Prerequisites and Prework

Is Machine Learning Crash Course right for you?

I have little or no machine learning background.

I have some background in machine learning, but I'd like a more current and complete understanding.

I know machine learning really well, but I know little or nothing about TensorFlow.

Please read through the following Prerequisites (#prerequisites) and Prework (#prework) sections before beginning Machine Learning Crash Course, to ensure you are prepared to complete all the modules.

Prerequisites

Machine Learning Crash Course does not presume or require any prior knowledge in machine learning. However, to understand the concepts presented and complete the exercises, we recommend that students meet the following prerequisites:

- **Mastery of intro-level algebra.** You should be comfortable with variables and coefficients, linear equations, graphs of functions, and histograms. (Familiarity with more advanced math concepts such as logarithms and derivatives is helpful, but not required.)
- **Proficiency in programming basics, and some experience coding in Python.** Programming exercises in Machine Learning Crash Course are coded in Python (https://www.python.org/) using TensorFlow (https://www.tensorflow.org/). No prior experience with TensorFlow is required, but you should feel comfortable reading and writing Python code that contains basic programming constructs, such as function definitions/invocations, lists and dicts, loops, and conditional expressions.

Note: See the Key Concepts and Tools (#key-concepts) section below for a detailed list of math and programming concepts used in Machine Learning Crash Course, with reference materials for each.

Prework

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.

Problem Framing

If you're new to machine learning, we recommend starting your journey by taking Introduction to Machine Learning Problem Framing (https://developers.google.com/machine-learning/problem-framing/). This one-hour course teaches you how to identify appropriate problems for machine learning.

Getting Started with pandas

The programming exercises in Machine Learning Crash Course use the pandas (http://pandas.pydata.org/) library for manipulating data sets. If you're unfamiliar with pandas, we recommend completing the 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) tutorial, which illustrates the key pandas features used in the exercises.

Key Concepts and Tools

Machine Learning Crash Course discusses and applies the following concepts and tools. For more information, see the linked resources.

Math

Algebra

- Variables (https://www.khanacademy.org/math/algebra/introduction-to-algebra/alg1-intro-to-variables/v/what-is-a-variable), coefficients (https://www.khanacademy.org/math/cc-sixth-grade-math/cc-6th-equivalent-exp/cc-6th-parts-of-expressions/v/expression-terms-factors-and-coefficients), and functions (https://www.khanacademy.org/math/algebra/algebra-functions)
- Linear equations (https://wikipedia.org/wiki/Linear_equation) such as $y = b + w_1 x_1 + w_2 x_2$
- Logarithms (https://wikipedia.org/wiki/Logarithm), and logarithmic equations such as $y = ln(1 + e^z)$
- Sigmoid function (https://wikipedia.org/wiki/Sigmoid_function)

Linear algebra

- Tensor and tensor rank (https://www.tensorflow.org/programmers_guide/tensors)
- Matrix multiplication (https://wikipedia.org/wiki/Matrix_multiplication)

Trigonometry

- Tanh (https://reference.wolfram.com/language/ref/Tanh.html) (discussed as an activation function (https://developers.google.com/machine-learning/glossary#activation_function); no prior knowledge needed)

Statistics

- Mean, median, outliers (https://www.khanacademy.org/math/probability/data-distributions-a1/summarizing-center-distributions/v/mean-median-and-mode), and standard deviation (https://wikipedia.org/wiki/Standard_deviation)
- Ability to read a histogram (https://wikipedia.org/wiki/Histogram)

Calculus (*optional, for advanced topics*)

- Concept of a derivative (https://wikipedia.org/wiki/Derivative) (you won't have to actually calculate derivatives)

- Gradient (https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/gradient-and-directional-derivatives/v/gradient) or slope
- Partial derivatives (https://wikipedia.org/wiki/Partial_derivative) (which are closely related to gradients)
- Chain rule (https://wikipedia.org/wiki/Chain_rule) (for a full understanding of the backpropagation algorithm (https://developers.google.com/machine-learning/crash-course/backprop-scroll/) for training neural networks)

Python Programming

Basic Python

The following Python basics are covered in The Python Tutorial (https://docs.python.org/3/tutorial/):

- Defining and calling functions (https://docs.python.org/3/tutorial/controlflow.html#defining-functions), using positional and keyword (https://docs.python.org/3/tutorial/controlflow.html#keyword-arguments) parameters
- Dictionaries (https://docs.python.org/3/tutorial/datastructures.html#dictionaries), lists (https://docs.python.org/3/tutorial/introduction.html#lists), sets (https://docs.python.org/3/tutorial/datastructures.html#sets) (creating, accessing, and iterating)
- `for` loops (https://docs.python.org/3/tutorial/controlflow.html#for-statements), `for` loops with multiple iterator variables (e.g., `for a, b in [(1,2), (3,4)]`)
- `if/else` conditional blocks (https://docs.python.org/3/tutorial/controlflow.html#if-statements) and conditional expressions (https://docs.python.org/2.5/whatsnew/pep-308.html)
- String formatting (https://docs.python.org/3/tutorial/inputoutput.html#old-string-formatting) (e.g., `'%.2f' % 3.14`)
- Variables, assignment, basic data types (https://docs.python.org/3/tutorial/introduction.html#using-python-as-a-calculator) (`int`, `float`, `bool`, `str`)
- The `pass` statement (https://docs.python.org/3/tutorial/controlflow.html#pass-statements)

Intermediate Python

The following more advanced Python features are also covered in The Python Tutorial (https://docs.python.org/3/tutorial/):

- List comprehensions (https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions)
- Lambda functions (https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions)

Third-Party Python Libraries

Machine Learning Crash Course code examples use the following features from third-party libraries. Prior familiarity with these libraries is not necessary; you can look up what you need to know when you need it.

Matplotlib (http://matplotlib.org/contents.html) **(for data visualization)**

- `pyplot` (http://matplotlib.org/api/pyplot_api.html) module
- `cm` (http://matplotlib.org/api/cm_api.html) module
- `gridspec` (http://matplotlib.org/api/gridspec_api.html) module

Seaborn (http://seaborn.pydata.org/index.html) **(for heatmaps)**

- `heatmap` (http://seaborn.pydata.org/generated/seaborn.heatmap.html) function

pandas (http://pandas.pydata.org/) **(for data manipulation)**

- `DataFrame` (http://pandas.pydata.org/pandas-docs/stable/dsintro.html#dataframe) class

NumPy (http://www.numpy.org/) **(for low-level math operations)**

- `linspace` (https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linspace.html) function
- `random` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.random.html#numpy.random.random) function
- `array` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html) function
- `arange` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html) function

scikit-learn (http://scikit-learn.org/) **(for evaluation metrics)**

- metrics (http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics) module

Bash Terminal / Cloud Console

To run the programming exercises on your local machine or in a cloud console, you should be comfortable working on the command line:

- Bash Reference Manual (https://tiswww.case.edu/php/chet/bash/bashref.html)
- Bash Cheatsheet (https://github.com/LeCoupa/awesome-cheatsheets/blob/master/languages/bash.sh)
- Learn Shell (http://www.learnshell.org/)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Last updated March 5, 2019.

Framing: Key ML Terminology

Estimated Time: 8 minutes

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:

$$x_1, x_2, \ldots x_N$$

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?

Key Terms

- classification model (https://developers.google.com/machine-learning/glossary#classification_model)
- feature
- example (https://developers.google.com/machine-learning/glossary#example)
- inference

(https://developers.google.com/machine-learning/glossary#feature)

- label (https://developers.google.com/machine-learning/glossary#label)
- regression model (https://developers.google.com/machine-learning/glossary#regression_model)

(https://developers.google.com/machine-learning/glossary#inference)

- model (https://developers.google.com/machine-learning/glossary#model)
- training (https://developers.google.com/machine-learning/glossary#training)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture (https://developers.google.com/machine-learning/crash-course/framing/video-lecture)

Next
Check Your Understanding → (https://developers.google.com/machine-learning/crash-course/framing/check-your-understanding)

Framing: Check Your Understanding

Estimated Time: 5 minutes

Supervised Learning

Explore the options below.

Suppose you want to develop a supervised machine learning model to predict whether a given email is "spam" or "not spam." Which of the following statements are true?

✓ Emails not marked as "spam" or "not spam" are unlabeled examples.

Because our label consists of the values "spam" and "not spam", any email not yet marked as spam or not spam is an unlabeled example.

1 of 2 correct answers.

🚫 Words in the subject header will make good labels.

Words in the subject header might make excellent features, but they won't make good labels.

Try again.

🚫 We'll use unlabeled examples to train the model.

We'll use **labeled** examples to train the model. We can then run the trained model against unlabeled examples to infer whether the unlabeled email messages are spam or not spam.

Try again.

✓ The labels applied to some examples might be unreliable.

Definitely. It's important to check how reliable your data is. The labels for this dataset probably come from email users who mark particular email messages as spam. Since most users do not mark every suspicious email message as spam, we may have trouble knowing whether an email is spam. Furthermore, spammers could intentionally poison our model by providing faulty labels.

2 of 2 correct answers.

Features and Labels

Explore the options below.

Suppose an online shoe store wants to create a supervised ML model that will provide personalized shoe recommendations to users. That is, the model will recommend certain pairs of shoes to Marty and different pairs of shoes to Janet. Which of the following statements are true?

✓ "Shoe size" is a useful feature.

"Shoe size" is a quantifiable signal that likely has a strong impact on whether the user will like the recommended shoes. For example, if Marty wears size 9, the model shouldn't recommend size 7 shoes.

1 of 2 correct answers.

✓ "The user clicked on the shoe's description" is a useful label.

Users probably only want to read more about those shoes that they like. Clicks by users is, therefore, an observable, quantifiable metric that could serve as a good training label.

2 of 2 correct answers.

🚫 "Shoe beauty" is a useful feature.

Good features are concrete and quantifiable. Beauty is too vague a concept to serve as a useful feature. Beauty is probably a blend of certain concrete features, such as style and color. Style and color would each be better features than beauty.

Try again.

🚫 "Shoes that a user adores" is a useful label.

Adoration is not an observable, quantifiable metric. The best we can do is search for observable proxy metrics for adoration.

Try again.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Key ML Terminology
(https://developers.google.com/machine-learning/crash-course/framing/ml-terminology)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/descending-into-ml/video-lecture)

Last updated March 5, 2019.

Descending into ML: Linear Regression

Estimated Time: 6 minutes

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 cataloged data on chirps-per-minute and temperature. As a birthday gift, your Aunt Ruth gives you her cricket database and asks you to learn a model to predict this relationship. Using this data, you want to explore this relationship.

First, examine your data by plotting it:

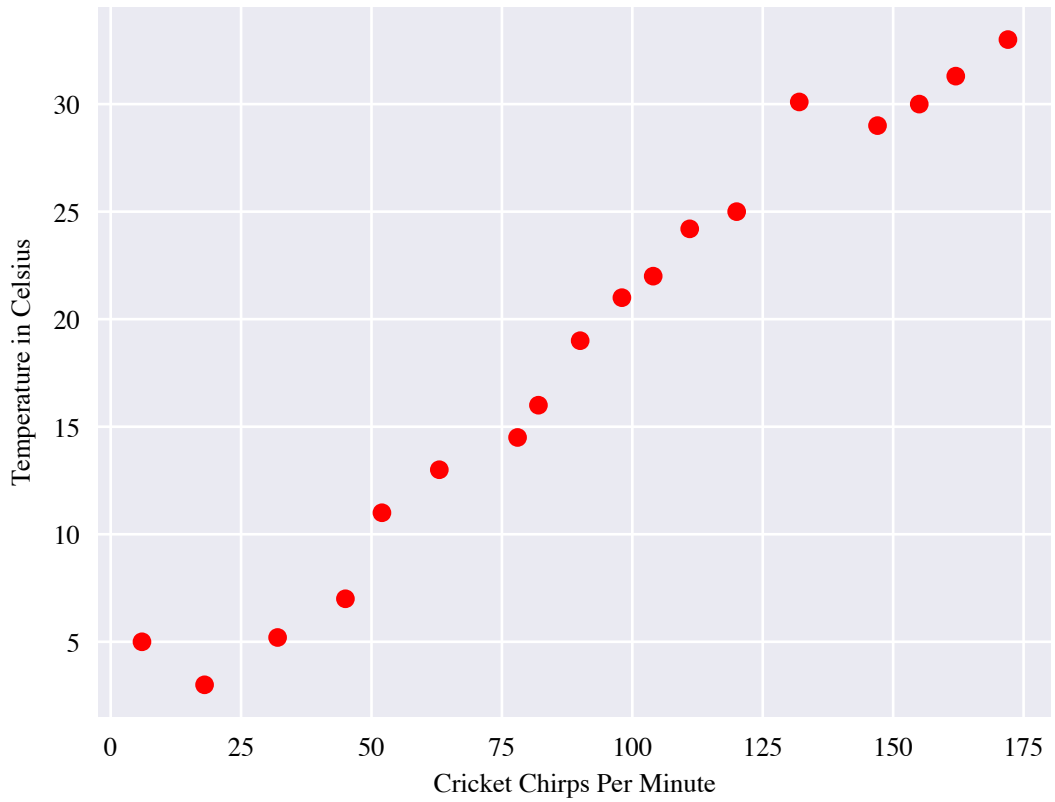


Figure 1. Chirps per Minute vs. Temperature in Celsius.

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

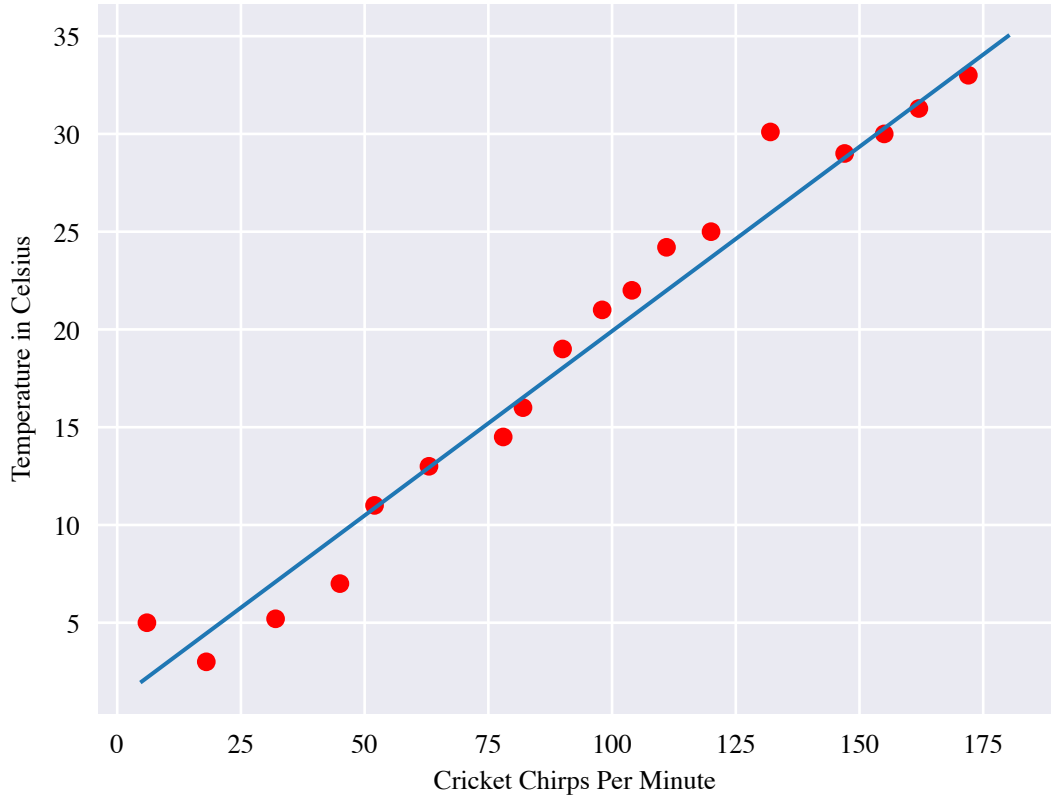


Figure 2. A linear relationship.

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

$$y = mx + b$$

where:

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

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

$$y' = b + w_1 x_1$$

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 w_0.
- w_1 is the weight of feature 1. Weight is the same concept as the "slope" m in the traditional equation of a line.
- x_1 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 x_1, just substitute the x_1 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 (w_1, w_2, etc.). For example, a model that relies on three features might look as follows:

$$y' = b + w_1 x_1 + w_2 x_2 + w_3 x_3$$

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)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture (https://developers.google.com/machine-learning/crash-course/descending-into-ml/video-lecture)

Next

Training and Loss →
(https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss)

Last updated March 5, 2019.

Descending into ML: Training and Loss

Estimated Time: 6 minutes

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.

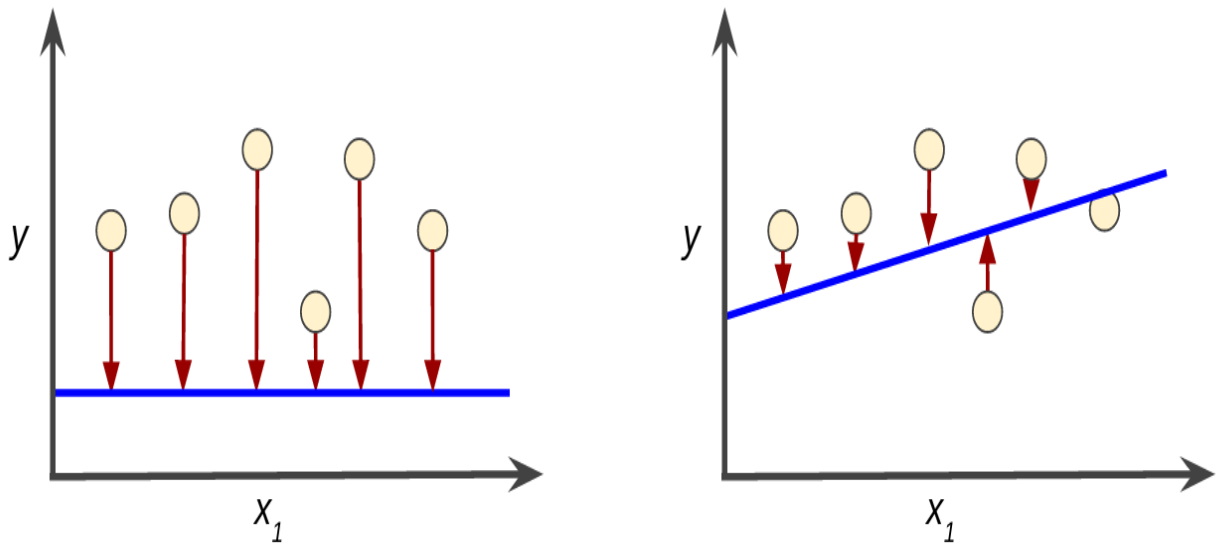


Figure 3. High loss in the left model; low loss in the right model.

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 **L_2 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:

$$MSE = \frac{1}{N} \sum_{(x,y) \in D} (y - prediction(x))^2$$

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.

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)
- training (https://developers.google.com/machine-learning/glossary#training)
- squared loss (https://developers.google.com/machine-learning/glossary#squared_loss)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Linear Regression (https://developers.google.com/machine-learning/crash-course/descending-into-ml/linear-regression)

Next
Check Your Understanding → (https://developers.google.com/machine-learning/crash-course/descending-into-ml/check-your-understanding)

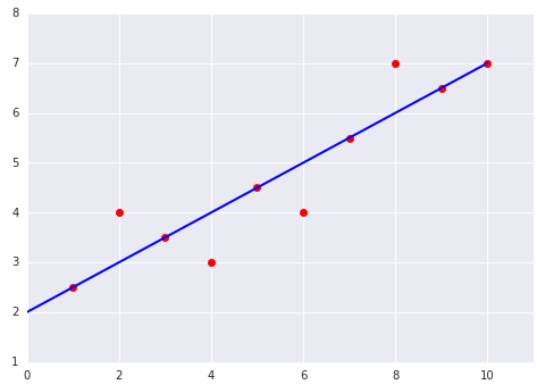
Last updated March 5, 2019.

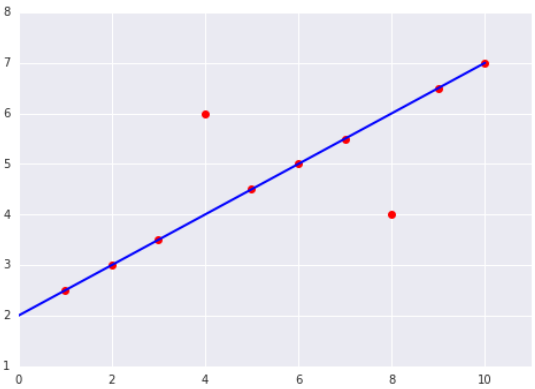
Descending into ML: Check Your Understanding

Estimated Time: 5 minutes

Mean Squared Error

Consider the following two plots:





Explore the options below.

Which of the two data sets shown in the preceding plots has the **higher** Mean Squared Error (MSE)?

✓ The dataset on the right.

The eight examples on the line incur a total loss of 0. However, although only two points lay off the line, both of those points are *twice* as far off the line as the outlier points in the left figure. Squared loss amplifies those differences, so an offset of two incurs a loss four times as great as an offset of one.

$$MSE = \frac{0^2 + 0^2 + 0^2 + 2^2 + 0^2 + 0^2 + 0^2 + 2^2 + 0^2 + 0^2}{10} = 0.8$$

Correct answer.

🚫 The dataset on the left.

The six examples on the line incur a total loss of 0. The four examples not on the line are not very far off the line, so even squaring their offset still yields a low value:

$$MSE = \frac{0^2 + 1^2 + 0^2 + 1^2 + 0^2 + 1^2 + 0^2 + 1^2 + 0^2 + 0^2}{10} = 0.4$$

Try again.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Training and Loss
(https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/reducing-loss/video-lecture)

Reducing Loss: An Iterative Approach

Estimated Time: 10 minutes

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 w_1 is 0.") and wait for the system to tell you what the loss is. Then, you'll try another guess ("The value of w_1 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:

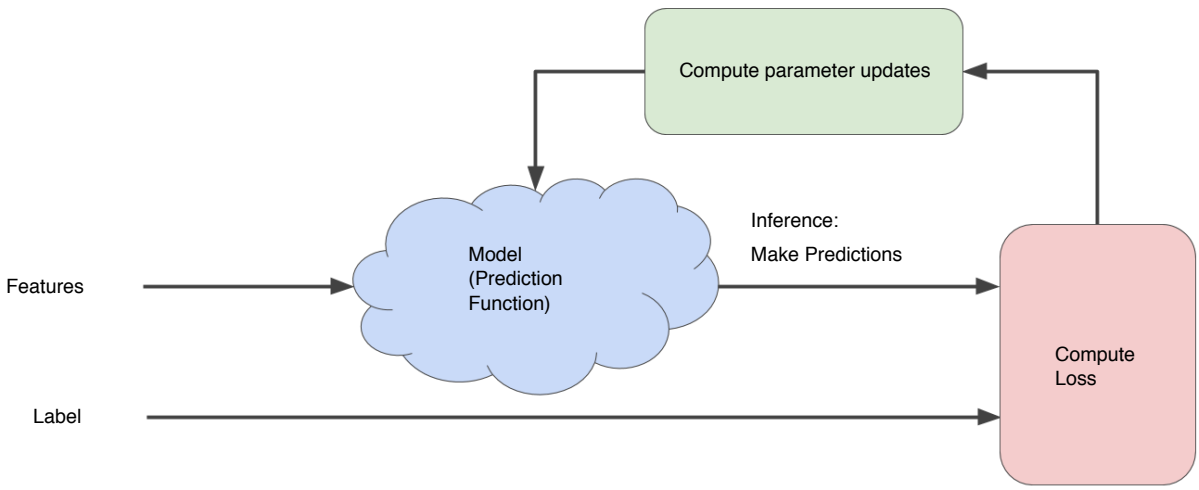


Figure 1. An iterative approach to training a model.

We'll use this same iterative approach throughout Machine Learning Crash Course, detailing various complications, particularly within that stormy cloud labeled "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:

$$y' = b + w_1 x_1$$

What initial values should we set for b and w_1? 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
- w_1 = 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 w_1. 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)
- training (https://developers.google.com/machine-learning/glossary#training)
- loss (https://developers.google.com/machine-learning/glossary#loss)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/reducing-loss/video-lecture)

Next
Gradient Descent →
(https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

Reducing Loss: Gradient Descent

Estimated Time: 10 minutes

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 w_1. For the kind of regression problems we've been examining, the resulting plot of loss vs. w_1 will always be convex. In other words, the plot will always be bowl-shaped, kind of like this:

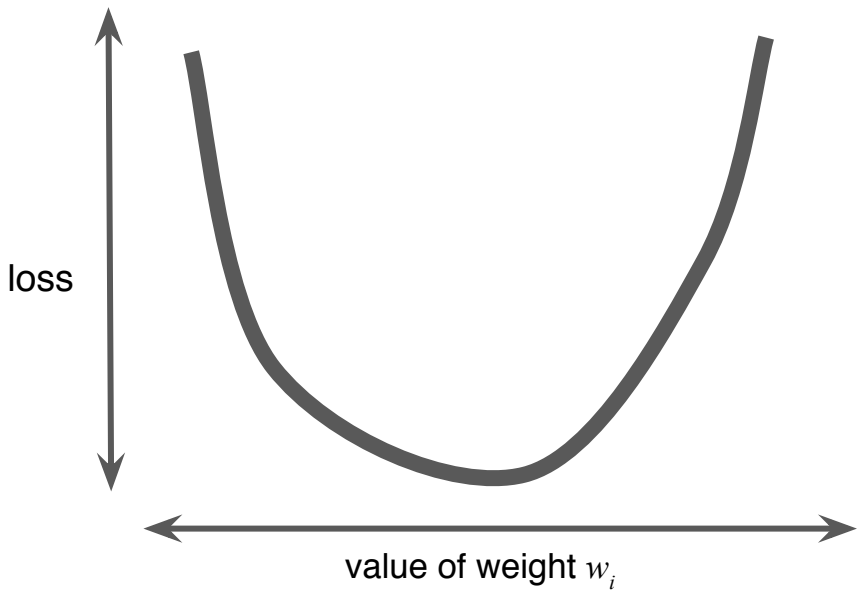


Figure 2. Regression problems yield convex loss vs weight plots.

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 w_1 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 w_1. The starting point doesn't matter much; therefore, many algorithms simply set w_1 to 0 or pick a random value. The following figure shows that we've picked a starting point slightly greater than 0:

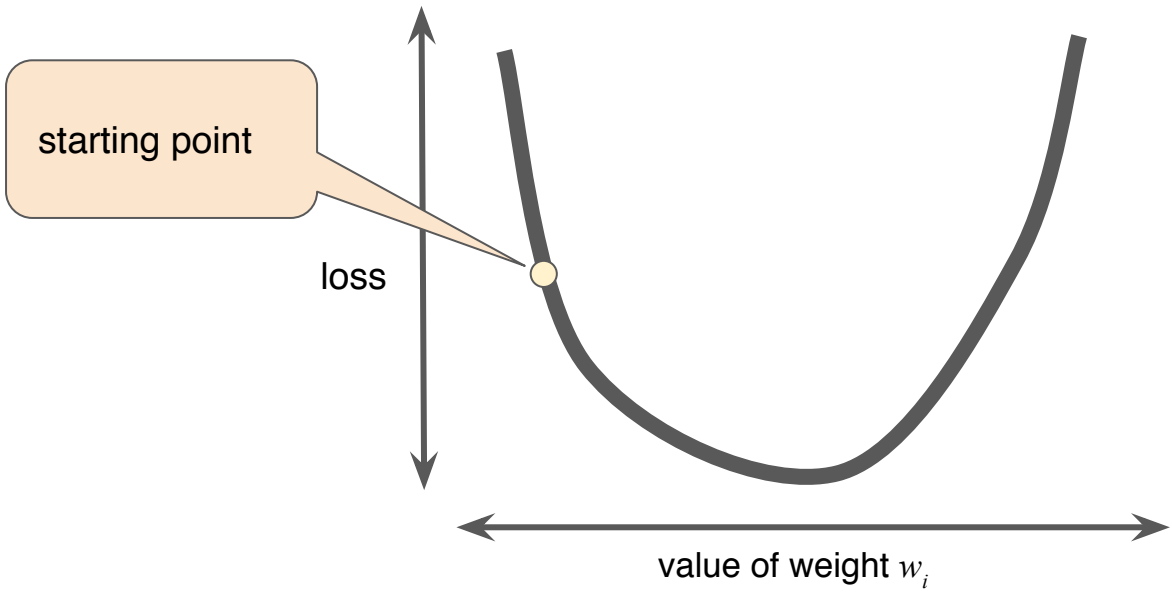


Figure 3. A starting point for gradient descent.

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.

Click the dropdown arrow to learn more about partial derivatives and gradients.

The math around machine learning is fascinating and we're delighted that you clicked the link to learn more. Please note, however, that TensorFlow handles all the gradient computations for you, so you don't actually have to understand the calculus provided here.

Partial derivatives

A **multivariable function** is a function with more than one argument, such as:

$$f(x, y) = e^{2y} \sin(x)$$

The **partial derivative f with respect to x**, denoted as follows:

$$\frac{\partial f}{\partial x}$$

is the derivative of f considered as a function of x alone. To find the following:

$$\frac{\partial f}{\partial x}$$

you must hold y constant (so f is now a function of one variable x), and take the regular derivative of f with respect to x. For example, when y is fixed at 1, the preceding function becomes:

$$f(x) = e^2 \sin(x)$$

This is just a function of one variable x, whose derivative is:

$$e^2 \cos(x)$$

In general, thinking of y as fixed, the partial derivative of f with respect to x is calculated as follows:

$$\frac{\partial f}{\partial x}(x, y) = e^{2y} \cos(x)$$

Similarly, if we hold x fixed instead, the partial derivative of f with respect to y is:

$$\frac{\partial f}{\partial y}(x, y) = 2e^{2y} \sin(x)$$

Intuitively, a partial derivative tells you how much the function changes when you perturb one variable a bit. In the preceding example:

$$\frac{\partial f}{\partial x}(0, 1) = e^2 \approx 7.4$$

So when you start at $(0, 1)$, hold y constant, and move x a little, f changes by about 7.4 times the amount that you changed x.

In machine learning, partial derivatives are mostly used in conjunction with the gradient of a function.

Gradients

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

$$\nabla f$$

For instance, if:

$$f(x,y) = e^{2y}\sin(x)$$

then:

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}(x,y), \frac{\partial f}{\partial y}(x,y)\right) = (e^{2y}\cos(x), 2e^{2y}\sin(x))$$

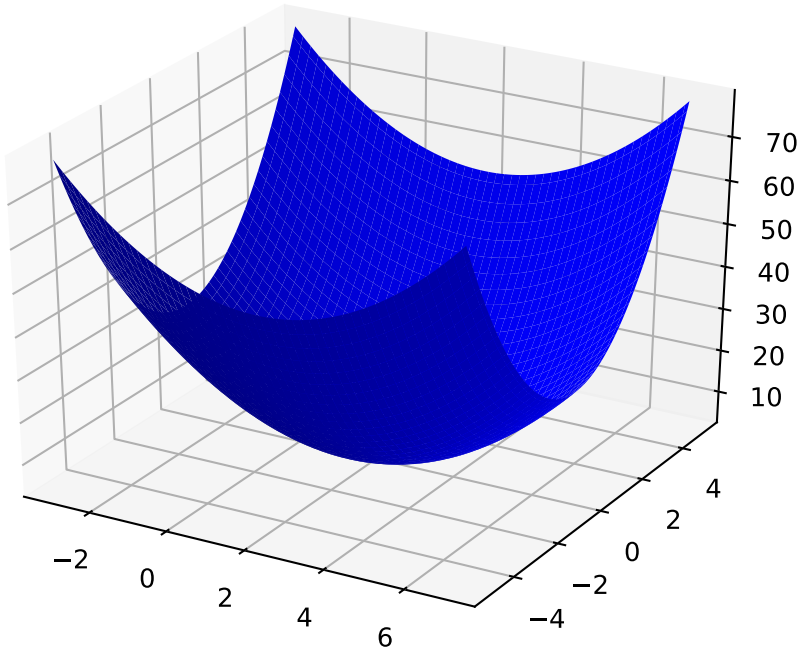
Note the following:

∇f	Points in the direction of greatest increase of the function.
$-\nabla f$	Points in the direction of greatest decrease of the function.

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 $f(x,y)$:

$$f(x,y) = 4 + (x-2)^2 + 2y^2$$

when viewed in three dimensions with $z = f(x,y)$ looks like a valley with a minimum at $(2,0,4)$:



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.

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.

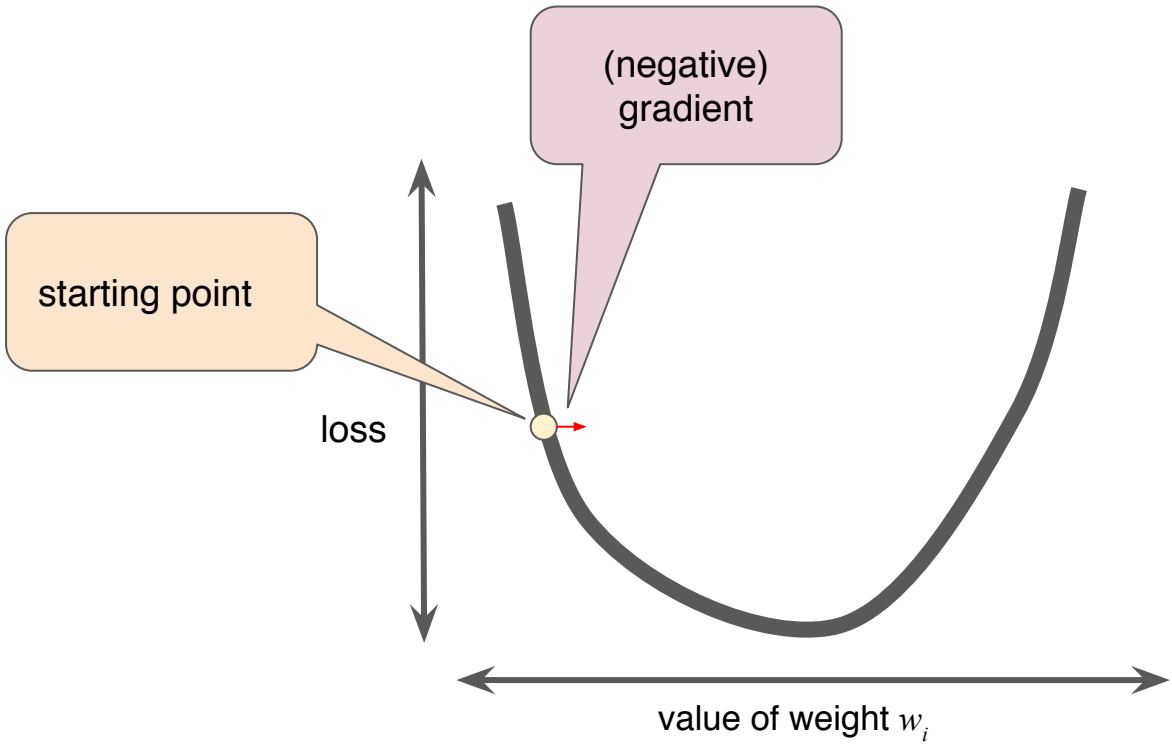


Figure 4. Gradient descent relies on negative gradients.

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:

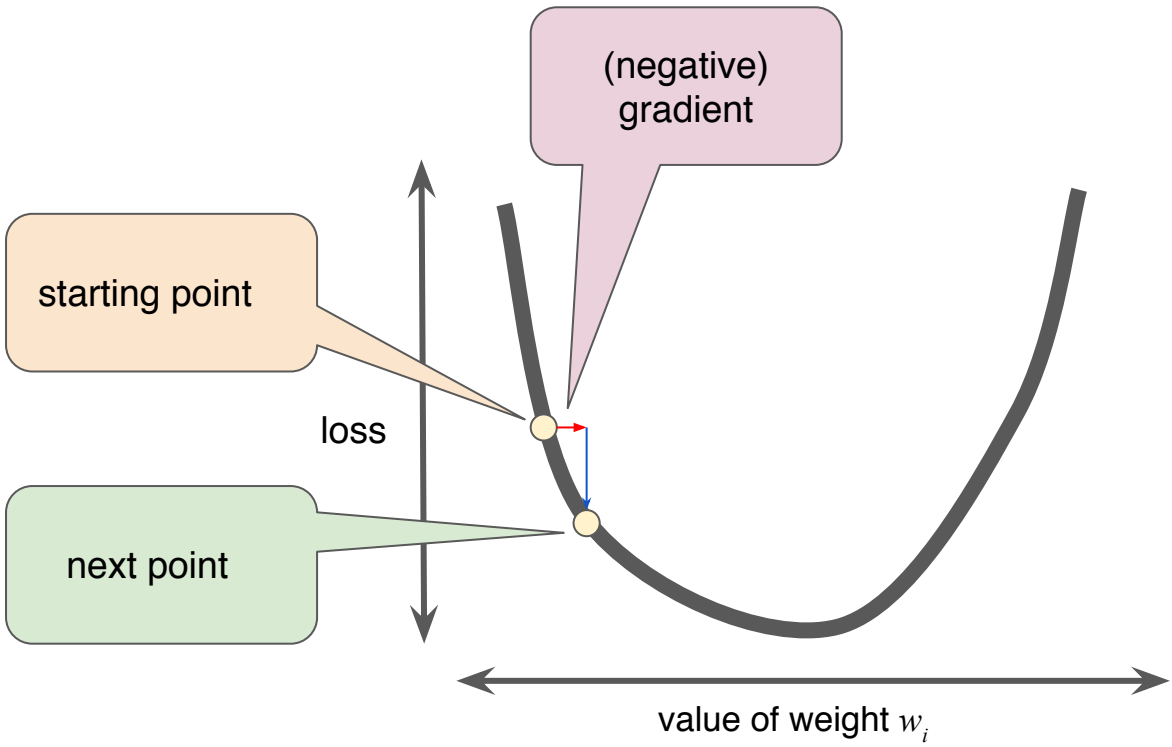


Figure 5. A gradient step moves us to the next point on the loss curve.

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 w_1 and the bias b, we calculate the gradients with respect to both w_1 and b. Next, we modify the values of w_1 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)

Previous
← An Iterative Approach (https://developers.google.com/machine-learning/crash-course/reducing-loss/an-iterative-approach)

Next
Learning Rate → (https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate)

Reducing Loss: Learning Rate

Estimated Time: 5 minutes

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:

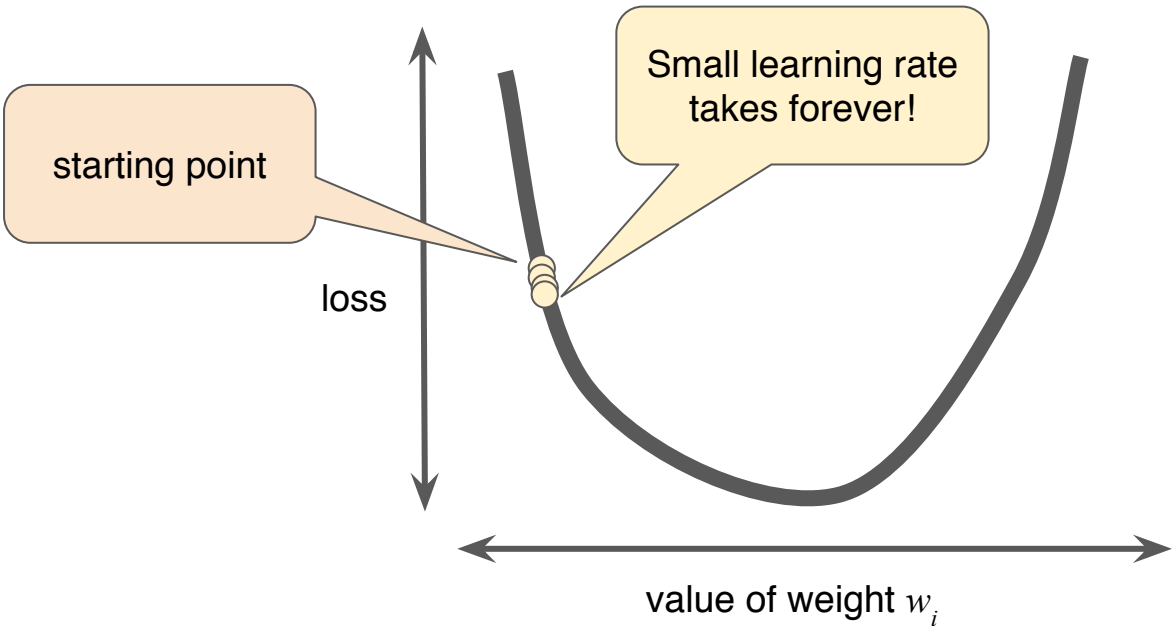


Figure 6. Learning rate is too small.

