**Google ML Crash Couse Notes**

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?

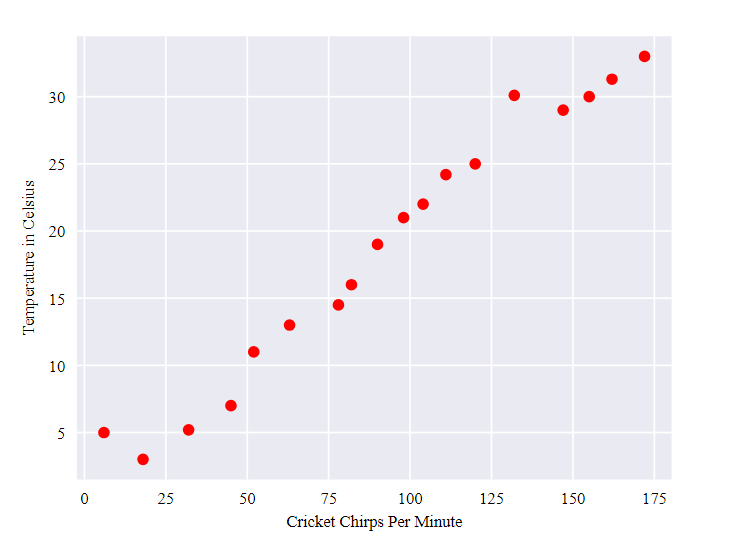
# Descending into ML

## Linear Regression

**Linear regression** is a method for finding the straight line or hyperplane that best fits a set of points. This module explores linear regression intuitively before laying the groundwork for a machine learning approach to linear regression

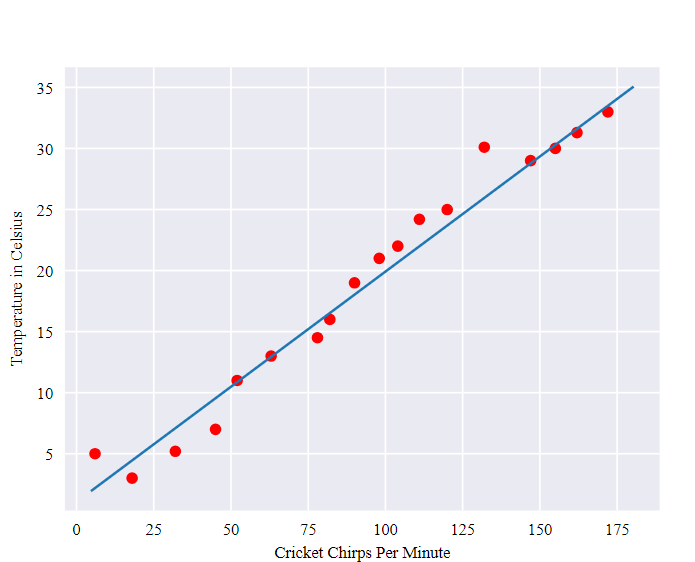
It has long been known that crickets (an insect species) chirp more frequently on hotter days than on cooler days. For decades, professional and amateur scientists have catalogued data on chirps-per-minute and temperature. As a birthday gift, your Aunt Ruth gives you her cricket database and asks you 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 w0.
* w1 is the weight of feature 1. Weight is the same concept as the "slope" m 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:

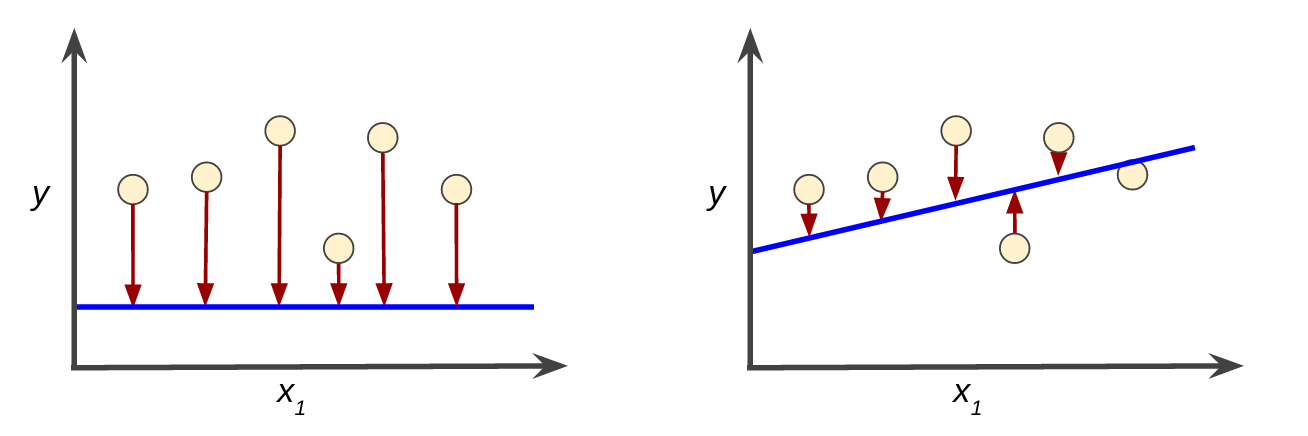
y′ =b + w1x1 + w2x2 + w3x3

## Training and Loss

**Training** a model simply means learning (determining) good values for all the weights and the bias from labelled 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.



**Figure 3. High loss in the left model; low loss in the right model.**

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**

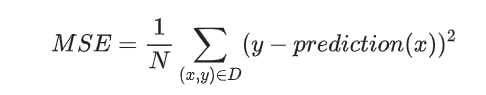
The linear regression models we'll examine here use a loss function called **squared loss** (also known as **L2 loss**). The squared loss for a single example is as follows:

= 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:



where:

* (x,y) is an example in which
  + x is the set of features (for example, chirps/minute, age, gender) that the model uses to make predictions.
  + y is the example's label (for example, temperature).
* prediction(x) is a function of the weights and bias in combination with the set of features x.
* D is a data set containing many labeled examples, which are (x,y) pairs.
* N is the number of examples in D.

Although MSE is commonly-used in machine learning, it is neither the only practical loss function nor the best loss function for all circumstances.

