This is self-study key concepts documentation for ML practitioners.

The concepts will help you to

1. Reduce programming time
2. Customize and scale the products

Machine Learning changes the way to solve the problem. We use assertions. From mathematical science to the natural science. Observations and experiments.

**SECTION 1**

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.

**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, x3, …..., 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."

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

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

**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?

**Key Terms**

|  |  |
| --- | --- |
|  [classification model](https://developers.google.com/machine-learning/glossary#classification_model) |  [example](https://developers.google.com/machine-learning/glossary#example) |
|  [feature](https://developers.google.com/machine-learning/glossary#feature) |  [inference](https://developers.google.com/machine-learning/glossary#inference) |
|  [label](https://developers.google.com/machine-learning/glossary#label) |  [model](https://developers.google.com/machine-learning/glossary#model) |
|  [regression model](https://developers.google.com/machine-learning/glossary#regression_model) |  [training](https://developers.google.com/machine-learning/glossary#training) |

**SECTION 2**

**Linear Regression**

**Chart, scatter chart

Description automatically generated**

**Fig – Linear Relationship**

By general mathematical convention, the linear relationship between the dependent and in-dependent variables is represent in following equation:

**y = mx + c**

y = 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.

c = is the y-intercept.

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

**y1 = w1x1 + b**

y1 = is the predicted label (a desired output).

x1 = is a feature (a known input).

w1 = is the weight of feature 1. Weight is the same concept as the "slope" m in the traditional equation of a line.

b = is the bias (the y-intercept), sometimes referred to as w0

The model that relies on three features might look as follows:

**y' = w1x1 + w2x2 + w3x3 + b**

**Key Terms**

|  |  |
| --- | --- |
|  [bias](https://developers.google.com/machine-learning/glossary#bias) |  [inference](https://developers.google.com/machine-learning/glossary#inference) |
|  [linear regression](https://developers.google.com/machine-learning/glossary#linear_regression) |  [weight](https://developers.google.com/machine-learning/glossary#weight) |

**SECTION 3**

**Training and Loss**

**What is training? – Learning weights and bias**

**What is Loss? – Difference between predicted and actual values.**

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.

Chart, diagram

Description automatically generated

**Squared loss: a popular loss function : L2 Loss**

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:

Text

Description automatically generated

**Key Terms**

|  |  |
| --- | --- |
|  [empirical risk minimization](https://developers.google.com/machine-learning/glossary#ERM) |  [loss](https://developers.google.com/machine-learning/glossary#loss) |
|  [mean squared error](https://developers.google.com/machine-learning/glossary#MSE) |  [squared loss](https://developers.google.com/machine-learning/glossary#squared_loss) |
|  [training](https://developers.google.com/machine-learning/glossary#training) |  |

**SECTION 4**

**Reducing Loss: An Iterative Approach**

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

Diagram

Description automatically generated

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 and w1 = 0

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

y’ = 0 + 0\*10 = 0

The "Compute Loss" part of the diagram is the loss function 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.

**Key Terms**

|  |  |
| --- | --- |
|  [convergence](https://developers.google.com/machine-learning/glossary#convergence) |  [loss](https://developers.google.com/machine-learning/glossary#loss) |
|  [training](https://developers.google.com/machine-learning/glossary#training) |  |

**SECTION 5**

**Reducing Loss: Gradient Descent**

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:

**Diagram

Description automatically generated**

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:

Diagram

Description automatically generated

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 (slope) of the curve.

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 to reduce loss as quickly as possible.

Diagram

Description automatically generated

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:

Diagram

Description automatically generated

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.