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:

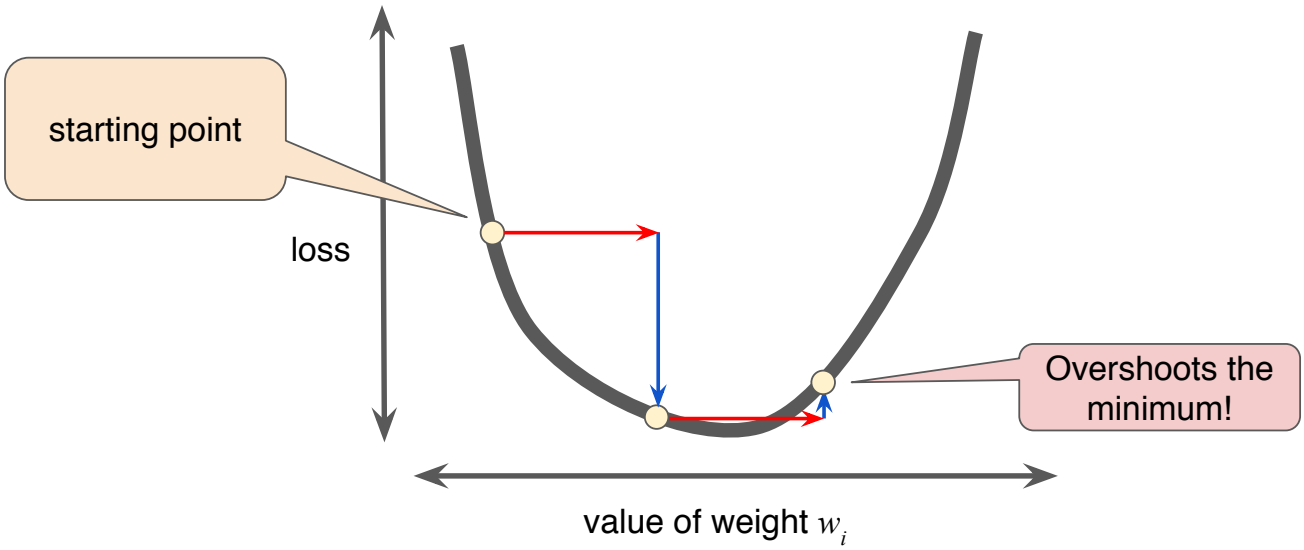


Figure 7. Learning rate is too large.

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.

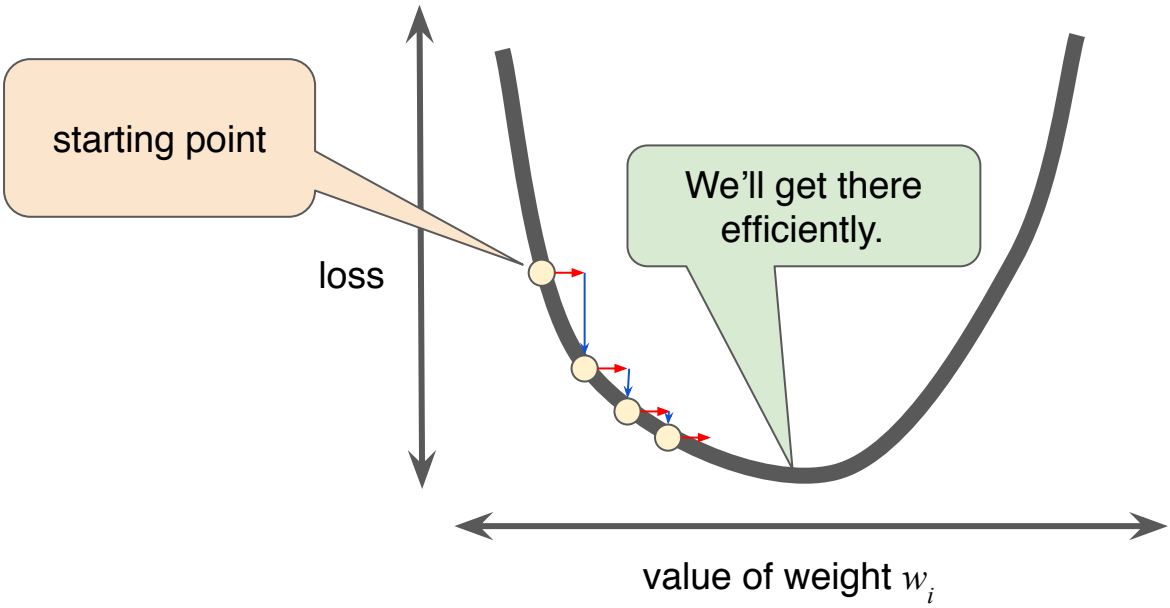


Figure 8. Learning rate is just right.

Click the dropdown arrow to learn more about the ideal learning rate.

The ideal learning rate in one-dimension is $\frac{1}{f(x)''}$ (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.

Key Terms

- hyperparameter (https://developers.google.com/machine-learning/glossary#hyperparameter)
- step size (https://developers.google.com/machine-learning/glossary#step_size)
- learning rate (https://developers.google.com/machine-learning/glossary#learning_rate)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Gradient Descent
(https://developers.google.com/machine-learning/crash-course/reducing-loss/gradient-descent)

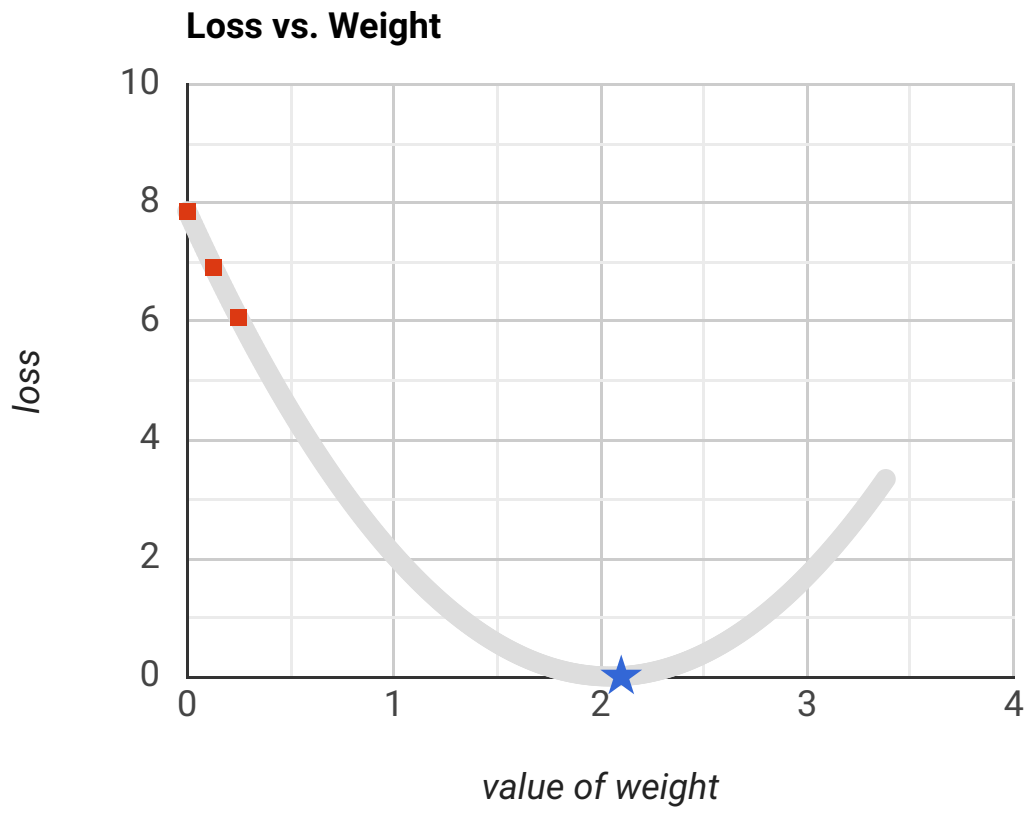
Next
Optimizing Learning Rate →
(https://developers.google.com/machine-learning/crash-course/fitter/graph)

Reducing Loss: Optimizing Learning Rate

Estimated Time: 15 minutes

Experiment with different learning rates and see how they affect the number of steps require
minimum of the loss curve. Try the exercises below the graph.

Set learning rate:		0.10
Execute single step:	STEP	2
Reset the graph:	RESET	



Exercise 1

Set a learning rate of 0.1 on the slider. Keep hitting the STEP button until the gradient descent algorithm reaches the minimum point of the loss curve. How many steps did it take?

Solution

Gradient descent reaches the minimum of the curve in 81 steps.

Exercise 2

Can you reach the minimum more quickly with a higher learning rate? Set a learning rate of 1, and keep hitting STEP until gradient descent reaches the minimum. How many steps did it take this time?

Solution

Gradient descent reaches the minimum of the curve in 6 steps.

Exercise 3

How about an even larger learning rate. Reset the graph, set a learning rate of 4, and try to reach the minimum of the loss curve. What happened this time?

Solution

Gradient descent **never reaches the minimum**. As a result, steps progressively increase in size. Each step jumps back and forth across the bowl, climbing the curve instead of descending to the bottom.

Optional Challenge

Can you find the Goldilocks (https://wikipedia.org/wiki/Goldilocks_principle) learning rate for this curve, where gradient descent reaches the minimum point in the fewest number of steps?

What is the fewest number of steps required to reach the minimum?

Solution

The Goldilocks learning rate for this data is 1.6, which reaches the minimum in 1 step.

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.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Learning Rate
(https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate)

Next
Stochastic Gradient Descent →
(https://developers.google.com/machine-learning/crash-course/reducing-loss/stochastic-gradient-descent)

Reducing Loss: Stochastic Gradient Descent

Estimated Time: 3 minutes

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)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Optimizing Learning Rate
(https://developers.google.com/machine-learning/crash-course/fitter/graph)

Next
Playground Exercise →
(https://developers.google.com/machine-learning/crash-course/reducing-loss/playground-exercise)

Last updated March 5, 2019.

Reducing Loss: Playground Exercise

Estimated Time: 10 minutes

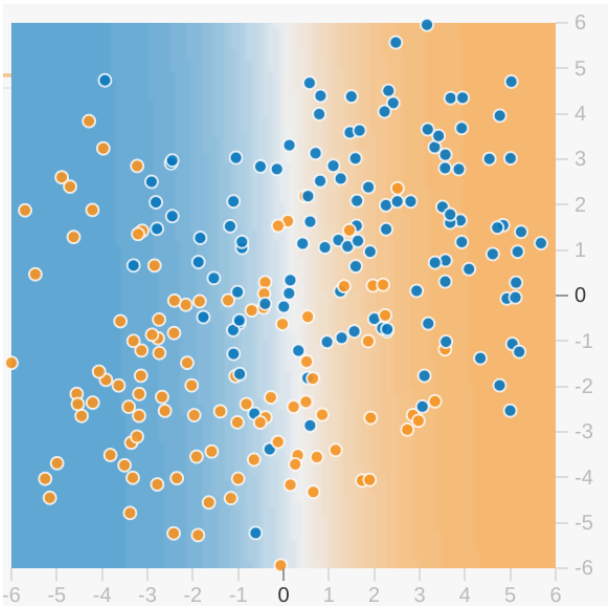
Learning Rate and Convergence

This is the first of several Playground exercises. Playground (http://playground.tensorflow.org) is a program developed especially for this course to teach machine learning principles.

Each Playground exercise generates a dataset. The label for this dataset has two possible values. You could think of those two possible values as spam vs. not spam or perhaps healthy trees vs. sick trees. The goal of most exercises is to tweak various hyperparameters to build a model that successfully classifies (separates or distinguishes) one label value from the other. Note that most data sets contain a certain amount of noise that will make it impossible to successfully classify *every* example.

Click the dropdown arrow for an explanation of model visualization.

Each Playground exercise displays a visualization of the current state of the model. For example, here's a visualization:



Note the following about the model visualization:

- Each axis represents a specific feature. In the case of spam vs. not spam, the features could be the word count and the number of recipients of the email.

★ **Note:** Appropriate axis values will depend on feature data. The axis values shown above would not make sense for word count or number of recipients, as neither can be negative.

- Each dot plots the feature values for one example of the data, such as an email.
- The color of the dot represents the class that the example belongs to. For example, the blue dots can represent non-spam emails while the orange dots can represent spam emails.
- The background color represents the model's prediction of where examples of that color should be found. A blue background around a blue dot means that the model is correctly predicting that example. Conversely, an orange background around a blue dot means that the model is incorrectly predicting that example.
- The background blues and oranges are scaled. For example, the left side of the visualization is solid blue but gradually fades to white in the center of the visualization. You can think of the color strength as suggesting the model's confidence in its guess. So solid blue means that the model is very confident about its guess and light blue means that the model is less confident. (The model visualization shown in the figure is doing a poor job of prediction.)

Use the visualization to judge your model's progress. ("Excellent—most of the blue dots have a blue background" or "Oh no! The blue dots have an orange background.") Beyond the colors, Playground also displays the model's current loss numerically. ("Oh no! Loss is going up instead of down.")

The interface for this exercise provides three buttons:

Icon	Name	What it Does
↺	Reset	Resets `Iterations` to 0. Resets any weights that model had already learned.
▶|	Step	Advance one iteration. With each iteration, the model changes—sometimes subtly and sometimes dramatically.
REGENERATE	Regenerate	Generates a new data set. Does not reset `Iterations`.

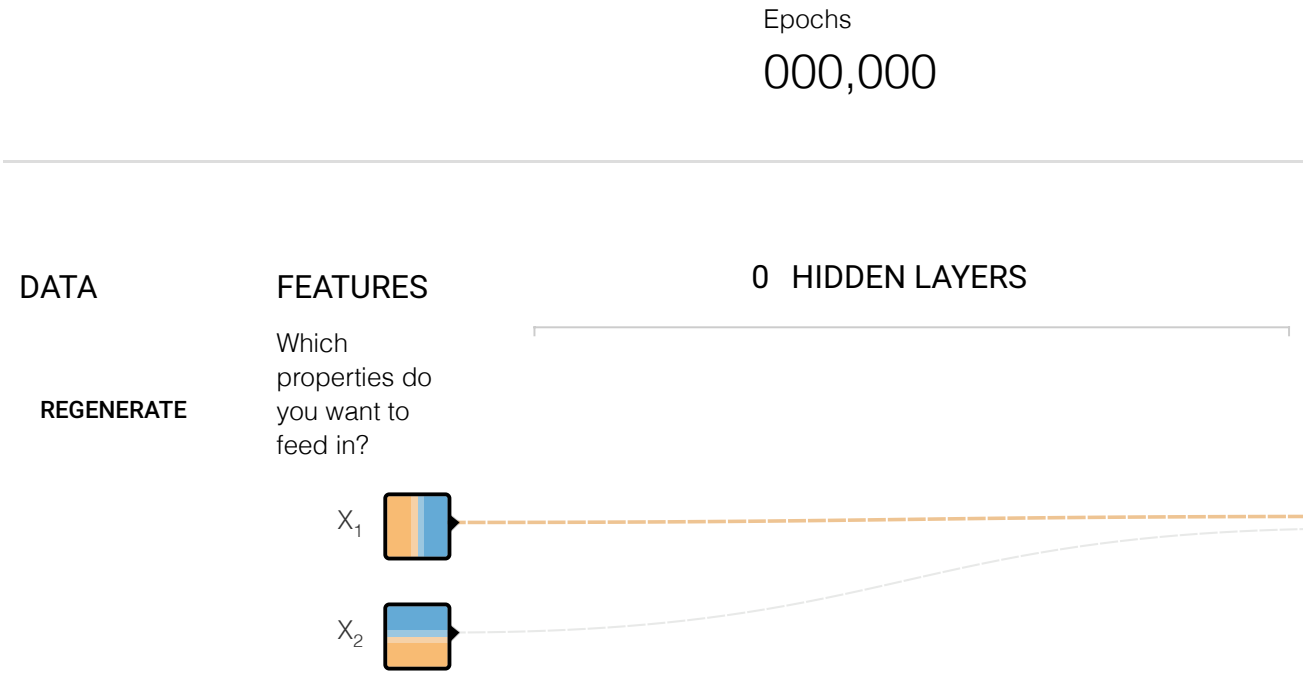
In this first Playground exercise, you'll experiment with learning rate by performing two tasks.

Task 1: Notice the **Learning rate** menu at the top-right of Playground. The given Learning rate—3—is very high. Observe how that high Learning rate affects your model by clicking the "Step" button 10 or 20 times. After each early iteration, notice how the model visualization changes dramatically. You might even see some instability *after* the model appears to have converged. Also notice the lines running from x_1 and x_2 to the model visualization. The weights of these lines indicate the weights of those features in the model. That is, a thick line indicates a high weight.

Task 2: Do the following:

1. Press the **Reset** button.
2. Lower the `Learning rate`.
3. Press the Step button a bunch of times.

How did the lower learning rate impact convergence? Examine both the number of steps needed for the model to converge, and also how smoothly and steadily the model converges. Experiment with even lower values of learning rate. Can you find a learning rate too slow to be useful? (You'll find a discussion just below the exercise.)



Click the dropdown arrow for a discussion about Task 2.

Due to the non-deterministic nature of Playground exercises, we can't always provide answers that will correspond exactly with your data set. That said, a learning rate of `0.1` converged efficiently for us. Smaller learning rates took much longer to converge; that is, smaller learning rates were too slow to be useful.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Stochastic Gradient Descent
(https://developers.google.com/machine-learning/crash-course/reducing-loss/stochastic-gradient-descent)

Next
Check Your Understanding →
(https://developers.google.com/machine-learning/crash-course/reducing-loss/check-your-understanding)

Last updated March 5, 2019.

Reducing Loss: Check Your Understanding

Estimated Time: 2 minutes

Check Your Understanding: Batch Size

Explore the options below.

When performing gradient descent on a large data set, which of the following batch sizes will likely be more efficient?

✓ A small batch or even a batch of one example (SGD).

Amazingly enough, performing gradient descent on a small batch or even a batch of one example is usually more efficient than the full batch. After all, finding the gradient of one example is far cheaper than finding the gradient of millions of examples. To ensure a good representative sample, the algorithm scoops up another random small batch (or batch of one) on every iteration.

Correct answer.

🚫 The full batch.

Computing the gradient from a full batch is inefficient. That is, the gradient can usually be computed far more efficiently (and just as accurately) from a smaller batch than from a vastly bigger full batch.

Try again.

Key Terms

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

(https://developers.google.com/machine-learning/glossary#mini-batch)

(https://developers.google.com/machine-learning/glossary#SGD)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Playground Exercise
(https://developers.google.com/machine-learning/crash-course/reducing-loss/playground-exercise)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/video-lecture)

Last updated March 5, 2019.

First Steps with TensorFlow: Toolkit

Estimated Time: 4 minutes

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:

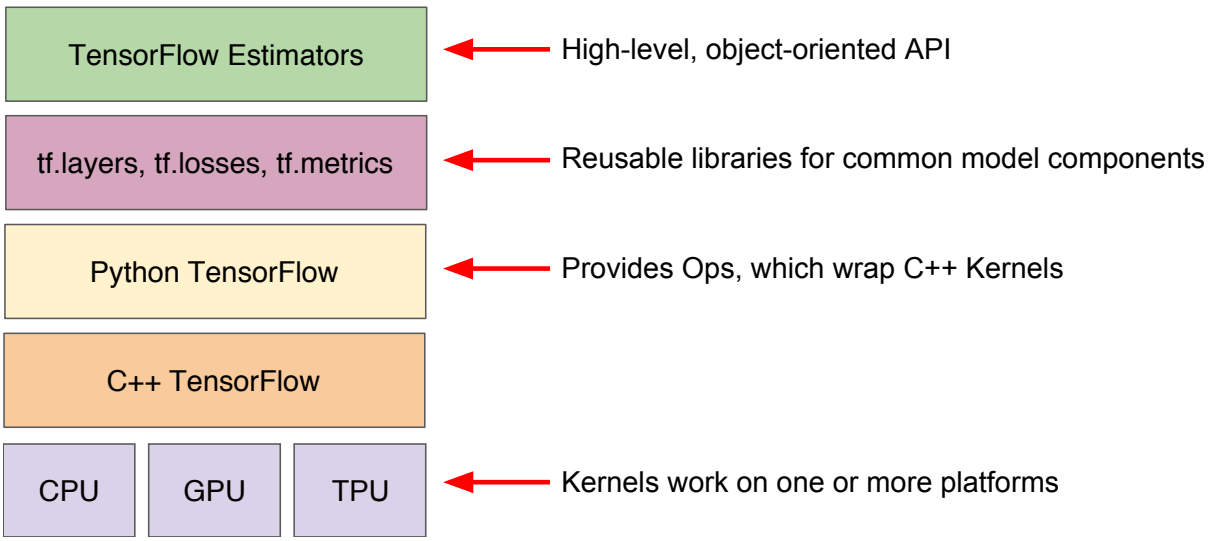


Figure 1. TensorFlow toolkit hierarchy.

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
- graph

(https://developers.google.com/machine-learning/glossary#Estimators) (https://developers.google.com/machine-learning/glossary#graph)

- tensor (https://developers.google.com/machine-learning/glossary#tensor)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/video-lecture)

Next
Programming Exercises →
(https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/programming-exercises)

Last updated March 5, 2019.

First Steps with TensorFlow: Programming Exercises

Estimated Time: 55 minutes

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 modeling, 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.
2. 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.
3. 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:

$$total\ number\ of\ trained\ examples = batch\ size * steps$$

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:

$$number\ of\ training\ examples\ in\ each\ period = \frac{batch\ size * steps}{periods}$$

Key Terms

Estimators (https://developers.google.com/machine-learning/glossary#Estimators)

operation (https://developers.google.com/machine-learning/glossary#Operation)

tensor (https://developers.google.com/machine-learning/glossary#tensor)

5. graph (https://developers.google.com/machine-learning/glossary#graph)
7. session (https://developers.google.com/machine-learning/glossary#session)

NEWEST TENSORFLOW QUESTIONS ON STACK OVERFLOW (HTTP://STACKOVERFLOW.COM/QUE

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Toolkit
(https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/generalization/video-lecture)

Last updated May 17, 2019.

Generalization: Peril of Overfitting

Estimated Time: 10 minutes

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.

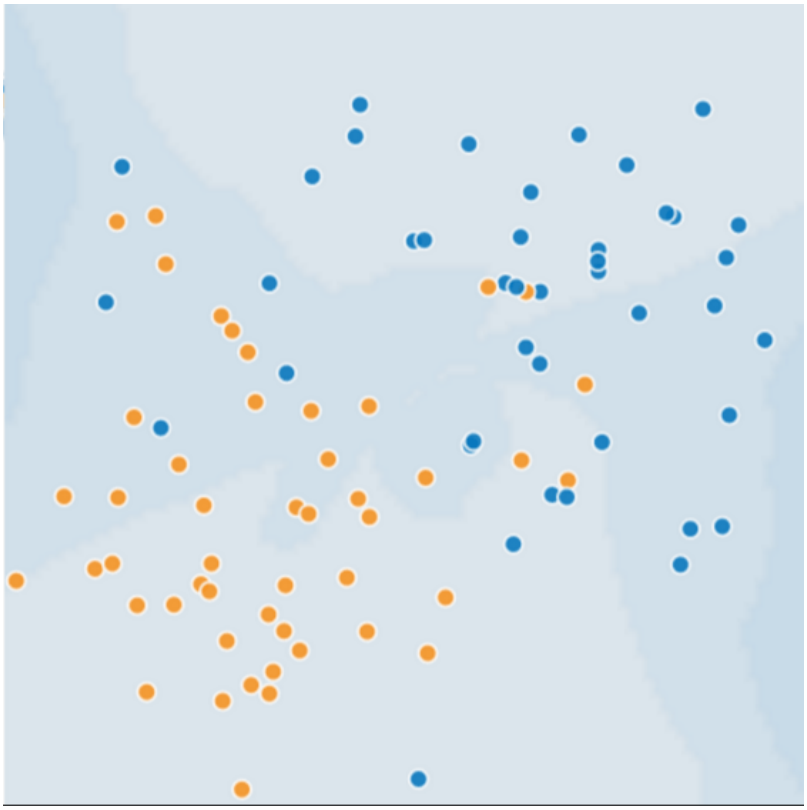


Figure 1. Sick (blue) and healthy (orange) trees.

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.

Click the dropdown arrow to see Figure 2.

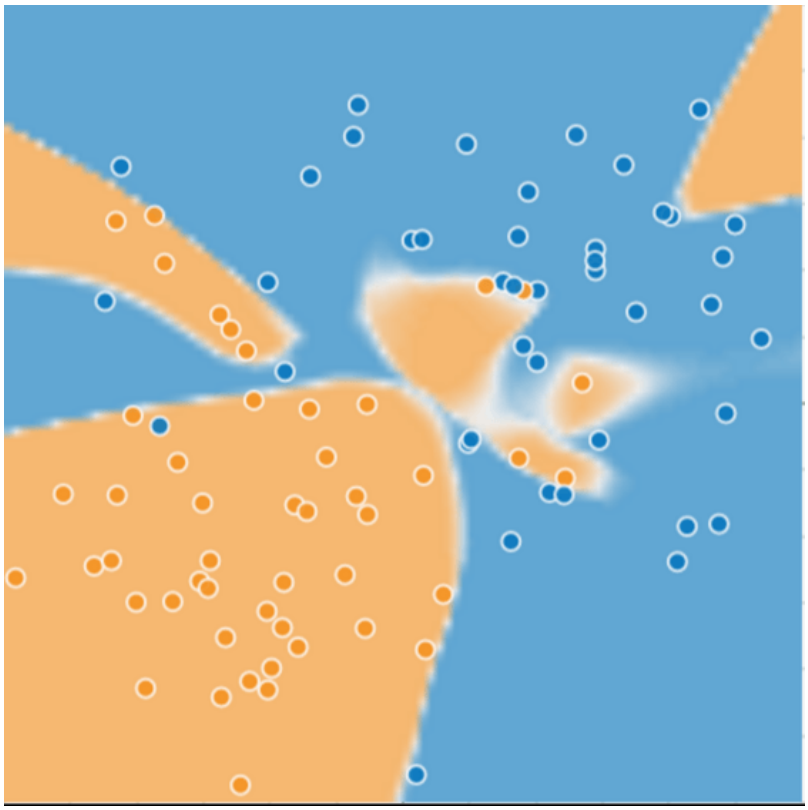


Figure 2. A complex model for distinguishing sick from healthy trees.

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.

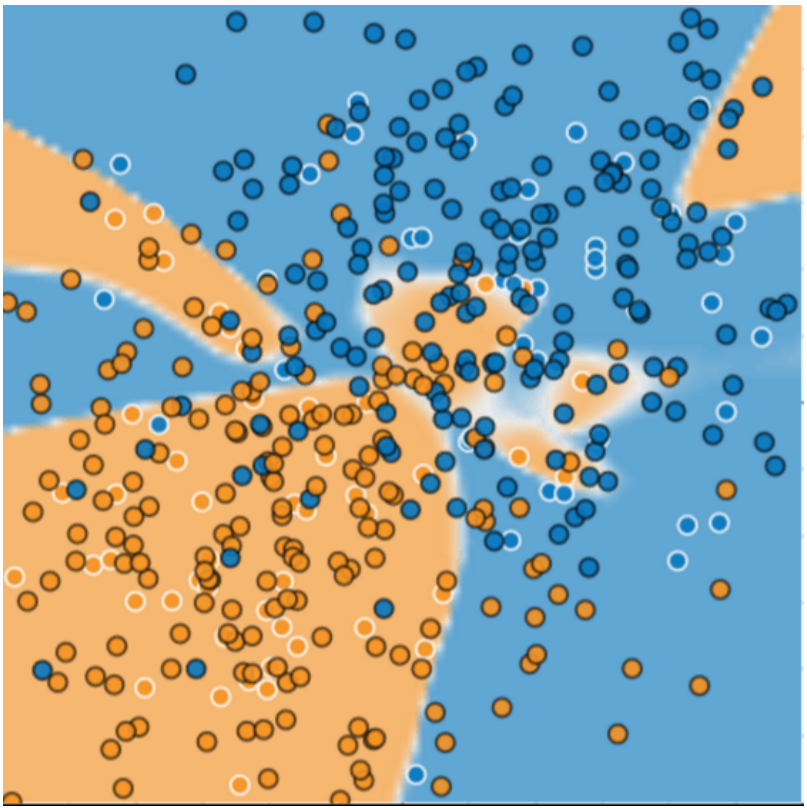


Figure 3. The model did a bad job predicting new data.

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)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/generalization/video-lecture)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/training-and-test-sets/video-lecture)

Training and Test Sets: Splitting Data

Estimated Time: 8 minutes

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:

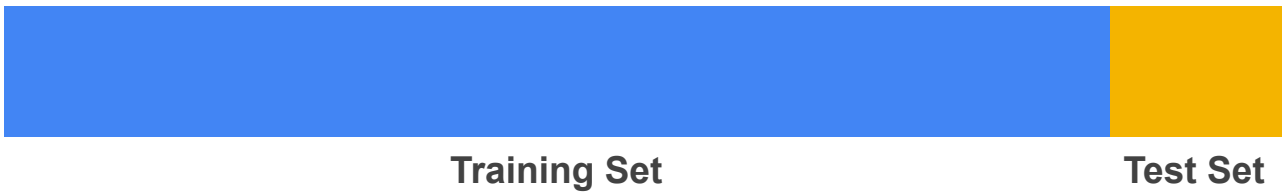
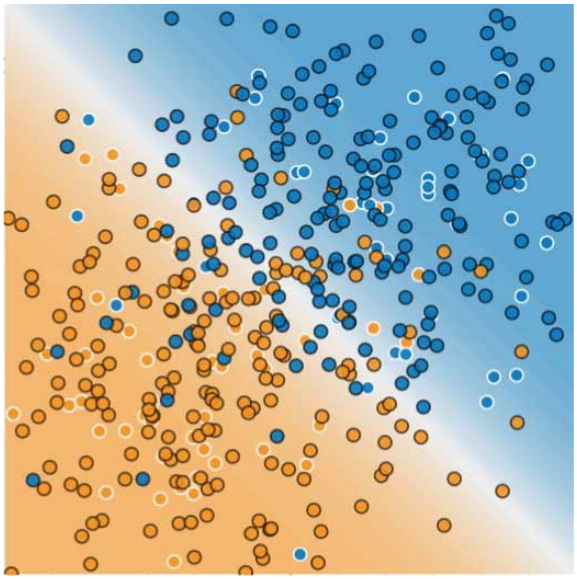


Figure 1. Slicing a single data set into a training set and test set.

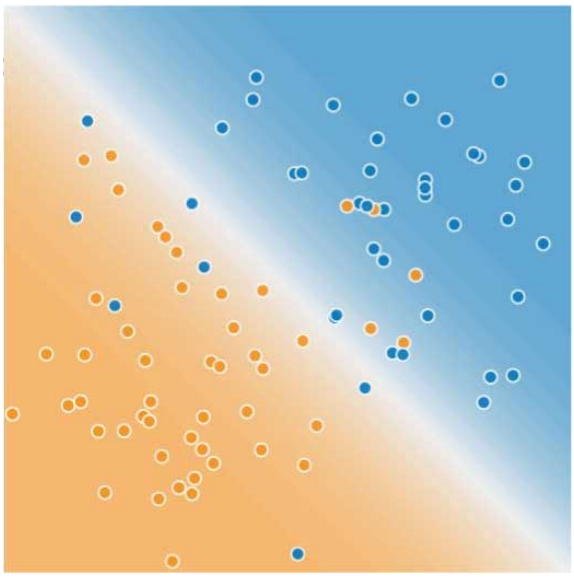
Make sure that your test set meets the following two conditions:

- 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.



Training Data



Test 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)
- training set (https://developers.google.com/machine-learning/glossary#training_set)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/training-and-test-sets/video-lecture)

Next
Playground Exercise →
(https://developers.google.com/machine-learning/crash-course/training-and-test-sets/playground-exercise)

Training and Test Sets: Playground Exercise

Estimated Time: 15 minutes

Training Sets and Test Sets

We return to Playground to experiment with training sets and test sets.

Click the dropdown arrow for a reminder of what the orange and blue dots mean.

In the visualization:

- Each blue dot signifies one example of one class of data (for example, spam).
- Each orange dot signifies one example of another class of data (for example, not spam).
- The background color represents the model's prediction of where examples of that color should be found. A blue background around a blue dot means that the model is correctly predicting that example. Conversely, an orange background around a blue dot means that the model is making an incorrect prediction for that example.

This exercise provides both a test set and a training set, both drawn from the same data set. By default, the visualization shows only the training set. If you'd like to also see the test set, click the **Show test data** checkbox just below the visualization. In the visualization, note the following distinction:

- The training examples have a white outline.
- The test examples have a black outline.

Task 1: Run Playground with the given settings by doing the following:

1. Click the Run/Pause button: 
2. Watch the Test loss and Training loss values change.
3. When the Test loss and Training loss values stop changing or only change once in a while, press the Run/Pause button again to pause Playground.

Note the delta between the Test loss and Training loss. We'll try to reduce this delta in the following tasks.

Task 2: Do the following:

1. Press the Reset button. 
2. Modify the Learning rate (https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate).
3. Press the Run/Pause button: 
4. Let Playground run for at least 150 epochs.

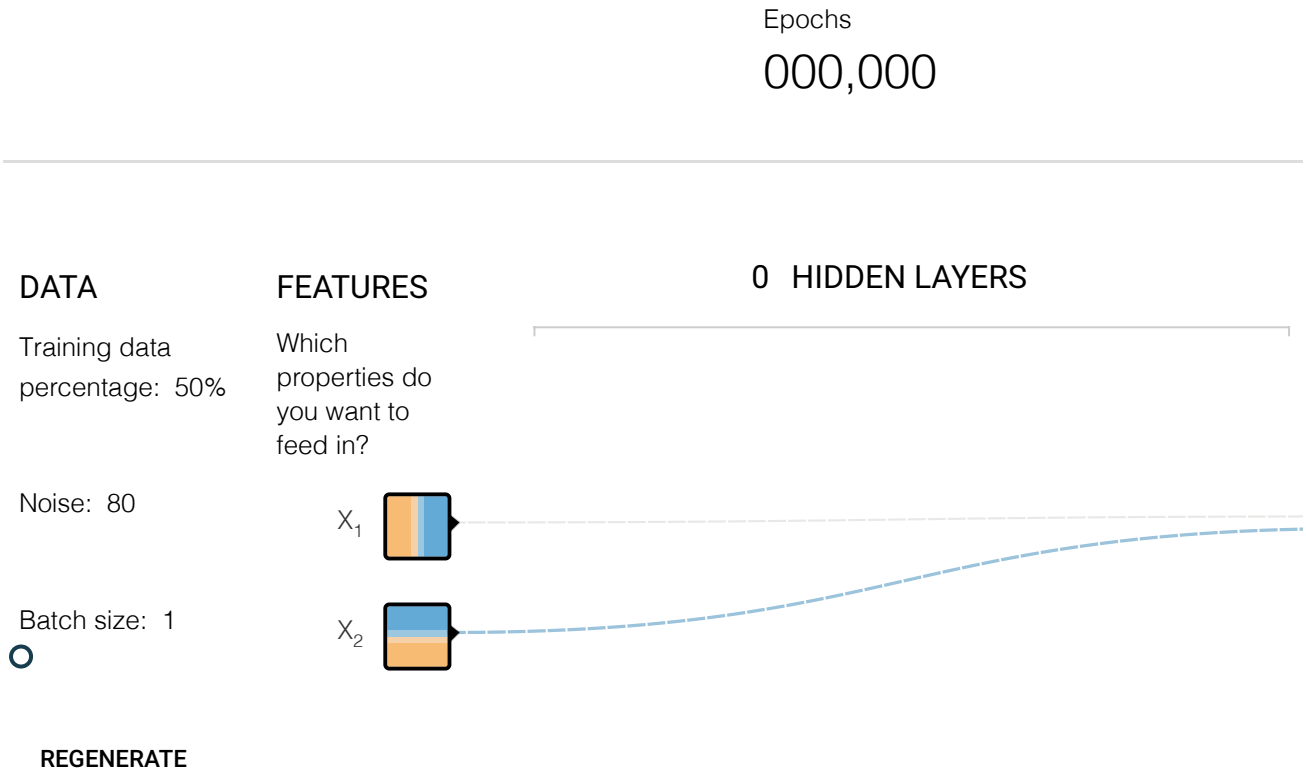
Is the delta between Test loss and Training loss lower or higher with this new Learning rate? What happens if you modify *both* Learning rate and batch size (https://developers.google.com/machine-learning/glossary#batch_size)?

Optional Task 3: A slider labeled **Training data percentage** lets you control the proportion of training data to test data. For example, when set to 90%, then 90% of the data is used for the training set and the remaining 10% is used for the test set.

Do the following:

1. Reduce the "Training data percentage" from 50% to 10%.
2. Experiment with Learning rate and Batch size, taking notes on your findings.

Does altering the training data percentage change the optimal learning settings that you discovered in Task 2? If so, why?



Click the dropdown arrow for the answer to Task 1.

With learning rate set to 3 (the initial setting), Test loss is significantly higher than Training loss.

Click the dropdown arrow for the answer to Task 2.

By reducing learning rate (for example, to `0.001`), Test loss drops to a value much closer to Training loss. In most runs, increasing Batch size does not influence Training loss or Test loss significantly. However, in a small percentage of runs, increasing Batch size to `20` or greater causes Test loss to drop slightly below Training loss.

Playground's data sets are randomly generated. Consequently, our answers may not always agree exactly with yours.

Click the dropdown arrow for the answer to Task 3.

Reducing the Training data percentage from 50% to 10% dramatically lowers the number of data points in the training set. With so little data, high batch size and high learning rate cause the training model to jump around chaotically (jumping repeatedly over the minimum point).

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Splitting Data
(https://developers.google.com/machine-learning/crash-course/training-and-test-sets/splitting-data)

Next
Check Your Intuition →
(https://developers.google.com/machine-learning/crash-course/validation/check-your-intuition)

Validation Set: Check Your Intuition

Estimated Time: 5 minutes

Before beginning this module, consider whether there are any pitfalls in using the training process outlined in Training and Test Sets (https://developers.google.com/machine-learning/crash-course/training-and-test-sets/video-lecture).

Explore the options below.

We looked at a process of using a test set and a training set to drive iterations of model development. On each iteration, we'd train on the training data and evaluate on the test data, using the evaluation results on test data to guide choices of and changes to various model hyperparameters like learning rate and features. Is there anything wrong with this approach? (Pick only one answer.)

🚫 This is computationally inefficient. We should just pick a default set of hyperparameters and live with them to save resources.

Although these sorts of iterations are expensive, they are a critical part of model development. Hyperparameter settings can make an enormous difference in model quality, and we should always budget some amount of time and computational resources to ensure we're getting the best quality we can.

Try again.

🚫 Totally fine, we're training on training data and evaluating on separate, held-out test data.

Actually, there's a subtle issue here. Think about what might happen if we did many, many iterations of this form.

Try again.

✓ Doing many rounds of this procedure might cause us to implicitly fit to the peculiarities of our specific test set.

Yes indeed! The more often we evaluate on a given test set, the more we are at risk for implicitly overfitting to that one test set. We'll look at a better protocol next.

Correct answer.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Playground Exercise
(https://developers.google.com/machine-learning/crash-course/training-and-test-sets/playground-exercise)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/validation/video-lecture)

Last updated March 5, 2019.