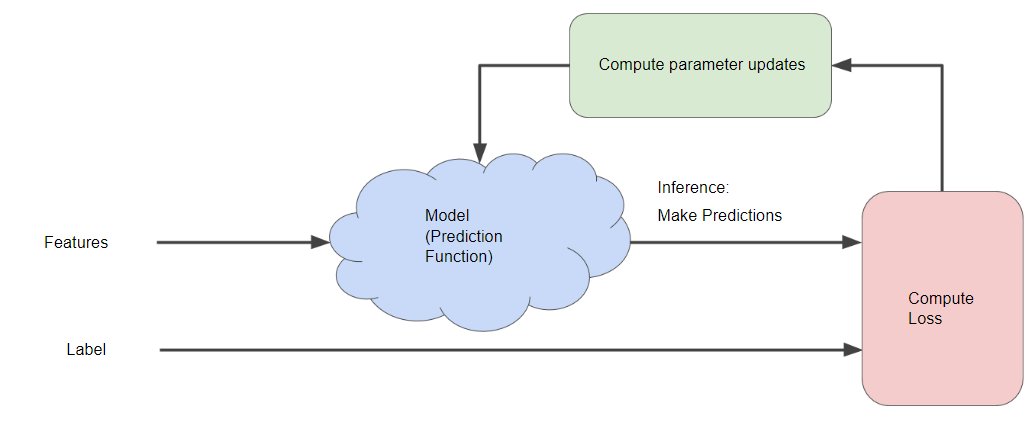
Reducing Loss

## 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 then 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:



**Figure 1. An iterative approach to training a model.**

We'll use this same iterative approach throughout Machine Learning Crash Course, detailing various complications, particularly within that stormy cloud labelled "Model (Prediction Function)." Iterative strategies are prevalent in machine learning, primarily because they scale so well to large data sets.

The "model" takes one or more features as input and returns one prediction (y') as output. To simplify, consider a model that takes one feature and returns one prediction:

y′ = b + w1x1

What initial values should we set for b and w1? For linear regression problems, it turns out that the starting values aren't important. We could pick random values, but we'll just take the following trivial values instead:

* b = 0
* w1 = 0

Suppose that the first feature value is 10. Plugging that feature value into the prediction function yields:

  y' = 0 + 0(10)  
  y' = 0

The "Compute Loss" part of the diagram is the [loss function](https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss) that the model will use. Suppose we use the squared loss function. The loss function takes in two input values:

* *y'*: The model's prediction for features *x*
* *y*: The correct label corresponding to features *x*.

At last, we've reached the "Compute parameter updates" part of the diagram. It is here that the machine learning system examines the value of the loss function and generates new values for b and w1. For now, just assume that this mysterious box devises new values and then the machine learning system re-evaluates all those features against all those labels, yielding a new value for the loss function, which yields new parameter values. And the learning continues iterating until the algorithm discovers the model parameters with the lowest possible loss. Usually, you iterate until overall loss stops changing or at least changes extremely slowly. When that happens, we say that the model has **converged**.

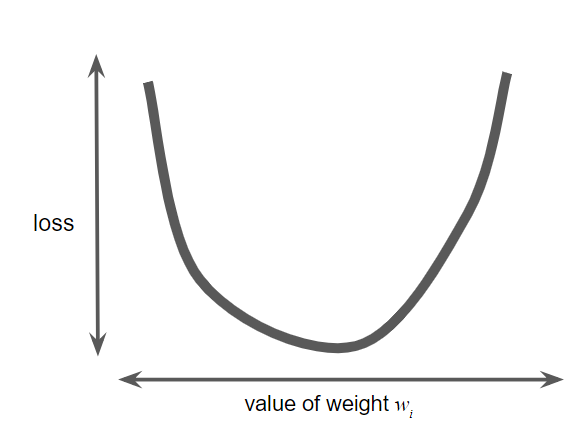
**Key Point:**

A Machine Learning model is trained by starting with an initial guess for the weights and bias and iteratively adjusting those guesses until learning the weights and bias with the lowest possible loss.

## Gradient Descent

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:

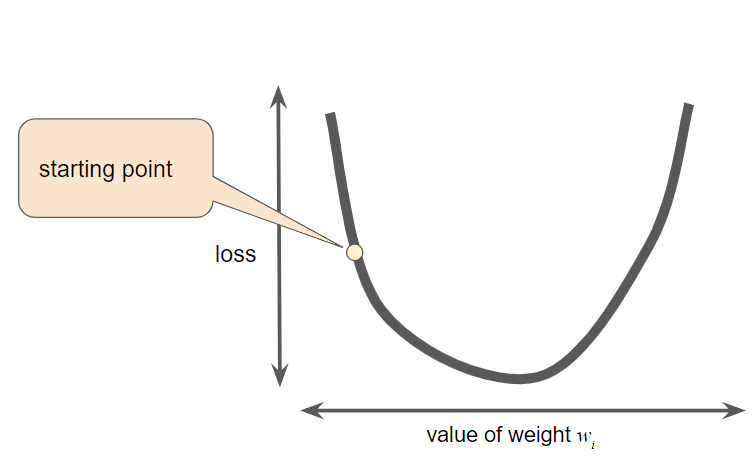


**Figure 2. Regression problems yield convex loss vs weight plots.**

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:



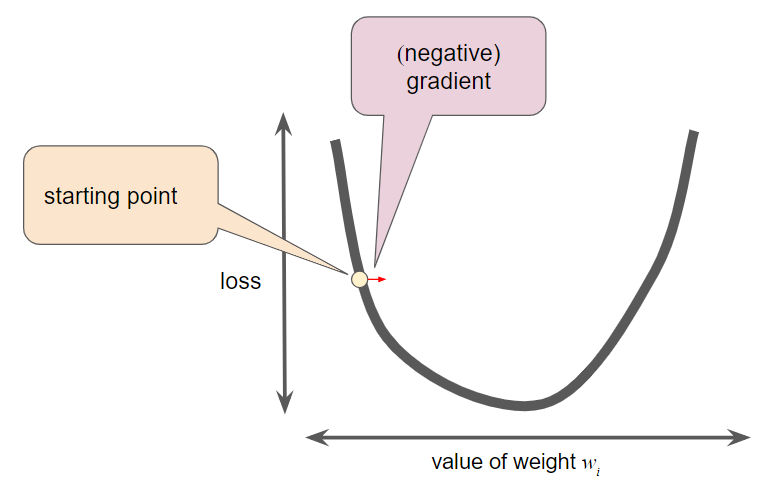
**Figure 3. A starting point for gradient descent.**

The gradient descent algorithm then calculates the gradient of the loss curve at the starting point. Here in Figure 3, the gradient of 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:

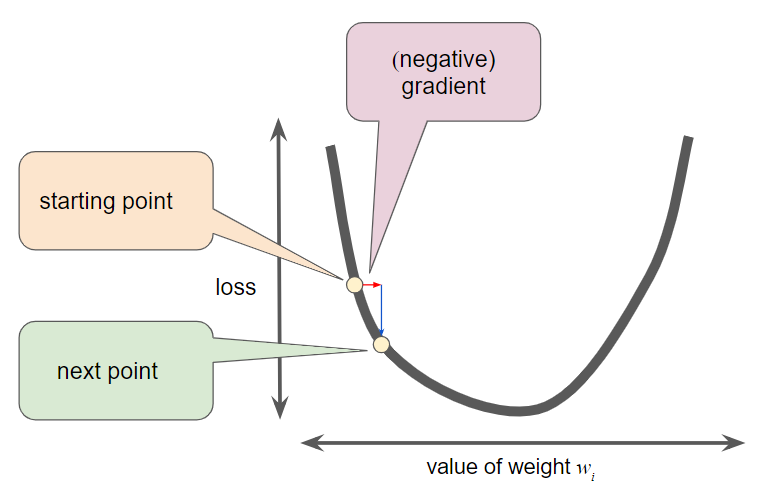
* a direction
* a magnitude

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.



