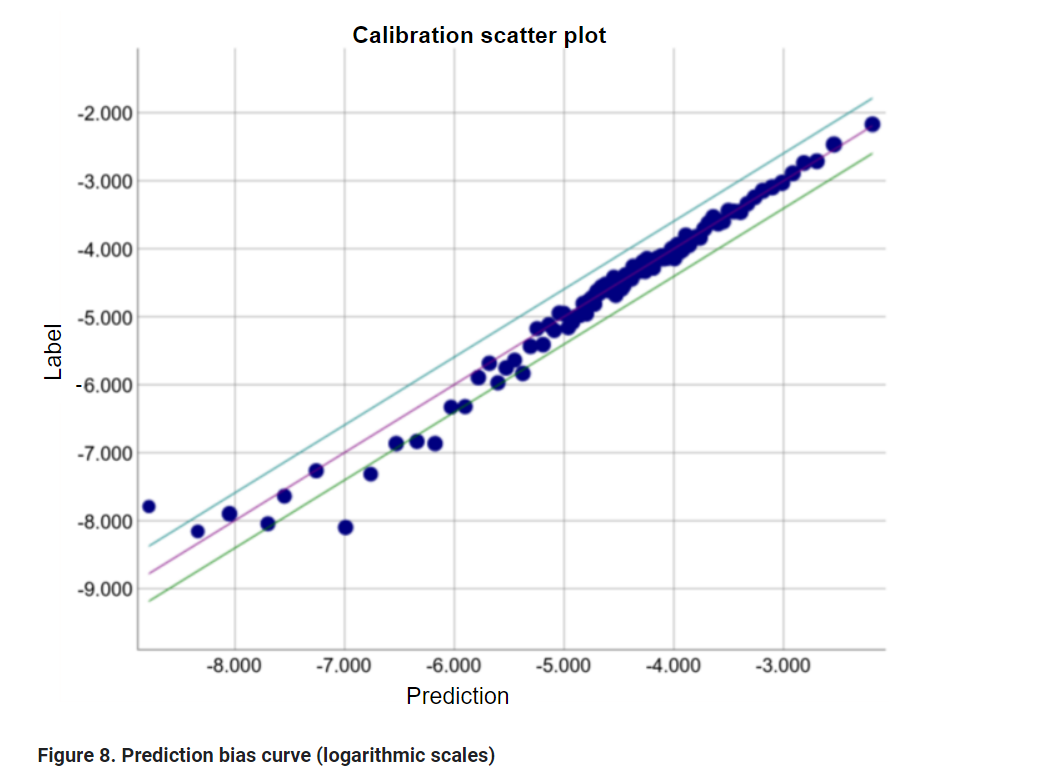
You can form buckets in the following ways:

* Linearly breaking up the target predictions.
* Forming quantiles.

Consider the following calibration plot from a particular model. Each dot represents a bucket of 1,000 values. The axes have the following meanings:

* The x-axis represents the average of values the model predicted for that bucket.
* The y-axis represents the actual average of values in the data set for that bucket.

Both axes are logarithmic scales.



Why are the predictions so poor for only part of the model? Here are a few possibilities:

* The training set doesn't adequately represent certain subsets of the data space.
* Some subsets of the data set are noisier than others.
* The model is overly [regularized](https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/video-lecture). (Consider reducing the value of [lambda](https://developers.google.com/machine-learning/glossary#lambda).)

# Regularization for Sparsity: L₁ Regularization

Sparse vectors often contain many dimensions. Creating a [feature cross](https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture) results in even more dimensions. Given such high-dimensional feature vectors, model size may become huge and require huge amounts of RAM.

In a high-dimensional sparse vector, it would be nice to encourage weights to drop to exactly 0 where possible. A weight of exactly 0 essentially removes the corresponding feature from the model. Zeroing out features will save RAM and may reduce noise in the model.

For example, consider a housing data set that covers not just California but the entire globe. Bucketing global latitude at the minute level (60 minutes per degree) gives about 10,000 dimensions in a sparse encoding; global longitude at the minute level gives about 20,000 dimensions. A feature cross of these two features would result in roughly 200,000,000 dimensions. Many of those 200,000,000 dimensions represent areas of such limited residence (for example, the middle of the ocean) that it would be difficult to use that data to generalize effectively. It would be silly to pay the RAM cost of storing these unneeded dimensions. Therefore, it would be nice to encourage the weights for the meaningless dimensions to drop to exactly 0, which would allow us to avoid paying for the storage cost of these model coefficients at inference time.

We might be able to encode this idea into the optimization problem done at training time, by adding an appropriately chosen regularization term.

Would L2 regularization accomplish this task? Unfortunately not. L2 regularization encourages weights to be small, but doesn't force them to exactly 0.0.

An alternative idea would be to try and create a regularization term that penalizes the count of non-zero coefficient values in a model. Increasing this count would only be justified if there was a sufficient gain in the model's ability to fit the data. Unfortunately, while this count-based approach is intuitively appealing, it would turn our convex optimization problem into a non-convex optimization problem. So this idea, known as L0 regularization isn't something we can use effectively in practice.

However, there is a regularization term called L1 regularization that serves as an approximation to L0, but has the advantage of being convex and thus efficient to compute. So we can use L1 regularization to encourage many of the uninformative coefficients in our model to be exactly 0, and thus reap RAM savings at inference time.

## L1 vs. L2 regularization.

L2 and L1 penalize weights differently:

* L2 penalizes weight2.
* L1 penalizes |weight|.

Consequently, L2 and L1 have different derivatives:

* The derivative of L2 is 2 \* weight.
* The derivative of L1 is k (a constant, whose value is independent of weight).

You can think of the derivative of L2 as a force that removes x% of the weight every time. As [Zeno](https://wikipedia.org/wiki/Zeno%27s_paradoxes#Dichotomy_paradox) knew, even if you remove x percent of a number *billions of times*, the diminished number will still never quite reach zero. (Zeno was less familiar with floating-point precision limitations, which could possibly produce exactly zero.) At any rate, L2 does not normally drive weights to zero.

You can think of the derivative of L1 as a force that subtracts some constant from the weight every time. However, thanks to absolute values, L1 has a discontinuity at 0, which causes subtraction results that cross 0 to become zeroed out. For example, if subtraction would have forced a weight from +0.1 to -0.2, L1 will set the weight to exactly 0. Eureka, L1 zeroed out the weight.

L1 regularization—penalizing the absolute value of all the weights—turns out to be quite efficient for wide models.

Note that this description is true for a one-dimensional model.