Validation Set: Another Partition

Estimated Time: 5 minutes

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:

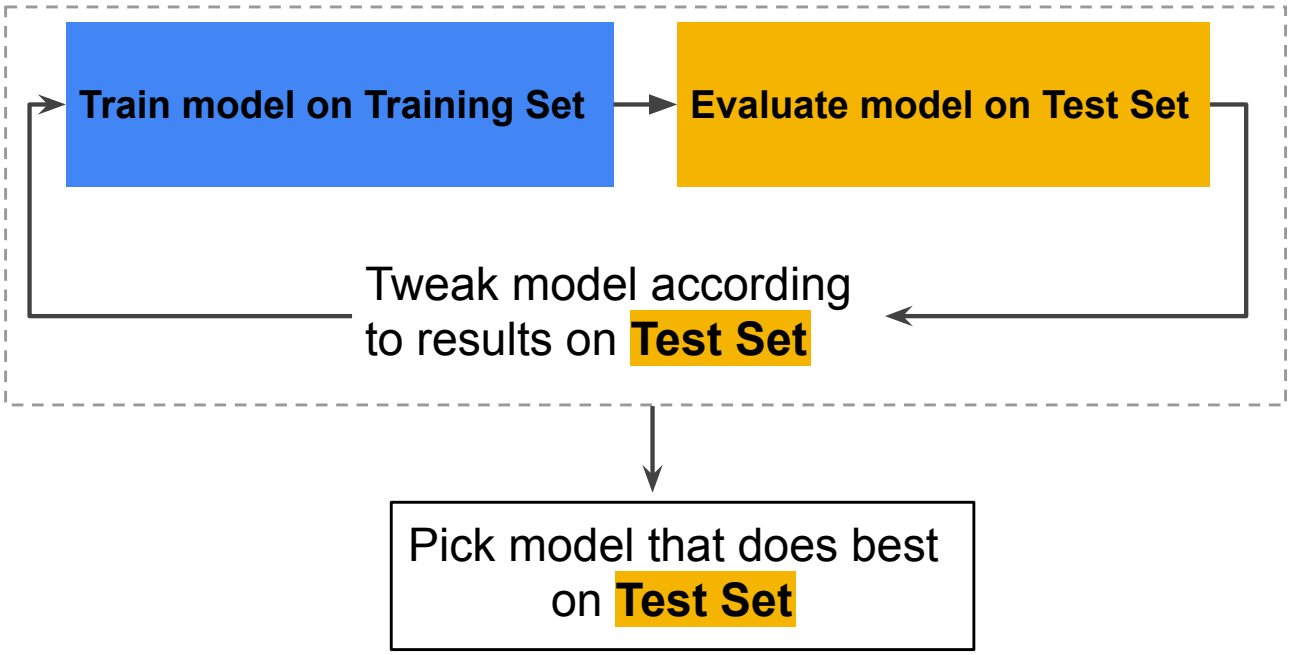


Figure 1. A possible workflow?

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 evaluation *after* the model has "passed" the validation set. The following figure shows this new workflow:

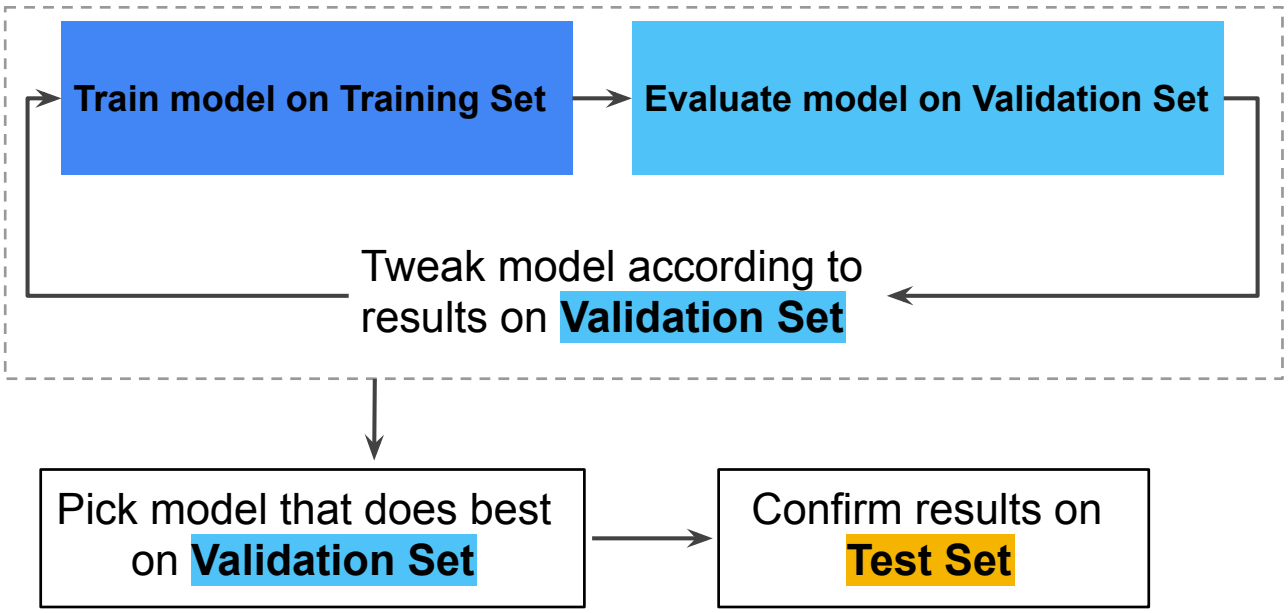


Figure 3. A better workflow.

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
- test set

(https://developers.google.com/machine-learning/glossary#overfitting)

- training set (https://developers.google.com/machine-learning/glossary#training_set)

(https://developers.google.com/machine-learning/glossary#test_set)

- validation set (https://developers.google.com/machine-learning/glossary#validation_set)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/validation/video-lecture)

Next
Programming Exercise →
(https://developers.google.com/machine-learning/crash-course/validation/programming-exercise)

Representation: Feature Engineering

Estimated Time: 9 minutes

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.

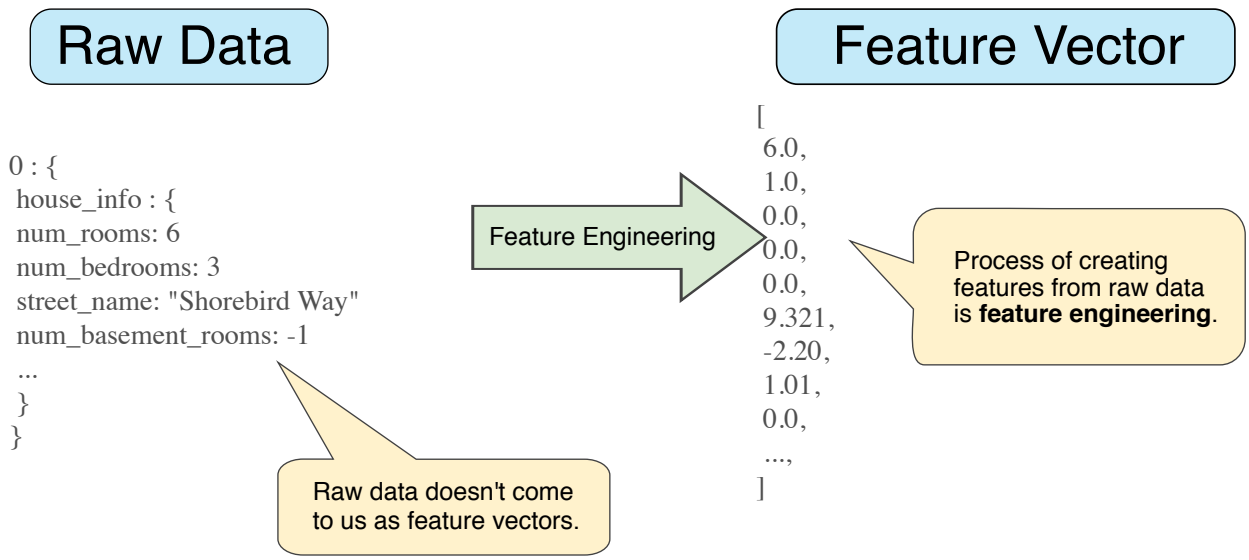


Figure 1. Feature engineering maps raw data to ML features.

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:

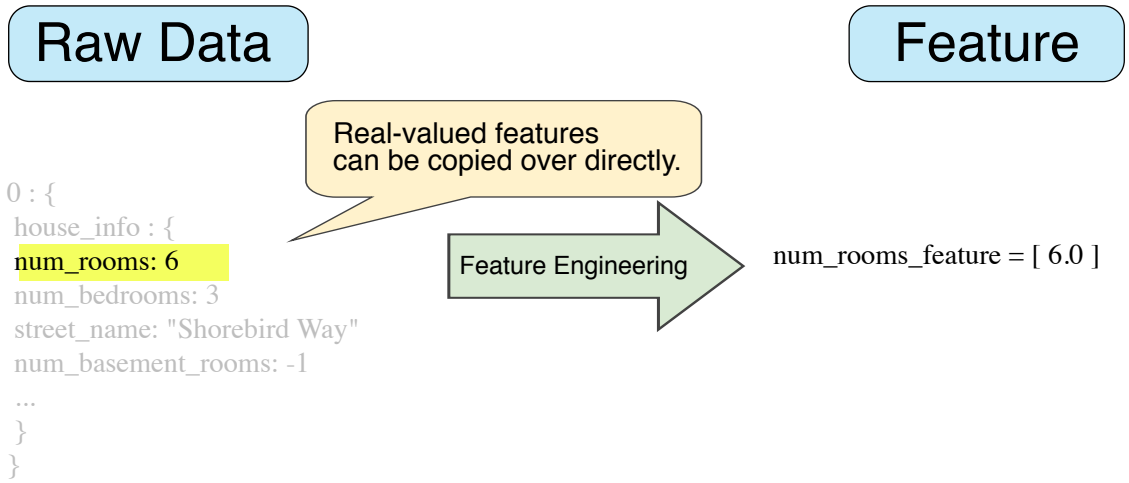


Figure 2. Mapping integer values to floating-point values.

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_name` with options that include:

```
{'Charleston Road', 'North Shoreline Boulevard', 'Shorebird Way', 'Rengstor
```

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`.

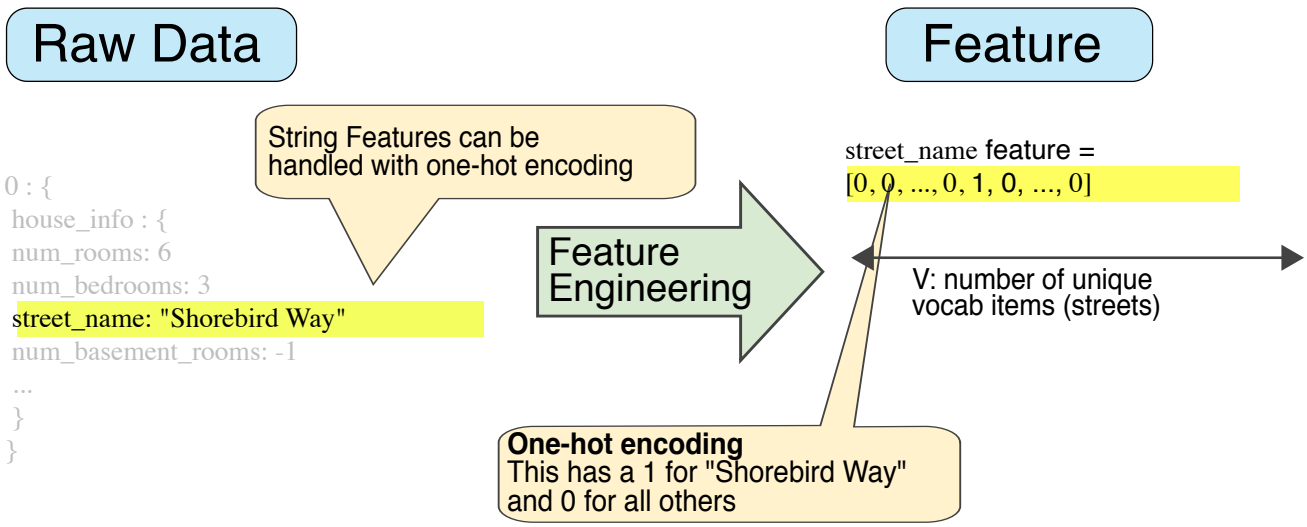


Figure 3. Mapping street address via one-hot encoding.

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 (/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)
- one-hot encoding (https://developers.google.com/machine-learning/glossary#one-hot_encoding)
- feature engineering (https://developers.google.com/machine-learning/glossary#feature_engineering)
- representation (https://developers.google.com/machine-learning/glossary#representation)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/representation/video-lecture)

Next
Qualities of Good Features →
(https://developers.google.com/machine-learning/crash-course/representation/qualities-of-good-features)

Last updated March 5, 2019.

Representation: Qualities of Good Features

Estimated Time: 10 minutes

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

Avoid rarely used discrete feature values

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

`house_type: victorian` ✔

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

`unique_house_id: 8SK982ZZ1242Z` ✘

Prefer clear and obvious meanings

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

`house_age_years: 27` ✔

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

`house_age: 851472000` ✘

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

appropriate values:

```
user_age_years: 277
```
✘

Don't mix "magic" values with actual data

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

```
quality_rating: 0.82
quality_rating: 0.37
```
✔

However, if a user didn't enter a **`quality_rating`**, perhaps the data set represented its absence with a magic value like the following:

```
quality_rating: -1
```
✘

To explicitly mark magic values, create a Boolean feature that indicates whether or not a **`quality_rating`** was supplied. Give this Boolean feature a name like **`is_quality_rating_defined`**.

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

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

Account for upstream instability

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

```
city_id: "br/sao_paulo"
```
✔

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

```
inferred_city_cluster: "219"
```

✘

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Feature Engineering
(https://developers.google.com/machine-learning/crash-course/representation/feature-engineering)

Next
Cleaning Data →
(https://developers.google.com/machine-learning/crash-course/representation/cleaning-data)

Last updated March 5, 2019.

Representation: Cleaning Data

Estimated Time: 10 minutes

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

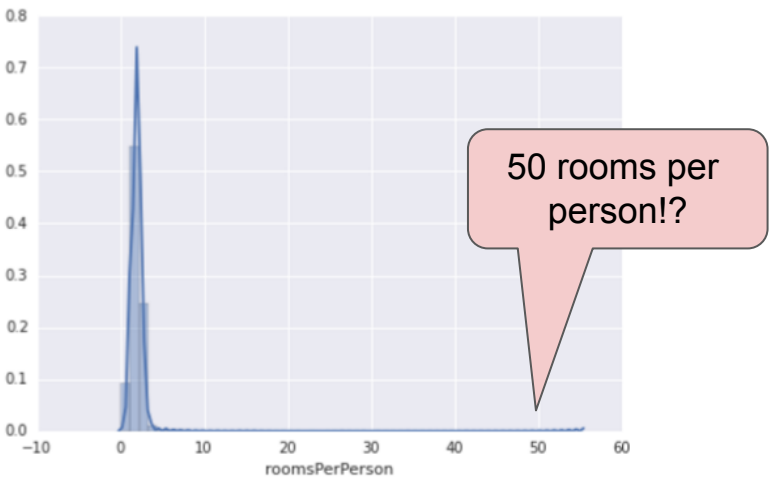
then:

```
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.




```
roomsPerPerson = totalRooms / population
```

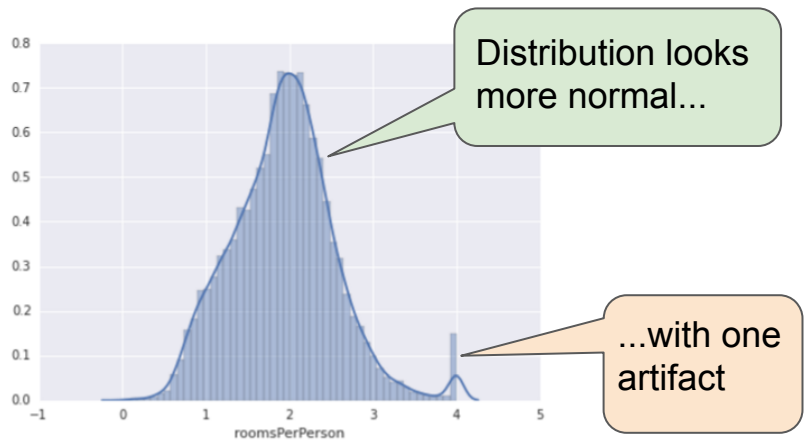
Figure 4. A verrrrry lonnnnnnng tail.

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


```
roomsPerPerson = log((totalRooms / population) + 1)
```

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?




```
roomsPerPerson = min(totalRooms / population, 4)
```

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.

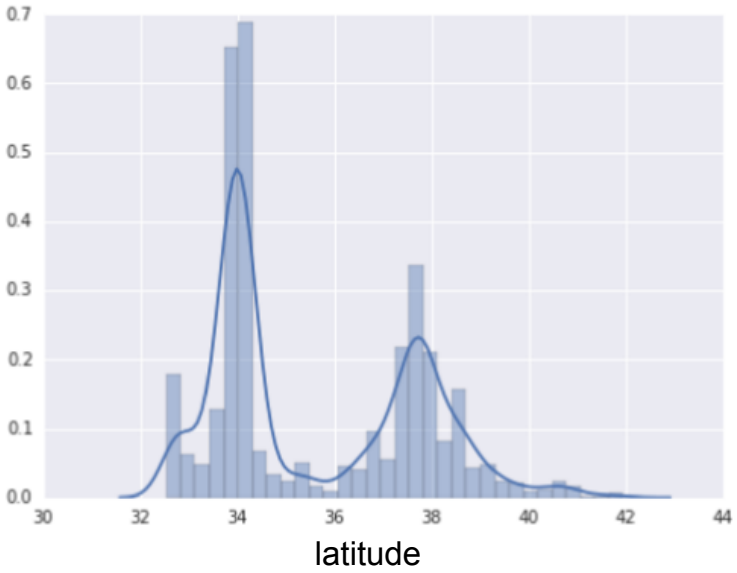


Figure 7. Houses per latitude.

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 $\frac{35}{34}$ 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:

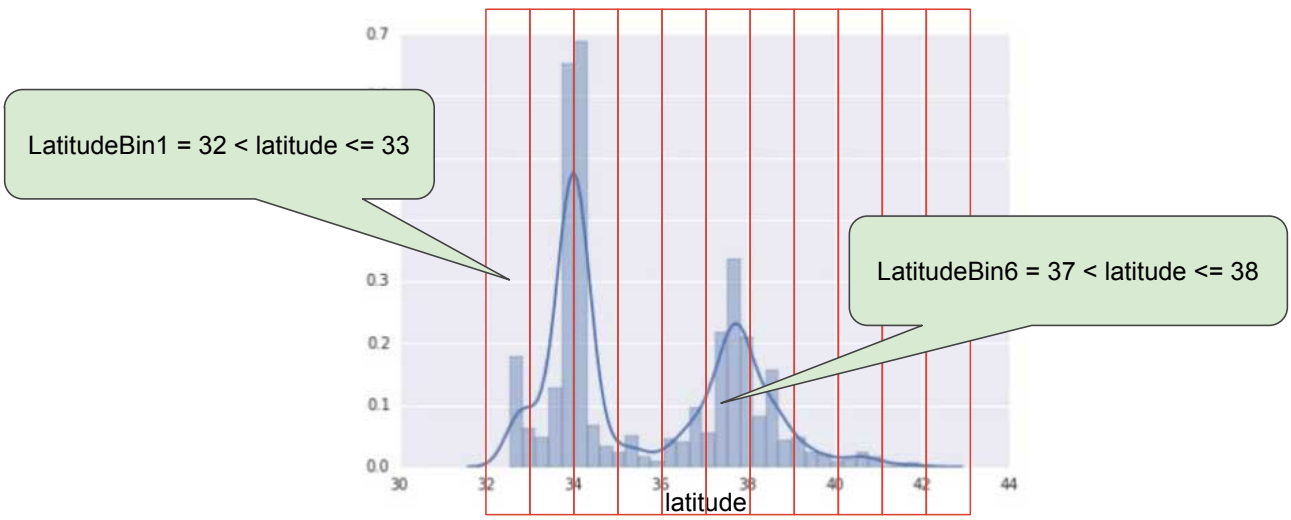


Figure 8. Binning values.

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)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Qualities of Good Features
(https://developers.google.com/machine-learning/crash-course/representation/qualities-of-good-features)

Next
Programming Exercise →
(https://developers.google.com/machine-learning/crash-course/representation/programming-exercise)

Last updated March 5, 2019.

Feature Crosses: Encoding Nonlinearity

Estimated Time: 7 minutes

In Figures 1 and 2, imagine the following:

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

Figure 1. Is this a linear problem?

Can you draw a line that neatly separates the sick trees from the healthy trees? Sure. This is a linear problem. The line won't be perfect. A sick tree or two might be on the "healthy" side, but your line will be a good predictor.

Now look at the following figure:

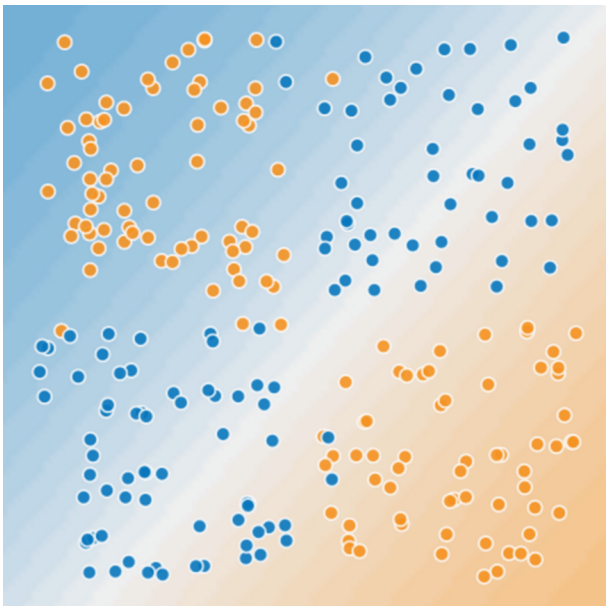


Figure 2. Is this a linear problem?

Can you draw a single straight line that neatly separates the sick trees from the healthy trees? No, you can't. This is a nonlinear problem. Any line you draw will be a poor predictor of tree health.

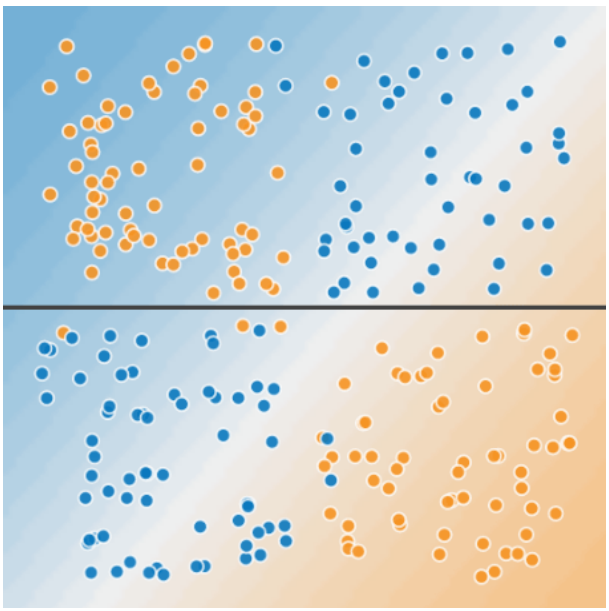


Figure 3. A single line can't separate the two classes.

To solve the nonlinear problem shown in Figure 2, create a feature cross. 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).) Let's create a feature cross named x_3 by crossing x_1 and x_2:

$$x_3 = x_1 x_2$$

We treat this newly minted x_3 feature cross just like any other feature. The linear formula becomes:

$$y = b + w_1 x_1 + w_2 x_2 + w_3 x_3$$

A linear algorithm can learn a weight for w_3 just as it would for w_1 and w_2. In other words, although w_3 encodes nonlinear information, you don't need to change how the linear model trains to determine the value of w_3.

Kinds of feature crosses

We can create many different 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)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture (https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture)
Next

Crossing One-Hot Vectors →
(https://developers.google.com/machine-learning/crash-course/feature-crosses/crossing-one-hot-vectors)

Feature Crosses: Crossing One-Hot Vectors

Estimated Time: 8 minutes

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

```
country:usa AND language:spanish
```

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

```
binned_latitude = [0, 0, 0, 1, 0]
binned_longitude = [0, 1, 0, 0, 0]
```

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

```
binned_latitude X binned_longitude
```

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

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

```
binned_latitude(lat) = [
  0  < lat <= 10
  10 < lat <= 20
  20 < lat <= 30
]

binned_longitude(lon) = [
  0  < lon <= 15
  15 < lon <= 30
]
```

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

```
binned_latitude_X_longitude(lat, lon) = [
  0  < lat <= 10 AND 0  < lon <= 15
  0  < lat <= 10 AND 15 < lon <= 30
  10 < lat <= 20 AND 0  < lon <= 15
  10 < lat <= 20 AND 15 < lon <= 30
  20 < lat <= 30 AND 0  < lon <= 15
  20 < lat <= 30 AND 15 < lon <= 30
]
```

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

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

If we build a feature cross from both these features:

```
[behavior type X time of day]
```

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

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

Key Terms

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

Previous

← Encoding Nonlinearity (https://developers.google.com/machine-learning/crash-course/feature-crosses/encoding-nonlinearity)

Next
Playground Exercises → (https://developers.google.com/machine-learning/crash-course/feature-crosses/playground-exercises)

Feature Crosses: Playground Exercises

Estimated Time: 15 minutes

Introducing Feature Crosses

Can a feature cross truly enable a model to fit nonlinear data? To find out, try this exercise.

Task: Try to create a model that separates the blue dots from the orange dots by *manually* changing the weights of the following three input features:

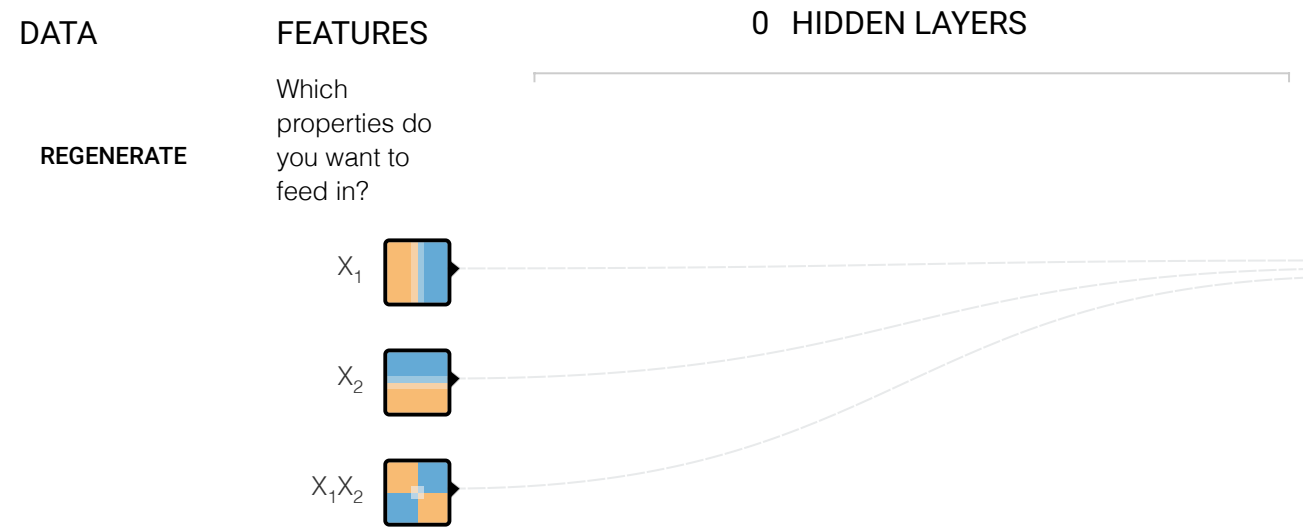
- x_1
- x_2
- $x_1 x_2$ (a feature cross)

To manually change a weight:

1. Click on a line that connects FEATURES to OUTPUT. An input form will appear.
2. Type a floating-point value into that input form.
3. Press Enter.

Note that the interface for this exercise does not contain a Step button. That's because this exercise does not iteratively train a model. Rather, you will manually enter the "final" weights for the model.

(Answers appear just below the exercise.)



Click the dropdown arrow for the answer.

- $w_1 = 0$
- $w_2 = 0$
- $x_1 x_2 = 1$ (or any positive value)

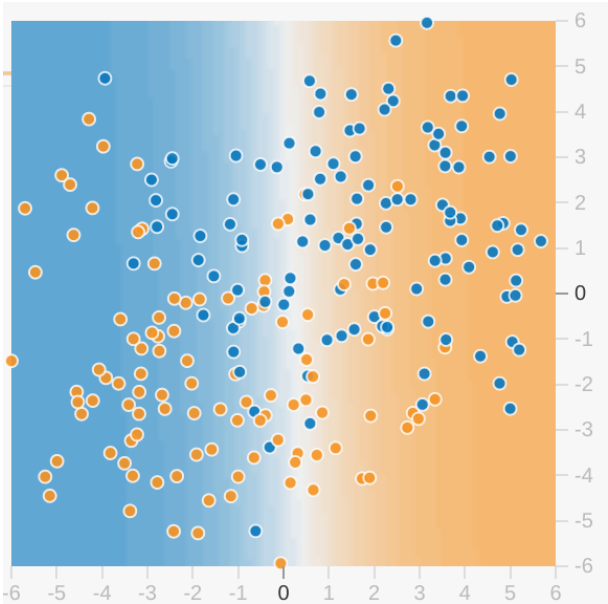
If you enter a negative value for the feature cross, the model will separate the blue dots from the orange dots but the predictions will be completely wrong. That is, the model will predict orange for the blue dots, and blue for the orange dots.

More Complex Feature Crosses

Now let's play with some advanced feature cross combinations. The data set in this Playground exercise looks a bit like a noisy bullseye from a game of darts, with the blue dots in the middle and the orange dots in an outer ring.

Click the dropdown arrow for an explanation of model visualization.

Each Playground exercise displays a visualization of the current state of the model. For example, here's a visualization:



Note the following about the model visualization:

- Each axis represents a specific feature. In the case of spam vs. not spam, the features could be the word count and the number of recipients of the email.

★ **Note:** Appropriate axis values will depend on feature data. The axis values shown above would not make sense for word count or number of recipients, as neither can be negative.

- Each dot plots the feature values for one example of the data, such as an email.
- The color of the dot represents the class that the example belongs to. For example, the blue dots can represent non-spam emails while the orange dots can represent spam emails.
- The background color represents the model's prediction of where examples of that color should be found. A blue background around a blue dot means that the model is correctly predicting that example. Conversely, an orange background around a blue dot means that the model is incorrectly predicting that example.
- The background blues and oranges are scaled. For example, the left side of the visualization is solid blue but gradually fades to white in the center of the visualization. You can think of the color strength as suggesting the model's confidence in its guess. So solid blue means that the model is very confident about its guess and light blue means that the model is less confident. (The model visualization shown in the figure is doing a poor job of prediction.)

Use the visualization to judge your model's progress. ("Excellent—most of the blue dots have a blue background" or "Oh no! The blue dots have an orange background.") Beyond the colors, Playground also displays the model's current loss numerically. ("Oh no! Loss is going up instead of down.")

Task 1: Run this linear model as given. Spend a minute or two (but no longer) trying different learning rate settings to see if you can find any improvements. Can a linear model produce effective results for this data set?

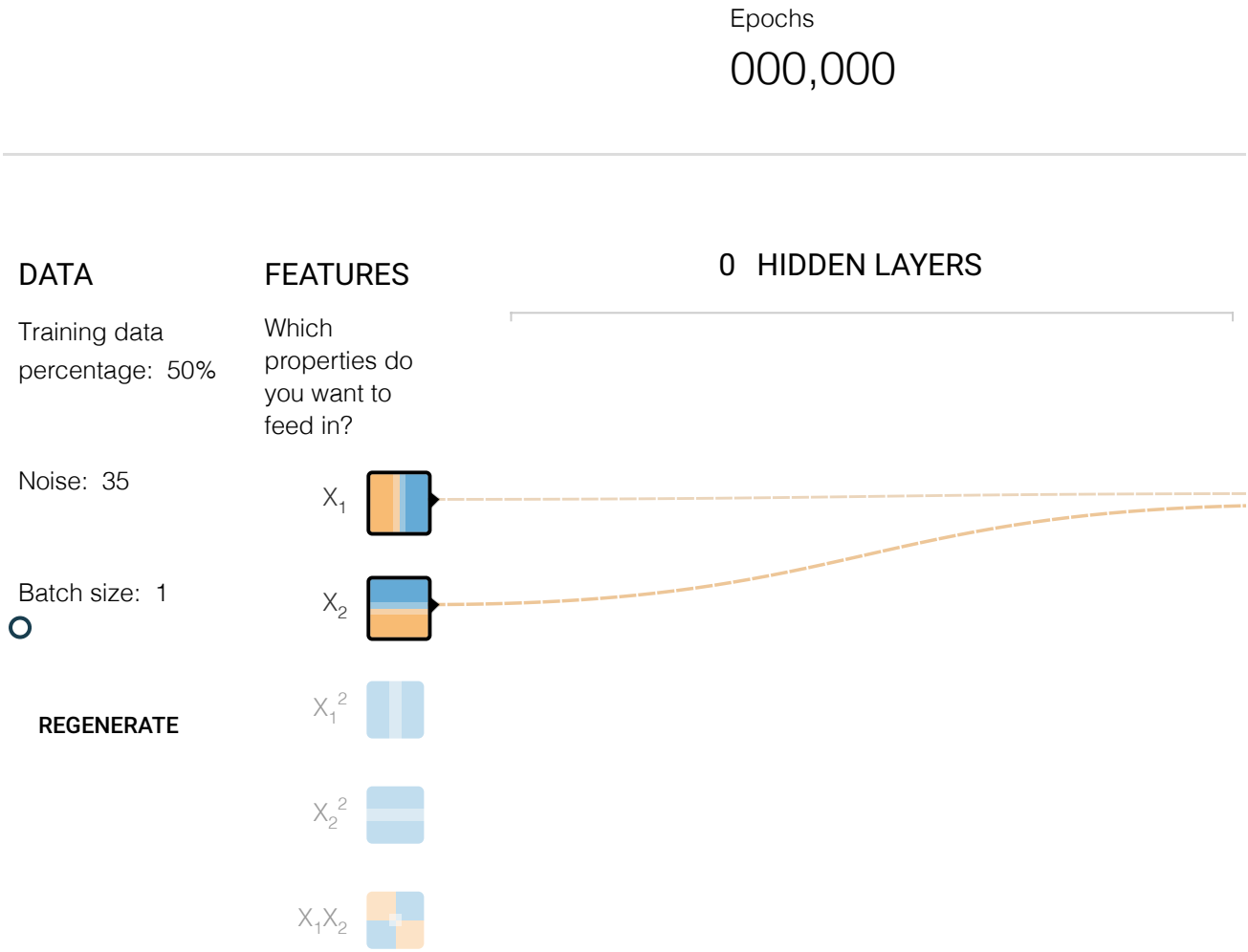
Task 2: Now try adding in cross-product features, such as x_1x_2, trying to optimize performance.

- Which features help most?
- What is the best performance that you can get?

Task 3: When you have a good model, examine the model output surface (shown by the background color).

1. Does it look like a linear model?
2. How would you describe the model?

(Answers appear just below the exercise.)



Click the dropdown arrow for the answer to Task 1.

No. A linear model cannot effectively model this data set. Reducing the learning rate reduces loss, but loss still converges at an unacceptably high value.

Click the dropdown arrow for an answer to Task 2.

Playground's data sets are randomly generated. Consequently, our answers may not always agree exactly with yours. In fact, if you regenerate the data set between runs, your own results won't always agree exactly with your previous runs. That said, you'll get better results by doing the following:

- Using both x_1^2 and x_2^2 as feature crosses. (Adding x_1x_2 as a feature cross doesn't appear to help.)
- Reducing the `Learning rate`, perhaps to `0.001`.

Click the dropdown arrow for an answer to Task 3.

The model output surface does not look like a linear model. Rather, it looks elliptical.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Crossing One-Hot Vectors
(https://developers.google.com/machine-learning/crash-course/feature-crosses/crossing-one-hot-vectors)

Next
Programming Exercise →
(https://developers.google.com/machine-learning/crash-course/feature-crosses/programming-exercise)

Feature Crosses: Check Your Understanding

Estimated Time: 5 minutes

Explore the options below.

Different cities in California have markedly different housing prices (https://developers.google.com/machine-learning/crash-course/california-housing-data-description). Suppose you must create a model to predict housing prices. Which of the following sets of features or feature crosses could learn *city-specific* relationships between `roomsPerPerson` and housing price?

🚫 Three separate binned features: `[binned latitude], [binned longitude], [binned roomsPerPerson]`

Binning is good because it enables the model to learn nonlinear relationships within a single feature. However, a city exists in more than one dimension, so learning city-specific relationships requires crossing latitude and longitude.

Try again.

🚫 Two feature crosses: `[binned latitude X binned roomsPerPerson]` and `[binned longitude X binned roomsPerPerson]`

Binning is a good idea; however, a city is the conjunction of latitude and longitude, so separate feature crosses prevent the model from learning city-specific prices.

Try again.

✓ One feature cross: `[binned latitude X binned longitude X binned roomsPerPerson]`

Crossing binned latitude with binned longitude enables the model to learn city-specific effects of roomsPerPerson. Binning prevents a change in latitude producing the same result as a change in

longitude. Depending on the granularity of the bins, this feature cross could learn city-specific or neighborhood-specific or even block-specific effects.

Correct answer.

🚫 One feature cross: `[latitude X longitude X roomsPerPerson]`

In this example, crossing real-valued features is not a good idea. Crossing the real value of, say, latitude with roomsPerPerson enables a 10% change in one feature (say, latitude) to be equivalent to a 10% change in the other feature (say, roomsPerPerson).

Try again.

Previous
← Programming Exercise
(https://developers.google.com/machine-learning/crash-course/feature-crosses/programming-exercise)

Next
Playground Exercise: Overcrossing? →
(https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/playground-exercise-overcrossing)

Regularization for Simplicity: Playground Exercise (Overcrossing?)

Estimated Time: 5 minutes

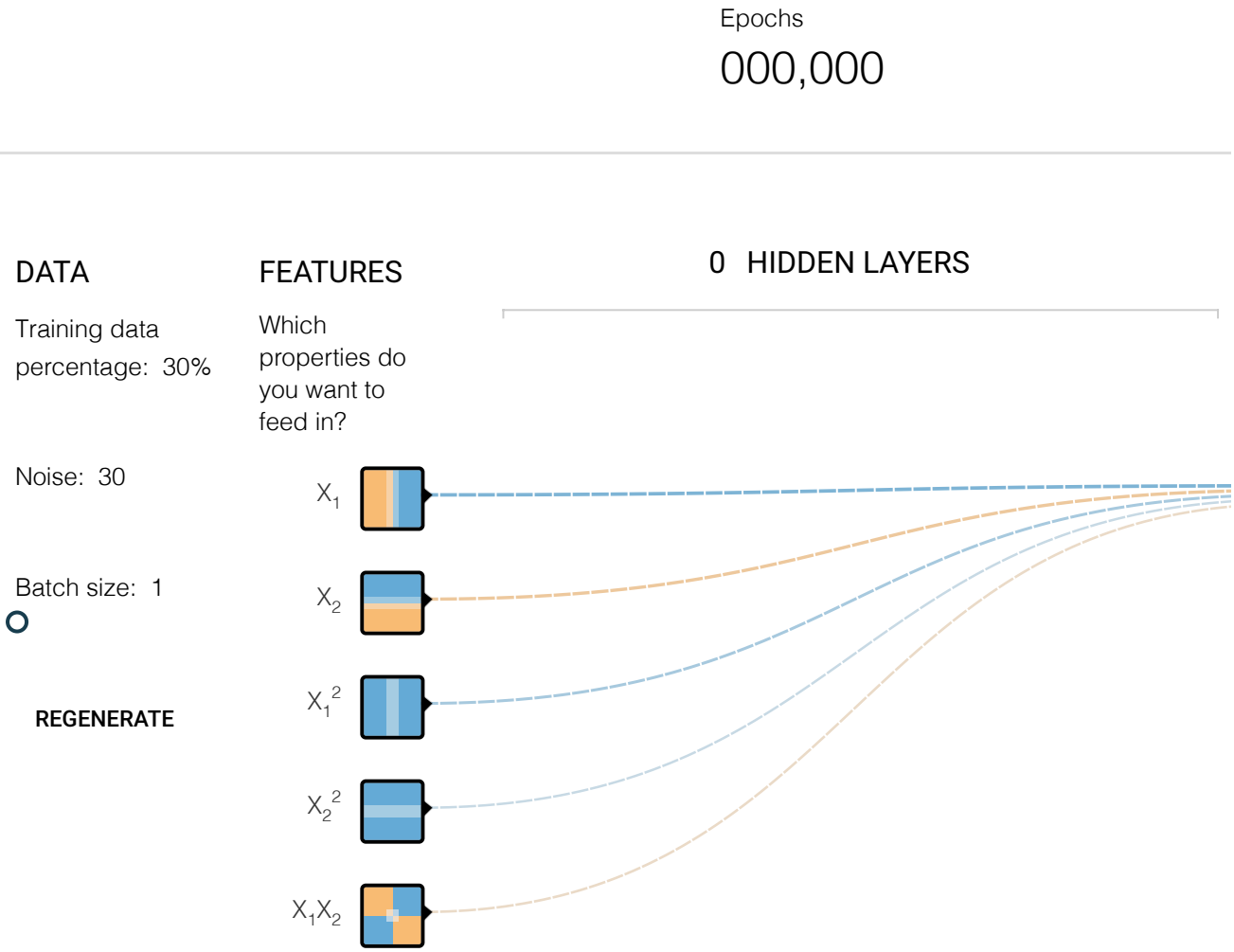
Overcrossing?

Before you watch the video or read the documentation, please complete this exercise that explores overuse of feature crosses.

Task 1: Run the model as is, with all of the given cross-product features. Are there any surprises in the way the model fits the data? What is the issue?

Task 2: Try removing various cross-product features to improve performance (albeit only slightly). Why would removing features improve performance?

(Answers appear just below the exercise.)



Click the dropdown arrow for an answer to Task 1.

Surprisingly, the model's decision boundary looks kind of crazy. In particular, there's a region in the upper left that's hinting towards blue, even though there's no visible support for that in the data.

Notice the relative thickness of the five lines running from INPUT to OUTPUT. These lines show the relative weights of the five features. The lines emanating from X_1

and X_2 are much thicker than those coming from the feature crosses. So, the feature crosses are contributing far less to the model than the normal (uncrossed) features.

Click the dropdown arrow for an answer to Task 2.

Removing all the feature crosses gives a saner model (there is no longer a curved boundary suggestive of overfitting) and makes the test loss converge.

After 1,000 iterations, test loss should be a slightly lower value than when the feature crosses were in play (although your results may vary a bit, depending on the data set).

The data in this exercise is basically linear data plus noise. If we use a model that is too complicated, such as one with too many crosses, we give it the opportunity to fit to the noise in the training data, often at the cost of making the model perform badly on test data.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Check Your Understanding
(https://developers.google.com/machine-learning/crash-course/feature-crosses/check-your-understanding)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/video-lecture)

Regularization for Simplicity: L_2 Regularization

Estimated Time: 7 minutes

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

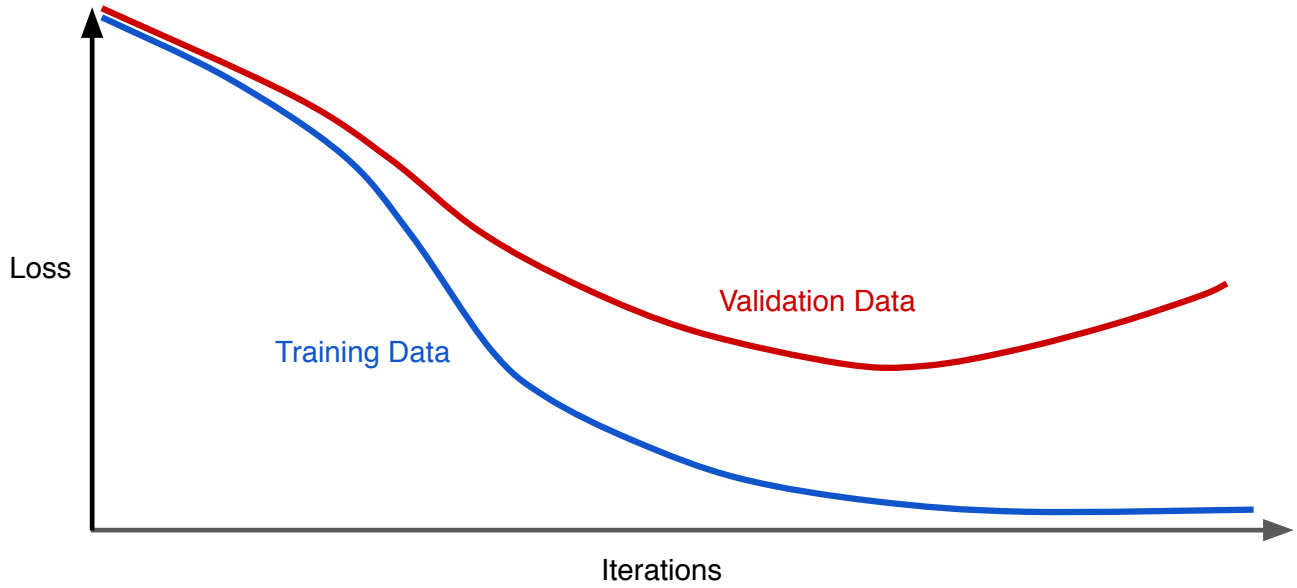


Figure 1. Loss on training set and validation set.

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 (https://developers.google.com/machine-learning/crash-course/generalization/peril-of-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):

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}))$$

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

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}) + \text{complexity}(\text{Model}))$$

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.

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 (https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/l1-regularization) 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 ***L_2* regularization** formula, which defines the regularization term as the sum of the squares of all the feature weights:

$$L_2 \text{ regularization term} = ||\boldsymbol{w}||_2^2 = w_1^2 + w_2^2 + \ldots + w_n^2$$

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:

$$\{w_1 = 0.2, w_2 = 0.5, w_3 = 5, w_4 = 1, w_5 = 0.25, w_6 = 0.75\}$$

Has an L_2 regularization term of 26.915:

$$w_1^2 + w_2^2 + \boldsymbol{w_3^2} + w_4^2 + w_5^2 + w_6^2$$
$$= 0.2^2 + 0.5^2 + \boldsymbol{5^2} + 1^2 + 0.25^2 + 0.75^2$$
$$= 0.04 + 0.25 + \boldsymbol{25} + 1 + 0.0625 + 0.5625$$
$$= 26.915$$

But $\boldsymbol{w_3}$ (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 L_2 regularization term.

Key Terms

- generalization curve (https://developers.google.com/machine-learning/glossary#generalization_curve)
- overfitting (https://developers.google.com/machine-learning/glossary#overfitting)
- structural risk minimization (https://developers.google.com/machine-learning/glossary#SRM)
- L_2 regularization (https://developers.google.com/machine-learning/glossary#L2_regularization)
- regularization (https://developers.google.com/machine-learning/glossary#regularization)

Previous
← Video Lecture (https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/video-lecture)

Next
Lambda → (https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/lambda)

Regularization for Simplicity: Lambda

Estimated Time: 8 minutes

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:

$$\text{minimize}(\text{Loss}(\text{Data}|\text{Model}) + \lambda \text{ complexity}(\text{Model}))$$

Performing L_2 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.

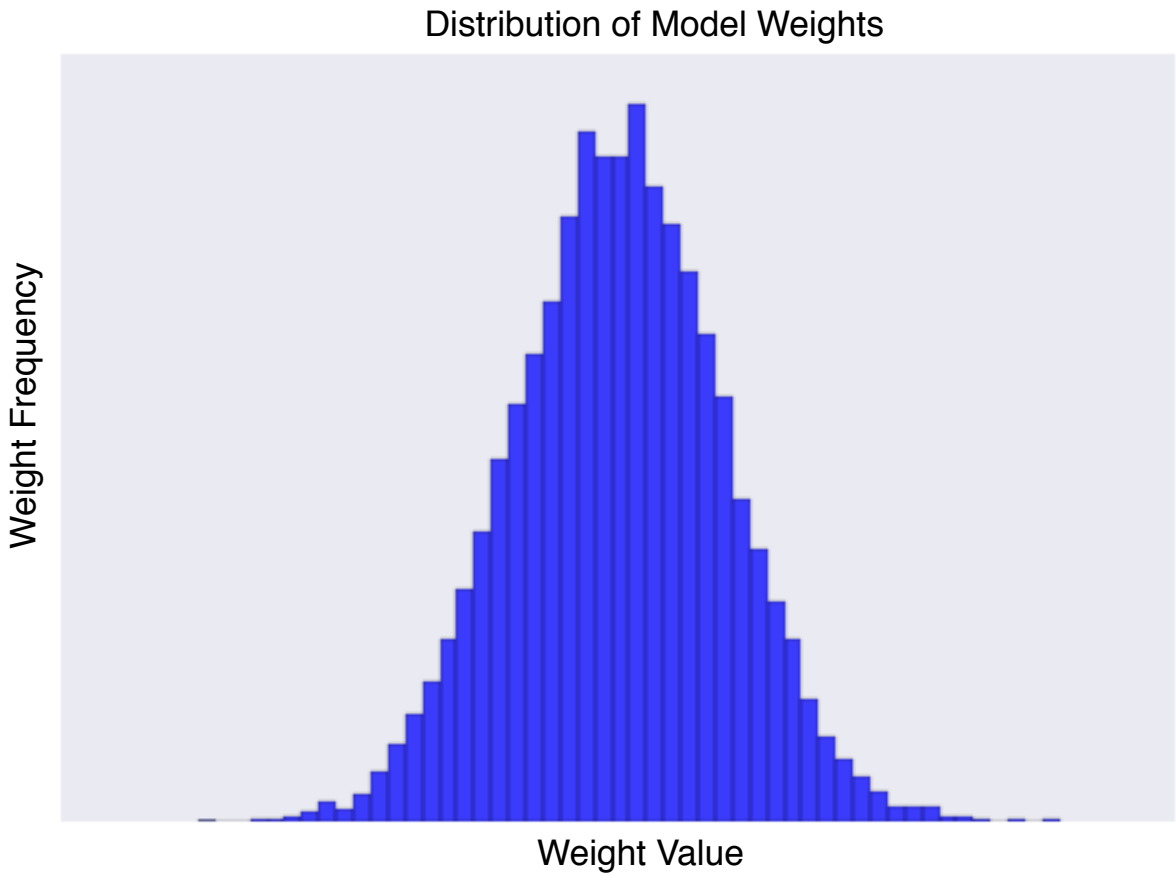


Figure 2. Histogram of weights.

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

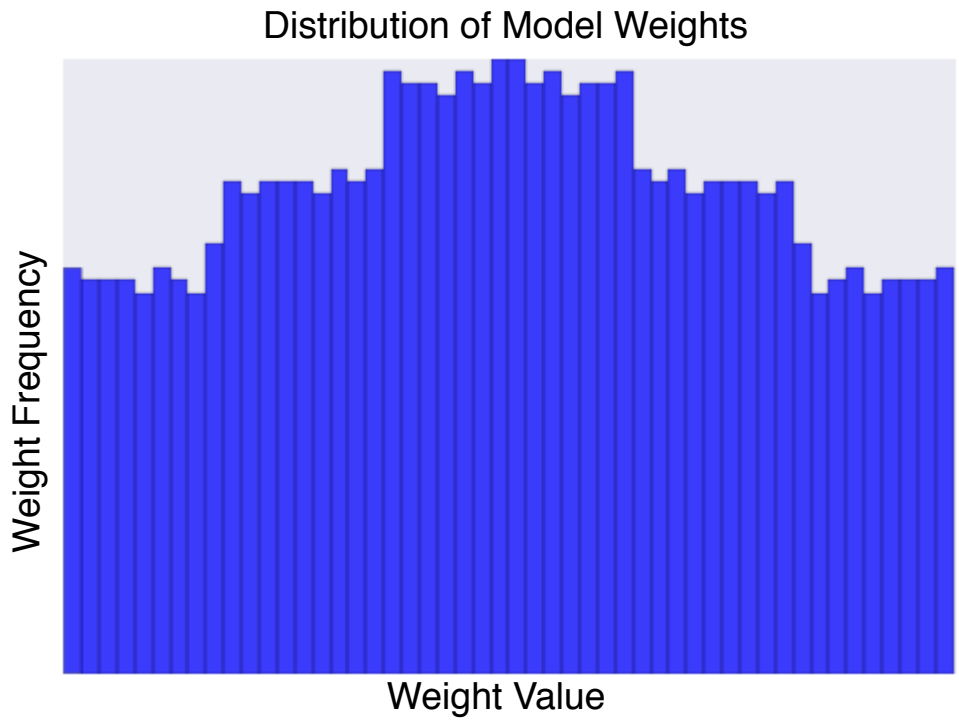


Figure 3. Histogram of weights produced by a lower lambda value.

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

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

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

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

Click the dropdown arrow to learn about L_2 regularization and learning rate.

There's a close connection between learning rate and lambda. Strong L_2 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 (https://developers.google.com/machine-learning/crash-course/production-ml-systems) (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)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← L2 Regularization (https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/l2-regularization)

Next
Playground Exercise: L2 Regularization → (https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/playground-exercise-examining-l2-regularization)

Last updated March 5, 2019.

Regularization for Simplicity: Playground Exercise (L2 Regularization)

Estimated Time: 10 minutes

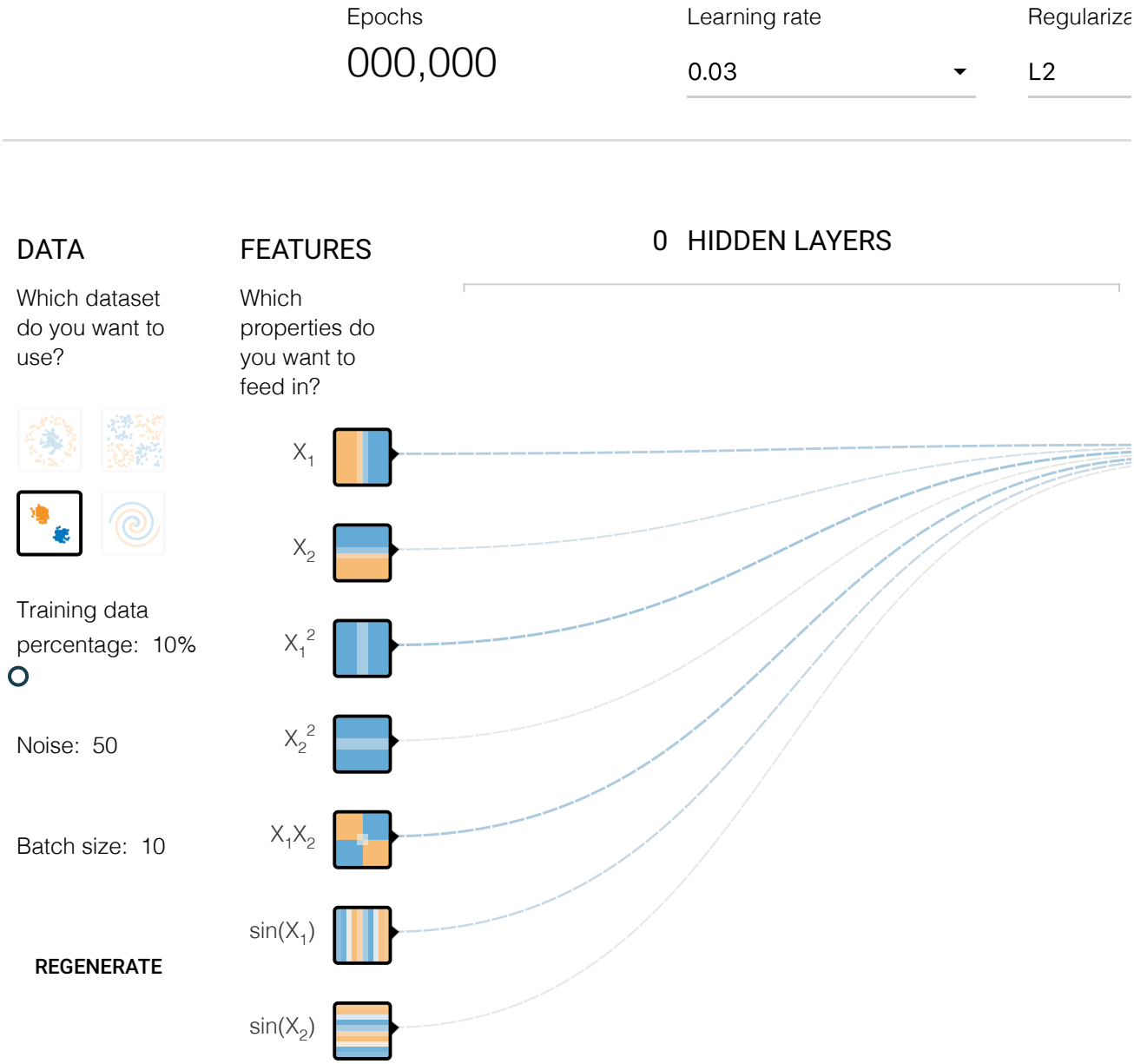
Examining L_2 regularization

This exercise contains a small, noisy training data set. In this kind of setting, overfitting is a real concern. Fortunately, regularization might help.

This exercise consists of three related tasks. To simplify comparisons across the three tasks, run each task in a separate tab.

- **Task 1:** Run the model as given for at least 500 epochs. Note the following:
  - Test loss.
  - The delta between Test loss and Training loss.
  - The learned weights of the features and the feature crosses. (The relative thickness of each line running from FEATURES to OUTPUT represents the learned weight for that feature or feature cross. You can find the exact weight values by hovering over each line.)
- **Task 2:** (*Consider doing this Task in a separate tab.*) Increase the regularization rate from `0` to `0.3`. Then, run the model for at least 500 epochs and find answers to the following questions:
  - How does the Test loss in Task 2 differ from the Test loss in Task 1?
  - How does the delta between Test loss and Training loss in Task 2 differ from that of Task 1?
  - How do the learned weights of each feature and feature cross differ from Task 2 to Task 1?
  - What do your results say about model complexity?
- **Task 3:** Experiment with regularization rate, trying to find the optimum value.

(Answers appear just below the exercise.)



Click the dropdown arrow for answers.

Increasing the regularization rate from 0 to 0.3 produces the following effects:

- Test loss drops significantly.

★ Note: While test loss decreases, training loss actually *increases*. This is expected, because you've added another term to the loss function to penalize complexity. Ultimately, all that matters is test loss, as that's the true measure of the model's ability to make good predictions on new data.

- The delta between Test loss and Training loss drops significantly.
- The weights of the features and some of the feature crosses have lower absolute values, which implies that model complexity drops.

Given the randomness in the data set, it is impossible to predict which regularization rate produced the best results for you. For us, a regularization rate of either 0.3 or 1 generally produced the lowest Test loss.

Previous
← Lambda
(https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/lambda)

Next
Check Your Understanding →
(https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/check-your-understanding)

Regularization for Simplicity: Check Your Understanding

Estimated Time: 5 minutes

L_2 Regularization

Explore the options below.

Imagine a linear model with 100 input features:

- 10 are highly informative.
- 90 are non-informative.

Assume that all features have values between -1 and 1. Which of the following statements are true?

✓ L_2 regularization may cause the model to learn a moderate weight for some **non-informative** features.

Surprisingly, this can happen when a non-informative feature happens to be correlated with the label. In this case, the model incorrectly gives such non-informative features some of the "credit" that should have gone to informative features.

1 of 2 correct answers.

🚫 L_2 regularization will encourage most of the non-informative weights to be exactly 0.0.

L_2 regularization does not tend to force weights to exactly 0.0. L_2 regularization penalizes larger weights more than smaller weights. As a weight gets close to 0.0, L_2 "pushes" less forcefully toward 0.0.

Try again.

✓ L_2 regularization will encourage many of the non-informative weights to be nearly (but not exactly) 0.0.

Yes, L_2 regularization encourages weights to be near 0.0, but not exactly 0.0.

2 of 2 correct answers.

L_2 Regularization and Correlated Features

Explore the options below.

Imagine a linear model with two strongly correlated features; that is, these two features are nearly identical copies of one another but one feature contains a small amount of random noise. If we train this model with L_2 regularization, what will happen to the weights for these two features?

🚫 One feature will have a large weight; the other will have a weight of **exactly** 0.0.

L_2 regularization rarely forces weights to exactly 0.0. By contrast, L_1 regularization (discussed later) *does* force weights to exactly 0.0.

Try again.

✓ Both features will have roughly equal, moderate weights.

L_2 regularization will force the features towards roughly equivalent weights that are approximately half of what they would have been had only one of the two features been in the model.

Correct answer.

🚫 One feature will have a large weight; the other will have a weight of **almost** 0.0.

L_2 regularization penalizes large weights more than small weights. So, even if one weight started to drop faster than the other, L_2 regularization would tend to force the bigger weight to drop more quickly than the smaller weight.

Try again.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Playground Exercise: L2 Regularization
(https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/playground-exercise-examining-l2-regularization)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/logistic-regression/video-lecture)

Last updated March 5, 2019.

Logistic Regression: Calculating a Probability

Estimated Time: 10 minutes

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 a `p(bark | night)` of 0.05, then over a year, the dog's owners should be startled awake approximately 18 times:

```
startled = p(bark | night) * nights
18 ~= 0.05 * 365
```

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 (https://developers.google.com/machine-learning/crash-course/classification/video-lecture) 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:

$$y = \frac{1}{1 + e^{-z}}$$

The sigmoid function yields the following plot:

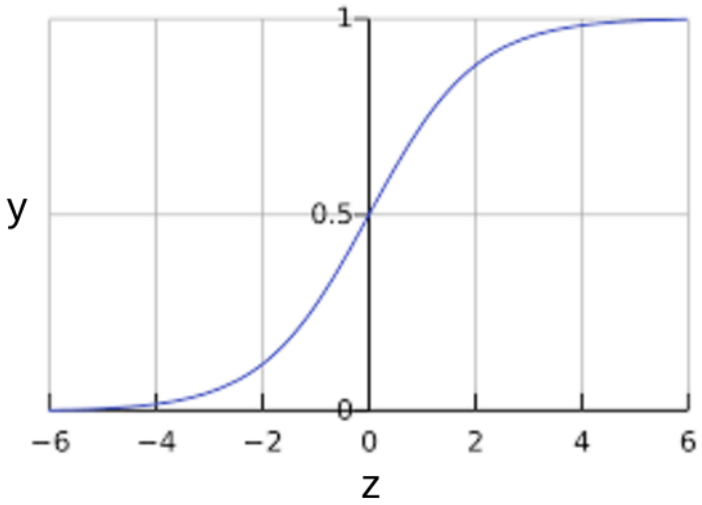


Figure 1: Sigmoid function.

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:

$$y' = \frac{1}{1 + e^{-(z)}}$$

where:

- y' is the output of the logistic regression model for a particular example.
- *z* is $b + w_1x_1 + w_2x_2 + \ldots w_Nx_N$
  - The *w* values are the model's learned weights, and *b* is the bias.
  - The *x* values are the feature values for a particular example.

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"):

$$z = log(\frac{y}{1 - y})$$

Here is the sigmoid function with ML labels:

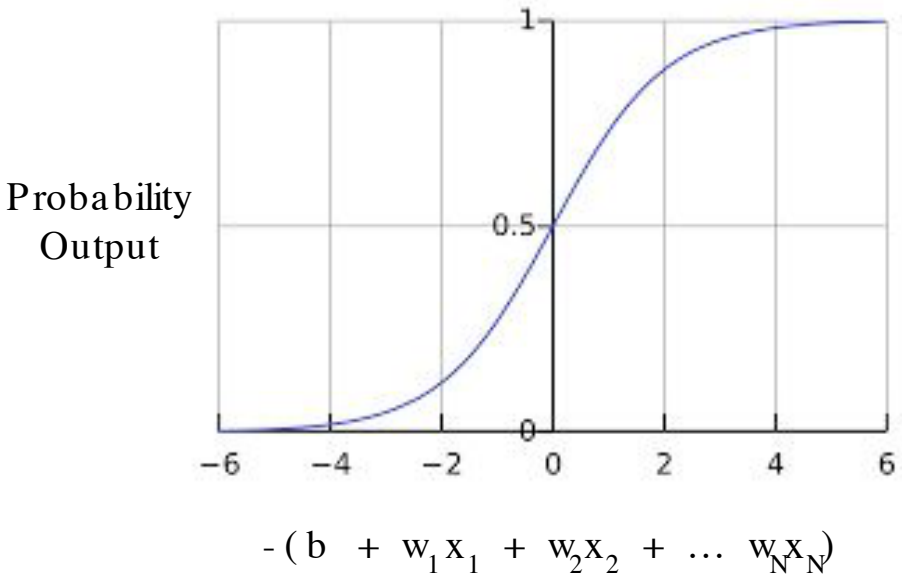


Figure 2: Logistic regression output.

Click the dropdown arrow to see a sample logistic regression inference calculation.

Suppose we had a logistic regression model with three features that learned the following bias and weights:

- b = 1
- w_1 = 2
- w_2 = -1
- w_3 = 5

Further suppose the following feature values for a given example:

- x_1 = 0
- x_2 = 10
- x_3 = 2

Therefore, the log-odds:

$$b + w_1 x_1 + w_2 x_2 + w_3 x_3$$

will be:

```
(1) + (2)(0) + (-1)(10) + (5)(2) = 1
```

Consequently, the logistic regression prediction for this particular example will be 0.731:

$$y' = \frac{1}{1 + e^{-(1)}} = 0.731$$

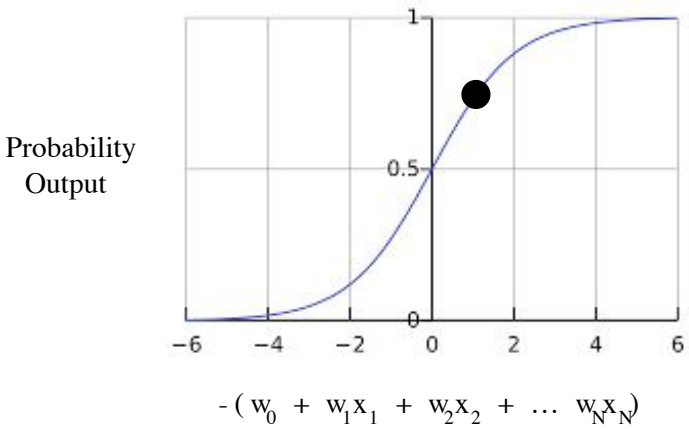


Figure 3: 73.1% probability.

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)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/logistic-regression/video-lecture)

Next
Loss and Regularization →
(https://developers.google.com/machine-learning/crash-course/logistic-regression/model-training)

Last updated March 5, 2019.

Logistic Regression: Loss and Regularization

Estimated Time: 6 minutes

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:

$$\text{Log Loss} = \sum_{(x,y)\in D} -y \log(y') - (1-y) \log(1-y')$$

where:

- $(x, y) \in D$ is the data set containing many labeled examples, which are (x, y) pairs.
- y is the label in a labeled example. Since this is logistic regression, every value of y must either be 0 or 1.
- y' is the predicted value (somewhere between 0 and 1), given the set of features in x.

The equation for Log Loss is closely related to Shannon's Entropy measure from Information Theory (https://wikipedia.org/wiki/Entropy_(information_theory)). It is also the negative logarithm of the likelihood function (https://wikipedia.org/wiki/Likelihood_function), assuming a Bernoulli distribution (https://wikipedia.org/wiki/Bernoulli_distribution) of y. Indeed, minimizing the loss function yields a maximum likelihood estimate.

Regularization in Logistic Regression

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

- L_2 regularization.
- Early stopping, that is, limiting the number of training steps or the learning rate.

(We'll discuss a third strategy—L_1 regularization—in a later module (https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/video-lecture)
.)

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

Fortunately, using L_2 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)
- L_1 regularization (https://developers.google.com/machine-learning/glossary#L1_regularization)
- log loss (https://developers.google.com/machine-learning/glossary#Log_Loss)
- L_2 regularization (https://developers.google.com/machine-learning/glossary#L2_regularization)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Calculating a Probability (https://developers.google.com/machine-learning/crash-course/logistic-regression/calculating-a-probability)

Next

Video Lecture →
(https://developers.google.com/machine-learning/crash-course/classification/video-lecture)

Last updated March 5, 2019.

Classification: Thresholding

Estimated Time: 2 minutes

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

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

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

Note: "Tuning" a threshold for logistic regression is different from tuning hyperparameters such as learning rate. Part of choosing a threshold is assessing how much you'll suffer for making a mistake. For example, mistakenly labeling a non-spam message as spam is very bad. However, mistakenly labeling a spam message as non-spam is unpleasant, but hardly the end of your job.

Key Terms

- binary classification (https://developers.google.com/machine-learning/glossary#binary_classification)
- classification model (https://developers.google.com/machine-learning/glossary#classification_model)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/classification/video-lecture)

Next
True vs. False; Positive vs. Negative
(https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative)

Last updated March 5, 2019.

Classification: True vs. False and Positive vs. Negative

Estimated Time: 5 minutes

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

> **An Aesop's Fable: The Boy Who Cried Wolf (*compressed*)**
>
> A shepherd boy gets bored tending the town's flock. To have some fun, he cries out, "Wolf!" even though no wolf is in sight. The villagers run to protect the flock, but then get really mad when they realize the boy was playing a joke on them.
>
> [Iterate previous paragraph *N* times.]
>
> One night, the shepherd boy sees a real wolf approaching the flock and calls out, "Wolf!" The villagers refuse to be fooled again and stay in their houses. The hungry wolf turns the flock into lamb chops. The town goes hungry. Panic ensues.

Let's make the following definitions:

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

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

True Positive (TP):

- Reality: A wolf threatened.
- Shepherd said: "Wolf."
- Outcome: Shepherd is a hero.

False Positive (FP):

- Reality: No wolf threatened.
- Shepherd said: "Wolf."
- Outcome: Villagers are angry at shepherd for waking them up.

False Negative (FN):

- Reality: A wolf threatened.
- Shepherd said: "No wolf."
- Outcome: The wolf ate all the sheep.

True Negative (TN):

- Reality: No wolf threatened.
- Shepherd said: "No wolf."
- Outcome: Everyone is fine.

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

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

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

Key Terms

- confusion matrix (https://developers.google.com/machine-learning/glossary#confusion_matrix)
- negative class (https://developers.google.com/machine-learning/glossary#negative_class)
- positive class (https://developers.google.com/machine-learning/glossary#p
- false negative (https://developers.google.com/machine-learning/glossary#FN)
- false positive (https://developers.google.com/machine-learning/glossary#FP)
- true negative (https://developers.google.com/machine-learning/glossary#TN)
- true positive (https://developers.google.com/machine-learning/glossary#TP)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Thresholding
(https://developers.google.com/machine-learning/crash-course/classification/thresholding)

Next
Accuracy →
(https://developers.google.com/machine-learning/crash-course/classification/accuracy)

Classification: Accuracy

Estimated Time: 6 minutes

Accuracy is one metric for evaluating classification models. Informally, **accuracy** is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where *TP* = True Positives, *TN* = True Negatives, *FP* = False Positives, and *FN* = False Negatives.

Let's try calculating accuracy for the following model that classified 100 tumors as malignant (https://wikipedia.org/wiki/Malignancy) (the positive class) or benign (https://wikipedia.org/wiki/Benign_tumor) (the negative class):

True Positive (TP): • Reality: Malignant • ML model predicted: Malignant • **Number of TP results: 1**	**False Positive (FP):** • Reality: Benign • ML model predicted: Malignant • **Number of FP results: 1**
False Negative (FN): • Reality: Malignant • ML model predicted: Benign • **Number of FN results: 8**	**True Negative (TN):** • Reality: Benign • ML model predicted: Benign • **Number of TN results: 90**

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1 + 90}{1 + 90 + 1 + 8} = 0.91$$

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

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

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

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

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

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

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

Key Terms

- accuracy (https://developers.google.com/machine-learning/glossary#accuracy)
- class-imbalanced data set (https://developers.google.com/machine-learning/glossary#class_imbalanced_data_set)

Previous
← True vs. False; Positive vs. Negative (https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative)

Next
Precision and Recall → (https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall)

Last updated March 5, 2019.

Classification: Precision and Recall

Estimated Time: 9 minutes

Precision

Precision attempts to answer the following question:

> What proportion of positive identifications was actually correct?

Precision is defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Note: A model that produces no false positives has a precision of 1.0.

Let's calculate precision for our ML model from the previous section (https://developers.google.com/machine-learning/crash-course/classification/accuracy) that analyzes tumors:

True Positives (TPs): 1	**False Positives (FPs): 1**
False Negatives (FNs): 8	**True Negatives (TNs): 90**

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{1}{1 + 1} = 0.5$$

Our model has a precision of 0.5—in other words, when it predicts a tumor is malignant, it is correct 50% of the time.

Recall

Recall attempts to answer the following question:

> What proportion of actual positives was identified correctly?

Mathematically, recall is defined as follows:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Note: A model that produces no false negatives has a recall of 1.0.

Let's calculate recall for our tumor classifier:

True Positives (TPs): 1	**False Positives (FPs): 1**
False Negatives (FNs): 8	**True Negatives (TNs): 90**

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{1}{1 + 8} = 0.11$$

Our model has a recall of 0.11—in other words, it correctly identifies 11% of all malignant tumors.

Precision and Recall: A Tug of War

To fully evaluate the effectiveness of a model, you must examine **both** precision and recall. Unfortunately, precision and recall are often in tension. That is, improving precision typically reduces recall and vice versa. Explore this notion by looking at the following figure, which shows 30 predictions made by an email classification model. Those to the right of the classification threshold are classified as "spam", while those to the left are classified as "not spam."

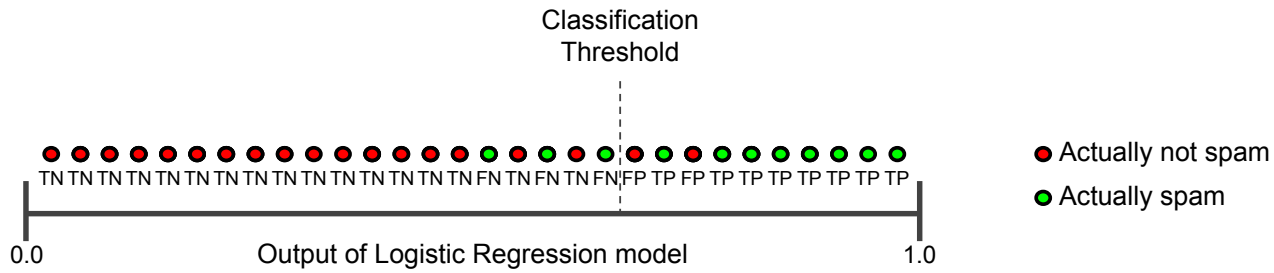


Figure 1. Classifying email messages as spam or not spam.

Let's calculate precision and recall based on the results shown in Figure 1:

True Positives (TP): 8	**False Positives (FP): 2**
False Negatives (FN): 3	**True Negatives (TN): 17**

Precision measures the percentage of **emails flagged as spam** that were correctly classified—that is, the percentage of dots to the right of the threshold line that are green in Figure 1:

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{8}{8 + 2} = 0.8$$

Recall measures the percentage of **actual spam emails** that were correctly classified—that is, the percentage of green dots that are to the right of the threshold line in Figure 1:

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{8}{8 + 3} = 0.73$$

Figure 2 illustrates the effect of increasing the classification threshold.

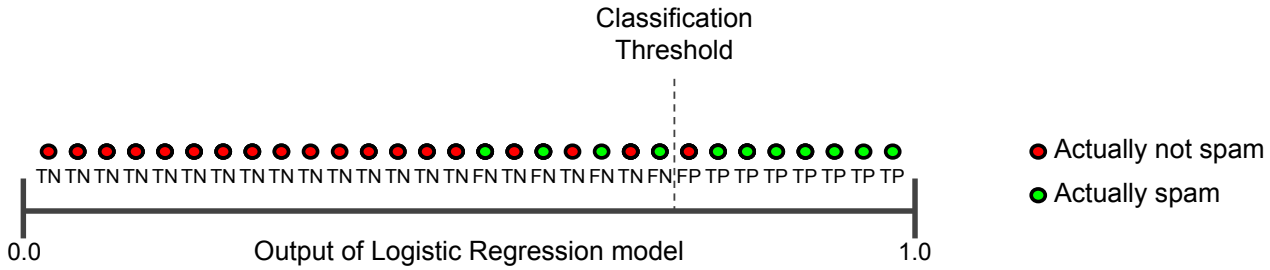


Figure 2. Increasing classification threshold.

The number of false positives decreases, but false negatives increase. As a result, precision increases, while recall decreases:

True Positives (TP): 7	**False Positives (FP): 1**
False Negatives (FN): 4	**True Negatives (TN): 18**

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{7}{7 + 1} = 0.88$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{7}{7 + 4} = 0.64$$

Conversely, Figure 3 illustrates the effect of decreasing the classification threshold (from its original position in Figure 1).

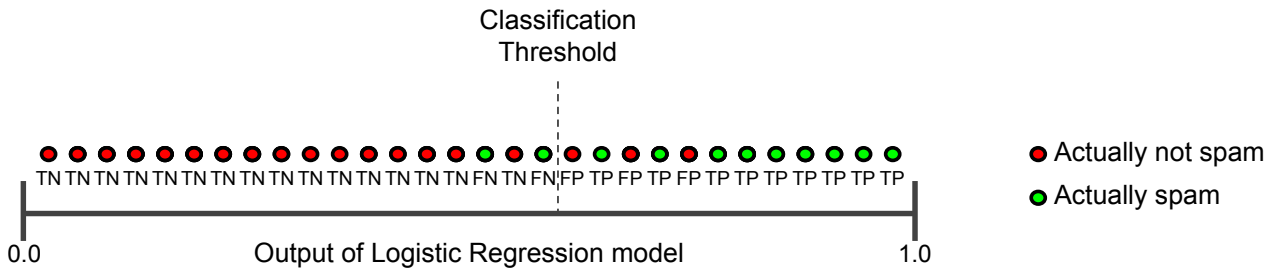


Figure 3. Decreasing classification threshold.

False positives increase, and false negatives decrease. As a result, this time, precision decreases and recall increases:

True Positives (TP): 9	**False Positives (FP): 3**
False Negatives (FN): 2	**True Negatives (TN): 16**

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{9}{9 + 3} = 0.75$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{9}{9 + 2} = 0.82$$

Various metrics have been developed that rely on both precision and recall. For example, see F1 score (https://wikipedia.org/wiki/F1_score).

Key Terms

- precision (https://developers.google.com/machine-learning/glossary#precision)
- recall (https://developers.google.com/machine-learning/glossary#recall)

Previous
← Accuracy
(https://developers.google.com/machine-learning/crash-course/classification/accuracy)

Next
Check Your Understanding: Accuracy, Precision, Recall →
(https://developers.google.com/machine-learning/crash-course/classification/check-your-understanding-accuracy-precision-recall)

Last updated March 5, 2019.

Classification: Check Your Understanding (Accuracy, Precision, Recall)

Estimated Time: 10 minutes

Accuracy

Explore the options below.

In which of the following scenarios would a high accuracy value suggest that the ML model is doing a good job?

🚫 An expensive robotic chicken crosses a very busy road a thousand times per day. An ML model evaluates traffic patterns and predicts when this chicken can safely cross the street with an accuracy of 99.99%.

A 99.99% accuracy value on a very busy road strongly suggests that the ML model is far better than chance. In some settings, however, the cost of making even a small number of mistakes is still too high. 99.99% accuracy means that the expensive chicken will need to be replaced, on average, every 10 days. (The chicken might also cause extensive damage to cars that it hits.)

Try again.

✓ In the game of roulette, a ball is dropped on a spinning wheel and eventually lands in one of 38 slots. Using visual features (the spin of the ball, the position of the wheel when the ball was dropped, the height of the ball over the wheel), an ML model can predict the slot that the ball will land in with an accuracy of 4%.

This ML model is making predictions far better than chance; a random guess would be correct 1/38 of the time—yielding an accuracy of 2.6%. Although the model's accuracy is "only" 4%, the benefits of success far outweigh the disadvantages of failure.

Correct answer.

🚫 A deadly, but curable, medical condition afflicts .01% of the population. An ML model uses symptoms as features and predicts this affliction with an accuracy of 99.99%.

Accuracy is a poor metric here. After all, even a "dumb" model that always predicts "not sick" would still be 99.99% accurate. Mistakenly predicting "not sick" for a person who actually is sick could be deadly.

Try again.

Precision

Explore the options below.

Consider a classification model that separates email into two categories: "spam" or "not spam." If you raise the classification threshold, what will happen to precision?

🚫 Definitely decrease.

In general, raising the classification threshold reduces false positives, thus raising precision.

Try again.

🚫 Probably decrease.

In general, raising the classification threshold reduces false positives, thus raising precision.

Try again.

🚫 Definitely increase.

Raising the classification threshold typically increases precision; however, precision is not guaranteed to increase monotonically as we raise the threshold.

Try again.

✓ Probably increase.

In general, raising the classification threshold reduces false positives, thus raising precision.

Correct answer.

Recall

Explore the options below.

Consider a classification model that separates email into two categories: "spam" or "not spam." If you raise the classification threshold, what will happen to recall?

🚫 Always stay constant.

Raising our classification threshold will cause the number of true positives to decrease or stay the same and will cause the number of false negatives to increase or stay the same. Thus, recall will either stay constant or decrease.

Try again.

✓ Always decrease or stay the same.

Raising our classification threshold will cause the number of true positives to decrease or stay the same and will cause the number of false negatives to increase or stay the same. Thus, recall will either stay constant or decrease.

Correct answer.

🚫 Always increase.

Raising the classification threshold will cause both of the following:

- The number of true positives will decrease or stay the same.
- The number of false negatives will increase or stay the same.

Thus, recall will never increase.

Try again.

Precision and Recall

Explore the options below.

Consider two models—A and B—that each evaluate the same dataset. Which one of the following statements is true?

🚫 If model A has better recall than model B, then model A is better.

While better recall is good, it might be coming at the expense of a large reduction in precision. In general, we need to look at both precision and recall together, or summary metrics like AUC, which we'll talk about next.

Try again.

🚫 If Model A has better precision than model B, then model A is better.

While better precision is good, it might be coming at the expense of a large reduction in recall. In general, we need to look at both precision and recall together, or summary metrics like AUC which we'll talk about next.

Try again.

✓ If model A has better precision and better recall than model B, then model A is probably better.

In general, a model that outperforms another model on both precision and recall is likely the better model. Obviously, we'll need to make sure that comparison is being done at a precision / recall point that is useful in practice for this to be meaningful. For example, suppose our spam detection model needs to have at least 90% precision to be useful and avoid unnecessary false alarms. In this case, comparing one model at {20% precision, 99% recall} to another at {15% precision, 98% recall} is not particularly instructive, as neither model meets the 90% precision requirement. But with that caveat in mind, this is a good way to think about comparing models when using precision and recall.

Correct answer.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Precision and Recall
(https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall)

Next
ROC Curve and AUC →
(https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc)

Last updated May 7, 2019.

Classification: ROC Curve and AUC

Estimated Time: 8 minutes

ROC curve

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

- True Positive Rate
- False Positive Rate

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

$$TPR = \frac{TP}{TP + FN}$$

False Positive Rate (**FPR**) is defined as follows:

$$FPR = \frac{FP}{FP + TN}$$

An ROC curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives. The following figure shows a typical ROC curve.

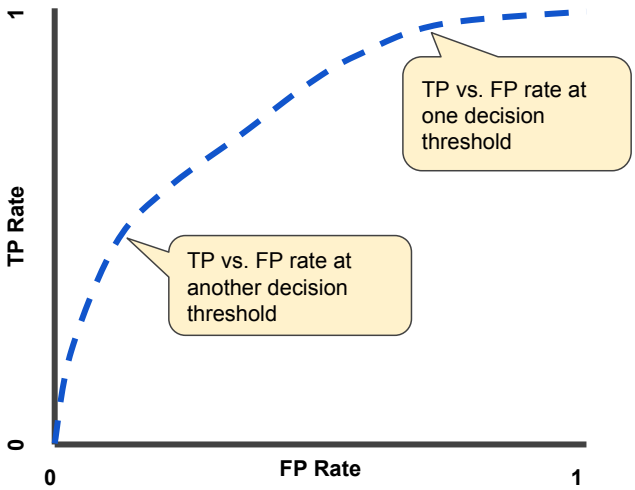


Figure 4. TP vs. FP rate at different classification thresholds.

To compute the points in an ROC curve, we could evaluate a logistic regression model many times with different classification thresholds, but this would be inefficient. Fortunately, there's an efficient, sorting-based algorithm that can provide this information for us, called AUC.

AUC: Area Under the ROC Curve

AUC stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1).

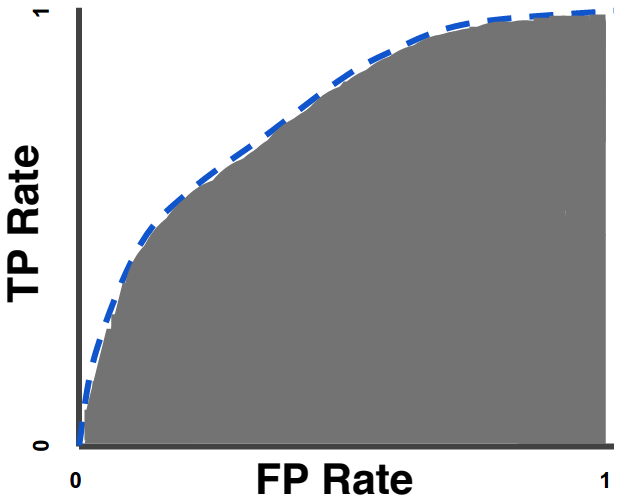


Figure 5. AUC (Area under the ROC Curve).

AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative example. For example, given the following examples, which are arranged from left to right in ascending order of logistic regression predictions:

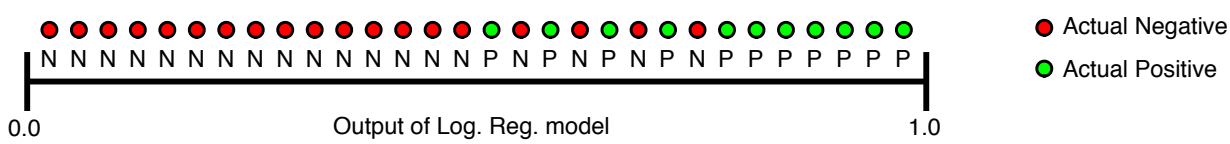


Figure 6. Predictions ranked in ascending order of logistic regression score.

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

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

AUC is desirable for the following two reasons:

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

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

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

Key Terms

- AUC (https://developers.google.com/machine-learning/glossary#AUC)
- ROC curve (https://developers.google.com/machine-learning/glossary#ROC)

Previous
← Check Your Understanding: Accuracy, Precision, Recall
(https://developers.google.com/machine-learning/crash-course/classification/check-your-understanding-accuracy-precision-recall)

Next
Check Your Understanding: ROC and AUC →
(https://developers.google.com/machine-learning/crash-course/classification/check-your-understanding-roc-and-auc)

Last updated March 5, 2019.

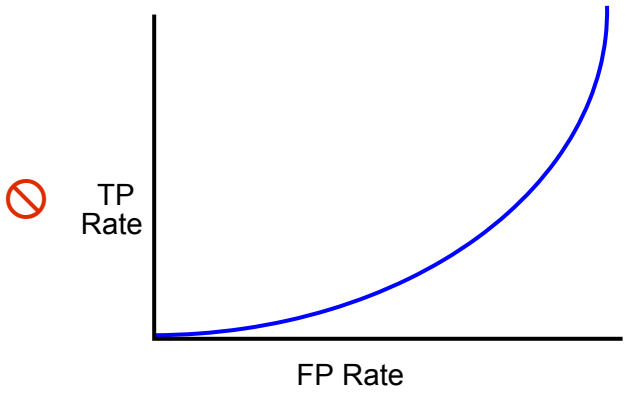
Classification: Check Your Understanding (ROC and AUC)

Estimated Time: 5 minutes

ROC and AUC

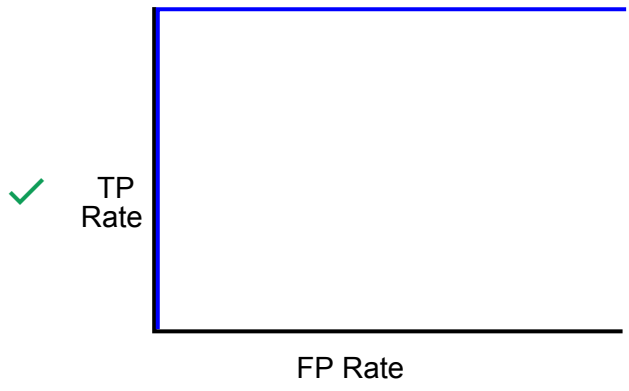
Explore the options below.

Which of the following ROC curves produce AUC values greater than 0.5?



This ROC curve has an AUC between 0 and 0.5, meaning it ranks a random positive example higher than a random negative example less than 50% of the time. The corresponding model actually performs worse than random guessing! If you see an ROC curve like this, it likely indicates there's a bug in your data.

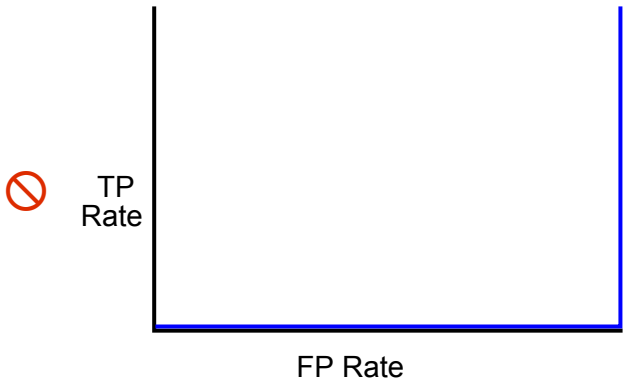
Try again.



This is the best possible ROC curve, as it ranks all positives above all negatives. It has an AUC of 1.0.

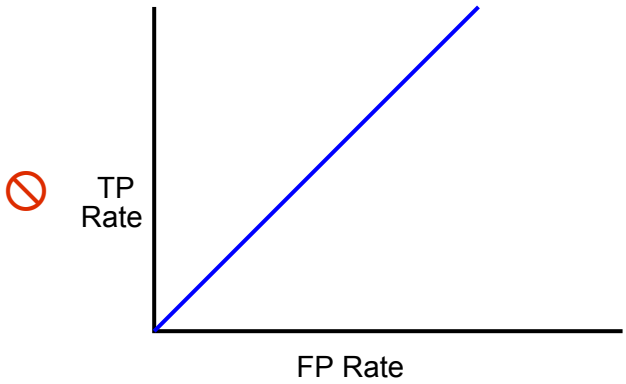
In practice, if you have a "perfect" classifier with an AUC of 1.0, you should be suspicious, as it likely indicates a bug in your model. For example, you may have overfit to your training data, or the label data may be replicated in one of your features.

1 of 2 correct answers.



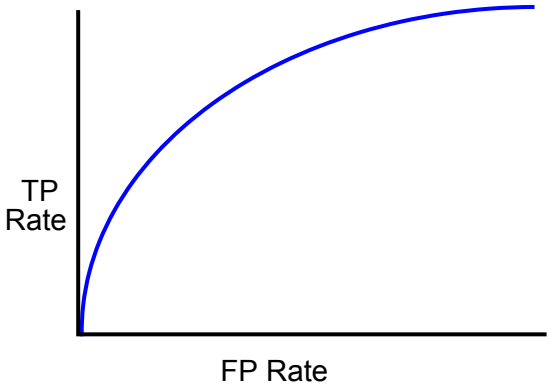
This is the worst possible ROC curve; it ranks all negatives above all positives, and has an AUC of 0.0. If you were to reverse every prediction (flip negatives to positives and postives to negatives), you'd actually have a perfect classifier!

Try again.



This ROC curve has an AUC of 0.5, meaning it ranks a random positive example higher than a random negative example 50% of the time. As such, the corresponding classification model is basically worthless, as its predictive ability is no better than random guessing.

Try again.



This ROC curve has an AUC between 0.5 and 1.0, meaning it ranks a random positive example higher than a random negative example more than 50% of the time. Real-world binary classification AUC values generally fall into this range.

2 of 2 correct answers.

AUC and Scaling Predictions

Explore the options below.

How would multiplying all of the predictions from a given model by 2.0 (for example, if the model predicts 0.4, we multiply by 2.0 to get a prediction of 0.8) change the model's performance as measured by AUC?

It would make AUC terrible, since the prediction values are now way off.

Interestingly enough, even though the prediction values are different (and likely farther from the truth), multiplying them all by 2.0 would keep the relative ordering of prediction values the same. Since AUC only cares about relative rankings, it is not impacted by any simple scaling of the predictions.

Try again.

It would make AUC better, because the prediction values are all farther apart.

The amount of spread between predictions does not actually impact AUC. Even a prediction score for a randomly drawn true positive is only a tiny epsilon greater than a randomly drawn negative, that will count that as a success contributing to the overall AUC score.

Try again.

✓ No change. AUC only cares about relative prediction scores.

Yes, AUC is based on the relative predictions, so any transformation of the predictions that preserves the relative ranking has no effect on AUC. This is clearly not the case for other metrics such as squared error, log loss, or prediction bias (discussed later).

Correct answer.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← ROC Curve and AUC
(https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc)

Next
Prediction Bias →
(https://developers.google.com/machine-learning/crash-course/classification/prediction-bias)

Classification: Prediction Bias

Estimated Time: 7 minutes

Logistic regression predictions should be unbiased. That is:

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

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

$$\text{prediction bias} = \text{average of predictions} - \text{average of labels in data set}$$

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

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

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

Possible root causes of prediction bias are:

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

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

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

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

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

Bucketing and Prediction Bias

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

You can form buckets in the following ways:

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

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

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

Both axes are logarithmic scales.

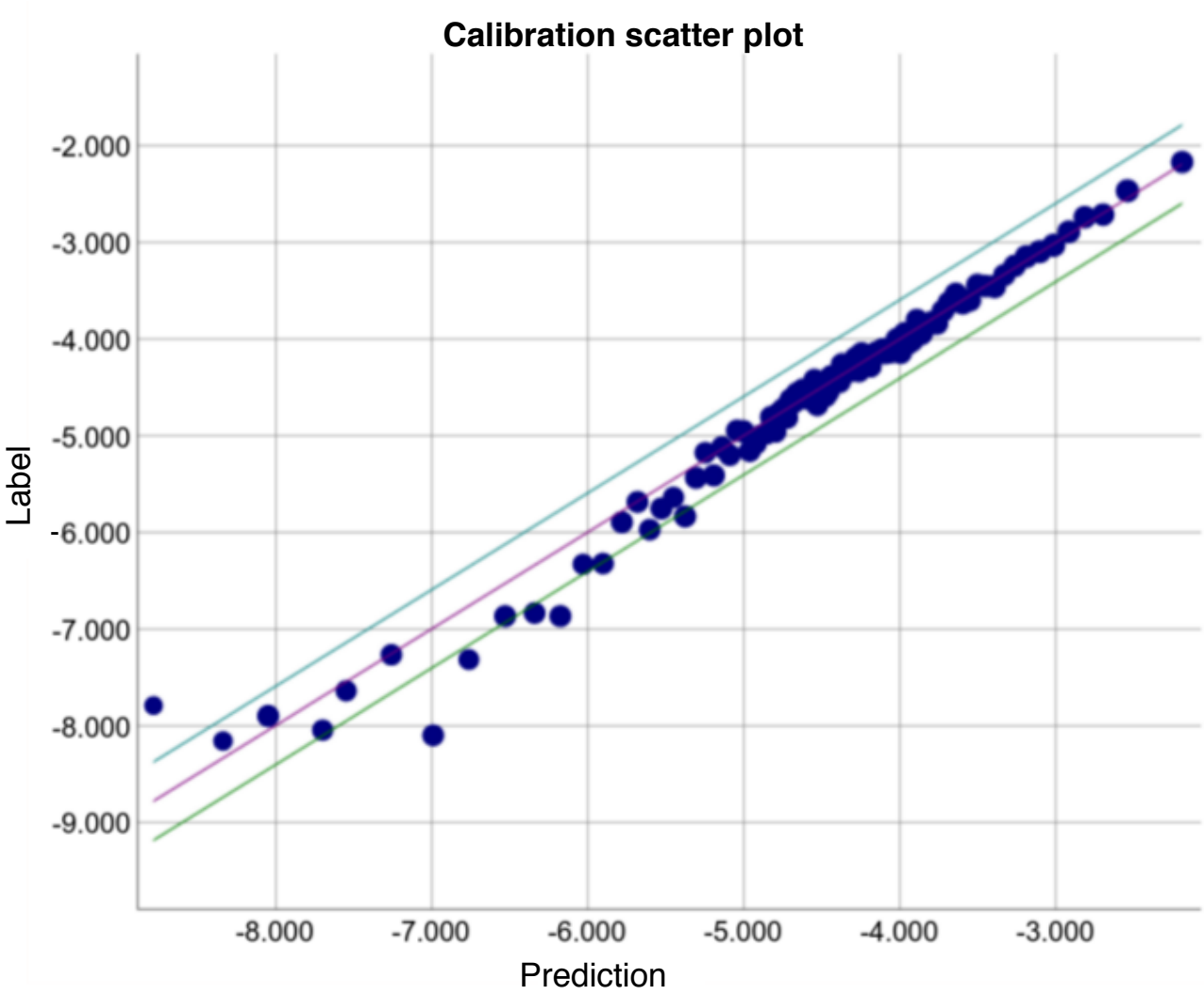


Figure 8. Prediction bias curve (logarithmic scales)

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

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

Key Terms

- bucketing (https://developers.google.com/machine-learning/glossary#bucketing)
- prediction bias
- calibration layer (https://developers.google.com/machine-learning/glossary#calibration_layer)

(https://developers.google.com/machine-learning/glossary#prediction_bias)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Check Your Understanding: ROC and AUC
(https://developers.google.com/machine-learning/crash-course/classification/check-your-understanding-roc-and-auc)

Next
Programming Exercise →
(https://developers.google.com/machine-learning/crash-course/classification/programming-exercise)

Last updated March 5, 2019.

Regularization for Sparsity: L_1 Regularization

Estimated Time: 5 minutes

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

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

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

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

Would L_2 regularization accomplish this task? Unfortunately not. L_2 regularization encourages weights to be small, but doesn't force them to exactly 0.0.

An alternative idea would be to try and create a regularization term that penalizes the count of non-zero coefficient values in a model. Increasing this count would only be justified if there was a sufficient gain in the model's ability to fit the data. Unfortunately, while this count-based approach is intuitively appealing, it would turn our convex optimization problem into a non-convex optimization problem that's NP-hard (https://wikipedia.org/wiki/NP-hardness). (If you squint, you can see a connection to the knapsack problem.) So this idea, known as L_0 regularization isn't something we can use effectively in practice.

However, there is a regularization term called L_1 regularization that serves as an approximation to L_0, but has the advantage of being convex and thus efficient to compute. So we can use L_1 regularization to encourage many of the uninformative coefficients in our model to be exactly 0, and thus reap RAM savings at inference time.

L_1 vs L_2 regularization.

L_2 and L_1 penalize weights differently:

- L_2 penalizes $weight^2$.
- L_1 penalizes $|weight|$.

Consequently, L_2 and L_1 have different derivatives:

- The derivative of L_2 is 2 * *weight*.
- The derivative of L_1 is *k* (a constant, whose value is independent of weight).

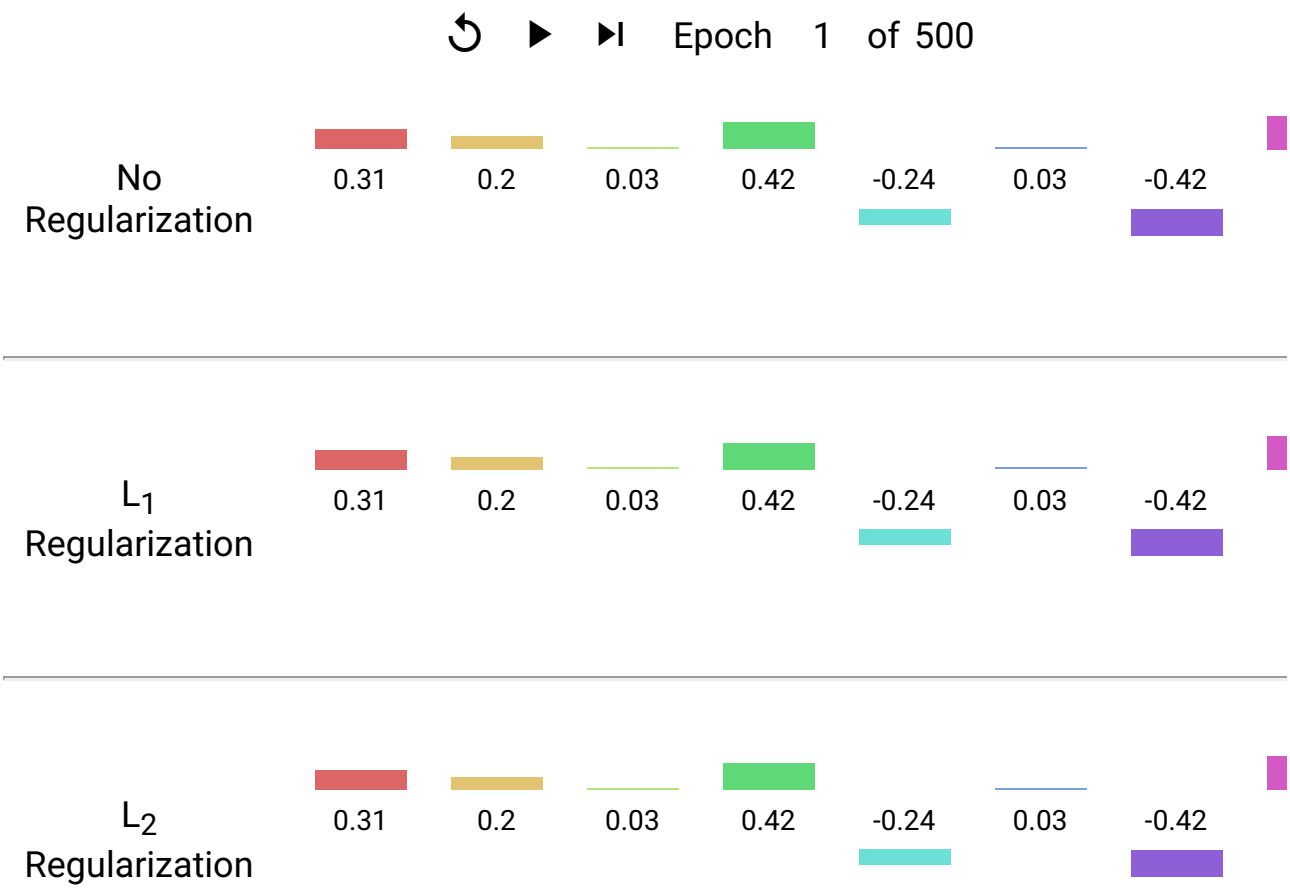
You can think of the derivative of L_2 as a force that removes x% of the weight every time. As Zeno (https://wikipedia.org/wiki/Zeno%27s_paradoxes#Dichotomy_paradox) knew, even if you remove x percent of a number *billions of times*, the diminished number will still never quite reach zero. (Zeno was less familiar with floating-point precision limitations, which could possibly produce exactly zero.) At any rate, L_2 does not normally drive weights to zero.

You can think of the derivative of L_1 as a force that subtracts some constant from the weight every time. However, thanks to absolute values, L_1 has a discontinuity at 0, which causes subtraction results that cross 0 to become zeroed out. For example, if subtraction would have forced a weight from +0.1 to -0.2, L_1 will set the weight to exactly 0. Eureka, L_1 zeroed out the weight.

L_1 regularization—penalizing the absolute value of all the weights—turns out to be quite efficient for wide models.

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

Click the Play button (▶) below to compare the effect L_1 and L_2 regularization have on a network of weights.



Key Terms

- convex optimization (https://developers.google.com/machine-learning/glossary#convex_optimization)
- L_2 regularization (https://developers.google.com/machine-learning/glossary#L2_regularization)
- L_1 regularization (https://developers.google.com/machine-learning/glossary#L1_regularization)
- one-hot encoding (https://developers.google.com/machine-learning/glossary/#one-hot_encoding)
- weight (https://developers.google.com/machine-learning/glossary#weight)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous

← Video Lecture
(https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/video-lecture)

Next: Playground Exercise →
(https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/playground-exercise)

Last updated March 5, 2019.

Regularization for Sparsity: Playground Exercise

Estimated Time: 8 minutes

Examining L_1 Regularization

This exercise contains a small, slightly noisy, training data set. In this kind of setting, overfitting is a real concern. Regularization might help, but which form of regularization?

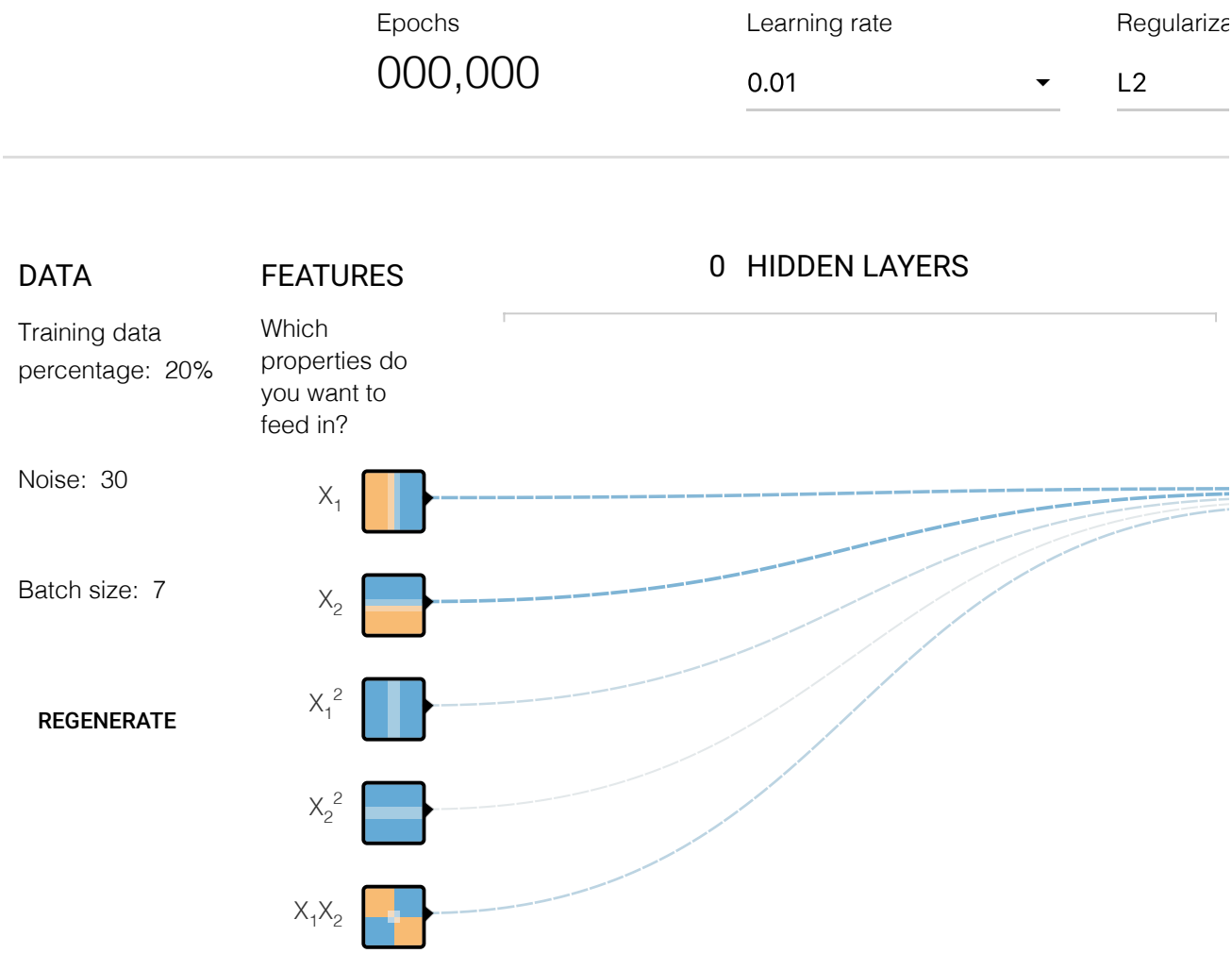
This exercise consists of five related tasks. To simplify comparisons across the five tasks, run each task in a separate tab. Notice that the thicknesses of the lines connecting FEATURES and OUTPUT represent the relative weights of each feature.

Task	Regularization Type	Regularization Rate (lambda)
1	L_2	0.1
2	L_2	0.3
3	L_1	0.1
4	L_1	0.3
5	L_1	experiment

Questions:

1. How does switching from L_2 to L_1 regularization influence the delta between test loss and training loss?
2. How does switching from L_2 to L_1 regularization influence the learned weights?
3. How does increasing the L_1 regularization rate (lambda) influence the learned weights?

(Answers appear just below the exercise.)



Click the dropdown arrow for answers.

1. Switching from L_2 to L_1 regularization dramatically reduces the delta between test loss and training loss.
2. Switching from L_2 to L_1 regularization dampens all of the learned weights.
3. Increasing the L_1 regularization rate generally dampens the learned weights; however, if the regularization rate goes too high, the model can't converge and

losses are very high.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← L1 Regularization
(https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/l1-regularization)

Next
Programming Exercise →
(https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/programming-exercise)

Regularization for Sparsity: Check Your Understanding

Estimated Time: 5 minutes

L_1 regularization

Explore the options below.

Imagine a linear model with 100 input features:

- 10 are highly informative.
- 90 are non-informative.

Assume that all features have values between -1 and 1. Which of the following statements are true?

✓ L1 regularization will encourage most of the non-informative weights to be exactly 0.0.

L1 regularization of sufficient lambda tends to encourage non-informative weights to become exactly 0.0. By doing so, these non-informative features leave the model.

1 of 2 correct answers.

🚫 L1 regularization will encourage many of the non-informative weights to be nearly (but not exactly) 0.0.

In general, L1 regularization of sufficient lambda tends to encourage non-informative features to weights of exactly 0.0. Unlike L2 regularization, L1 regularization "pushes" just as hard toward 0.0 no matter how far the weight is from 0.0.

Try again.

✓ L1 regularization may cause informative features to get a weight of exactly 0.0.

Be careful--L1 regularization may cause the following kinds of features to be given weights of exactly 0:

- Weakly informative features.
- Strongly informative features on different scales.
- Informative features strongly correlated with other similarly informative features.

2 of 2 correct answers.

L_1 vs. L_2 Regularization

Explore the options below.

Imagine a linear model with 100 input features, all having values between -1 and 1:

- 10 are highly informative.
- 90 are non-informative.

Which type of regularization will produce the smaller model?

✓ L_1 regularization.

L_1 regularization tends to reduce the number of features. In other words, L_1 regularization often reduces the model size.

Correct answer.

🚫 L_2 regularization.

L_2 regularization rarely reduces the number of features. In other words, L_2 regularization rarely reduces the model size.

Try again.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Programming Exercise
(https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/programming-exercise)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/video-lecture)

Last updated March 5, 2019.

Neural Networks: Structure

Estimated Time: 7 minutes

If you recall from the Feature Crosses unit (https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture), the following classification problem is nonlinear:

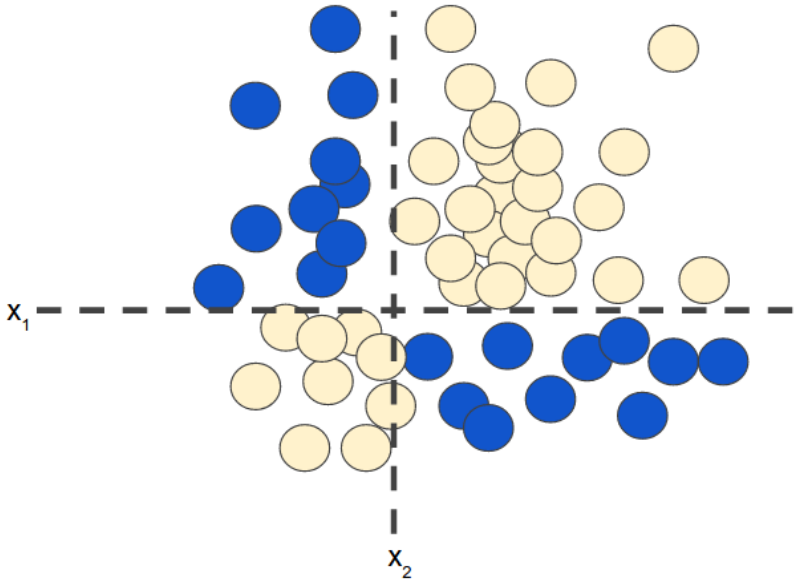


Figure 1. Nonlinear classification problem.