**Figure 4. Gradient descent relies on negative gradients.**

To determine the next point along the loss function curve, the gradient descent algorithm adds some fraction of the gradient's magnitude to the starting point as shown in the following figure:



**Figure 5. A gradient step moves us to the next point on the loss curve.**

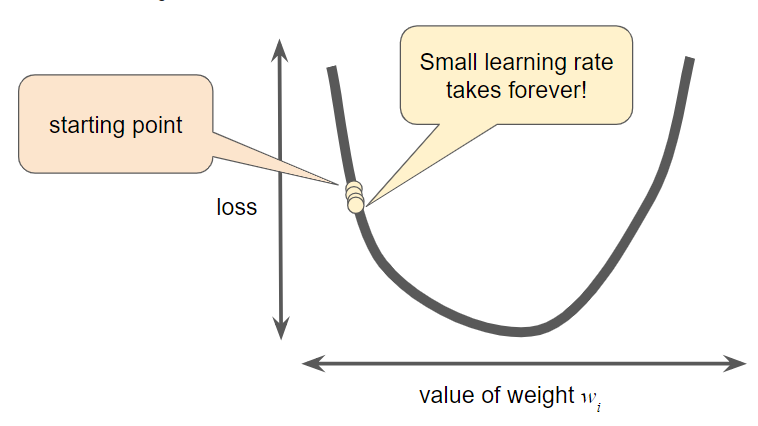
The gradient descent then repeats this process, edging ever closer to the minimum.

**Note:** When performing gradient descent, we generalize the above process to tune all the model parameters simultaneously. For example, to find the optimal values of both w1 and the bias b, we calculate the gradients with respect to both w1 and b. Next, we modify the values of w1 and b based on their respective gradients. Then we repeat these steps until we reach minimum loss.

## Learning Rate

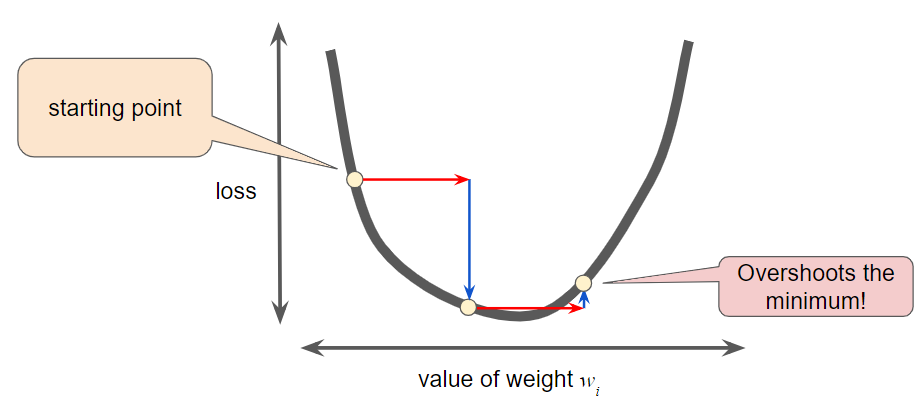
As noted, the gradient vector has both a direction and a magnitude. Gradient descent algorithms multiply the gradient by a scalar known as the **learning rate** (also sometimes called **step size**) to determine the next point. For example, if the gradient magnitude is 2.5 and the learning rate is 0.01,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



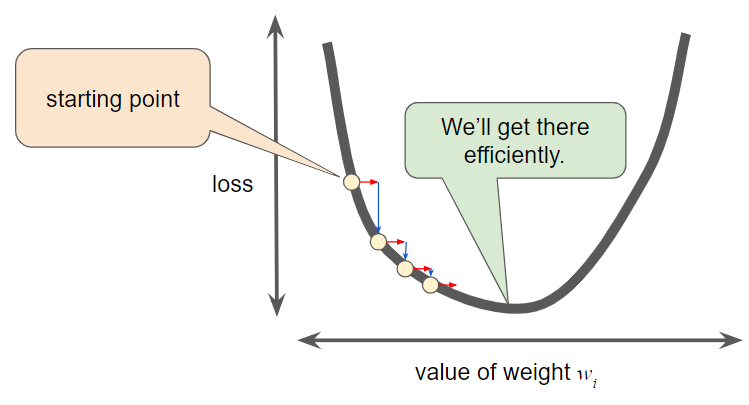
**Figure 6. Learning rate is too small.**

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:



**Figure 7. Learning rate is too large.**

There's a [Goldilocks](https://wikipedia.org/wiki/Goldilocks_principle) learning rate for every regression problem. The Goldilocks value is related to how flat the loss function is. If you know the gradient of the loss function is small then you can safely try a larger learning rate, which compensates for the small gradient and results in a larger step size.



**Figure 8. Learning rate is just right.**

learn more about the ideal learning rate.

The ideal learning rate in one-dimension is 1/f(x)″ (the inverse of the second derivative of f(x) at x).

The ideal learning rate for 2 or more dimensions is the inverse of the [Hessian](https://wikipedia.org/wiki/Hessian_matrix) (matrix of second partial derivatives).

The story for general convex functions is more complex.

## Stochastic Gradient Descent

In gradient descent, a **batch** is the total number of examples you use to calculate the gradient in a single 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.

**Key Terms**

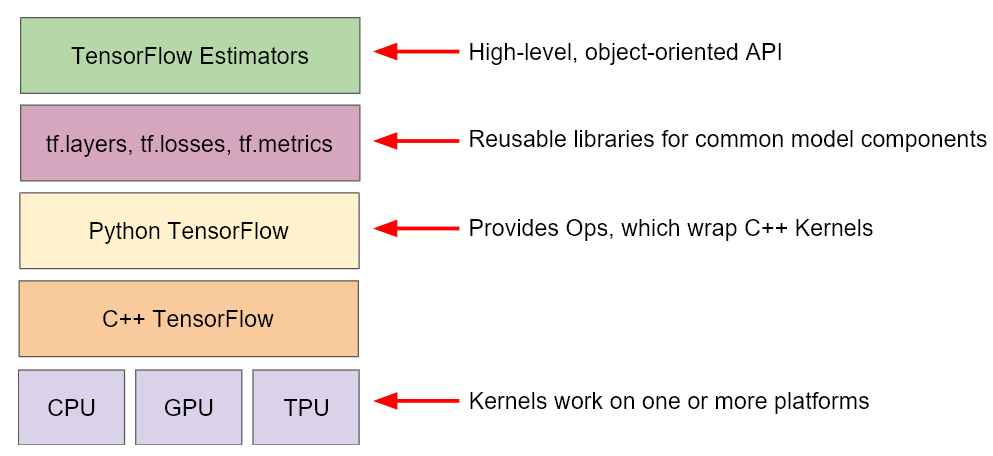
|  |  |
| --- | --- |
|  [batch](https://developers.google.com/machine-learning/glossary#batch) |  [batch size](https://developers.google.com/machine-learning/glossary#batch_size) |
|  [mini-batch](https://developers.google.com/machine-learning/glossary#mini-batch) |  [stochastic gradient descent (SGD)](https://developers.google.com/machine-learning/glossary#SGD) |