**Key Terms**

|  |  |
| --- | --- |
|  [gradient descent](https://developers.google.com/machine-learning/glossary#gradient_descent) |  [step](https://developers.google.com/machine-learning/glossary#step) |

<https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent#expandable-1>

The gradient of a function, denoted as follows, is the vector of partial derivatives with respect to all the independent variables:

Text

Description automatically generated

The number of dimensions in the vector is equal to the number of variables in the formula for f; in other words, the vector falls within the domain space of the function. For instance, the graph of the following function f(x,y):

Chart, surface chart

Description automatically generated

The gradient of f(x,y) is a two-dimensional vector that tells you in which (x,y) direction to move for the maximum increase in height. Thus, the negative of the gradient moves you in the direction of maximum decrease in height. In other words, the negative of the gradient vector points into the valley.

In machine learning, gradients are used in gradient descent. We often have a loss function of many variables that we are trying to minimize, and we try to do this by following the negative of the gradient of the function.

**SECTION 6**

**Reducing 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:**

**Diagram

Description automatically generated**

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:

Diagram

Description automatically generated

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.

Diagram

Description automatically generated

**Key Terms**

|  |  |
| --- | --- |
|  [hyperparameter](https://developers.google.com/machine-learning/glossary#hyperparameter) |  [learning rate](https://developers.google.com/machine-learning/glossary#learning_rate) |
|  [step size](https://developers.google.com/machine-learning/glossary#step_size)  **NOTE:** In practice, finding a "perfect" (or near-perfect) learning rate is not essential for successful model training. The goal is to find a learning rate large enough that gradient descent converges efficiently, but not so large that it never converges. |  |

**SECTION 7**

**Reducing Loss: 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) |

**SECTION 8**

**Introduction to Tensor Flow**

**Learning Objectives**

* Learn enough about NumPy and pandas to understand tf.keras code.
* Learn how to use Colabs.
* Become familiar with linear regression code in tf.keras.
* Evaluate loss curves.
* Tune hyperparameters.

TensorFlow is an end-to-end open-source platform for machine learning. TensorFlow is a rich system for managing all aspects of a machine learning system; however, this class focuses on using a particular TensorFlow API to develop and train machine learning models. See the [TensorFlow documentation](https://tensorflow.org/) for complete details on the broader TensorFlow system.

TensorFlow APIs are arranged hierarchically, with the high-level APIs built on the low-level APIs. Machine learning researchers use the low-level APIs to create and explore new machine learning algorithms. In this class, you will use a high-level API named tf.keras to define and train machine learning models and to make predictions. tf.keras is the TensorFlow variant of the open source [Keras](https://keras.io/) API.

A picture containing diagram

Description automatically generated

**Section 9 – Programming Exercises**

**NumPy and Pandas**

**Using tf.keras requires at least a little understanding of the following two open-source Python libraries:**

[NumPy](https://numpy.org/), which simplifies representing arrays and performing linear algebra operations.

[Pandas](https://pandas.pydata.org/), which provides an easy way to represent datasets in memory.

If you are unfamiliar with NumPy or pandas, please begin by doing the following two Colab exercises:

[NumPy Ultraquick Tutorial](https://colab.research.google.com/github/google/eng-edu/blob/main/ml/cc/exercises/numpy_ultraquick_tutorial.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=numpy_tf2-colab&hl=en) Colab exercise, which provides all the NumPy information you need for this course.

[Pandas UltraQuick Tutorial](https://colab.research.google.com/github/google/eng-edu/blob/main/ml/cc/exercises/pandas_dataframe_ultraquick_tutorial.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=pandas_tf2-colab&hl=en) Colab exercise, which provides all the pandas information you need for this course.

**Linear regression with tf.keras**

After gaining competency in NumPy and pandas, do the following two Colab exercises to explore linear regression and hyperparameter tuning in tf.keras:

[Linear Regression with Synthetic Data](https://colab.research.google.com/github/google/eng-edu/blob/main/ml/cc/exercises/linear_regression_with_synthetic_data.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=linear_regression_synthetic_tf2-colab&hl=en) Colab exercise, which explores linear regression with a toy dataset.