"Nonlinear" means that you can't accurately predict a label with a model of the form $b + w_1 x_1 + w_2 x_2$ In other words, the "decision surface" is not a line. Previously, we looked at feature crosses (https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture) as one possible approach to modeling nonlinear problems.

Now consider the following data set:

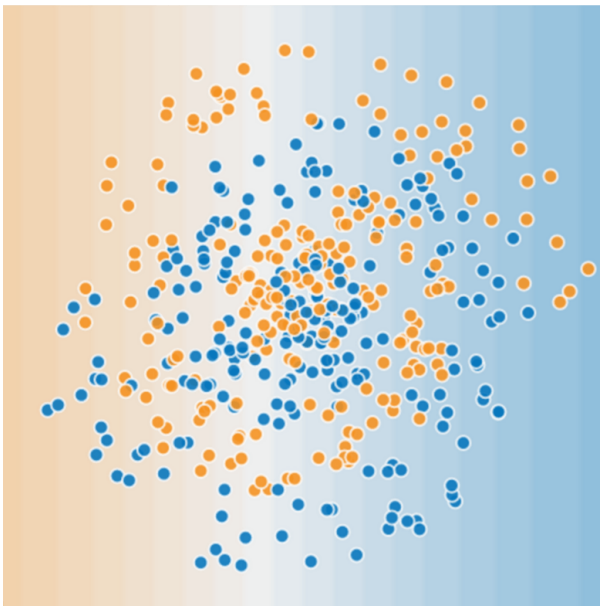


Figure 2. A more difficult nonlinear classification problem.

The data set shown in Figure 2 can't be solved with a linear model.

To see how neural networks might help with nonlinear problems, let's start by representing a linear model as a graph:

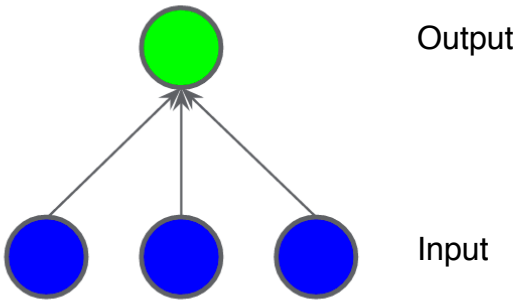


Figure 3. Linear model as graph.

Each blue circle represents an input feature, and the green circle represents the weighted sum of the inputs.

How can we alter this model to improve its ability to deal with nonlinear problems?

Hidden Layers

In the model represented by the following graph, we've added a "hidden layer" of intermediary values. Each yellow node in the hidden layer is a weighted sum of the blue

input node values. The output is a weighted sum of the yellow nodes.

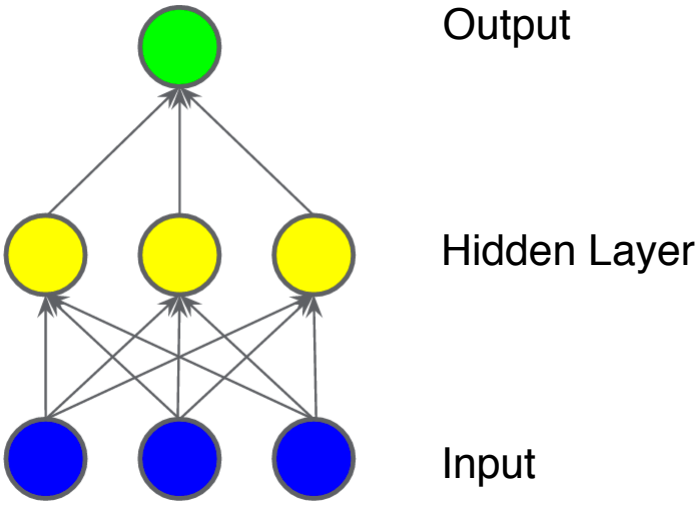


Figure 4. Graph of two-layer model.

Is this model linear? Yes—its output is still a linear combination of its inputs.

In the model represented by the following graph, we've added a second hidden layer of weighted sums.

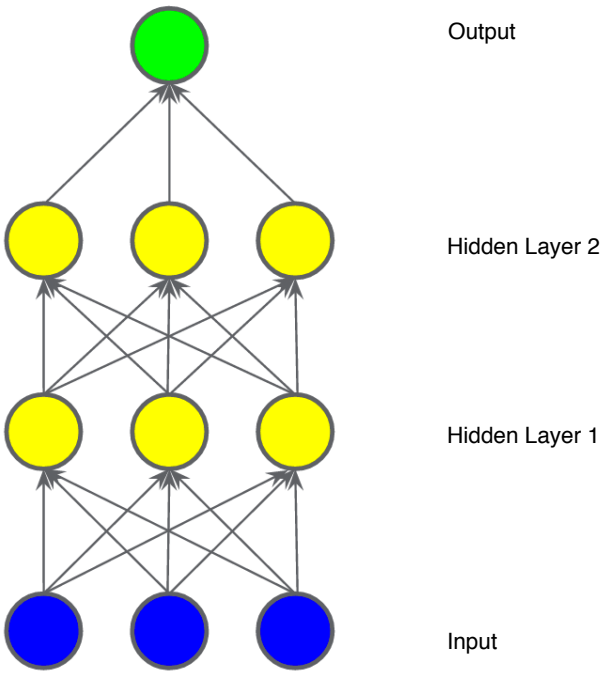


Figure 5. Graph of three-layer model.

Is this model still linear? Yes, it is. When you express the output as a function of the input and simplify, you get just another weighted sum of the inputs. This sum won't effectively model the nonlinear problem in Figure 2.

Activation Functions

To model a nonlinear problem, we can directly introduce a nonlinearity. We can pipe each hidden layer node through a nonlinear function.

In the model represented by the following graph, the value of each node in Hidden Layer 1 is transformed by a nonlinear function before being passed on to the weighted sums of the next layer. This nonlinear function is called the activation function.

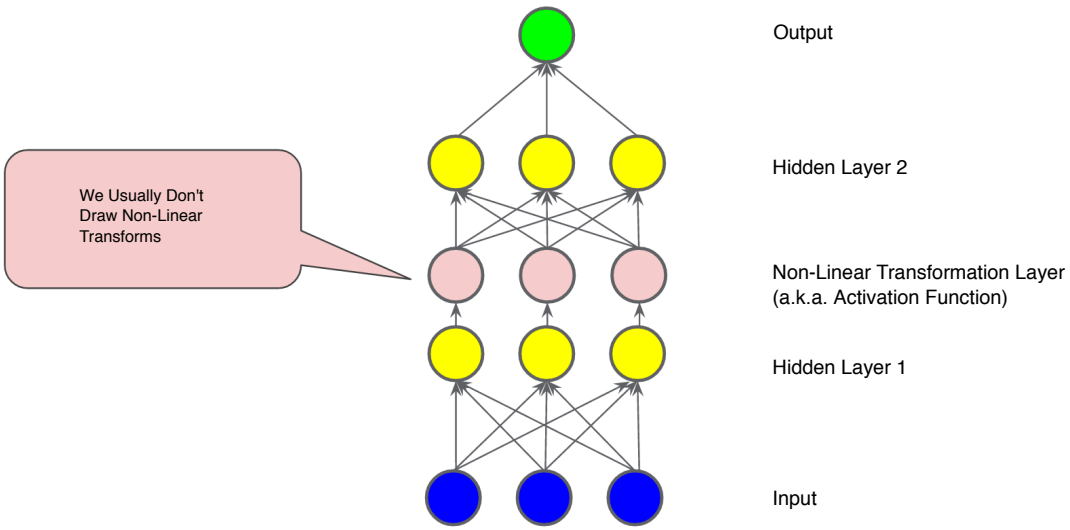


Figure 6. Graph of three-layer model with activation function.

Now that we've added an activation function, adding layers has more impact. Stacking nonlinearities on nonlinearities lets us model very complicated relationships between the inputs and the predicted outputs. In brief, each layer is effectively learning a more complex, higher-level function over the raw inputs. If you'd like to develop more intuition on how this works, see Chris Olah's excellent blog post (http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/).

Common Activation Functions

The following **sigmoid** activation function converts the weighted sum to a value between 0 and 1.

$$F(x) = \frac{1}{1 + e^{-x}}$$

Here's a plot:

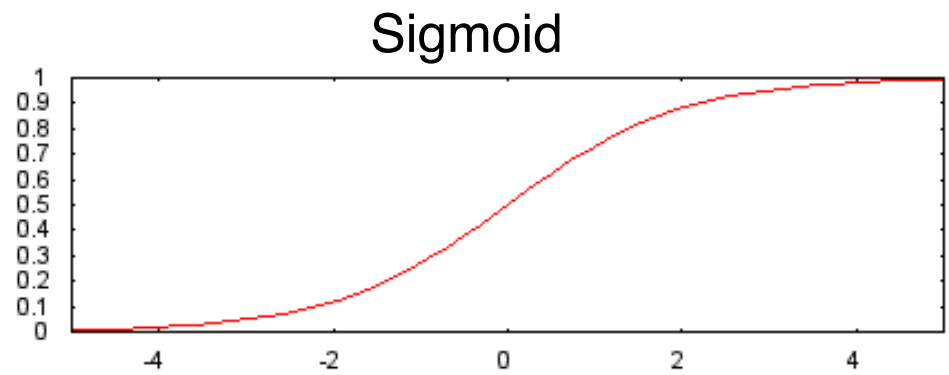


Figure 7. Sigmoid activation function.

The following **rectified linear unit** activation function (or **ReLU**, for short) often works a little better than a smooth function like the sigmoid, while also being significantly easier to compute.

$$F(x) = max(0, x)$$

The superiority of ReLU is based on empirical findings, probably driven by ReLU having a more useful range of responsiveness. A sigmoid's responsiveness falls off relatively quickly on both sides.

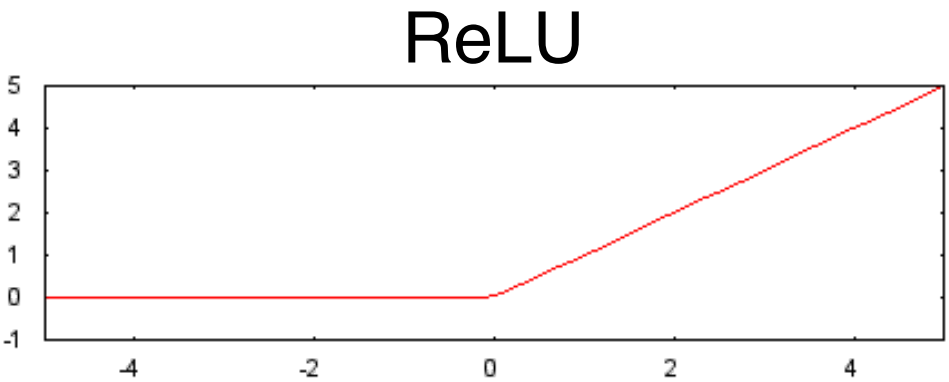


Figure 8. ReLU activation function.

In fact, any mathematical function can serve as an activation function. Suppose that σ represents our activation function (Relu, Sigmoid, or whatever). Consequently, the value of a node in the network is given by the following formula:

$$\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

TensorFlow provides out-of-the-box support for a wide variety of activation functions (https://www.tensorflow.org/api_docs/python/nn.html). That said, we still recommend starting with ReLU.

Summary

Now our model has all the standard components of what people usually mean when they say "neural network":

- A set of nodes, analogous to neurons, organized in layers.
- A set of weights representing the connections between each neural network layer and the layer beneath it. The layer beneath may be another neural network layer, or some other kind of layer.
- A set of biases, one for each node.
- An activation function that transforms the output of each node in a layer. Different layers may have different activation functions.

A caveat: neural networks aren't necessarily always better than feature crosses, but neural networks do offer a flexible alternative that works well in many cases.

Key Terms

- activation function (https://developers.google.com/machine-learning/glossary#activation_function)
- hidden layer (https://developers.google.com/machine-learning/glossary#hidden_layer)
- neural network (https://developers.google.com/machine-learning/glossary#neural_network)
- neuron (https://developers.google.com/machine-learning/glossary#neuron)
- rectified linear unit (ReLU) (https://developers.google.com/machine-learning/glossary#ReLU)
- sigmoid function (https://developers.google.com/machine-learning/glossary#sigmoid_function)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/video-lecture)

Next
Playground Exercises →
(https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/playground-exercises)

Last updated March 5, 2019.

Neural Networks: Playground Exercises

Estimated Time: 20 minutes

A First Neural Network

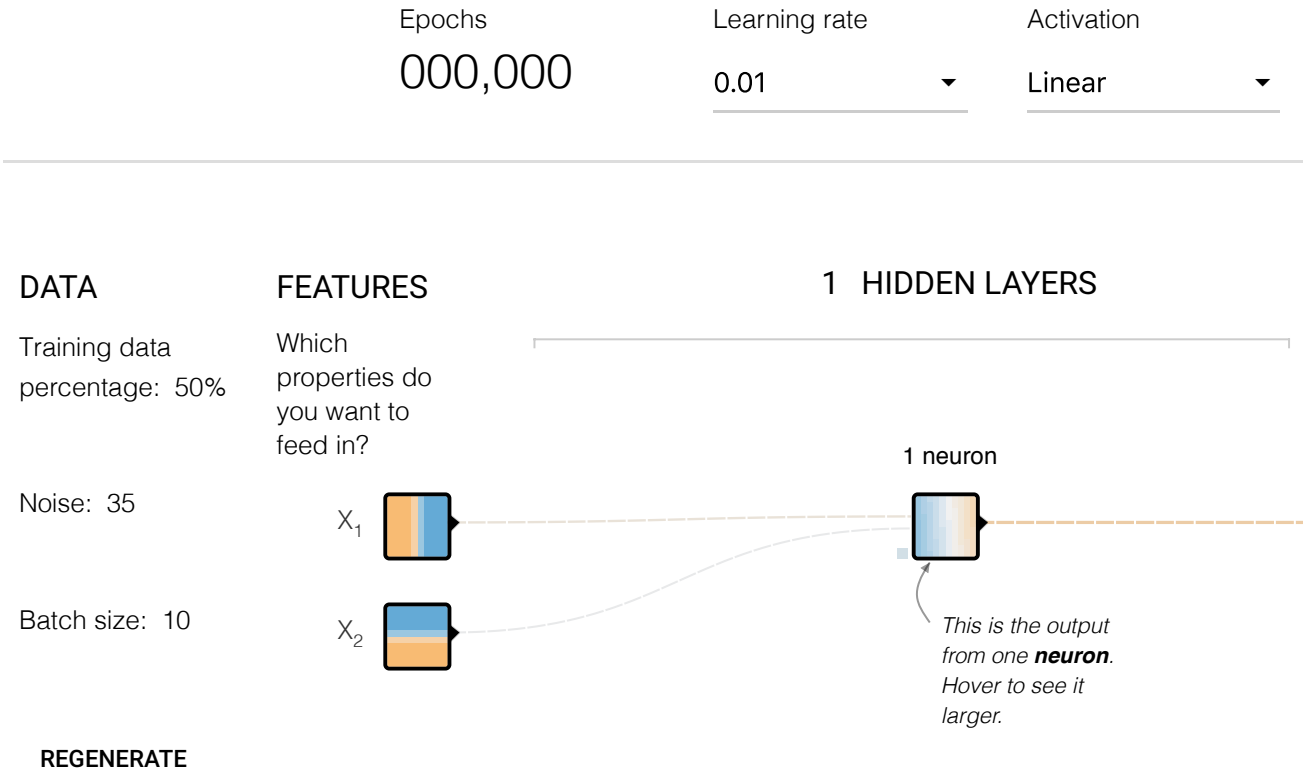
In this exercise, we will train our first little neural net. Neural nets will give us a way to learn nonlinear models without the use of explicit feature crosses.

Task 1: The model as given combines our two input features into a single neuron. Will this model learn any nonlinearities? Run it to confirm your guess.

Task 2: Try increasing the number of neurons in the hidden layer from 1 to 2, and also try changing from a Linear activation to a nonlinear activation like ReLU. Can you create a model that can learn nonlinearities?

Task 3: Continue experimenting by adding or removing hidden layers and neurons per layer. Also feel free to change learning rates, regularization, and other learning settings. What is the smallest number of nodes and layers you can use that gives test loss of 0.177 or lower?

(Answers appear just below the exercise.)



Click the dropdown arrow for an answer to Task 1.

The Activation is set to `Linear`, so this model cannot learn any nonlinearities. The loss is very high.

Click the dropdown arrow for an answer to Task 2.

The nonlinear Activation function can learn nonlinear models. However, a single hidden layer with 2 neurons will take a while to learn the model. These exercises are nondeterministic, so some runs will not learn an effective model, while other runs will do a pretty good job.

Click the dropdown arrow for an answer to Task 3.

Playground's nondeterministic nature shines through on this exercise. Some runs produce very low test loss with 3 Hidden Layers, arranged as follows:

- First layer had 3 neurons.
- Second layer had 3 neurons.
- Third layer had 2 neurons.

However, other runs with the same configuration yielded very high loss.

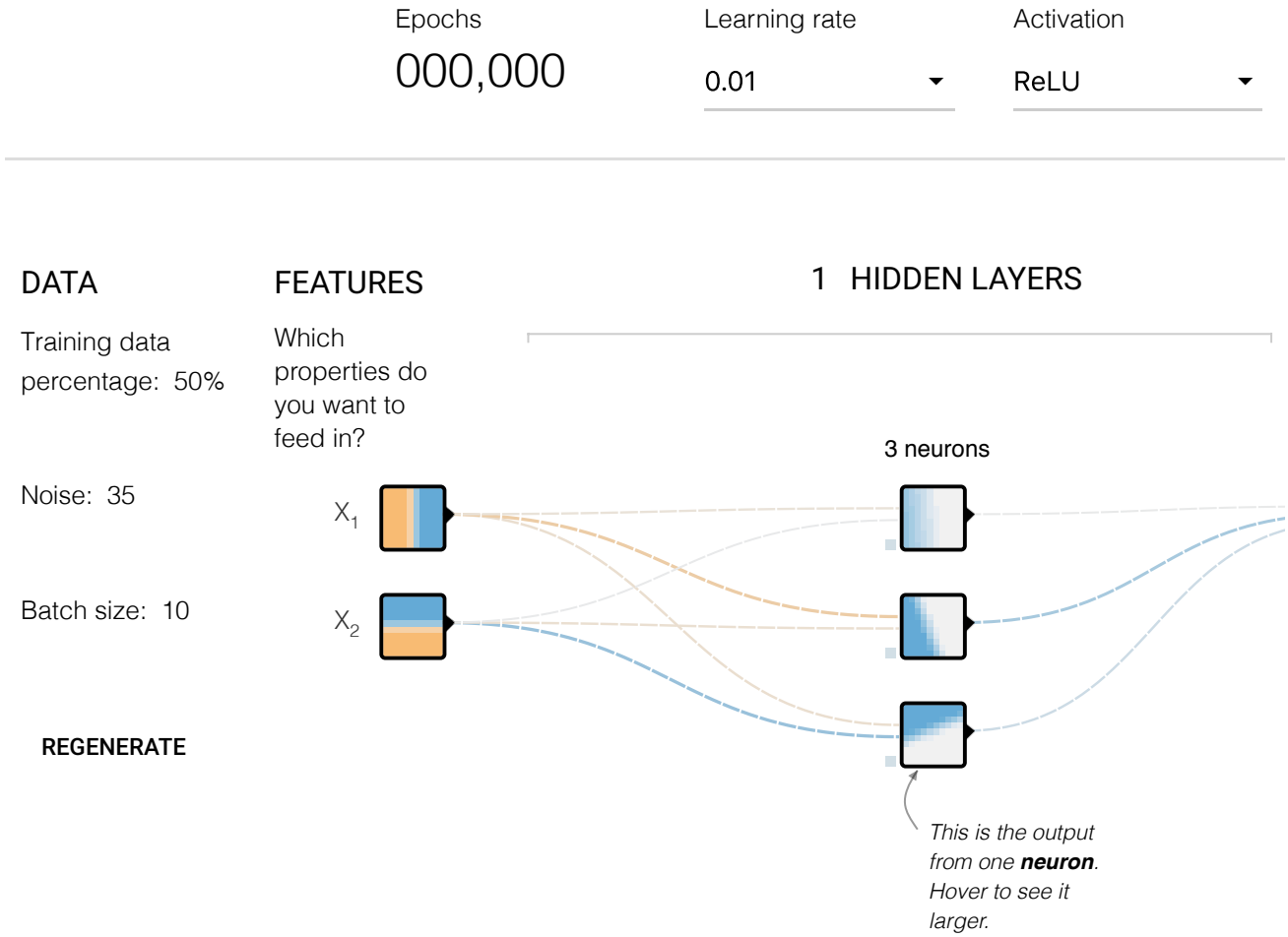
Neural Net Initialization

This exercise uses the XOR data again, but looks at the repeatability of training Neural Nets and the importance of initialization.

Task 1: Run the model as given four or five times. Before each trial, hit the **Reset the network** button to get a new random initialization. (The **Reset the network** button is the circular reset arrow just to the left of the Play button.) Let each trial run for at least 500 steps to ensure convergence. What shape does each model output converge to? What does this say about the role of initialization in non-convex optimization?

Task 2: Try making the model slightly more complex by adding a layer and a couple of extra nodes. Repeat the trials from Task 1. Does this add any additional stability to the results?

(Answers appear just below the exercise.)



Click the dropdown arrow for an answer to Task 1.

The learned model had different shapes on each run. The converged test loss varied almost 2X from lowest to highest.

Click the dropdown arrow for an answer to Task 2.

Adding the layer and extra nodes produced more repeatable results. On each run, the resulting model looked roughly the same. Furthermore, the converged test loss showed less variance between runs.

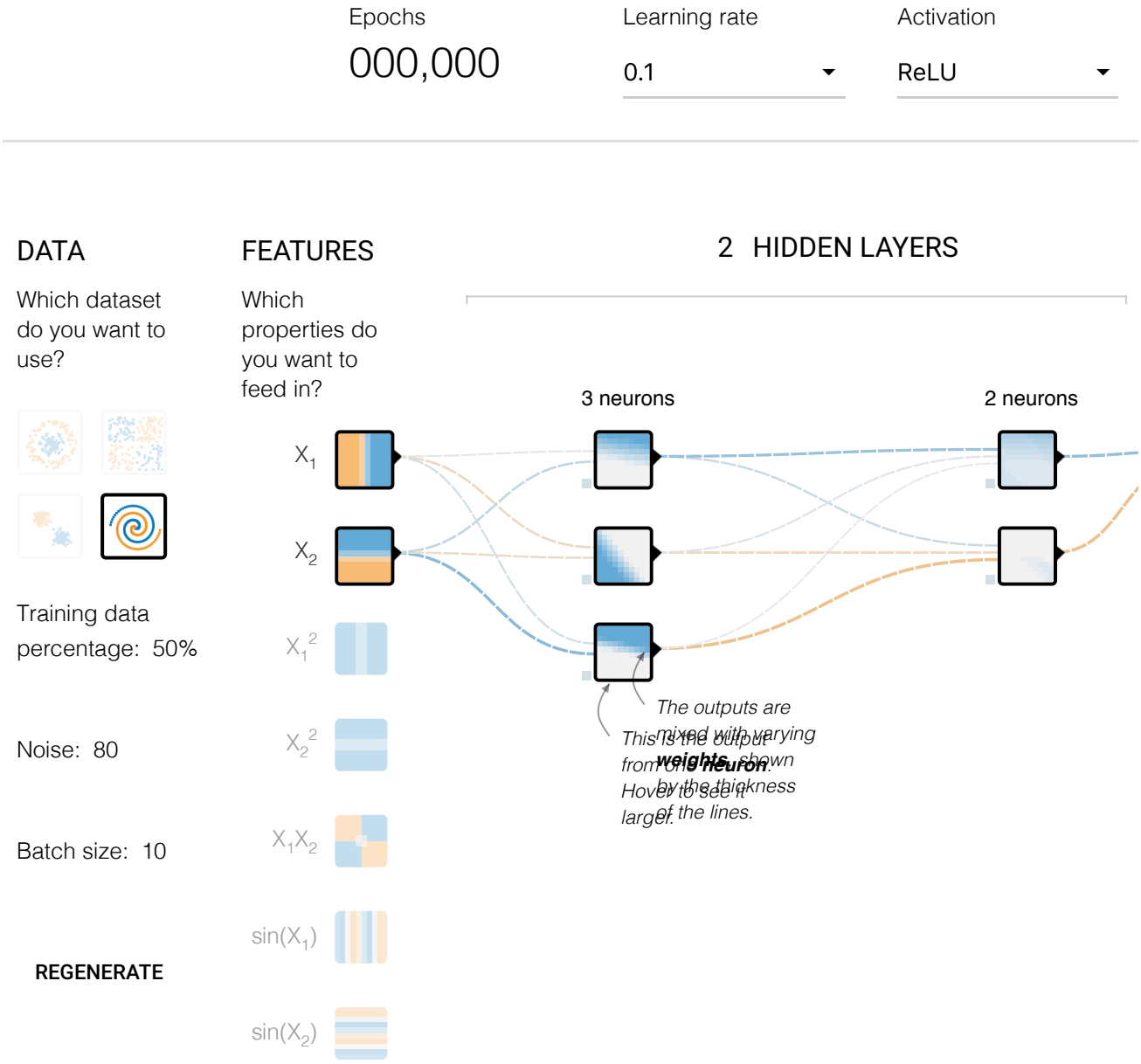
Neural Net Spiral

This data set is a noisy spiral. Obviously, a linear model will fail here, but even manually defined feature crosses may be hard to construct.

Task 1: Train the best model you can, using just X_1 and X_2. Feel free to add or remove layers and neurons, change learning settings like learning rate, regularization rate, and batch size. What is the best test loss you can get? How smooth is the model output surface?

Task 2: Even with Neural Nets, some amount of feature engineering is often needed to achieve best performance. Try adding in additional cross product features or other transformations like $\sin(X_1)$ and $\sin(X_2)$. Do you get a better model? Is the model output surface any smoother?

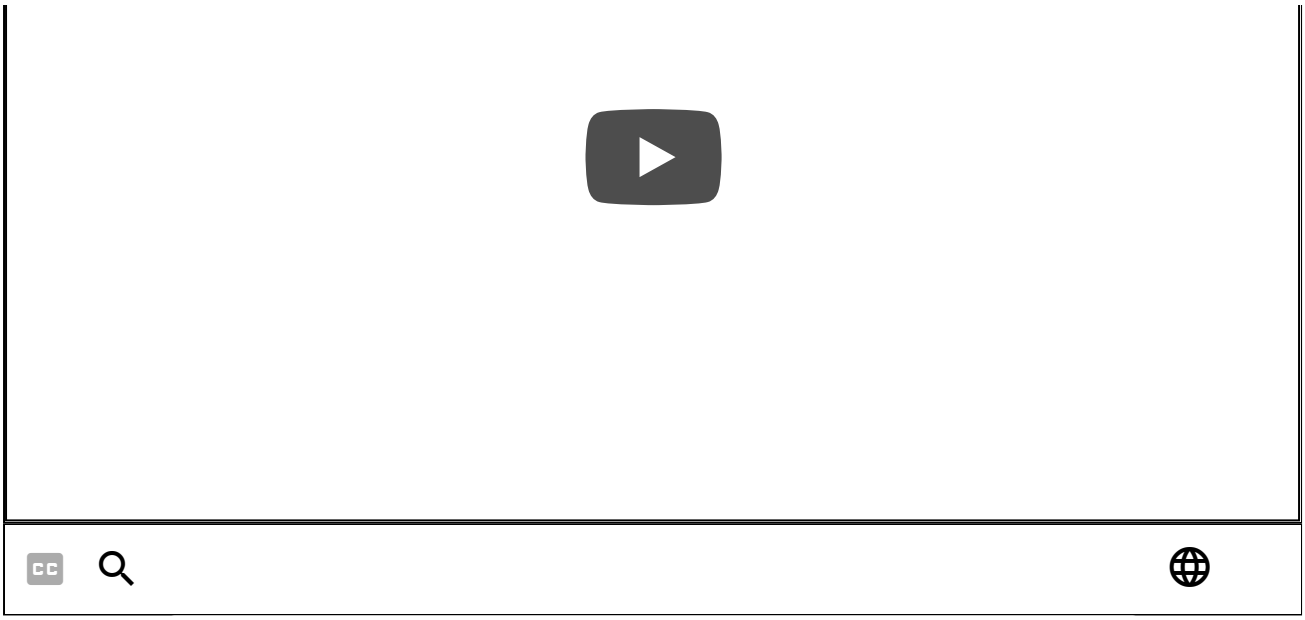
(Answers appear just below the exercise.)



Click the dropdown arrow for possible answers.

The following video walks through how to choose hyperparameters in Playground to train a model for the spiral data that minimizes test loss.

ML Spiral Solution



HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Structure
(https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/anatomy)

Next
Programming Exercise →
(https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/programming-exercise)

Training Neural Networks: Best Practices

Estimated Time: 5 minutes

This section explains backpropagation's failure cases and the most common way to regularize a neural network.

Failure Cases

There are a number of common ways for backpropagation to go wrong.

Vanishing Gradients

The gradients for the lower layers (closer to the input) can become very small. In deep networks, computing these gradients can involve taking the product of many small terms.

When the gradients vanish toward 0 for the lower layers, these layers train very slowly, or not at all.

The ReLU activation function can help prevent vanishing gradients.

Exploding Gradients

If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms. In this case you can have exploding gradients: gradients that get too large to converge.

Batch normalization can help prevent exploding gradients, as can lowering the learning rate.

Dead ReLU Units

Once the weighted sum for a ReLU unit falls below 0, the ReLU unit can get stuck. It outputs 0 activation, contributing nothing to the network's output, and gradients can no longer flow through it during backpropagation. With a source of gradients cut off, the input to the ReLU may not ever change enough to bring the weighted sum back above 0.

Lowering the learning rate can help keep ReLU units from dying.

Dropout Regularization

Yet another form of regularization, called **Dropout**, is useful for neural networks. It works by randomly "dropping out" unit activations in a network for a single gradient step. The more you drop out, the stronger the regularization:

- 0.0 = No dropout regularization.
- 1.0 = Drop out everything. The model learns nothing.
- Values between 0.0 and 1.0 = More useful.

Key Terms

- activation function (https://developers.google.com/machine-learning/glossary#activation_function)
- backpropagation (https://developers.google.com/machine-learning/glossary#backpropagation)
- dropout regularization (https://developers.google.com/machine-learning/glossary#dropout_regularization)
- gradient descent (https://developers.google.com/machine-learning/glossary#gradient_descent)
- rectified linear unit (ReLU) (https://developers.google.com/machine-learning/glossary#ReLU)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/training-neural-networks/video-lecture)

Next
Programming Exercise →
(https://developers.google.com/machine-learning/crash-course/training-neural-networks/programming-exercise)

Last updated March 5, 2019.

Multi-Class Neural Networks

Earlier, you encountered binary classification models that could pick between one of *two* possible choices, such as whether:

- A given email is spam or not spam.
- A given tumor is malignant or benign.

In this module, we'll investigate **multi-class** classification, which can pick from *multiple* possibilities. For example:

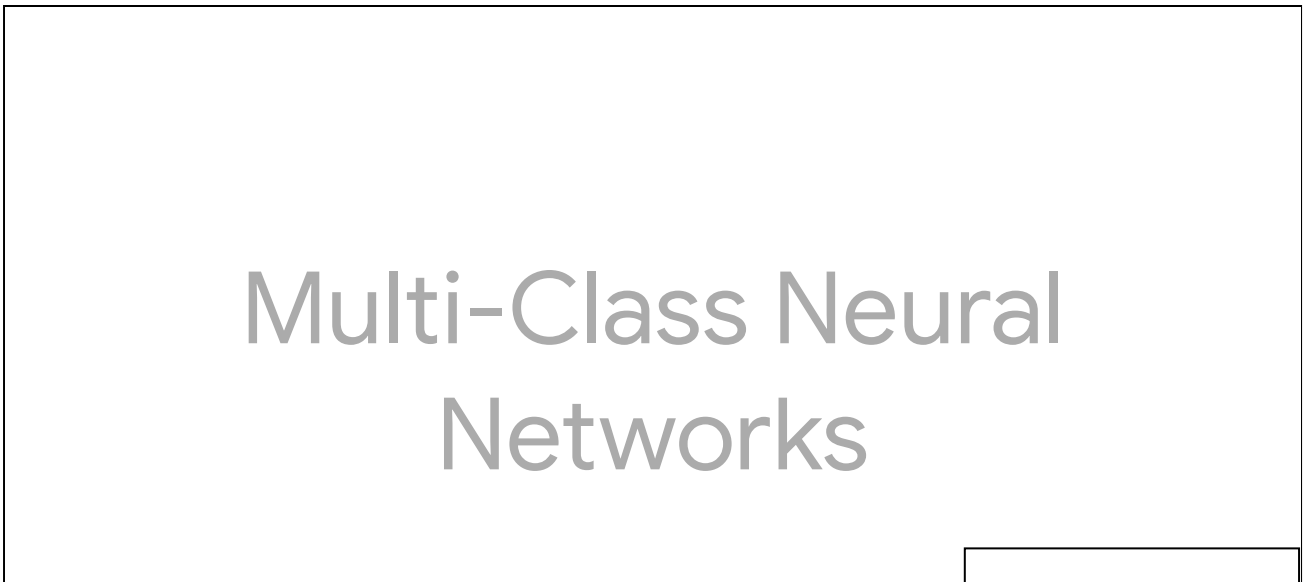
- Is this dog a beagle, a basset hound, or a bloodhound?
- Is this flower a Siberian Iris, Dutch Iris, Blue Flag Iris, or Dwarf Bearded Iris?
- Is that plane a Boeing 747, Airbus 320, Boeing 777, or Embraer 190?
- Is this an image of an apple, bear, candy, dog, or egg?

Some real-world multi-class problems entail choosing from *millions* of separate classes. For example, consider a multi-class classification model that can identify the image of just about anything.

Estimated Time: 5 minutes

Learning Objectives

- Develop an understanding of multi-class classification problems, particularly Softmax.
- Develop Softmax solutions in TensorFlow.



Video not displaying?

Please see the community page (https://support.google.com/machinelearningeducation/thread/51878) for troubleshooting assistance.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Programming Exercise
(https://developers.google.com/machine-learning/crash-course/training-neural-networks/programming-exercise)

Next
One vs. All →
(https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/one-vs-all)

Last updated March 5, 2019.

Multi-Class Neural Networks: One vs. All

Estimated Time: 2 minutes

One vs. all provides a way to leverage binary classification. Given a classification problem with N possible solutions, a one-vs.-all solution consists of N separate binary classifiers—one binary classifier for each possible outcome. During training, the model runs through a sequence of binary classifiers, training each to answer a separate classification question. For example, given a picture of a dog, five different recognizers might be trained, four seeing the image as a negative example (not a dog) and one seeing the image as a positive example (a dog). That is:

1. Is this image an apple? No.
2. Is this image a bear? No.
3. Is this image candy? No.
4. Is this image a dog? Yes.
5. Is this image an egg? No.

This approach is fairly reasonable when the total number of classes is small, but becomes increasingly inefficient as the number of classes rises.

We can create a significantly more efficient one-vs.-all model with a deep neural network in which each output node represents a different class. The following figure suggests this approach:

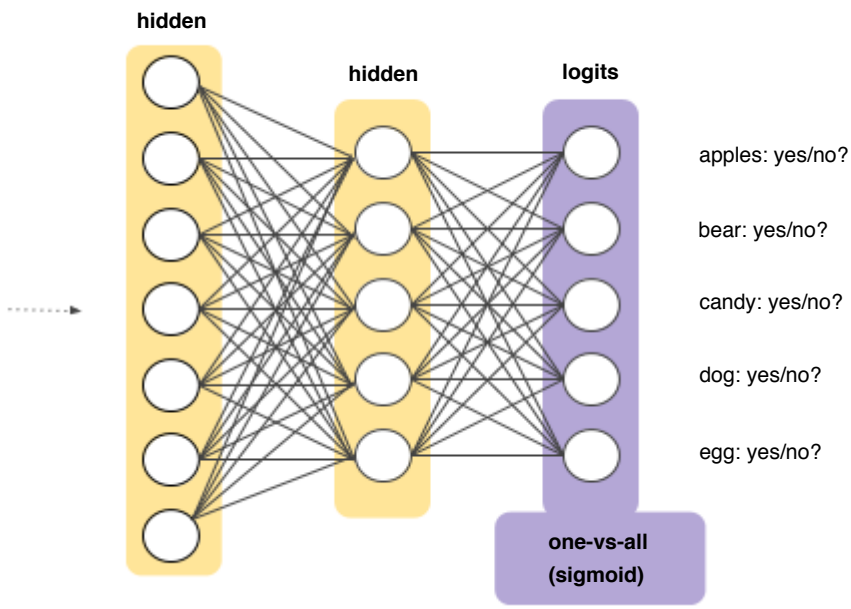


Figure 1. A one-vs.-all neural network.

Key Terms

- multi-class (https://developers.google.com/machine-learning/glossary#multi-class)
- one-vs.-all (https://developers.google.com/machine-learning/glossary#one-vs.-all)

Previous
← Video Lecture (https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/video-lecture)

Next
Softmax → (https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax)

Multi-Class Neural Networks: Softmax

Estimated Time: 5 minutes

Recall that logistic regression (https://developers.google.com/machine-learning/crash-course/logistic-regression) produces a decimal between 0 and 1.0. For example, a logistic regression output of 0.8 from an email classifier suggests an 80% chance of an email being spam and a 20% chance of it being not spam. Clearly, the sum of the probabilities of an email being either spam or not spam is 1.0.

Softmax extends this idea into a multi-class world. That is, Softmax assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would.

For example, returning to the image analysis we saw in Figure 1, Softmax might produce the following likelihoods of an image belonging to a particular class:

Class	Probability
apple	0.001
bear	0.04
candy	0.008
dog	0.95
egg	0.001

Softmax is implemented through a neural network layer just before the output layer. The Softmax layer must have the same number of nodes as the output layer.

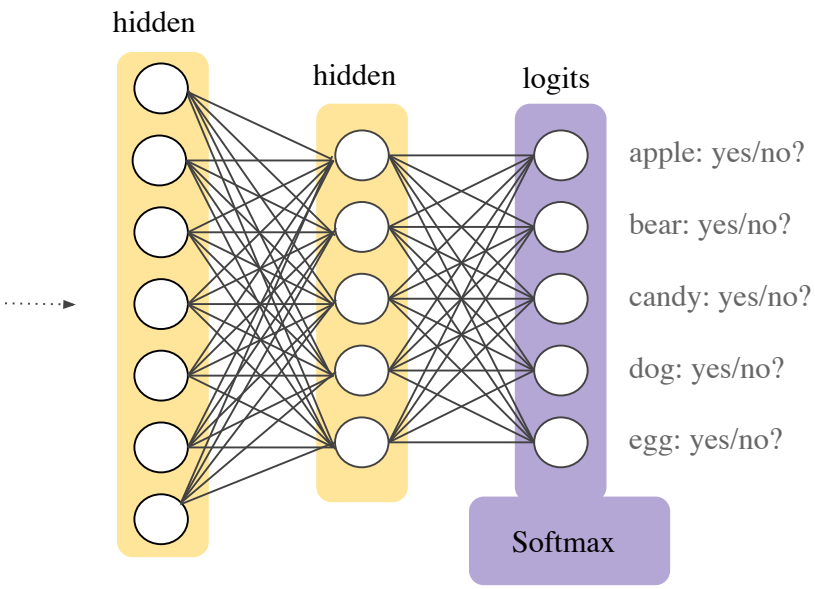


Figure 2. A Softmax layer within a neural network.

Click the dropdown arrow to see the Softmax equation.

The Softmax equation is as follows:

$$p(y = j|\mathbf{x}) = \frac{e^{(\mathbf{w}_j^T \mathbf{x} + b_j)}}{\sum_{k \in K} e^{(\mathbf{w}_k^T \mathbf{x} + b_k)}}$$

Note that this formula basically extends the formula for logistic regression into multiple classes.

Softmax Options

Consider the following variants of Softmax:

- **Full Softmax** is the Softmax we've been discussing; that is, Softmax calculates a probability for every possible class.
- **Candidate sampling** means that Softmax calculates a probability for all the positive labels but only for a random sample of negative labels. For example, if we are interested in determining whether an input image is a beagle or a bloodhound, we don't have to provide probabilities for every non-doggy example.

Full Softmax is fairly cheap when the number of classes is small but becomes prohibitively expensive when the number of classes climbs. Candidate sampling can improve efficiency in problems having a large number of classes.

One Label vs. Many Labels

Softmax assumes that each example is a member of exactly one class. Some examples, however, can simultaneously be a member of multiple classes. For such examples:

- You may not use Softmax.
- You must rely on multiple logistic regressions.

For example, suppose your examples are images containing exactly one item—a piece of fruit. Softmax can determine the likelihood of that one item being a pear, an orange, an apple, and so on. If your examples are images containing all sorts of things—bowls of different kinds of fruit—then you'll have to use multiple logistic regressions instead.

Key Terms

- candidate sampling (https://developers.google.com/machine-learning/glossary#candidate_sampling)
- multi-class (https://developers.google.com/machine-learning/glossary#multi-class)
- logistic regression (https://developers.google.com/machine-learning/glossary#logistic_regression)
- softmax (https://developers.google.com/machine-learning/glossary#softmax)

Previous
← One vs. All (https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/one-vs-all)

Next
Programming Exercise → (https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/programming-exercise)

Last updated March 5, 2019.

Embeddings

An **embedding** is a relatively low-dimensional space into which you can translate high-dimensional vectors. Embeddings make it easier to do machine learning on large inputs like sparse vectors representing words. Ideally, an embedding captures some of the semantics of the input by placing semantically similar inputs close together in the embedding space. An embedding can be learned and reused across models.

Estimated Time: 15 minutes

Learning Objectives

- Learn what an embedding is and what it's for.
- Learn how embeddings encode semantic relations.
- Learn how to use embeddings.
- Learn how to train meaningful embeddings (using word2vec, for example).

Video not displaying?

Please see the community page (https://support.google.com/machinelearningeducation/thread/51878) for troubleshooting assistance.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Programming Exercise
(https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/programming-exercise)

Next
Motivation from Collaborative Filtering →
(https://developers.google.com/machine-learning/crash-course/embeddings/motivation-from-collaborative-filtering)

Last updated March 5, 2019.

Embeddings: Motivation From Collaborative Filtering

Estimated Time: 10 minutes

Collaborative filtering is the task of making predictions about the interests of a user based on interests of many other users. As an example, let's look at the task of movie recommendation. Suppose we have 1,000,000 users, and a list of the movies each user has watched (from a catalog of 500,000 movies). Our goal is to recommend movies to users.

To solve this problem some method is needed to determine which movies are similar to each other. We can achieve this goal by embedding the movies into a low-dimensional space created such that similar movies are nearby.

Before describing how we can learn the embedding, we first explore the type of qualities we want the embedding to have, and how we will represent the training data for learning the embedding.

Arrange Movies on a One-Dimensional Number Line

To help develop intuition about embeddings, on a piece of paper, try to arrange the following movies on a one-dimensional number line so that the movies nearest each other are the most closely related:

Movie	Rating (https://wikipedia.org/wiki/Motion_Picture_Association_of_Ame
Bleu (http://www.imdb.com/title/tt0108394/)	R
The Dark Knight Rises (http://www.imdb.com/title/tt1345836)	PG-13

Movie	Rating (https://wikipedia.org/wiki/Motion_Picture_Association_of_Ame
Harry Potter and the Sorcerer's Stone (http://www.imdb.com/title/tt0241527/)	PG
The Incredibles (http://www.imdb.com/title/tt0317705/)	PG
Shrek (http://www.imdb.com/title/tt0126029/)	PG
Star Wars (http://www.imdb.com/title/tt0076759)	PG
The Triplets of Belleville (http://www.imdb.com/title/tt0286244)	PG-13
Memento (http://www.imdb.com/title/tt0209144/)	R

Click the dropdown arrow for one possible (highly imperfect) solution.

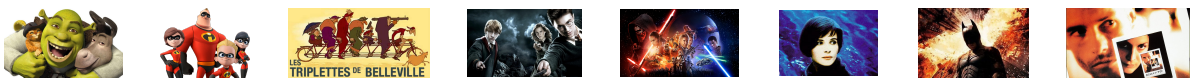


Figure 1. A possible one-dimensional arrangement

While this embedding does help capture how much the movie is geared towards children versus adults, there are many more aspects of a movie that one would

want to capture when making recommendations. Let's take this example one step further, adding a second embedding dimension.

Arrange Movies in a Two-Dimensional Space

Try the same exercise as before, but this time arrange the same movies in a two-dimensional space.

Click the dropdown arrow for another possible solution.

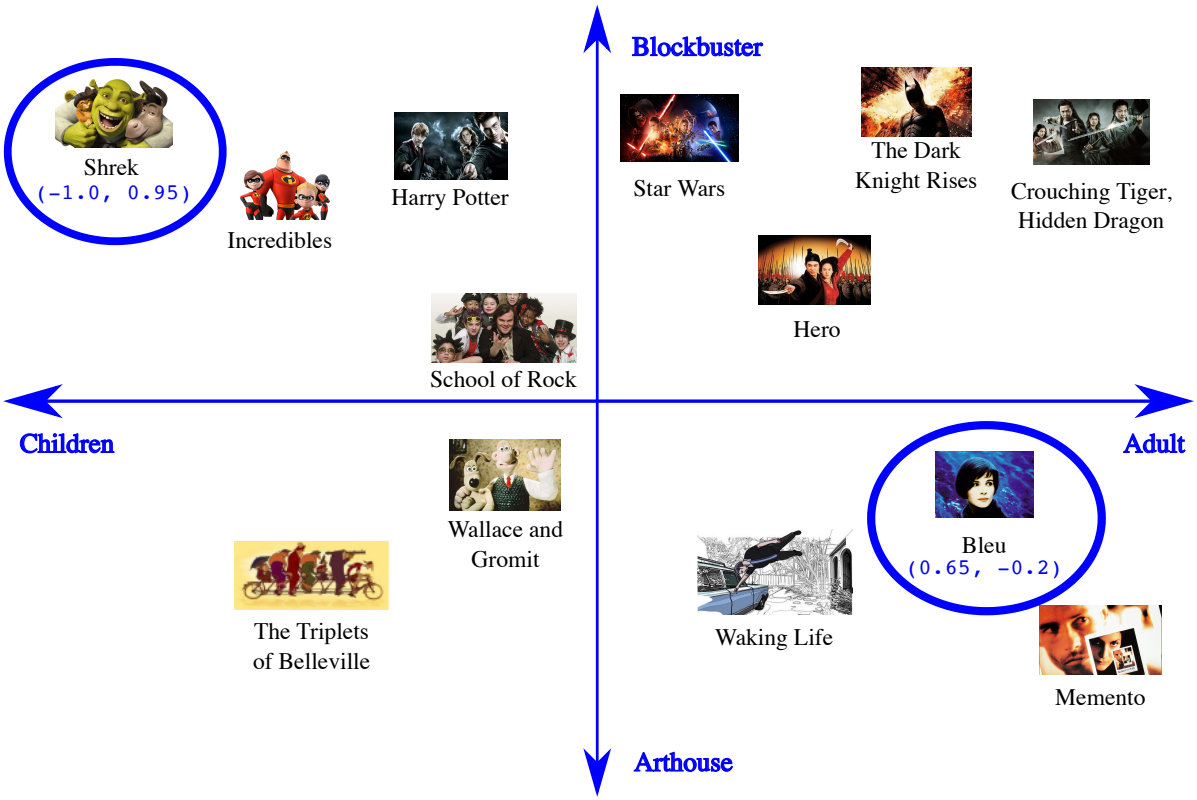


Figure 2. A possible two-dimensional arrangement

With this two-dimensional embedding we define a distance between movies such that movies are nearby (and thus inferred to be similar) if they are both alike in the extent to which they are geared towards children versus adults, as well as the extent to which they are blockbuster movies versus arthouse movies. These, of course, are just two of many characteristics of movies that might be important.

More generally, what we've done is mapped these movies into an **embedding space**, where each word is described by a two-dimensional set of coordinates. For

example, in this space, "Shrek" maps to (-1.0, 0.95) and "Bleu" maps to (0.65, -0.2). In general, when learning a *d*-dimensional embedding, each movie is represented by *d* real-valued numbers, each one giving the coordinate in one dimension.

In this example, we have given a name to each dimension. When learning embeddings, the individual dimensions are not learned with names. Sometimes, we can look at the embeddings and assign semantic meanings to the dimensions, and other times we cannot. Often, each such dimension is called a **latent dimension**, as it represents a feature that is not explicit in the data but rather inferred from it.

Ultimately, it is the distances between movies in the embedding space that are meaningful, rather than a single movie's values along any given dimension.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture)

Next
Categorical Input Data →
(https://developers.google.com/machine-learning/crash-course/embeddings/categorical-input-data)

Embeddings: Categorical Input Data

Estimated Time: 10 minutes

Categorical data refers to input features that represent one or more discrete items from a finite set of choices. For example, it can be the set of movies a user has watched, the set of words in a document, or the occupation of a person.

Categorical data is most efficiently represented via **sparse tensors**, which are tensors with very few non-zero elements. For example, if we're building a movie recommendation model, we can assign a unique ID to each possible movie, and then represent each user by a sparse tensor of the movies they have watched, as shown in Figure 3.

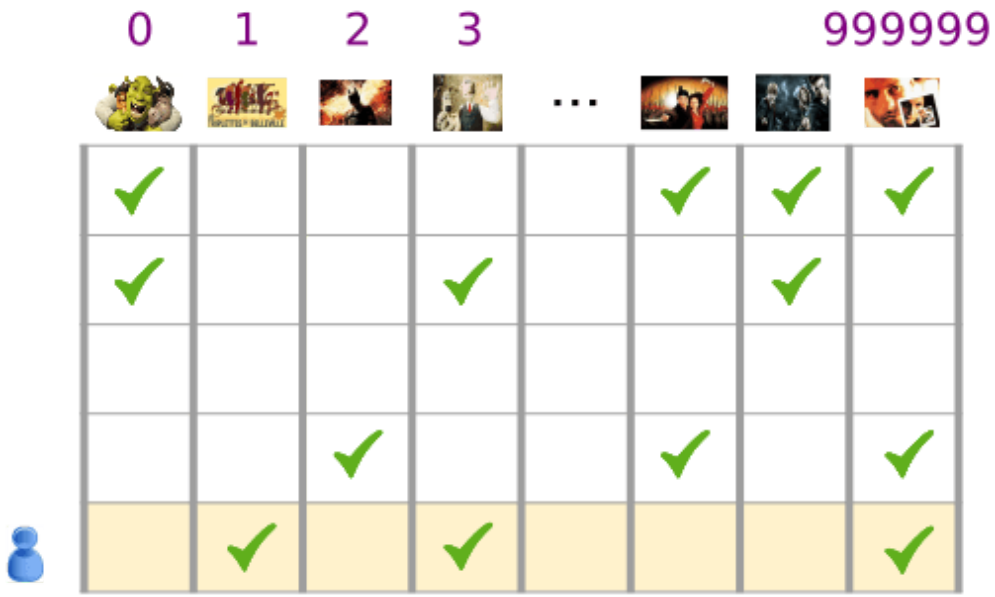


Figure 3. Data for our movie recommendation problem.

Each row of the matrix in Figure 3 is an example capturing a user's movie-viewing history, and is represented as a sparse tensor because each user only watches a small fraction of all possible movies. The last row corresponds to the sparse tensor [1, 3, 999999], using the vocabulary indices shown above the movie icons.

Likewise one can represent words, sentences, and documents as sparse vectors where each word in the vocabulary plays a role similar to the movies in our recommendation example.

In order to use such representations (https://developers.google.com/machine-learning/crash-course/representation/video-lecture) within a

machine learning system, we need a way to represent each sparse vector as a vector of numbers so that semantically similar items (movies or words) have similar distances in the vector space. But how do you represent a word as a vector of numbers?

The simplest way is to define a giant input layer with a node for every word in your vocabulary, or at least a node for every word that appears in your data. If 500,000 unique words appear in your data, you could represent a word with a length 500,000 vector and assign each word to a slot in the vector.

If you assign "horse" to index 1247, then to feed "horse" into your network you might copy a 1 into the 1247th input node and 0s into all the rest. This sort of representation is called a **one-hot encoding**, because only one index has a non-zero value.

More typically your vector might contain counts of the words in a larger chunk of text. This is known as a "bag of words" representation. In a bag-of-words vector, several of the 500,000 nodes would have non-zero value.

But however you determine the non-zero values, one-node-per-word gives you very *sparse* input vectors—very large vectors with relatively few non-zero values. Sparse representations have a couple of problems that can make it hard for a model to learn effectively.

Size of Network

Huge input vectors mean a super-huge number of weights for a neural network. If there are M words in your vocabulary and N nodes in the first layer of the network above the input, you have MxN weights to train for that layer. A large number of weights causes further problems:

- **Amount of data**. The more weights in your model, the more data you need to train effectively.
- **Amount of computation**. The more weights, the more computation required to train and use the model. It's easy to exceed the capabilities of your hardware.

Lack of Meaningful Relations Between Vectors

If you feed the pixel values of RGB channels into an image classifier, it makes sense to talk about "close" values. Reddish blue is close to pure blue, both semantically and in terms of the geometric distance between vectors. But a vector with a 1 at index 1247 for "horse" is

not any closer to a vector with a 1 at index 50,430 for "antelope" than it is to a vector with a 1 at index 238 for "television".

The Solution: Embeddings

The solution to these problems is to use **embeddings**, which translate large sparse vectors into a lower-dimensional space that preserves semantic relationships. We'll explore embeddings intuitively, conceptually, and programmatically in the following sections of this module.

Key Terms

- input layer (https://developers.google.com/machine-learning/glossary#input_layer)
- one-hot encoding (https://developers.google.com/machine-learning/glossary#one-hot_encoding)
- sparse features (https://developers.google.com/machine-learning/glossary#sparse_features)

Previous
← Motivation from Collaborative Filtering (https://developers.google.com/machine-learning/crash-course/embeddings/motivation-from-collaborative-filtering)

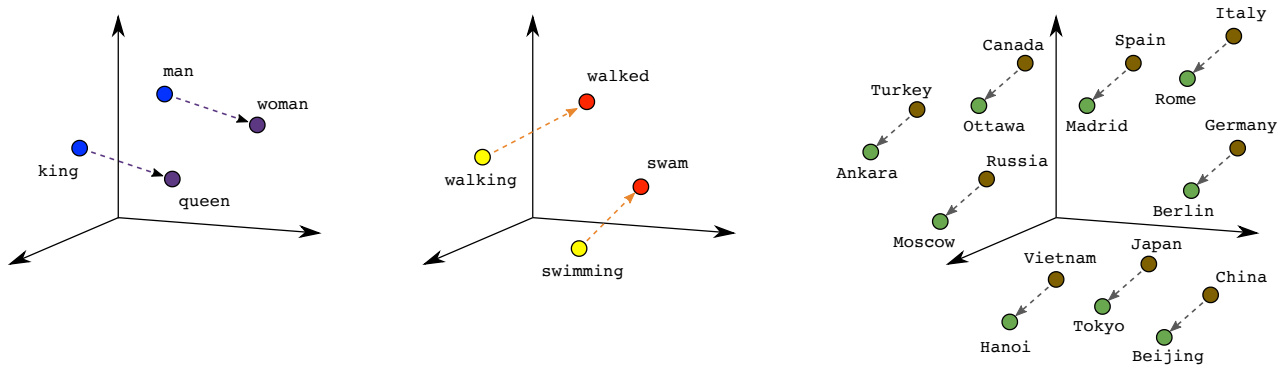
Next
Translating to a Lower-Dimensional Space → (https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space)

Embeddings: Translating to a Lower-Dimensional Space

Estimated Time: 5 minutes

You can solve the core problems of sparse input data by mapping your high-dimensional data into a lower-dimensional space.

As you can see from the paper exercises, even a small multi-dimensional space provides the freedom to group semantically similar items together and keep dissimilar items far apart. Position (distance and direction) in the vector space can encode semantics in a good embedding. For example, the following visualizations of real embeddings show geometrical relationships that capture semantic relations like the relation between a country and its capital:



Male-Female Verb Tense Country-Capital

Figure 4. Embeddings can produce remarkable analogies.

This sort of meaningful space gives your machine learning system opportunities to detect patterns that may help with the learning task.

Shrinking the network

While we want enough dimensions to encode rich semantic relations, we also want an embedding space that is small enough to allow us to train our system more quickly. A useful embedding may be on the order of hundreds of dimensions. This is likely several orders of magnitude smaller than the size of your vocabulary for a natural language task.

Embeddings as lookup tables

An embedding is a matrix in which each column is the vector that corresponds to an item in your vocabulary. To get the dense vector for a single vocabulary item, you retrieve the column corresponding to that item.

But how would you translate a sparse bag of words vector? To get the dense vector for a sparse vector representing multiple vocabulary items (all the words in a sentence or paragraph, for example), you could retrieve the embedding for each individual item and then add them together.

If the sparse vector contains counts of the vocabulary items, you could multiply each embedding by the count of its corresponding item before adding it to the sum.

These operations may look familiar.

Embedding lookup as matrix multiplication

The lookup, multiplication, and addition procedure we've just described is equivalent to matrix multiplication. Given a 1 X N sparse representation S and an N X M embedding table E, the matrix multiplication S X E gives you the 1 X M dense vector.

But how do you get E in the first place? We'll take a look at how to obtain embeddings in the next section.

Key Terms

- embeddings (https://developers.google.com/machine-learning/glossary#embeddings)

Previous
← Categorical Input Data
(https://developers.google.com/machine-learning/crash-course/embeddings/categorical-input-data)

Next

Obtaining Embeddings →
(https://developers.google.com/machine-learning/crash-course/embeddings/obtaining-embeddings)

Last updated March 5, 2019.

Embeddings: Obtaining Embeddings

Estimated Time: 8 minutes

There are a number of ways to get an embedding, including a state-of-the-art algorithm created at Google.

Standard Dimensionality Reduction Techniques

There are many existing mathematical techniques for capturing the important structure of a high-dimensional space in a low dimensional space. In theory, any of these techniques could be used to create an embedding for a machine learning system.

For example, principal component analysis (https://wikipedia.org/wiki/Principal_component_analysis) (PCA) has been used to create word embeddings. Given a set of instances like bag of words vectors, PCA tries to find highly correlated dimensions that can be collapsed into a single dimension.

Word2vec

Word2vec is an algorithm invented at Google for training word embeddings. Word2vec relies on the **distributional hypothesis** to map semantically similar words to geometrically close embedding vectors.

The distributional hypothesis states that words which often have the same neighboring words tend to be semantically similar. Both "dog" and "cat" frequently appear close to the word "vet", and this fact reflects their semantic similarity. As the linguist John Firth put it in 1957, "You shall know a word by the company it keeps".

Word2Vec exploits contextual information like this by training a neural net to distinguish actually co-occurring groups of words from randomly grouped words. The input layer takes a sparse representation of a target word together with one or more context words. This input connects to a single, smaller hidden layer.

In one version of the algorithm, the system makes a negative example by substituting a random noise word for the target word. Given the positive example "the plane flies", the system might swap in "jogging" to create the contrasting negative example "the jogging flies".

The other version of the algorithm creates negative examples by pairing the true target word with randomly chosen context words. So it might take the positive examples (the, plane), (flies, plane) and the negative examples (compiled, plane), (who, plane) and learn to identify which pairs actually appeared together in text.