# First Steps with TensorFlow

## Toolkit

Tensorflow is a computational framework for building machine learning models. TensorFlow provides a variety of different toolkits that allow you to construct models at your preferred level of abstraction. You can use lower-level APIs to build models by defining a series of mathematical operations. Alternatively, you can use higher-level APIs (like tf.estimator) to specify predefined architectures, such as linear regressors or neural networks.

The following figure shows the current hierarchy of TensorFlow toolkits:



**Figure 1. TensorFlow toolkit hierarchy.**

The following table summarizes the purposes of the different layers:

|  |  |
| --- | --- |
| Toolkit(s) | Description |
| Estimator (tf.estimator) | High-level, OOP API. |
| tf.layers/tf.losses/tf.metrics | Libraries for common model components. |
| TensorFlow | Lower-level APIs |

TensorFlow consists of the following two components:

* a [graph protocol buffer](https://www.tensorflow.org/extend/tool_developers/#protocol_buffers)
* a runtime that executes the (distributed) graph

These two components are analogous to Python code and the Python interpreter. Just as the Python interpreter is implemented on multiple hardware platforms to run Python code, TensorFlow can run the graph on multiple hardware platforms, including CPU, GPU, and [TPU](https://wikipedia.org/wiki/Tensor_processing_unit).

Which API(s) should you use? You should use the highest level of abstraction that solves the problem. The higher levels of abstraction are easier to use, but are also (by design) less flexible. We recommend you start with the highest-level API first and get everything working. If you need additional flexibility for some special modeling concerns, move one level lower. Note that each level is built using the APIs in lower levels, so dropping down the hierarchy should be reasonably straightforward.

### **tf.estimator API**

We'll use tf.estimator for the majority of exercises in Machine Learning Crash Course. Everything you'll do in the exercises could have been done in lower-level (raw) TensorFlow, but using tf.estimator dramatically lowers the number of lines of code.

tf.estimator is compatible with the scikit-learn API. [Scikit-learn](http://scikit-learn.org/) is an extremely popular open-source ML library in Python, with over 100k users, including many at Google.

Very broadly speaking, here's the pseudocode for a linear classification program implemented in tf.estimator:

import tensorflow as tf  
  
# Set up a linear classifier.  
classifier = tf.estimator.LinearClassifier(feature\_columns)  
  
# Train the model on some example data.  
classifier.train(input\_fn=train\_input\_fn, steps=2000)  
  
# Use it to predict.  
predictions = classifier.predict(input\_fn=predict\_input\_fn)

**Key Terms**

|  |  |
| --- | --- |
|  [Estimators](https://developers.google.com/machine-learning/glossary#Estimators) |  [graph](https://developers.google.com/machine-learning/glossary#graph) |
|  [tensor](https://developers.google.com/machine-learning/glossary#tensor) | |

## Programming Exercises

As you progress through Machine Learning Crash Course, you'll put the principles and techniques you learn into practice by coding models using tf.estimator, a high-level [TensorFlow](https://www.tensorflow.org/) API.

The programming exercises in Machine Learning Crash Course use a data-analysis platform that combines code, output, and descriptive text into one collaborative document.

Programming exercises run directly in your browser (no setup required!) using the [Colaboratory](https://colab.research.google.com/) platform. Colaboratory is supported on most major browsers, and is most thoroughly tested on desktop versions of Chrome and Firefox. If you'd prefer to download and run the exercises offline, see [these instructions](https://developers.google.com/machine-learning/crash-course/running-exercises-locally) for setting up a local environment.

Run the following three exercises in the provided order:

1. [Quick Introduction to pandas](https://colab.research.google.com/notebooks/mlcc/intro_to_pandas.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=pandas-colab&hl=en): pandas is an important library for data analysis and modelling and is widely used in TensorFlow coding. This tutorial provides all the pandas information you need for this course. If you already know pandas, you can skip this exercise.

*Introduction to Pandas*

 *NumPy and Python*

1. [First Steps with TensorFlow](https://colab.research.google.com/notebooks/mlcc/first_steps_with_tensor_flow.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=firststeps-colab&hl=en): This exercise explores linear regression.
2. [Synthetic Features and Outliers](https://colab.research.google.com/notebooks/mlcc/synthetic_features_and_outliers.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=syntheticfeatures-colab&hl=en): This exercise explores synthetic features and the effect of input outliers.

### **Common hyperparameters in Machine Learning Crash Course exercises**

Many of the coding exercises contain the following hyperparameters:

* + **steps**, which is the total number of training iterations. One step calculates the loss from *one batch* and uses that value to modify the model's weights *once*.
  + **batch size**, which is the number of examples (chosen at random) for a single step. For example, the batch size for SGD is 1.

The following formula applies:

Totalnumberoftrainedexamples = batchsize ∗ steps

### **A convenience variable in Machine Learning Crash Course exercises**

The following convenience variable appears in several exercises:

* + **periods**, which controls the granularity of reporting. For example, if periods is set to 7 and steps is set to 70, then the exercise will output the loss value every 10 steps (or 7 times). Unlike hyperparameters, we don't expect you to modify the value of periods. Note that modifying periods does not alter what your model learns.