[Linear Regression with a Real Dataset](https://colab.research.google.com/github/google/eng-edu/blob/main/ml/cc/exercises/linear_regression_with_a_real_dataset.ipynb?utm_source=mlcc&utm_campaign=colab-external&utm_medium=referral&utm_content=linear_regression_real_tf2-colab&hl=en) Colab exercise, which guides you through the kinds of analysis you should do on a real dataset.

**Section 10**

**Generalization and the Problem of Overfitting**

The opposite of overfitting is generalization in ML.

Example will explain this further –

Chart, scatter chart

Description automatically generated

A picture containing chart

Description automatically generated

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.

**LOW LOSS BUT, STILL BAD MODEL**

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, **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

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) |

**Section 11**

**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:

**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.

If 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.

**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) |
|  [training set](https://developers.google.com/machine-learning/glossary#training_set) |  |

**Section 12**

The previous module 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:

Diagram

Description automatically generated

A picture containing text

Description automatically generated

Diagram

Description automatically generated

In this improved workflow:

* Pick the model that does best on the validation set.
* 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.

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) |

**Section 13**

**Representation: Feature Engineering**

**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.

Diagram

Description automatically generated

**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:

**Mapping categorical values**

Categorical features have a discrete set of possible values.

Generally, we need to convert the strings to integers as further we must multiply model weights to those integer 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.**

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

1. **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.

1. **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.

Diagram

Description automatically generated

TIP:

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) |

**Section 14**

**Representation: Qualities of Good Features**

1. **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.

**Conversely, if a feature's value appears only once or very rarely, the model can't make predictions based on that feature**.

1. **Prefer clear and obvious meanings**

Each feature should have a clear and obvious meaning to anyone on the project.

1. **Don't mix "magic" values with actual data**

Good floating-point features don't contain peculiar out-of-range discontinuities or "magic" values.

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. Ex. Other.
* **For continuous variables**, ensure missing values do not affect the model by using the **mean value** of the feature's data.
* In case a rating column has values from 0 to 1 value with “magic” value of -1 in few columns then, to explicitly mark such 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.

1. **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.)

**Section 15**

**Representation: Cleaning Data**

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**.

1. **Scaling feature values**

[**Scaling**](https://developers.google.com/machine-learning/crash-course/representation/cleaning-data#expandable-1) **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 (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 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.

1. **Handling extreme outliers**

The following plot represents a feature called roomsPerPerson from the California Housing data set. 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.

**Chart

Description automatically generated with medium confidence**

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

**Chart

Description automatically generated with medium confidence**

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?

A picture containing diagram

Description automatically generated

**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.

1. **Binning – [hint: Bucketizing]**

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.

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 more expensive (or less expensive) than houses at latitude 34. And yet, individual latitudes probably are a pretty good predictor of house values.

Chart, histogram

Description automatically generated

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

Diagram

Description automatically generated

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.

1. **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?

1. **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.

**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) | |

**Section 16**

**Feature Crosses: Encoding Nonlinearity**

A **feature cross** is a synthetic feature that encodes nonlinearity in the feature space by multiplying two or more input features together.

The term cross comes from [cross product](https://wikipedia.org/wiki/Cross_product).

x3 = x1.x2

We treat this newly minted feature cross w3 just like any other feature. The linear formula becomes:

Y = w1x1 + w2x3 + w3x3 + b

linear algorithm can learn a weight for w3 just as it would for w1 and w2. In other words, although w3 encodes nonlinear information, you don’t need to change how the linear model trains to determine the value of w3.

### Kinds of feature crosses

We can create many kinds of feature crosses. For example:

* [A X B]: a feature cross formed by multiplying the values of two features.
* [A x B x C x D x E]: a feature cross formed by multiplying the values of five features.
* [A x A]: a feature cross formed by squaring a single feature.

Thanks to [stochastic gradient descent](https://developers.google.com/machine-learning/crash-course/reducing-loss/stochastic-gradient-descent), linear models can be trained efficiently. Consequently, supplementing scaled linear models with feature crosses has traditionally been an efficient way to train on massive-scale data sets.