The classifier is not the real goal for either version of the system, however. After the model has been trained, you have an embedding. You can use the weights connecting the input layer with the hidden layer to map sparse representations of words to smaller vectors. This embedding can be reused in other classifiers.

For more information about word2vec, see the tutorial on tensorflow.org (https://www.tensorflow.org/tutorials/word2vec/index.html)

Training an Embedding as Part of a Larger Model

You can also learn an embedding as part of the neural network for your target task. This approach gets you an embedding well customized for your particular system, but may take longer than training the embedding separately.

In general, when you have sparse data (or dense data that you'd like to embed), you can create an embedding unit that is just a special type of hidden unit of size *d*. This embedding layer can be combined with any other features and hidden layers. As in any DNN, the final layer will be the loss that is being optimized. For example, let's say we're performing collaborative filtering, where the goal is to predict a user's interests from the interests of other users. We can model this as a supervised learning problem by randomly setting aside (or holding out) a small number of the movies that the user has watched as the positive labels, and then optimize a softmax loss.

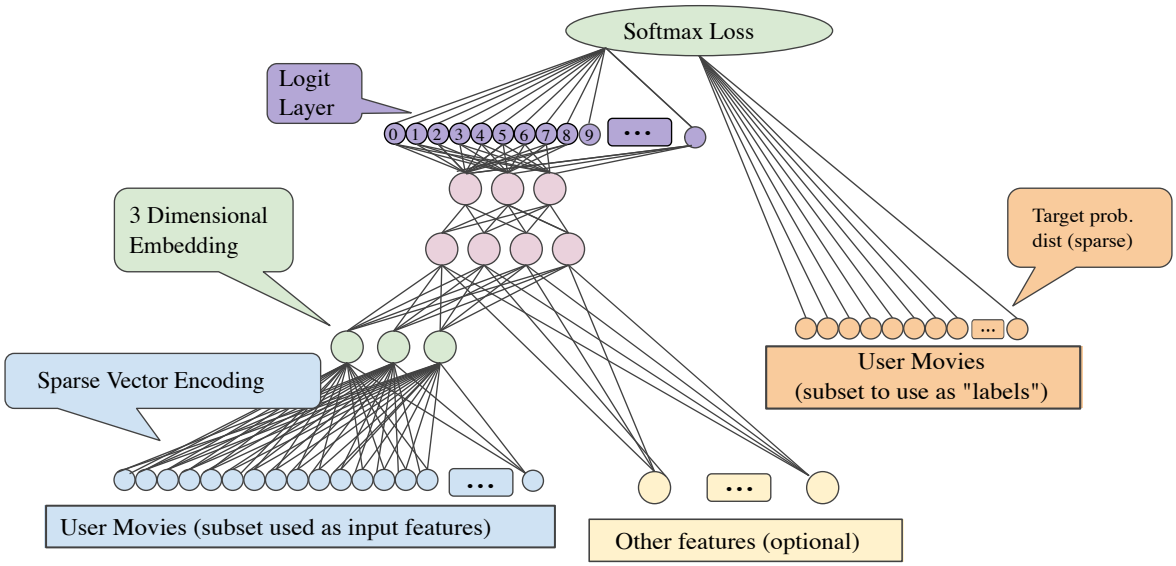


Figure 5. A sample DNN architecture for learning movie embeddings from collaborative filtering data.

As another example if you want to create an embedding layer for the words in a real-estate ad as part of a DNN to predict housing prices then you'd optimize an L_2 Loss using the known sale price of homes in your training data as the label.

When learning a *d*-dimensional embedding each item is mapped to a point in a *d*-dimensional space so that the similar items are nearby in this space. Figure 6 helps to illustrate the relationship between the weights learned in the embedding layer and the geometric view. The edge weights between an input node and the nodes in the *d*-dimensional embedding layer correspond to the coordinate values for each of the *d* axes.

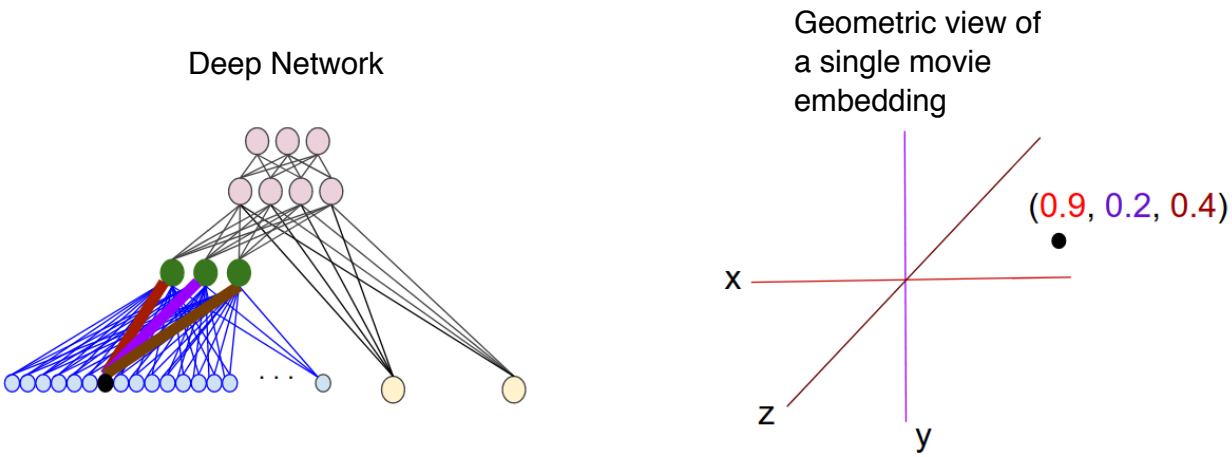


Figure 6. A geometric view of the embedding layer weights.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Translating to a Lower-Dimensional Space
(https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space)

Next
Programming Exercise →
(https://developers.google.com/machine-learning/crash-course/embeddings/programming-exercise)

Last updated March 5, 2019.

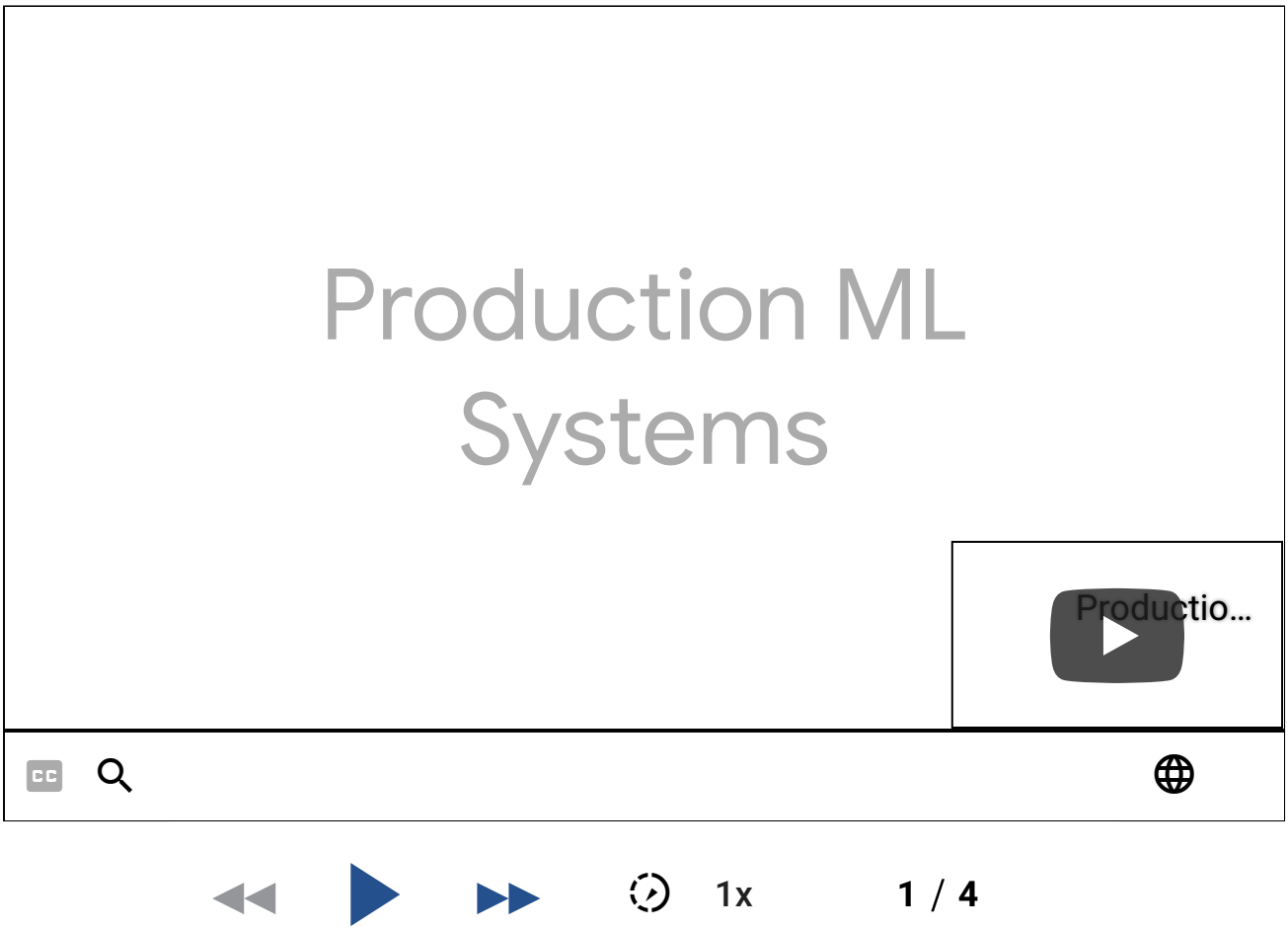
Production ML Systems

There's a lot more to machine learning than just implementing an ML algorithm. A production ML system involves a significant number of components.

Estimated Time: 3 minutes

Learning Objectives

- Understand the breadth of components in a production ML system.



Video not displaying?

Please see the community page (https://support.google.com/machinelearningeducation/thread/51878) for troubleshooting assistance.

Video Lecture Summary

So far, Machine Learning Crash Course has focused on building ML models. However, as the following figure suggests, real-world production ML systems are large ecosystems of which the model is just a single part.

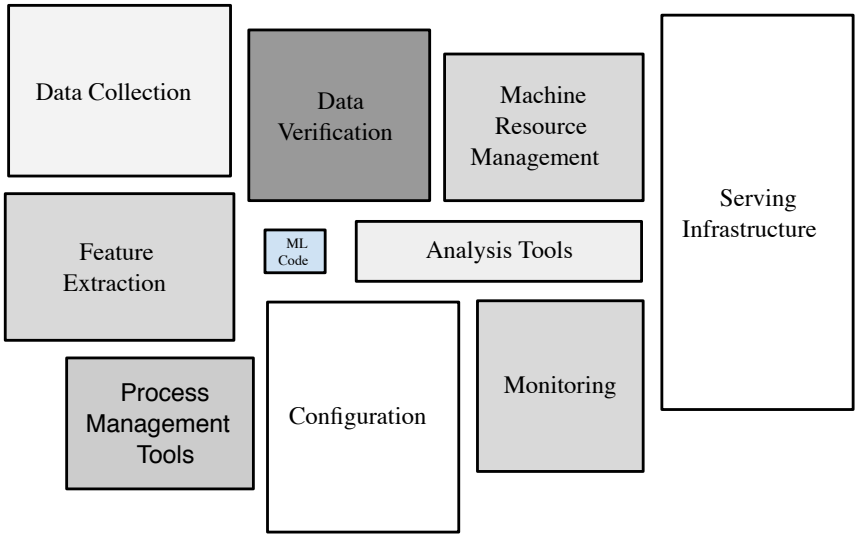


Figure 1. Real-world production ML system.

The ML code is at the heart of a real-world ML production system, but that box often represents only 5% or less of the overall code of that total ML production system. (That's not a misprint.) Notice that a ML production system devotes considerable resources to input data—collecting it, verifying it, and extracting features from it. Furthermore, notice that a serving infrastructure must be in place to put the ML model's predictions into practical use in the real world.

Fortunately, many of the components in the preceding figure are reusable. Furthermore, you don't have to build all the components in Figure 1 yourself.

TensorFlow provides many of these components, but other options are available from other platforms such as Spark or Hadoop.

Subsequent modules will help guide your design decisions in building a production ML system.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous

← Programming Exercise (https://developers.google.com/machine-learning/crash-course/embeddings/programming-exercise)

Next
Video Lecture → (https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-training/video-lecture)

Static vs. Dynamic Training

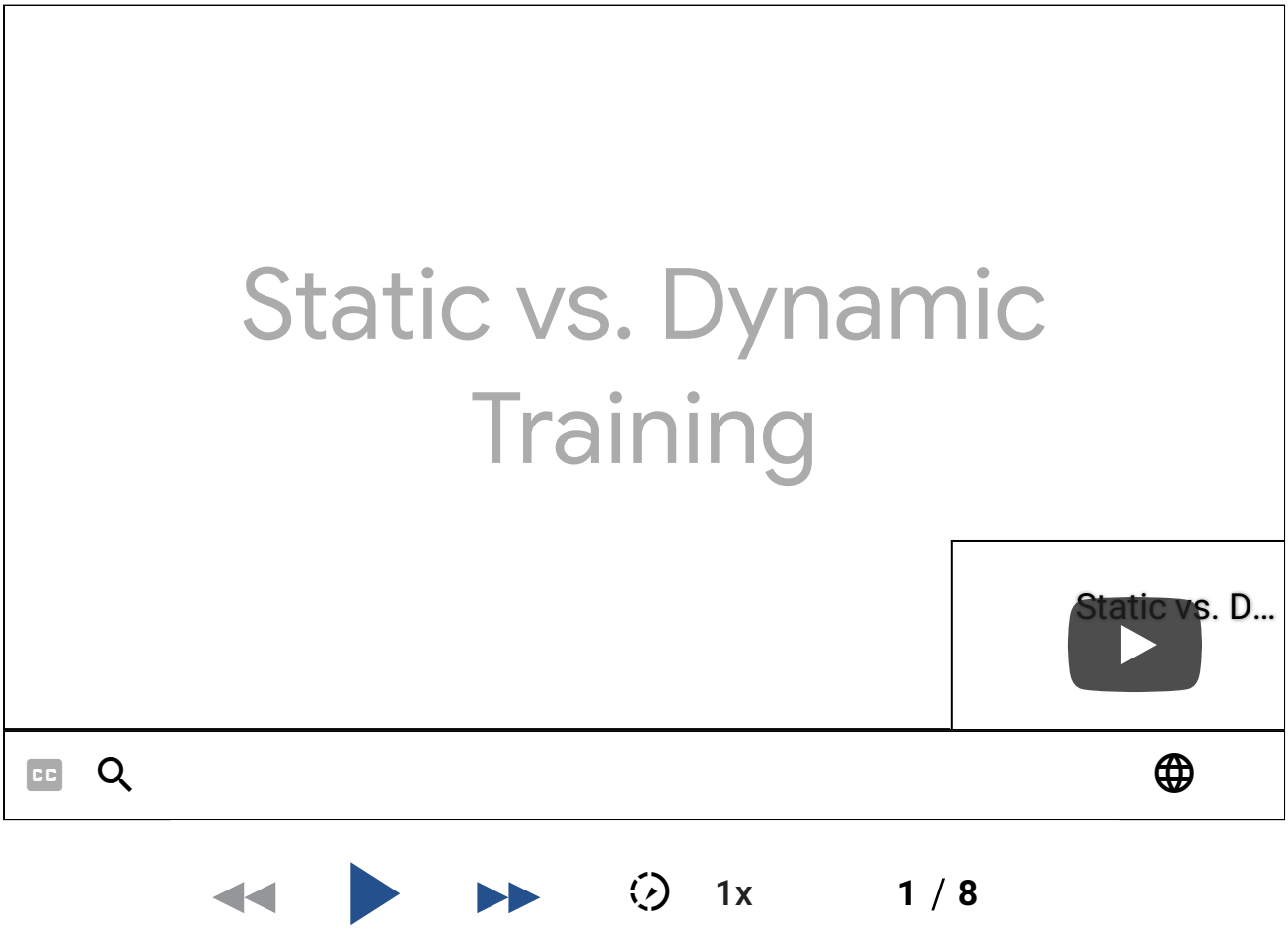
Broadly speaking, there are two ways to train a model:

- A **static model** is trained offline. That is, we train the model exactly once and then use that trained model for a while.
- A **dynamic model** is trained online. That is, data is continually entering the system and we're incorporating that data into the model through continuous updates.

Estimated Time: 3 minutes

Learning Objective

- Identify the pros and cons of static and dynamic training.



Video not displaying?

Please see the community page (https://support.google.com/machinelearningeducation/thread/51878) for troubleshooting assistance.

Video Lecture Summary

Broadly speaking, the following points dominate the static vs. dynamic training decision:

- Static models are easier to build and test.
- Dynamic models adapt to changing data. The world is a highly changeable place. Sales predictions built from last year's data are unlikely to successfully predict next year's results.

If your data set truly isn't changing over time, choose static training because it is cheaper to create and maintain than dynamic training. However, many information sources really do change over time, even those with features that you think are as constant as, say, sea level. The moral: even with static training, you must still monitor your input data for change.

For example, consider a model trained to predict the probability that users will buy flowers. Because of time pressure, the model is trained only once using a dataset of flower buying behavior during July and August. The model is then shipped off to serve predictions in production, *but is never updated.* The model works fine for several months, but then makes terrible predictions around Valentine's Day (https://wikipedia.org/wiki/Valentine's_Day) because user behavior during that holiday period changes dramatically.

Key Terms

- dynamic model (https://developers.google.com/machine-learning/glossary#dynamic_model)
- static model (https://developers.google.com/machine-learning/glossary#static_model)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Production ML Systems (3 min)
(https://developers.google.com/machine-learning/crash-course/production-ml-systems)

Next

Check Your Understanding →

(https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-training/check-your-understanding)

Last updated March 5, 2019.

Static vs. Dynamic Training: Check Your Understanding

Estimated Time: 4 minutes

Dynamic (Online) Training

Explore the options below.

Which one of the following statements is true of dynamic (online) training?

🚫 Very little monitoring of input data needs to be done at inference time.

Just like a static, offline model, it is also important to monitor the inputs to the dynamically updated models. We are likely not at risk for large seasonality effects, but sudden, large changes to inputs (such as an upstream data source going down) can still cause unreliable predictions.

Try again.

✓ The model stays up to date as new data arrives.

This is the primary benefit of online training—we can avoid many staleness issues by allowing the model to train on new data as it comes in.

Correct answer.

🚫 Very little monitoring of training jobs needs to be done.

Actually, you must continuously monitor training jobs to ensure that they are healthy and working as intended. You'll also need supporting infrastructure like the ability to roll a model back to a previous snapshot in case something goes wrong in training, such as a buggy job or corruption in input data.

Try again.

Static (Offline) Training

Explore the options below.

Which of the following statements are true about static (offline) training?

✓ Offline training requires less monitoring of training jobs than online training.

In general, monitoring requirements at training time are more modest for offline training, which insulates us from many production considerations. However, the more frequently you train your model, the higher the investment you'll need to make in monitoring. You'll also want to validate regularly to ensure that changes to your code (and its dependencies) don't adversely affect model quality.

1 of 2 correct answers.

🚫 The model stays up to date as new data arrives.

Actually, if we train offline, then the model has no way to incorporate new data as it arrives. This can lead to model staleness, if the distribution we are trying to learn from changes over time.

Try again.

✓ You can verify the model before applying it in production.

Yes, offline training gives ample opportunity to verify model performance before introducing the model in production.

2 of 2 correct answers.

🚫 Very little monitoring of input data needs to be done at inference time.

Counterintuitively, you do need to monitor input data at serving time. If the input distributions change, then our model's predictions may become unreliable. Imagine, for example, a model trained only on summertime clothing data suddenly being used to predict clothing buying behavior in wintertime.

Try again.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-training/video-lecture)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-inference/video-lecture)

Last updated March 5, 2019.

Static vs. Dynamic Inference

You can choose either of the following inference strategies:

- **offline inference**, meaning that you make all possible predictions in a batch, using a MapReduce or something similar. You then write the predictions to an SSTable or Bigtable, and then feed these to a cache/lookup table.
- **online inference**, meaning that you predict on demand, using a server.

Estimated Time: 3 minutes

Learning Objectives

- Understand the pros and cons of static and dynamic inference.
- Estimate training and serving needs for real-world scenarios.

Learn more about static vs. dynamic inference in the following video (2 min).

Video not displaying?

Please see the community page (https://support.google.com/machinelearningeducation/thread/51878) for troubleshooting assistance.

Video Lecture Summary

Here are the pros and cons of offline inference:

- Pro: Don't need to worry much about cost of inference.
- Pro: Can likely use batch quota or some giant MapReduce.
- Pro: Can do post-verification of predictions before pushing.
- Con: Can only predict things we know about — bad for long tail.
- Con: Update latency is likely measured in hours or days.

Here are the pros and cons of online inference:

- Pro: Can make a prediction on any new item as it comes in — great for long tail.
- Con: Compute intensive, latency sensitive—may limit model complexity.
- Con: Monitoring needs are more intensive.

Key Terms

- offline inference (https://developers.google.com/machine-learning/glossary#offline_inference)
- online inference (https://developers.google.com/machine-learning/glossary#online_inference)

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Check Your Understanding (https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-training/check-your-understanding)

Next

Check Your Understanding →
(https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-inference/check-your-understanding)

Last updated March 5, 2019.

Static vs. Dynamic Inference: Check Your Understanding

Estimated Time: 4 minutes

Static (Offline) Inference

Explore the options below.

In offline inference, we make predictions on a big batch of data all at once. We then put those predictions in a look-up table for later use. Which of the following are true of offline inference?

🚫 We will be able to react quickly to changes in the world.

No, this is a drawback of offline inference. We'll need to wait until a new set of predictions have been written to the look-up table before we can respond differently based on any changes in the world.

Try again.

🚫 We will need to carefully monitor our input signals over a long period of time.

This is the one case where we don't actually need to monitor input signals over a long period of time. This is because once the predictions have been written to a look-up table, we're no longer dependent on the input features. Note that any subsequent update of the model will require a new round of input verification.

Try again.

✓ For a given input, we can serve a prediction more quickly than with *online* inference.

One of the great things about offline inference is that once the predictions have been written to some look-up table, they can be served with minimal latency. No feature computation or model inference needs to be done at request time.

1 of 2 correct answers.

✓ After generating the predictions, we can verify them before applying them.

This is indeed one useful thing about offline inference. We can sanity check and verify all of our predictions before they are used.

2 of 2 correct answers.

🚫 We will have predictions for all possible inputs.

No, we will not have predictions for all possible inputs. This is one of the drawbacks of offline inference. We will only be able to serve a prediction for those examples that we already know about. This is fine if the set of things that we're predicting is limited, like world cities. But for things like user queries that have a long tail of unusual or rare items, we may not be able to provide full coverage with an offline-inference system.

Try again.

Dynamic (Online) Inference

Explore the options below.

Dynamic (online) inference means making predictions on demand. That is, in online inference, we put the trained model on a server and issue inference requests as needed. Which of the following are true of dynamic inference?

🚫 You can do post-verification of predictions before they are used.

In general, it's not possible to do a post-verification of all predictions before they get used because predictions are being made on demand. You can, however, potentially monitor *aggregate*

prediction qualities to provide some level of sanity checking, but these will signal fire alarms only after the fire has already spread.

Try again.

🚫 When performing online inference, you do not need to worry about prediction latency (the lag time for returning predictions) as much as when performing offline inference.

Prediction latency is often a real concern in online inference. Unfortunately, you can't necessarily fix prediction latency issues by adding more inference servers.

Try again.

✓ You can provide predictions for all possible items.

Yes, this is a strength of online inference. Any request that comes in will be given a score. Online inference handles long-tail distributions (those with many rare items), like the space of all possible sentences written in movie reviews.

1 of 2 correct answers.

✓ You must carefully monitor input signals.

Yes. Signals could change suddenly due to upstream issues, harming our predictions.

2 of 2 correct answers.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-inference/video-lecture)

Next
Video Lecture →
(https://developers.google.com/machine-learning/crash-course/data-dependencies/video-lecture)

Last updated March 5, 2019.

Data Dependencies

Data is as important to ML developers as code is to traditional programmers. This lesson focuses on the kinds of questions you should be asking of your data.

Estimated Time: 10 minutes

Learning Objectives

- Understand data dependencies in production ML systems.

Video not displaying?

Please see the community page (https://support.google.com/machinelearningeducation/thread/51878) for troubleshooting assistance.

Video Lecture Summary

The behavior of an ML system is dependent on the behavior and qualities of its input features (https://developers.google.com/machine-learning/crash-course/representation). As the input data for those features changes, so too will your model. Sometimes that change is desirable, but sometimes it is not.

In traditional software development, you focus more on code than on data. In machine learning development, although coding is still part of the job, your focus must widen to include data. For example, on traditional software development projects, it is a best practice to write unit tests to validate your code. On ML projects, you must also continuously test, verify, and monitor your input data.

For example, you should continuously monitor your model to remove unused (or little used) features. Imagine a certain feature that has been contributing little or nothing to the model. If the input data for that feature abruptly changes, your model's behavior might also abruptly change in undesirable ways.

Reliability

Some questions to ask about the reliability of your input data:

- Is the signal always going to be available or is it coming from an unreliable source? For example:
- Is the signal coming from a server that crashes under heavy load?
- Is the signal coming from humans that go on vacation every August?

Versioning

Some questions to ask about versioning:

- Does the system that computes this data ever change? If so:
- How often?
- How will you know when that system changes?

Sometimes, data comes from an upstream process. If that process changes abruptly, your model can suffer.

Consider creating your own copy of the data you receive from the upstream process. Then, only advance to the next version of the upstream data when you are certain that it is safe to

do so.

Necessity

The following question might remind you of regularization (https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/video-lecture):

- Does the usefulness of the feature justify the cost of including it?

It is always tempting to add more features to the model. For example, suppose you find a new feature whose addition makes your model slightly more accurate. More accuracy certainly *sounds* better than less accuracy. However, now you've just added to your maintenance burden. That additional feature could degrade unexpectedly, so you've got to monitor it. Think carefully before adding features that lead to minor short-term wins.

Correlations

Some features correlate (positively or negatively) with other features. Ask yourself the following question:

- Are any features so tied together that you need additional strategies to tease them apart?

Feedback Loops

Sometimes a model can affect its own training data. For example, the results from some models, in turn, are directly or indirectly input features to that same model.

Sometimes a model can affect another model. For example, consider two models for predicting stock prices:

- Model A, which is a bad predictive model.
- Model B.

Since Model A is buggy, it mistakenly decides to buy stock in Stock X. Those purchases drive up the price of Stock X. Model B uses the price of Stock X as an input feature, so Model B can easily come to some false conclusions about the value of Stock X stock. Model B could, therefore, buy or sell shares of Stock X based on the buggy behavior of

Model A. Model B's behavior, in turn, can affect Model A, possibly triggering a tulip mania (https://wikipedia.org/wiki/Tulip_mania) or a slide in Company X's stock

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Check Your Understanding (https://developers.google.com/machine-learning/crash-course/static-vs-dynamic-inference/check-your-understanding)

Next
Check Your Understanding → (https://developers.google.com/machine-learning/crash-course/data-dependencies/check-your-understanding)

Data Dependencies: Check Your Understanding

Estimated Time: 4 minutes

Explore the options below.

Which of the following models are susceptible to a feedback loop?

🚫 A housing-value model that predicts house prices, using size (area in square meters), number of bedrooms, and geographic location as features.

It is not possible to quickly change a house's location, size, or number of bedrooms in response to price forecasts, making a feedback loop unlikely. However, there is potentially a correlation between size and number of bedrooms (larger homes are likely to have more rooms) that may need to be teased apart.

Try again.

🚫 An election-results model that forecasts the winner of a mayoral race by surveying 2% of voters after the polls have closed.

If the model does not publish its forecast until after the polls have closed, it is not possible for its predictions to affect voter behavior.

Try again.

🚫 A face-attributes model that detects whether a person is smiling in a photo, which is regularly trained on a database of stock photography that is automatically updated monthly.

There is no feedback loop here, as model predictions don't have any impact on our photo database. However, versioning of our input data is a concern here, as these monthly updates could potentially have unforeseen effects on the model.

Try again.

✓ A book-recommendation model that suggests novels its users may like based on their popularity (i.e., the number of times the books have been purchased).

Book recommendations are likely to drive purchases, and these additional sales will be fed back into the model as input, making it more likely to recommend these same books in the future.

1 of 3 correct answers.

✓ A university-ranking model that rates schools in part by their selectivity—the percentage of students who applied that were admitted.

The model's rankings may drive additional interest to top-rated schools, increasing the number of applications they receive. If these schools continue to admit the same number of students, selectivity will increase (the percentage of students admitted will go down). This will boost these schools' rankings, which will further increase prospective student interest, and so on…

2 of 3 correct answers.

✓ A traffic-forecasting model that predicts congestion at highway exits near the beach, using beach crowd size as one of its features.

Some beachgoers are likely to base their plans on the traffic forecast. If there is a large beach crowd and traffic is forecast to be heavy, many people may make alternative plans. This may depress beach turnout, resulting in a lighter traffic forecast, which then may increase attendance, and the cycle repeats.

3 of 3 correct answers.

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/data-dependencies/video-lecture)

Next

Video Lecture →
(https://developers.google.com/machine-learning/crash-course/fairness/video-lecture)

Last updated March 5, 2019.

Fairness: Types of Bias

Estimated Time: 5 minutes

Machine learning models are not inherently objective. Engineers train models by feeding them a data set of training examples, and human involvement in the provision and curation of this data can make a model's predictions susceptible to bias.

When building models, it's important to be aware of common human biases that can manifest in your data, so you can take proactive steps to mitigate their effects.

WARNING: The following inventory of biases provides just a small selection of biases that are often uncovered in machine learning data sets; this list *is not intended to be exhaustive*. Wikipedia's catalog of cognitive biases (https://wikipedia.org/wiki/List_of_cognitive_biases) enumerates over 100 different types of human bias that can affect our judgment. When auditing your data, you should be on the lookout for any and all potential sources of bias that might skew your model's predictions.

Reporting Bias

Reporting bias occurs when the frequency of events, properties, and/or outcomes captured in a data set does not accurately reflect their real-world frequency. This bias can arise because people tend to focus on documenting circumstances that are unusual or especially memorable, assuming that the ordinary can "go without saying."

EXAMPLE: A sentiment-analysis model is trained to predict whether book reviews are positive or negative based on a corpus of user submissions to a popular website. The majority of reviews in the training data set reflect extreme opinions (reviewers who either loved or hated a book), because people were less likely to submit a review of a book if they did not respond to it strongly. As a result, the model is less able to correctly predict sentiment of reviews that use more subtle language to describe a book.

Automation Bias

Automation bias is a tendency to favor results generated by automated systems over those generated by non-automated systems, irrespective of the error rates of each.

EXAMPLE: Software engineers working for a sprocket manufacturer were eager to deploy the new "groundbreaking" model they trained to identify tooth defects, until the factory supervisor pointed out that the model's precision and recall rates were both 15% lower than those of human inspectors.

Selection Bias

Selection bias occurs if a data set's examples are chosen in a way that is not reflective of their real-world distribution. Selection bias can take many different forms:

- **Coverage bias**: Data is not selected in a representative fashion.

EXAMPLE: A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product. Consumers who instead opted to buy a competing product were not surveyed, and as a result, this group of people was not represented in the training data.

- **Non-response bias** (or **participation bias**): Data ends up being unrepresentative due to participation gaps in the data-collection process.

EXAMPLE: A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product and with a sample of consumers who bought a competing product. Consumers who bought the competing product were 80% more likely to refuse to complete the survey, and their data was underrepresented in the sample.

- **Sampling bias**: Proper randomization is not used during data collection.

EXAMPLE: A model is trained to predict future sales of a new product based on phone surveys conducted with a sample of consumers who bought the product and with a sample of consumers who bought a competing product. Instead of randomly targeting consumers, the surveyer chose the first 200 consumers that responded to an email, who might have been more enthusiastic about the product than average purchasers.

Group Attribution Bias

Group attribution bias is a tendency to generalize what is true of individuals to an entire group to which they belong. Two key manifestations of this bias are:

- **In-group bias**: A preference for members of a group to which *you also belong*, or for characteristics that you also share.

EXAMPLE: Two engineers training a résumé-screening model for software developers are predisposed to believe that applicants who attended the same computer-science academy as they both did are more qualified for the role.

- **Out-group homogeneity bias**: A tendency to stereotype individual members of a group to which *you do not belong*, or to see their characteristics as more uniform.

EXAMPLE: Two engineers training a résumé-screening model for software developers are predisposed to believe that all applicants who did not attend a computer-science academy do not have sufficient expertise for the role.

Implicit Bias

Implicit bias occurs when assumptions are made based on one's own mental models and personal experiences that do not necessarily apply more generally.

EXAMPLE: An engineer training a gesture-recognition model uses a head shake (https://wikipedia.org/wiki/Head_shake) as a feature to indicate a person is communicating the word "no." However, in some regions of the world, a head shake actually signifies "yes."

A common form of implicit bias is **confirmation bias**, where model builders unconsciously process data in ways that affirm preexisting beliefs and hypotheses. In some cases, a model builder may actually keep training a model until it produces a result that aligns with their original hypothesis; this is called **experimenter's bias**.

EXAMPLE: An engineer is building a model that predicts aggressiveness in dogs based on a variety of features (height, weight, breed, environment). The engineer had an unpleasant encounter with a hyperactive toy poodle as a child, and ever since has associated the breed with aggression. When the trained model predicted most toy poodles to be relatively docile, the engineer retrained the model several more times until it produced a result showing smaller poodles to be more violent.

Key Terms

- automation bias
- confirmation bias

(https://developers.google.com/machine-learning/glossary#automation_bias)

- coverage bias (https://developers.google.com/machine-learning/glossary#selection_bias)
- in-group bias (https://developers.google.com/machine-learning/glossary#in-group_bias)
- implicit bias (https://developers.google.com/machine-learning/glossary#implicit_bias)
- out-group homogeneity bias (https://developers.google.com/machine-learning/glossary#out-group_homogeneity_bias)
- sampling bias (https://developers.google.com/machine-learning/glossary#selection_bias)

(https://developers.google.com/machine-learning/glossary#confirmation_bias)

- experimenter's bias (https://developers.google.com/machine-learning/glossary#confirmation_bias)
- group attribution bias (https://developers.google.com/machine-learning/glossary#group_attribution_bias)
- non-response bias (https://developers.google.com/machine-learning/glossary#selection_bias)
- reporting bias (https://developers.google.com/machine-learning/glossary#reporting_bias)
- selection bias (https://developers.google.com/machine-learning/glossary#selection_bias)

Previous
← Video Lecture
(https://developers.google.com/machine-learning/crash-course/fairness/video-lecture)

Next
Identifying Bias →
(https://developers.google.com/machine-learning/crash-course/fairness/identifying-bias)

Last updated March 26, 2019.

Fairness: Identifying Bias

Estimated Time: 10 minutes

As you explore your data to determine how best to represent (https://developers.google.com/machine-learning/crash-course/representation/) it in your model, it's important to also keep issues of fairness in mind and proactively audit for potential sources of bias.

Where might bias lurk? Here are three red flags to look out for in your data set.

Missing Feature Values

If your data set has one or more features that have missing values for a large number of examples, that could be an indicator that certain key characteristics of your data set are under-represented.

For example, the table below shows a summary of key stats for a subset of features in the California Housing dataset (https://developers.google.com/machine-learning/crash-course/california-housing-data-description), stored in a pandas `DataFrame` and generated via `DataFrame.describe` (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.describe.html). Note that all features have a `count` of 17000, indicating there are no missing values:

	longitude	latitude	total_rooms	population	households	median_income	median_house_va
count	17000.0	17000.0	17000.0	17000.0	17000.0	17000.0	17000.0
mean	-119.6	35.6	2643.7	1429.6	501.2	3.9	207.3
std	2.0	2.1	2179.9	1147.9	384.5	1.9	116.0
min	-124.3	32.5	2.0	3.0	1.0	0.5	15.0
25%	-121.8	33.9	1462.0	790.0	282.0	2.6	119.4
50%	-118.5	34.2	2127.0	1167.0	409.0	3.5	180.4

	longitude	latitude	total_rooms	population	households	median_income	median_house_va
75%	-118.0	37.7	3151.2	1721.0	605.2	4.8	265.0
max	-114.3	42.0	37937.0	35682.0	6082.0	15.0	500.0

Suppose instead that three features (`population`, `households`, and `median_income`) only had a count of `3000`—in other words, that there were 14,000 missing values for each feature:

	longitude	latitude	total_rooms	population	households	median_income	median_house_va
count	17000.0	17000.0	17000.0	3000.0	3000.0	3000.0	17000.0
mean	-119.6	35.6	2643.7	1429.6	501.2	3.9	207.3
std	2.0	2.1	2179.9	1147.9	384.5	1.9	116.0
min	-124.3	32.5	2.0	3.0	1.0	0.5	15.0
25%	-121.8	33.9	1462.0	790.0	282.0	2.6	119.4
50%	-118.5	34.2	2127.0	1167.0	409.0	3.5	180.4
75%	-118.0	37.7	3151.2	1721.0	605.2	4.8	265.0
max	-114.3	42.0	37937.0	35682.0	6082.0	15.0	500.0

These 14,000 missing values would make it much more difficult to accurately correlate average income of households with median house prices. Before training a model on this data, it would be prudent to investigate the cause of these missing values to ensure that there are no latent biases responsible for missing income and population data.

Unexpected Feature Values

When exploring data, you should also look for examples that contain feature values that stand out as especially uncharacteristic or unusual. These unexpected feature values could

indicate problems that occurred during data collection or other inaccuracies that could introduce bias.

For example, take a look at the following excerpted examples from the California housing data set:

	longitude	latitude	total_rooms	population	households	median_income	median_house_value
1	-121.7	38.0	7105.0	3523.0	1088.0	5.0	0.2
2	-122.4	37.8	2479.0	1816.0	496.0	3.1	0.3
3	-122.0	37.0	2813.0	1337.0	477.0	3.7	0.3
4	-103.5	43.8	2212.0	803.0	144.0	5.3	0.2
5	-117.1	32.8	2963.0	1162.0	556.0	3.6	0.2
6	-118.0	33.7	3396.0	1542.0	472.0	7.4	0.4

Can you pinpoint any unexpected feature values?