The following formula applies:

Numberoftrainingexamplesineachperiod = (batchsize ∗ stepsperiods) / period

**Key Terms**

|  |  |
| --- | --- |
|  [Estimators](https://developers.google.com/machine-learning/glossary#Estimators) |  [graph](https://developers.google.com/machine-learning/glossary#graph) |
|  [operation](https://developers.google.com/machine-learning/glossary#Operation) |  [session](https://developers.google.com/machine-learning/glossary#session) |

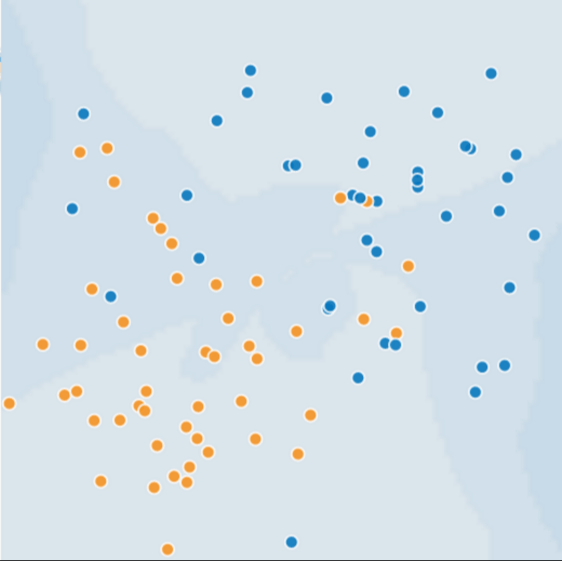
# Generalization

## 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:

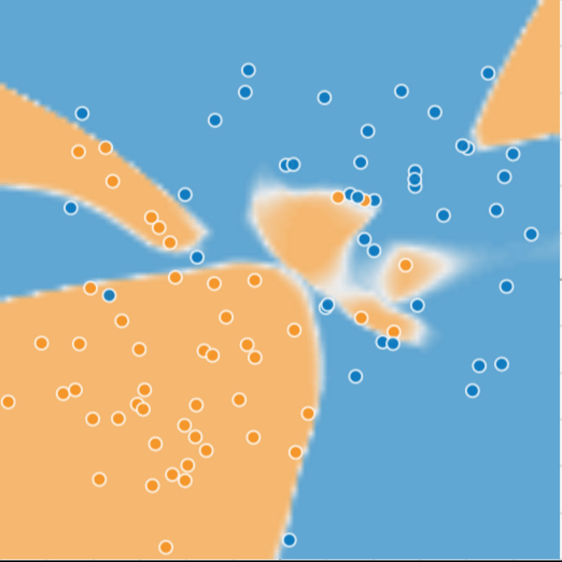
* The blue dots represent sick trees.
* The orange dots represent healthy trees.

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



**Figure 1. Sick (blue) and healthy (orange) trees.**

Can you imagine a good model for predicting subsequent sick or healthy trees? Take a moment to mentally draw an arc that divides the blues from the oranges, or mentally lasso a batch of oranges or blues. Then, look at Figure 2, which shows how a certain machine learning model separated the sick trees from the healthy trees. Note that this model produced a very low loss.

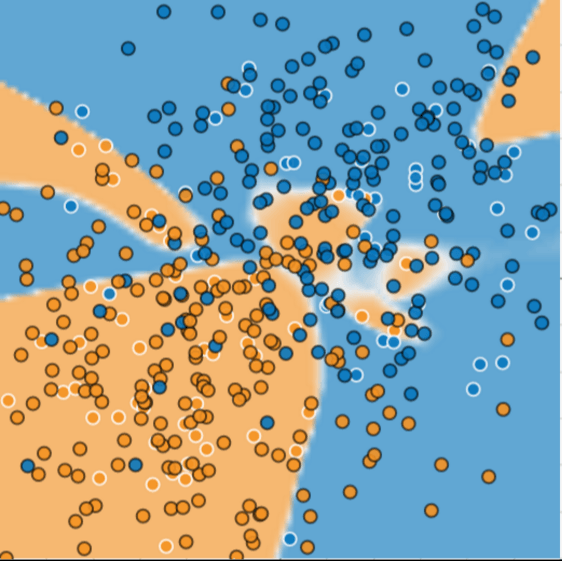


**Figure 2. A complex model for distinguishing sick from healthy trees.**

At first glance, the model shown in Figure 2 appeared to do an excellent job of separating the healthy trees from the sick ones. Or did it?

**Low loss, but still a bad model?**

Figure 3 shows what happened when we added new data to the model. It turned out that the model adapted very poorly to the new data. Notice that the model miscategorized much of the new data.



**Figure 3. The model did a bad job predicting new data.**

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.

**Summary**

* Overfitting occurs when a model tries to fit the training data so closely that it does not generalize well to new data.
* If the key assumptions of supervised ML are not met, then we lose important theoretical guarantees on our ability to predict on new data.

**Key Terms**

|  |  |
| --- | --- |
|  [generalization](https://developers.google.com/machine-learning/glossary#generalization) |  [overfitting](https://developers.google.com/machine-learning/glossary#overfitting) |
|  [prediction](https://developers.google.com/machine-learning/glossary#prediction) |  [stationarity](https://developers.google.com/machine-learning/glossary#stationarity) |
|  [test set](https://developers.google.com/machine-learning/glossary#test_set) |  [training set](https://developers.google.com/machine-learning/glossary#training_set) |

## 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:

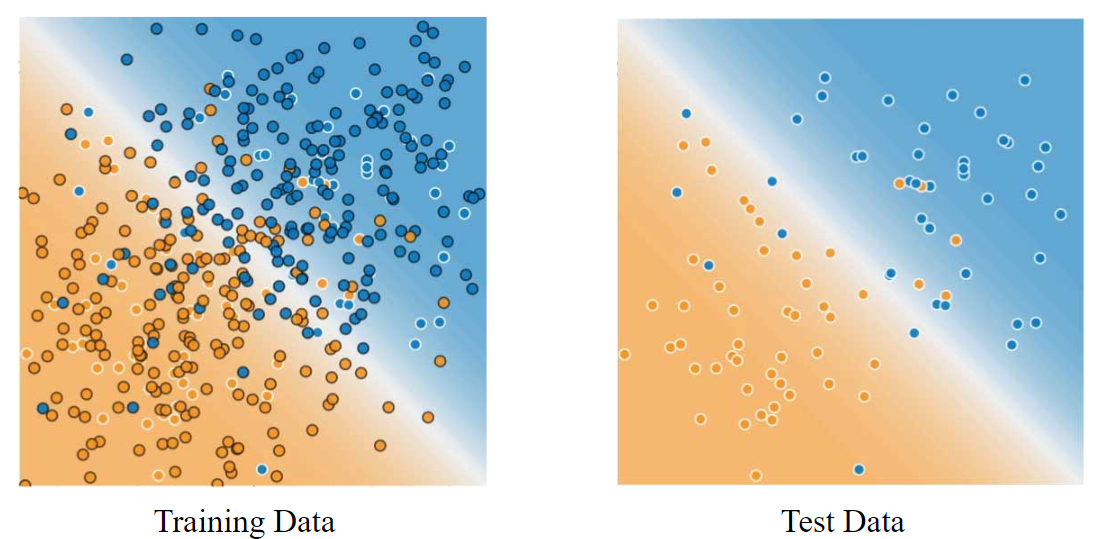


**Figure 1. Slicing a single data set into a training set and test set.**

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.