**Key Terms**

|  |
| --- |
|  [feature crosses](https://developers.google.com/machine-learning/glossary#feature_cross) |
|  [synthetic feature](https://developers.google.com/machine-learning/glossary#synthetic_feature) |

**Section 17**

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]

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 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.

**Key Terms**

|  |
| --- |
|  [one-hot encoding](https://developers.google.com/machine-learning/glossary#one-hot_encoding) |

**Section 18**

**Regularization for Simplicity: L₂ Regularization**

Consider the following generalization curve, which shows the loss for both the training set and validation set against the number of training iterations.

A picture containing diagram

Description automatically generated

Figure 1 shows a model in which **training loss** gradually **decreases**, but **validation loss** eventually **goes up**.

In other words, this generalization curve shows that the **model is overfitting** to the data in the training set. Channeling our inner [Ockham](https://developers.google.com/machine-learning/crash-course/generalization/peril-of-overfitting#ockham), perhaps we could prevent overfitting by penalizing complex models, a principle called **regularization**.

In other words, **instead** of simply aiming to **minimize loss** (**empirical risk minimization**):

**Minimize(Loss(Data|Model))**

we'll now **minimize loss+complexity**, which is called **structural risk minimization**:

**Minimize(Loss(Data|Model) + Complexity(Model))**

Our training optimization algorithm is now a function of two terms: **the loss term**, which measures how well the model fits the data, and the **regularization term**, which measures model complexity.

**What is Model Complexity?**

Machine Learning Crash Course focuses on two common (and somewhat related) ways to think of **model complexity**:

* Model complexity as **a function of the weights** of all the features in the model.
* Model complexity as **a function of the total number of features with nonzero weights.** (A later module covers this approach.)

If **model complexity is a function of weights**, a feature weight with a high absolute value is more complex than a feature weight with a low absolute value.

We can **quantify complexity** using the **L2 regularization formula**, which defines the regularization term as the sum of the squares of all the feature weights:



In this formula, weights close to zero have little effect on model complexity, while outlier weights can have a huge impact.

For example, a linear model with the following weights:



Has an *L2* regularization term of 26.915:

Text, letter

Description automatically generated

But w3 (bolded above), with a squared value of 25, contributes nearly all the complexity. The sum of the squares of all five other weights adds just 1.915 to the L2 regularization term.

**Key Terms**

|  |  |
| --- | --- |
|  [generalization curve](https://developers.google.com/machine-learning/glossary#generalization_curve) |  [*L2* regularization](https://developers.google.com/machine-learning/glossary#L2_regularization) |
|  [overfitting](https://developers.google.com/machine-learning/glossary#overfitting) |  [regularization](https://developers.google.com/machine-learning/glossary#regularization) |
|  [structural risk minimization](https://developers.google.com/machine-learning/glossary#SRM) |  |

**Section 18**

**Regularization for Simplicity: Lambda**

Model developers tune the overall impact of the regularization term by multiplying its value by a scalar known as **lambda** (also called the **regularization rate**).

That is, model developers aim to do the following:



Performing L2 regularization has the following effect on a model

* Encourages **weight values** **toward 0** (**but not exactly 0**)
* Encourages the **mean of the weights** **toward 0**, with a normal (**bell-shaped or Gaussian**) distribution.

**Increasing the lambda value strengthens the regularization effect**. For example, the histogram of weights for a high value of lambda might look as shown in Figure 2.

Chart, histogram

Description automatically generated

**Lowering the value of lambda tends to yield a flatter histogram, as shown in Figure 3**

**Chart, histogram

Description automatically generated**

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, t**raining 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.