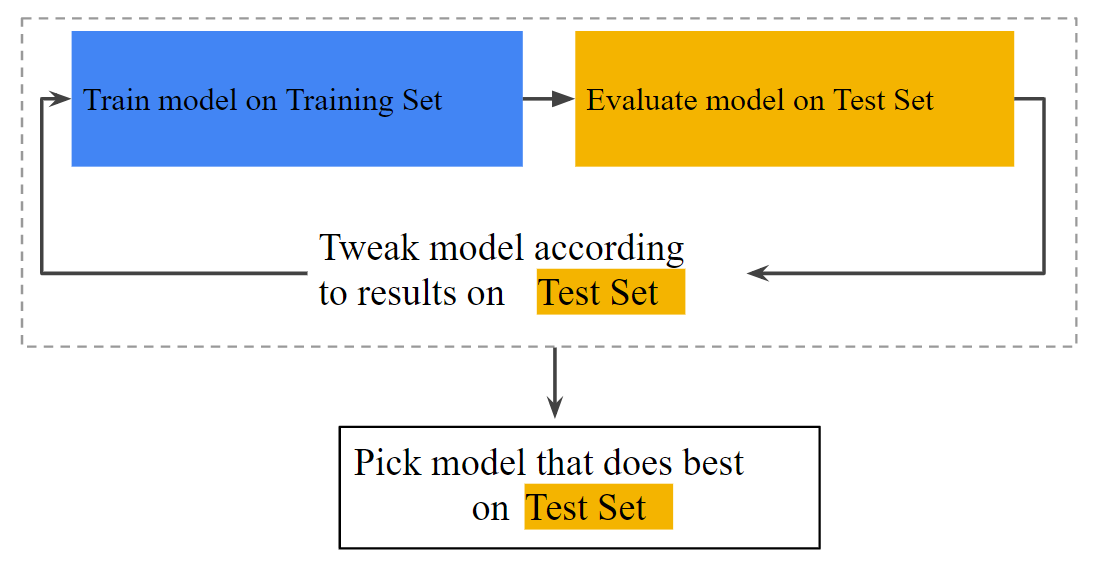
For example, consider a model that predicts whether an email is spam, using the subject line, email body, and sender's email address as features. We apportion the data into training and test sets, with an 80-20 split. After training, the model achieves 99% precision on both the training set and the test 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.

**Key Terms**

|  |  |
| --- | --- |
|  [overfitting](https://developers.google.com/machine-learning/glossary#overfitting) |  [test set](https://developers.google.com/machine-learning/glossary#test_set) |

## 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?**

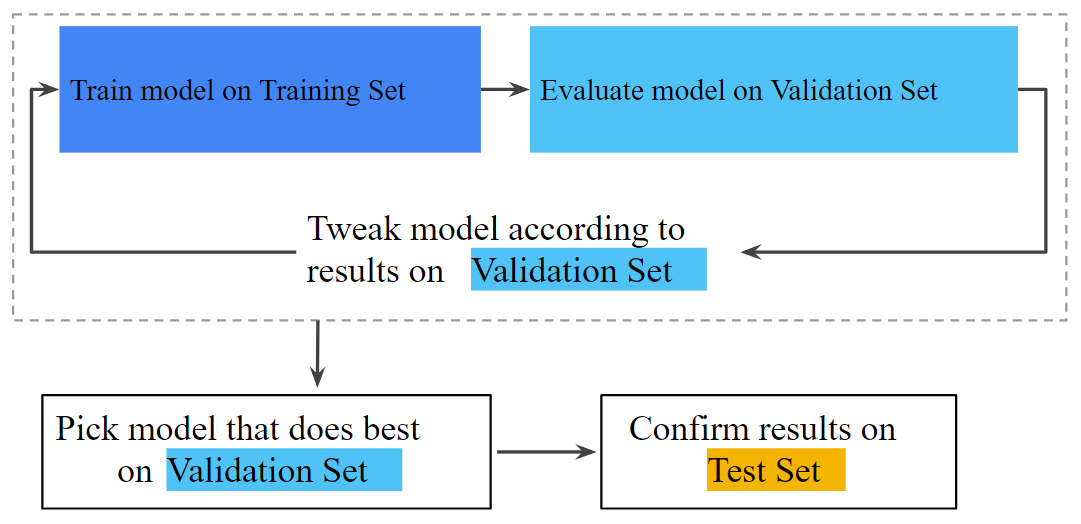
In the figure, "Tweak model" means adjusting anything about the model you can dream up—from changing the learning rate, to adding or removing features, to designing a completely new model from scratch. At the end of this workflow, you pick the model that does best on the *test set*.

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:



**Figure 2. Slicing a single data set into three subsets.**

Use the **validation set** to evaluate results from the training set. Then, use the test set to double-check your evaluationafter the model has "passed" the validation set. The following figure shows this new workflow:



**Figure 3. A better workflow.**

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.

**Tip**

Test sets and validation sets "wear out" with repeated use. That is, the more you use the same data to make decisions about hyperparameter settings or other model improvements, the less confidence you'll have that these results actually generalize to new, unseen data. Note that validation sets typically wear out more slowly than test sets.

If possible, it's a good idea to collect more data to "refresh" the test set and validation set. Starting anew is a great reset.

**Key Terms**

|  |  |
| --- | --- |
|  [overfitting](https://developers.google.com/machine-learning/glossary#overfitting) |  [test set](https://developers.google.com/machine-learning/glossary#test_set) |
|  [training set](https://developers.google.com/machine-learning/glossary#training_set) |  [validation set](https://developers.google.com/machine-learning/glossary#validation_set) |

# 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:

✔This is a good

example: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:

The following is an example of a unique value. This should

be avoided.✘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:

✔The meaning of the following value is clear from the label and

value.house\_age\_years: 27

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

✘The following is an

example of a value that is unclear. This should be avoidedhouse\_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:

✘The following is an example of noisy/bad data. This should be avoided.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:

✔The following is a

good example: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:

✘The following is an

example of a magic value. This should be avoided.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 nameprobably won't change. (Note that we'll still need to convert a string like "br/sao\_paulo" to a one-hot vector.)

✔This is a good

example: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:

✘The following is an

example of a value that could change. This should be avoided.

inferred\_city\_cluster: "219"

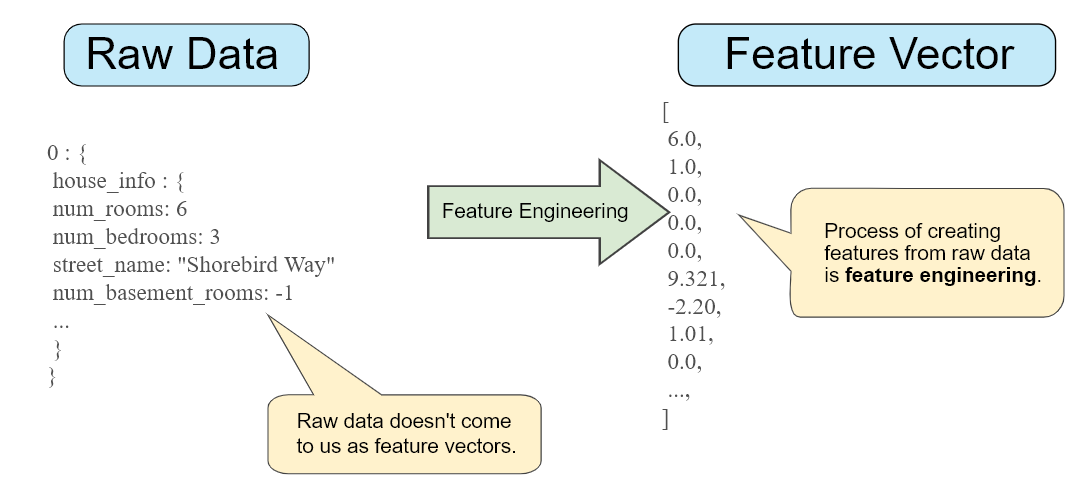
## Feature Engineering

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.

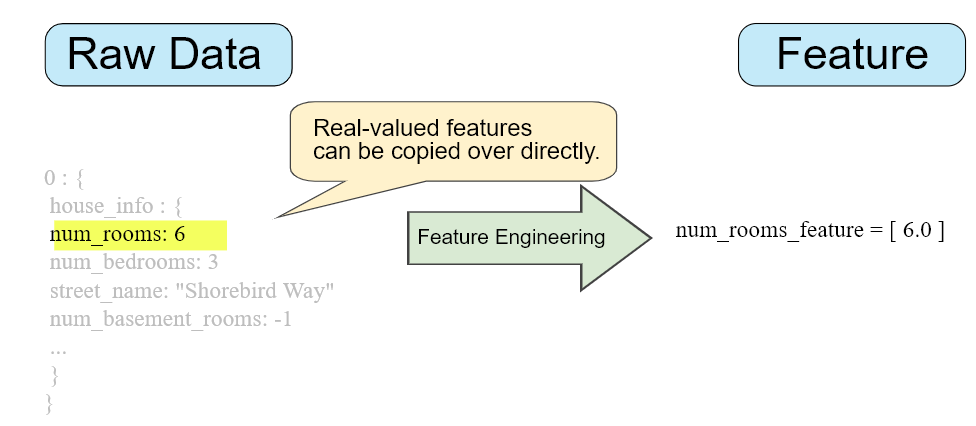
Many machine learning models must represent the features as real-numbered vectors since the feature values must be multiplied by the model weights.



**Figure 1. Feature engineering maps raw data to ML features.**

### Mapping numeric values

Integer and floating-point data don't need a special encoding because they can be multiplied by a numeric weight. As suggested in Figure 2, converting the raw integer value 6 to the feature value 6.0 is trivial:



**Figure 2. Mapping integer values to floating-point values.**

### Mapping categorical values

[Categorical features](https://developers.google.com/machine-learning/glossary#categorical_data) have a discrete set of possible values. For example, there might be a feature called street\_namewith options that include:

{'Charleston Road', 'North Shoreline Boulevard', 'Shorebird Way', 'Rengstorff Avenue'}

Since models cannot multiply strings by the learned weights, we use feature engineering to convert strings to numeric values.

We can accomplish this by defining a mapping from the feature values, which we'll refer to as the **vocabulary** of possible values, to integers. Since not every street in the world will appear in our dataset, we can group all other streets into a catch-all "other" category, known as an **OOV (out-of-vocabulary) bucket**.

Using this approach, here's how we can map our street names to numbers:

* map Charleston Road to 0
* map North Shoreline Boulevard to 1
* map Shorebird Way to 2
* map Rengstorff Avenue to 3
* map everything else (OOV) to 4

However, if we incorporate these index numbers directly into our model, it will impose some constraints that might be problematic:

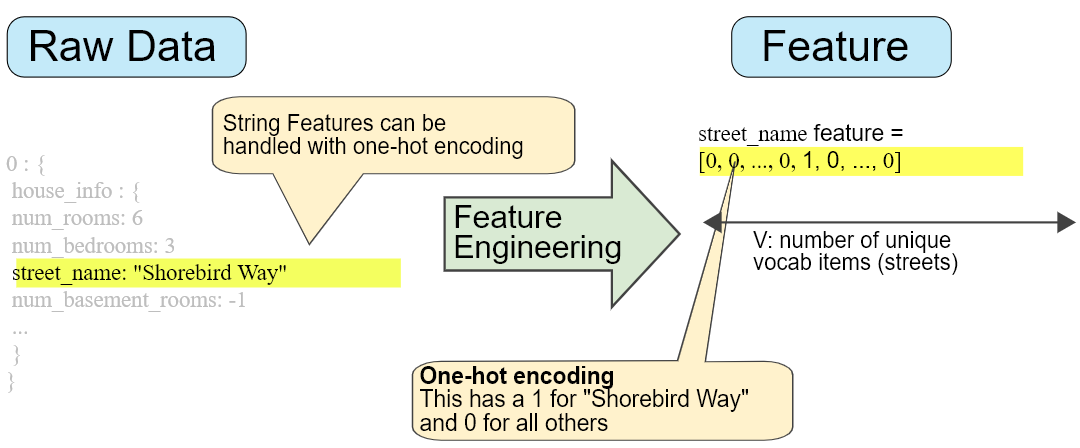
* We'll be learning a single weight that applies to all streets. For example, if we learn a weight of 6 for street\_name, then we will multiply it by 0 for Charleston Road, by 1 for North Shoreline Boulevard, 2 for Shorebird Way and so on. Consider a model that predicts house prices using street\_name as a feature. It is unlikely that there is a linear adjustment of price based on the street name, and furthermore this would assume you have ordered the streets based on their average house price. Our model needs the flexibility of learning different weights for each street that will be added to the price estimated using the other features.
* We aren't accounting for cases where street\_name may take multiple values. For example, many houses are located at the corner of two streets, and there's no way to encode that information in the street\_name value if it contains a single index.

To remove both these constraints, we can instead create a binary vector for each categorical feature in our model that represents values as follows:

* For values that apply to the example, set corresponding vector elements to 1.
* Set all other elements to 0.

The length of this vector is equal to the number of elements in the vocabulary. This representation is called a **one-hot encoding** when a single value is 1, and a **multi-hot encoding** when multiple values are 1.

Figure 3 illustrates a one-hot encoding of a particular street: Shorebird Way. The element in the binary vector for Shorebird Way has a value of 1, while the elements for all other streets have values of 0.



**Figure 3. Mapping street address via one-hot encoding.**

This approach effectively creates a Boolean variable for every feature value (e.g., street name). Here, if a house is on Shorebird Way then the binary value is 1 only for Shorebird Way. Thus, the model uses only the weight for Shorebird Way.