There's a **close connection between learning rate and lambda**:

**Doing L2 Regularization and Lowering Learning Rates can create compound effect.**

**Strong L2 regularization** values tend to drive feature weights closer to 0. **Lower learning rates** (with early stopping) often produce the same effect because the steps away from 0 aren't as large. Consequently, tweaking learning rate and lambda simultaneously may have confounding effects.

**Early stopping** means ending training before the model fully reaches convergence.

In practice, we often end up with some amount of implicit early stopping when training in an online (continuous) fashion. That is, some new trends just haven't had enough data yet to converge.

As noted, the effects from changes to regularization parameters can be confounded with the effects from changes in learning rate or number of iterations.

One useful practice (when training across a fixed batch of data) is to give yourself a high enough number of iterations that early stopping doesn't play into things.

**Key Terms**

|  |  |
| --- | --- |
|  [early stopping](https://developers.google.com/machine-learning/glossary#early_stopping) |  [lambda](https://developers.google.com/machine-learning/glossary#lambda) |
|  [regularization rate](https://developers.google.com/machine-learning/glossary#regularization_rate) |  |

**Section 19**

**Logistic Regression: Calculating a Probability**

Many problems require a probability estimate as output. Logistic regression is an extremely efficient mechanism for **calculating probabilities.** Practically speaking, you can use the returned probability in either of the following two ways:

* "As is"
* Converted to a binary category.

Let's consider how we might use the **probability "as is."**

Suppose we create a logistic regression model to predict the probability that a dog will bark during the middle of the night. We'll call that probability:

P(bark|night)

If the logistic regression model predicts P(bark|night) = 0.05, then over a year, the dog's owners should be startled awake approximately 18 times:

Startled = P(bark|night) .nights = 0.05 \* 365 = 18

In many cases, you'll map the logistic regression output into the solution to a binary classification problem, in which the goal is to correctly predict one of two possible labels (e.g., "spam" or "not spam"). A later module focuses on that.

You might be wondering how a logistic regression model can ensure output that always falls between 0 and 1. As it happens, **a sigmoid function**, defined as follows, produces output having those same characteristics:

A picture containing text, clock

Description automatically generated

Chart

Description automatically generated

If **z**represents the output of the linear layer of a model trained with logistic regression,

then **sigmoid(z)** will yield a **value (a probability) between 0 and 1**. In mathematical terms:

Graphical user interface, text, application, email

Description automatically generated

Note that z is also referred to as the log-odds

because the inverse of the sigmoid states that z can be defined as the log of the probability of the 1 label (e.g., "dog barks") divided by the probability of the 0 label (e.g., "dog doesn't bark"):

Diagram

Description automatically generated

Chart, line chart, histogram

Description automatically generated

[Example of L.G](https://developers.google.com/machine-learning/crash-course/logistic-regression/calculating-a-probability#expandable-1)

**Key Terms**

|  |  |
| --- | --- |
|  [binary classification](https://developers.google.com/machine-learning/glossary#binary_classification) |  [logistic regression](https://developers.google.com/machine-learning/glossary#logistic_regression) |
|  [sigmoid function](https://developers.google.com/machine-learning/glossary#sigmoid_function) |  |

**Section 20**

**Logistic Regression: Loss and Regularization**

**Loss function for Logistic Regression**

The loss function for linear regression is squared loss. The loss function for logistic regression is Log Loss, which is defined as follows:

Text, letter

Description automatically generated

**Regularization in Logistic Regression**

Regularization 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.

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.

**Summary**

Logistic regression models generate probabilities.

Log Loss is the loss function for logistic regression.

Logistic regression is widely used by many practitioners.

**Key Terms**

|  |  |
| --- | --- |
|  [early stopping](https://developers.google.com/machine-learning/glossary#early_stopping) |  [log loss](https://developers.google.com/machine-learning/glossary#Log_Loss) |
|  [L1 regularization](https://developers.google.com/machine-learning/glossary#L1_regularization) |  [L2 regularization](https://developers.google.com/machine-learning/glossary#L2_regularization) |