Similarly, if a house is at the corner of two streets, then two binary values are set to 1, and the model uses both their respective weights.

One-hot encoding extends to numeric data that you do not want to directly multiply by a weight, such as a postal code.

### **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.

**Key Terms**

|  |  |
| --- | --- |
|  [discrete feature](https://developers.google.com/machine-learning/glossary#discrete_feature) |  [feature engineering](https://developers.google.com/machine-learning/glossary#feature_engineering) |
|  [one-hot encoding](https://developers.google.com/machine-learning/glossary#one-hot_encoding) |  [representation](https://developers.google.com/machine-learning/glossary#representation) |

## 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.

#### **Click the dropdown arrow to learn more about scaling.**

One obvious way to scale numerical data is to linearly map [min value, max value] to a small scale, such as [-1, +1].

Another popular scaling tactic is to calculate the Z score of each value. The Z score relates the number of standard deviations away from the mean. In other words:

scaledvalue=(value−mean)/stddev.

For example, given:

* mean = 100
* standard deviation = 20
* original value = 130

then:

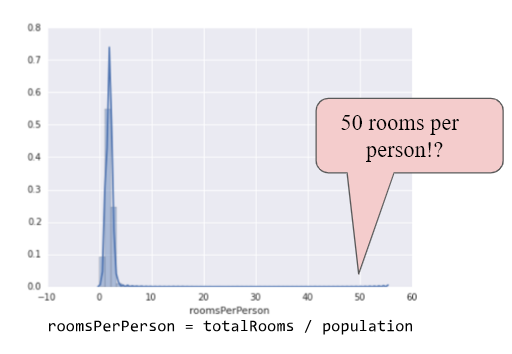
scaled\_value = (130 - 100) / 20

scaled\_value = 1.5

Scaling with Z scores means that most scaled values will be between -3 and +3, but a few values will be a little higher or lower than that range.

### 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.



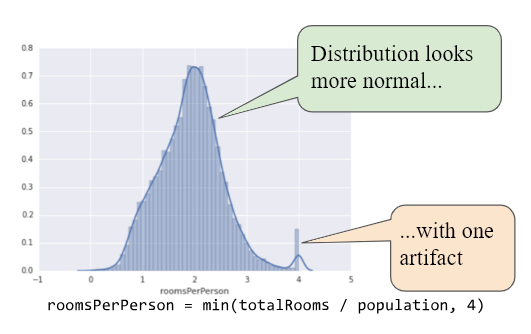
**Figure 4. A verrrrry lonnnnnnng tail.**

How could we minimize the influence of those extreme outliers? Well, one way would be to take the log of every value:



**Figure 5. Logarithmic scaling still leaves a tail.**

Log scaling does a slightly better job, but there's still a significant tail of outlier values. Let's pick yet another approach. What if we simply "cap" or "clip" the maximum value of roomsPerPerson at an arbitrary value, say 4.0?

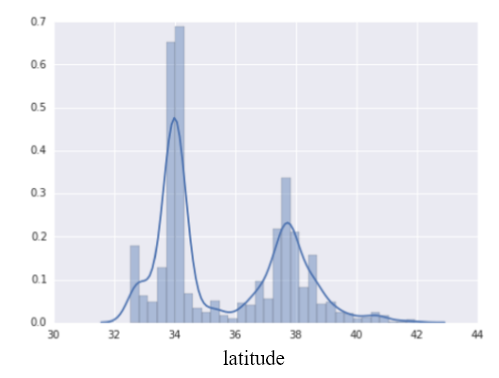


**Figure 6. Clipping feature values at 4.0**

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

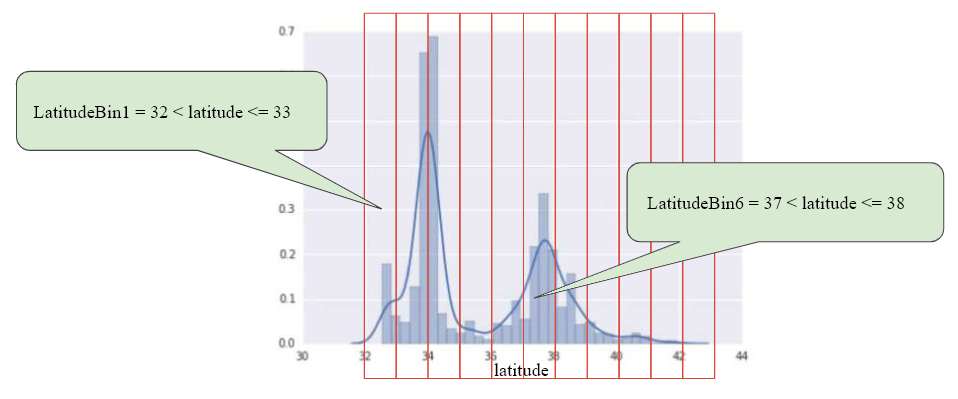
The following plot shows the relative prevalence of houses at different latitudes in California. Notice the clustering—Los Angeles is about at latitude 34 and San Francisco is roughly at latitude 38.



**Figure 7. Houses per latitude.**

In the data set, latitude is a floating-point value. However, it doesn't make sense to represent latitude as a floating-point feature in our model. That's because no linear relationship exists between latitude and housing values. For example, houses in latitude 35 are not 3534 more expensive (or less expensive) than houses at latitude 34. And yet, individual latitudes probably are a pretty good predictor of house values.

To make latitude a helpful predictor, let's divide latitudes into "bins" as suggested by the following figure:



**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.

#### **Click the dropdown arrow to learn more about binning boundaries.**

For simplicity's sake in the latitude example, we used whole numbers as bin boundaries. Had we wanted finer-grain resolution, we could have split bin boundaries at, say, every tenth of a degree. Adding more bins enables the model to learn different behaviors from latitude 37.4 than latitude 37.5, but only if there are sufficient examples at each tenth of a latitude.

Another approach is to bin by [quantile](https://wikipedia.org/wiki/Quantile), which ensures that the number of examples in each bucket is equal. Binning by quantile completely removes the need to worry about outliers.

### 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.

### Additional Information

Rules of Machine Learning, [ML Phase II: Feature Engineering](https://developers.google.com/machine-learning/rules-of-ml/#ml_phase_ii_feature_engineering)

**Key Terms**

|  |  |
| --- | --- |
|  [binning](https://developers.google.com/machine-learning/glossary#binning) |  [feature set](https://developers.google.com/machine-learning/glossary#feature_set) |
|  [NaN trap](https://developers.google.com/machine-learning/glossary#NaN_trap) |  [outliers](https://developers.google.com/machine-learning/glossary#outliers) |
|  [scaling](https://developers.google.com/machine-learning/glossary#scaling) | |