Click the dropdown arrow to see the answer

	longitude	latitude	total_rooms	population	households	median_income	median_house
1	-121.7	38.0	7105.0	3523.0	1088.0	5.0	0.2
2	-122.4	37.8	2479.0	1816.0	496.0	3.1	0.3
3	-122.0	37.0	2813.0	1337.0	477.0	3.7	0.3
4	-103.5	43.8	2212.0	803.0	144.0	5.3	0.2
5	-117.1	32.8	2963.0	1162.0	556.0	3.6	0.2
6	-118.0	33.7	3396.0	1542.0	472.0	7.4	0.4

The longitude and latitude coordinates in example 4 (-103.5 and 43.8, respectively) do not fall within the U.S. state of California. In fact, they are the approximate coordinates of Mount Rushmore National Memorial (https://wikipedia.org/wiki/Mount_Rushmore) in the state of South Dakota. This is a bogus example that we inserted into the data set.

Data Skew

Any sort of skew in your data, where certain groups or characteristics may be under- or over-represented relative to their real-world prevalence, can introduce bias into your model.

If you completed the Validation programming exercise (https://developers.google.com/machine-learning/crash-course/validation/programming-exercise), you may recall discovering how a failure to randomize the California housing data set prior to splitting it into training and validation sets resulted in a pronounced data skew. Figure 1 visualizes a subset of data drawn from the full data set that exclusively represents the northwest region of California.

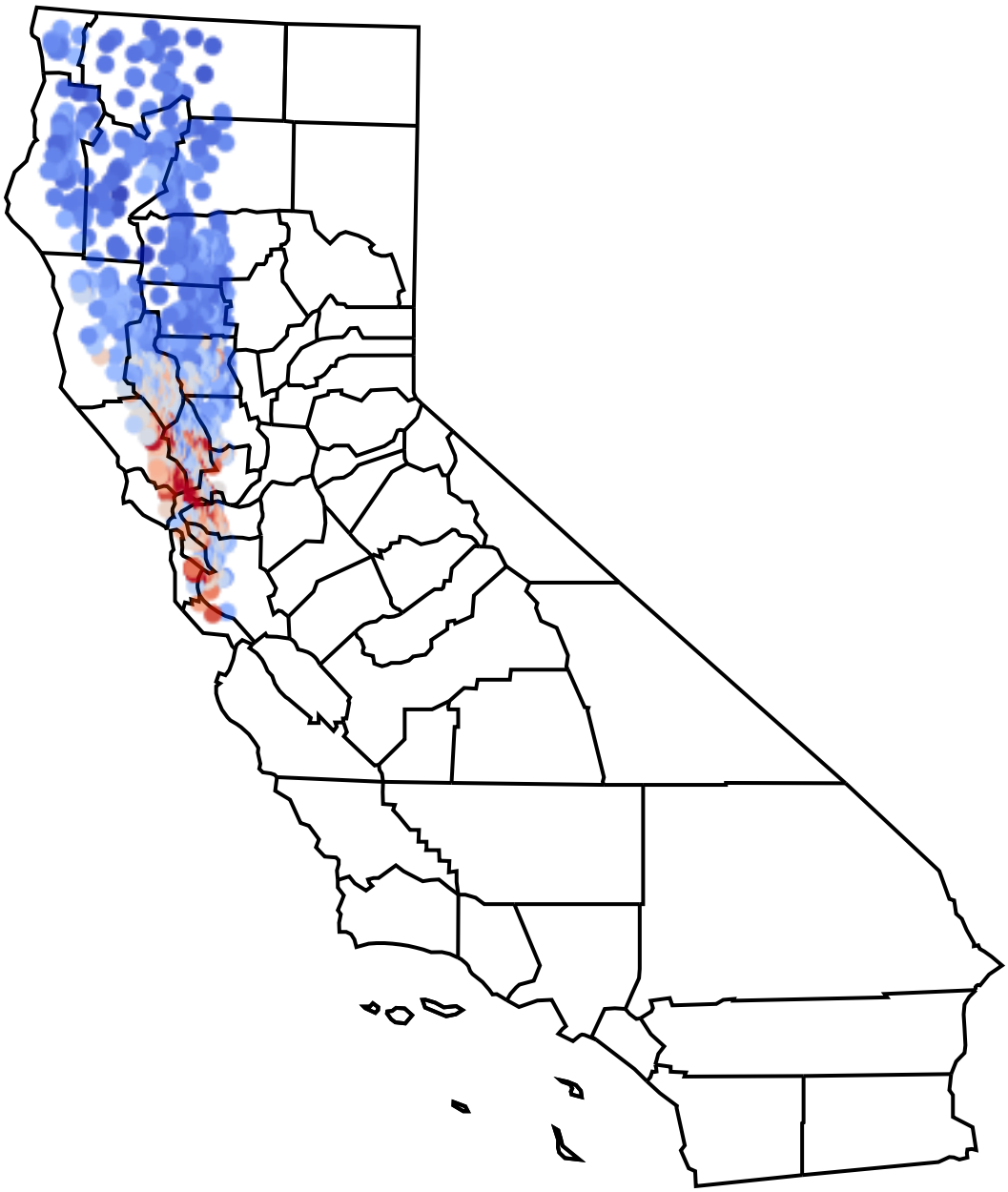


Figure 1. California state map overlaid with data from the California Housing data set. Each dot represents a housing block, with colors ranging from blue to red corresponding to median house price ranging from low to high, respectively.

If this unrepresentative sample were used to train a model to predict California housing prices statewide, the lack of housing data from southern portions of California would be problematic. The geographical bias encoded in the model might adversely affect homebuyers in unrepresented communities.

Previous

← Types of Bias
(https://developers.google.com/machine-learning/crash-course/fairness/types-of-bias)

Next
Evaluating for Bias →
(https://developers.google.com/machine-learning/crash-course/fairness/evaluating-for-bias)

Last updated April 18, 2019.

Fairness: Evaluating for Bias

Estimated Time: 5 minutes

When evaluating a model, metrics calculated against an entire test or validation set don't always give an accurate picture of how fair the model is.

Consider a new model developed to predict the presence of tumors that is evaluated against a validation set of 1,000 patients' medical records. 500 records are from female patients, and 500 records are from male patients. The following confusion matrix (https://developers.google.com/machine-learning/glossary#confusion_matrix) summarizes the results for all 1,000 examples:

True Positives (TPs): 16	**False Positives (FPs): 4**
False Negatives (FNs): 6	**True Negatives (TNs): 974**

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{16}{16 + 4} = 0.800$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{16}{16 + 6} = 0.727$$

These results look promising: precision of 80% and recall of 72.7%. But what happens if we calculate the result separately for each set of patients? Let's break out the results into two separate confusion matrices: one for female patients and one for male patients.

Female Patient Results	
True Positives (TPs): 10	False Positives (FPs): 1
False Negatives (FNs): 1	True Negatives (TNs): 488
$\text{Precision} = \frac{TP}{TP+FP} = \frac{10}{10+1} = 0.9$	
$\text{Recall} = \frac{TP}{TP+FN} = \frac{10}{10+1} = 0.90$	

Male Patient Results	
True Positives (TPs): 6	False Positives (FPs): 3
False Negatives (FNs): 5	True Negatives (TNs): 486
$\text{Precision} = \frac{TP}{TP+FP} = \frac{6}{6+3} = 0.66$	
$\text{Recall} = \frac{TP}{TP+FN} = \frac{6}{6+5} = 0.545$	

When we calculate metrics separately for female and male patients, we see stark differences in model performance for each group.

Female patients:

- Of the 11 female patients who actually have tumors, the model correctly predicts positive for 10 patients (recall rate: 90.9%). In other words, **the model misses a tumor diagnosis in 9.1% of female cases**.
- Similarly, when the model returns positive for tumor in female patients, it is correct in 10 out of 11 cases (precision rate: 90.9%); in other words, **the model incorrectly predicts tumor in 9.1% of female cases**.

Male patients:

- However, of the 11 male patients who actually have tumors, the model correctly predicts positive for only 6 patients (recall rate: 54.5%). That means **the model misses a tumor diagnosis in 45.5% of male cases**.
- And when the model returns positive for tumor in male patients, it is correct in only 6 out of 9 cases (precision rate: 66.7%); in other words, **the model incorrectly predicts tumor in 33.3% of male cases**.

We now have a much better understanding of the biases inherent in the model's predictions, as well as the risks to each subgroup if the model were to be released for medical use in the general population.

Additional Fairness Resources

Fairness is a relatively new subfield within the discipline of machine learning. To learn more about research and initiatives devoted to developing new tools and techniques for identifying and mitigating bias in machine learning models, check out Google's Machine Learning Fairness resources page (https://developers.google.com/machine-learning/fairness-overview/).

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Identifying Bias (https://developers.google.com/machine-learning/crash-course/fairness/identifying-bias)

Next
Programming Exercise → (https://developers.google.com/machine-learning/crash-course/fairness/programming-exercise)

Last updated March 26, 2019.

Fairness: Check Your Understanding

Estimated Time: 5 minutes

Types of Bias

Explore the options below.

Which of the following model's predictions have been affected by selection bias?

🚫 Engineers built a model to predict the likelihood of a person developing diabetes based on their daily food intake. The model was trained on 10,000 "food diaries" collected from a randomly chosen group of people worldwide representing a variety of different age groups, ethnic backgrounds, and genders. However, when the model was deployed, it had very poor accuracy. Engineers subsequently discovered that food diary participants were reluctant to admit the true volume of unhealthy foods they ate, and were more likely to document consumption of nutritious food than less healthy snacks.

There is no selection bias in this model; participants who provided training data were a representative sampling of users and were chosen randomly. Instead, this model was affected by **reporting bias**. Ingestion of unhealthy foods was reported at a much lower frequency than true real-world occurrence.

Try again.

✓ Engineers at a company developed a model to predict staff turnover rates (the percentage of employees quitting their jobs each year) based on data collected from a survey sent to all employees. After several years of use, engineers determined that the model underestimated turnover by more than 20%. When conducting exit interviews with employees leaving the company, they learned that more than 80% of people who were dissatisfied with their jobs chose not to complete the survey, compared to a company-wide opt-out rate of 15%.

This model was affected by a type of selection bias called **non-response bias**. People who were dissatisfied with their jobs were underrepresented in the training data set because they opted out of the company-wide survey at much higher rates than the entire employee population.

1 of 2 correct answers.

✓ A German handwriting recognition smartphone app uses a model that frequently incorrectly classifies ß (https://wikipedia.org/wiki/%C3%9F) (Eszett) characters as B (https://wikipedia.org/wiki/B) characters, because it was trained on a corpus of American handwriting samples, mostly written in English.

This model was affected by a type of selection bias called **coverage bias**: the training data (American English handwriting) was not representative of the type of data provided by the model's target audience (German handwriting).

2 of 2 correct answers.

🚫 Engineers developing a movie-recommendation system hypothesized that people who like horror movies will also like science-fiction movies. When they trained a model on 50,000 users' watchlists, however, it showed no such correlation between preferences for horror and for sci-fi; instead it showed a strong correlation between preferences for horror and for documentaries. This seemed odd to them, so they retrained the model five more times using different hyperparameters. Their final trained model showed a 70% correlation between preferences for horror and for sci-fi, so they confidently released it into production.

There is no evidence of selection bias, but this model may have instead been affected by **experimenter's bias**, as the engineers kept iterating on their model until it confirmed their preexisting hypothesis.

Try again.

Evaluating for Bias

A sarcasm (https://wikipedia.org/wiki/Sarcasm)-detection model was trained on 80,000 text messages: 40,000 messages sent by adults (18 years and older) and 40,000 messages

sent by minors (less than 18 years old). The model was then evaluated on a test set of 20,000 messages: 10,000 from adults and 10,000 from minors. The following confusion matrices show the results for each group (a positive prediction signifies a classification of "sarcastic"; a negative prediction signifies a classificaton of "not sarcastic"):

Adults	
True Positives (TPs): 512	**False Positives (FPs): 51**
False Negatives (FNs): 36	**True Negatives (TNs): 9401**

$$\text{Precision} = \frac{TP}{TP + FP} = 0.909$$

$$\text{Recall} = \frac{TP}{TP + FN} = 0.934$$

Minors	
True Positives (TPs): 2147	**False Positives (FPs): 96**
False Negatives (FNs): 2177	**True Negatives (TNs): 5580**

$$\text{Precision} = \frac{TP}{TP + FP} = 0.957$$

$$\text{Recall} = \frac{TP}{TP + FN} = 0.497$$

Explore the options below.

Which of the following statements about the model's test-set performance are true?

✓ The 10,000 messages sent by adults are a class-imbalanced (https://developers.google.com/machine-learning/glossary/#class_imbalanced_data_set) dataset.

If we compare the number of messages from adults that are actually sarcastic (TP+FN = 548) with the number of messages that are actually not sarcastic (TN + FP = 9452), we see that "not sarcastic" labels outnumber "sarcastic" labels by a ratio of approximately 17:1.

1 of 3 correct answers.

🚫 Approximately 50% of messages sent by minors are classified as "sarcastic" incorrectly.

The precision rate of 0.957 indicates that over 95% of minors' messages classified as "sarcastic" are actually sarcastic.

Try again.

✓ Overall, the model performs better on examples from adults than on examples from minors.

The model achieves both precision and recall rates over 90% when detecting sarcasm in text messages from adults.

While the model achieves a slightly higher precision rate for minors than adults, the recall rate is substantially lower for minors, resulting in less reliable predictions for this group.

2 of 3 correct answers.

🚫 The 10,000 messages sent by minors are a class-imbalanced (https://developers.google.com/machine-learning/glossary/#class_imbalanced_data_set) dataset.

If we compare the number of messages from minors that are actually sarcastic (TP+FN = 4324) with the number of messages that are actually not sarcastic (TN + FP = 5676), we see that there is a 1.3:1 ratio of "not sarcastic" labels to "sarcastic" labels. Given that the distribution of labels between the two classes is quite close to 50/50, this is not a class-imbalanced dataset.

Try again.

✓ The model fails to classify approximately 50% of sarcastic messages as "sarcastic."

The recall rate of 0.497 for minors indicates that the model predicts "not sarcastic" for approximately 50% of minors' sarcastic texts.

3 of 3 correct answers.

Explore the options below.

Engineers are working on retraining this model to address inconsistencies in sarcasm-detection accuracy across age demographics, but the model has already been released into production. Which of the following stopgap strategies will help mitigate errors in the model's predictions?

✓ Restrict the model's usage to text messages sent by adults.

The model performs well on text messages from adults (with precision and recall rates both above 90%), so restricting its use to this group will sidestep the systematic errors in classifying minors' text messages.

1 of 2 correct answers.

🚫 Restrict the model's usage to text messages sent by minors.

The systematic errors in this model are specific to text messages sent by minors. Restricting the model's use to the group more susceptible to error would not help.

Try again.

✓ When the model predicts "not sarcastic" for text messages sent by minors, adjust the output so the model returns a value of "unsure" instead.

The precision rate for text messages sent by minors is high, which means that when the model predicts "sarcastic" for this group, it is nearly always correct.

The problem is that recall is very low for minors; The model fails to identify sarcasm in approximately 50% of examples. Given that the model's negative predictions for minors are no better than random guesses, we can avoid these errors by not providing a prediction in these cases.

2 of 2 correct answers.

🚫 Adjust the model output so that it returns "sarcastic" for all text messages sent by minors, regardless of what the model originally predicted.

Always predicting "sarcastic" for minors' text messages would increase the recall rate from 0.497 to 1.0, as the model would no longer fail to identify any messages as sarcastic. However, this increase in recall would come at the expense of precision. All the true negatives would be changed to false positives:

True Positives (TPs): 4324	**False Positives (FPs): 5676**
False Negatives (FNs): 0	**True Negatives (TNs): 0**

which would decrease the precision rate from 0.957 to 0.432. So, adding this calibration would change the type of error but would not mitigate the magnitude of the error.

Try again.

HELP CENTER (HTTPS://SUPPORT.GOOGLE.COM/MACHINELEARNINGEDUCATION)

Previous
← Programming Exercise
(https://developers.google.com/machine-learning/crash-course/fairness/programming-exercise)

Next
Cancer Prediction (5 min) →
(https://developers.google.com/machine-learning/crash-course/cancer-prediction)

Last updated March 26, 2019.

Rules of Machine Learning:

Best Practices for ML Engineering

Martin Zinkevich

This document is intended to help those with a basic knowledge of machine learning get the benefit of Google's best practices in machine learning. It presents a style for machine learning, similar to the Google C++ Style Guide and other popular guides to practical programming. If you have taken a class in machine learning, or built or worked on a machine-learned model, then you have the necessary background to read this document.

Martin Zinkevich introduces 10 of his favorite rules of machine learning. Read on to learn all 43 rules!

Terminology

The following terms will come up repeatedly in our discussion of effective machine learning:

- **Instance**: The thing about which you want to make a prediction. For example, the instance might be a web page that you want to classify as either "about cats" or "not about cats".

- **Label**: An answer for a prediction task either the answer produced by a machine learning system, or the right answer supplied in training data. For example, the label for a web page might be "about cats".
- **Feature**: A property of an instance used in a prediction task. For example, a web page might have a feature "contains the word 'cat'".
- **Feature Column**: A set of related features, such as the set of all possible countries in which users might live. An example may have one or more features present in a feature column. "Feature column" is Google-specific terminology. A feature column is referred to as a "namespace" in the VW system (at Yahoo/Microsoft), or a field (https://www.csie.ntu.edu.tw/%7Ecjlin/libffm/).
- **Example**: An instance (with its features) and a label.
- **Model**: A statistical representation of a prediction task. You train a model on examples then use the model to make predictions.
- **Metric**: A number that you care about. May or may not be directly optimized.
- **Objective**: A metric that your algorithm is trying to optimize.
- **Pipeline**: The infrastructure surrounding a machine learning algorithm. Includes gathering the data from the front end, putting it into training data files, training one or more models, and exporting the models to production.
- **Click-through Rate** The percentage of visitors to a web page who click a link in an ad.

Overview

To make great products:

do machine learning like the great engineer you are, not like the great machine learning expert you aren't.

Most of the problems you will face are, in fact, engineering problems. Even with all the resources of a great machine learning expert, most of the gains come from great features, not great machine learning algorithms. So, the basic approach is:

1. Make sure your pipeline is solid end to end.
2. Start with a reasonable objective.
3. Add common-sense features in a simple way.
4. Make sure that your pipeline stays solid.

This approach will work well for a long period of time. Diverge from this approach only when there are no more simple tricks to get you any farther. Adding complexity slows future releases.

Once you've exhausted the simple tricks, cutting-edge machine learning might indeed be in your future. See the section on Phase III (#ml_phase_iii_slowed_growth_optimization_refinement_and_complex_models) machine learning projects.

This document is arranged as follows:

1. The first part (#before_machine_learning) should help you understand whether the time is right for building a machine learning system.
2. The second part (#ml_phase_i_your_first_pipeline) is about deploying your first pipeline.
3. The third part (#ml_phase_ii_feature_engineering) is about launching and iterating while adding new features to your pipeline, how to evaluate models and training-serving skew.
4. The final part (#ml_phase_iii_slowed_growth_optimization_refinement_and_complex_models) is about what to do when you reach a plateau.
5. Afterwards, there is a list of related work (#related_work) and an appendix (#appendix) with some background on the systems commonly used as examples in this document.

Before Machine Learning

Rule #1: Don't be afraid to launch a product without machine learning.

Machine learning is cool, but it requires data. Theoretically, you can take data from a different problem and then tweak the model for a new product, but this will likely underperform basic **heuristics** (https://developers.google.com/machine-learning/glossary/#heuristic). If you think that machine learning will give you a 100% boost, then a heuristic will get you 50% of the way there.

For instance, if you are ranking apps in an app marketplace, you could use the install rate or number of installs as heuristics. If you are detecting spam, filter out publishers that have sent spam before. Don't be afraid to use human editing either. If you need to rank contacts, rank the most recently used highest (or even rank alphabetically). If machine learning is not absolutely required for your product, don't use it until you have data.

Rule #2: First, design and implement metrics.

Before formalizing what your machine learning system will do, track as much as possible in your current system. Do this for the following reasons:

1. It is easier to gain permission from the system's users earlier on.
2. If you think that something might be a concern in the future, it is better to get historical data now.
3. If you design your system with metric instrumentation in mind, things will go better for you in the future. Specifically, you don't want to find yourself grepping for strings in logs to instrument your metrics!
4. You will notice what things change and what stays the same. For instance, suppose you want to directly optimize one-day active users. However, during your early manipulations of the system, you may notice that dramatic alterations of the user experience don't noticeably change this metric.

Google Plus (#google_plus_overview) team measures expands per read, reshares per read, plusones per read, comments/read, comments per user, reshares per user, etc. which they use in computing the goodness of a post at serving time. **Also, note that an experiment framework, in which you can group users into buckets and aggregate statistics by experiment, is important.** See Rule #12 (#rule_12_don_t_overthink_which_objective_you_choose_to_directly_optimize).

By being more liberal about gathering metrics, you can gain a broader picture of your system. Notice a problem? Add a metric to track it! Excited about some quantitative change on the last release? Add a metric to track it!

Rule #3: Choose machine learning over a complex heuristic.

A simple heuristic can get your product out the door. A complex heuristic is unmaintainable. Once you have data and a basic idea of what you are trying to accomplish, move on to machine learning. As in most software engineering tasks, you will want to be constantly updating your approach, whether it is a heuristic or a machine-learned model, and you will find that the machine-learned model is easier to update and maintain (see Rule #16 (#rule_16_plan_to_launch_and_iterate)).

ML Phase I: Your First Pipeline

Focus on your system infrastructure for your first pipeline. While it is fun to think about all the imaginative machine learning you are going to do, it will be hard to figure out what is happening if you don't first trust your pipeline.

Rule #4: Keep the first model simple and get the infrastructure right.

The first model provides the biggest boost to your product, so it doesn't need to be fancy. But you will run into many more infrastructure issues than you expect. Before anyone can use your fancy new machine learning system, you have to determine:

- How to get examples to your learning algorithm.
- A first cut as to what "good" and "bad" mean to your system.
- How to integrate your model into your application. You can either apply the model live, or precompute the model on examples offline and store the results in a table. For example, you might want to preclassify web pages and store the results in a table, but you might want to classify chat messages live.

Choosing simple features makes it easier to ensure that:

- The features reach your learning algorithm correctly.
- The model learns reasonable weights.
- The features reach your model in the server correctly.

Once you have a system that does these three things reliably, you have done most of the work. Your simple model provides you with baseline metrics and a baseline behavior that you can use to test more complex models. Some teams aim for a "neutral" first launch: a first launch that explicitly deprioritizes machine learning gains, to avoid getting distracted.

Rule #5: Test the infrastructure independently from the machine learning.

Make sure that the infrastructure is testable, and that the learning parts of the system are encapsulated so that you can test everything around it. Specifically:

1. Test getting data into the algorithm. Check that feature columns that should be populated are populated. Where privacy permits, manually inspect the input to your training algorithm. If possible, check statistics in your pipeline in comparison to statistics for the same data processed elsewhere.
2. Test getting models out of the training algorithm. Make sure that the model in your training environment gives the same score as the model in your serving environment (see Rule #37 (#rule_37_measure_training_serving_skew)).

Machine learning has an element of unpredictability, so make sure that you have tests for the code for creating examples in training and serving, and that you can load and use a fixed model during serving. Also, it is important to understand your data: see Practical Advice for Analysis of Large, Complex Data Sets (http://www.unofficialgoogledatascience.com/2016/10/practical-advice-for-analysis-of-large.html).

Rule #6: Be careful about dropped data when copying pipelines.

Often we create a pipeline by copying an existing pipeline (i.e., cargo cult programming (https://wikipedia.org/wiki/Cargo_cult_programming)), and the old pipeline drops data that we need for the new pipeline. For example, the pipeline for Google Plus (#google_plus_overview) What's Hot drops older posts (because it is trying to rank fresh posts). This pipeline was copied to use for Google Plus (#google_plus_overview) Stream, where older posts are still meaningful, but the pipeline was still dropping old posts. Another common pattern is to only log data that was seen by the user. Thus, this data is useless if we want to model why a particular post was not seen by the user, because all the negative examples have been dropped. A similar issue occurred in Play. While working on Play Apps Home, a new pipeline was created that also contained examples from the landing page for Play Games without any feature to disambiguate where each example came from.

Rule #7: Turn heuristics into features, or handle them externally.

Usually the problems that machine learning is trying to solve are not completely new. There is an existing system for ranking, or classifying, or whatever problem you are trying to solve. This means that there are a bunch of rules and heuristics. **These same heuristics can give you a lift when tweaked with machine learning.** Your heuristics should be mined for whatever information they have, for two reasons. First, the transition to a machine learned system will be smoother. Second, usually those rules contain a lot of the intuition about the system you don't want to throw away. There are four ways you can use an existing heuristic:

- Preprocess using the heuristic. If the feature is incredibly awesome, then this is an option. For example, if, in a spam filter, the sender has already been blacklisted, don't try to relearn what "blacklisted" means. Block the message. This approach makes the most sense in binary classification tasks.
- Create a feature. Directly creating a feature from the heuristic is great. For example, if you use a heuristic to compute a relevance score for a query result, you can include the score as the value of a feature. Later on you may want to use machine learning techniques to massage the value (for example, converting the value into one of a finite set of discrete values, or combining it with other features) but start by using the raw value produced by the heuristic.

- Mine the raw inputs of the heuristic. If there is a heuristic for apps that combines the number of installs, the number of characters in the text, and the day of the week, then consider pulling these pieces apart, and feeding these inputs into the learning separately. Some techniques that apply to ensembles apply here (see Rule #40 (#rule_40_keep_ensembles_simple)).
- Modify the label. This is an option when you feel that the heuristic captures information not currently contained in the label. For example, if you are trying to maximize the number of downloads, but you also want quality content, then maybe the solution is to multiply the label by the average number of stars the app received. There is a lot of leeway here. See "Your First Objective" (#your_first_objective).

Do be mindful of the added complexity when using heuristics in an ML system. Using old heuristics in your new machine learning algorithm can help to create a smooth transition, but think about whether there is a simpler way to accomplish the same effect.

Monitoring

In general, practice good alerting hygiene, such as making alerts actionable and having a dashboard page.

Rule #8: Know the freshness requirements of your system.

How much does performance degrade if you have a model that is a day old? A week old? A quarter old? This information can help you to understand the priorities of your monitoring. If you lose significant product quality if the model is not updated for a day, it makes sense to have an engineer watching it continuously. Most ad serving systems have new advertisements to handle every day, and must update daily. For instance, if the ML model for Google Play Search (#google_play_overview) is not updated, it can have a negative impact in under a month. Some models for What's Hot in Google Plus (#google_plus_overview) have no post identifier in their model so they can export these models infrequently. Other models that have post identifiers are updated much more frequently. Also notice that freshness can change over time, especially when feature columns are added or removed from your model.

Rule #9: Detect problems before exporting models.

Many machine learning systems have a stage where you export the model to serving. If there is an issue with an exported model, it is a user-facing issue.

Do sanity checks right before you export the model. Specifically, make sure that the model's performance is reasonable on held out data. Or, if you have lingering concerns with the

data, don't export a model. Many teams continuously deploying models check the area under the ROC curve (https://wikipedia.org/wiki/Receiver_operating_characteristic) (or AUC) before exporting. **Issues about models that haven't been exported require an email alert, but issues on a user-facing model may require a page.** So better to wait and be sure before impacting users.

Rule #10: Watch for silent failures.

This is a problem that occurs more for machine learning systems than for other kinds of systems. Suppose that a particular table that is being joined is no longer being updated. The machine learning system will adjust, and behavior will continue to be reasonably good, decaying gradually. Sometimes you find tables that are months out of date, and a simple refresh improves performance more than any other launch that quarter! The coverage of a feature may change due to implementation changes: for example a feature column could be populated in 90% of the examples, and suddenly drop to 60% of the examples. Play once had a table that was stale for 6 months, and refreshing the table alone gave a boost of 2% in install rate. If you track statistics of the data, as well as manually inspect the data on occassion, you can reduce these kinds of failures.

Rule #11: Give feature columns owners and documentation.

If the system is large, and there are many feature columns, know who created or is maintaining each feature column. If you find that the person who understands a feature column is leaving, make sure that someone has the information. Although many feature columns have descriptive names, it's good to have a more detailed description of what the feature is, where it came from, and how it is expected to help.

Your First Objective

You have many metrics, or measurements about the system that you care about, but your machine learning algorithm will often require a single **objective, a number that your algorithm is "trying" to optimize.** I distinguish here between objectives and metrics: a metric is any number that your system reports, which may or may not be important. See also Rule #2 (#rule_2_first_design_and_implement_metrics).

Rule #12: Don't overthink which objective you choose to directly optimize.

You want to make money, make your users happy, and make the world a better place. There are tons of metrics that you care about, and you should measure them all (see Rule #2 (#rule_2_first_design_and_implement_metrics)). However, early in the machine learning process,

you will notice them all going up, even those that you do not directly optimize. For instance, suppose you care about number of clicks and time spent on the site. If you optimize for number of clicks, you are likely to see the time spent increase.

So, keep it simple and don't think too hard about balancing different metrics when you can still easily increase all the metrics. Don't take this rule too far though: do not confuse your objective with the ultimate health of the system (see Rule #39 (#rule_39_launch_decisions_are_a_proxy_for_long_term_product_goals)). And, **if you find yourself increasing the directly optimized metric, but deciding not to launch, some objective revision may be required.**

Rule #13: Choose a simple, observable and attributable metric for your first objective.

Often you don't know what the true objective is. You think you do but then as you stare at the data and side-by-side analysis of your old system and new ML system, you realize you want to tweak the objective. Further, different team members often can't agree on the true objective. **The ML objective should be something that is easy to measure and is a proxy for the "true" objective.** In fact, there is often no "true" objective (see Rule#39 (#rule_39_launch_decisions_are_a_proxy_for_long_term_product_goals)). So train on the simple ML objective, and consider having a "policy layer" on top that allows you to add additional logic (hopefully very simple logic) to do the final ranking.

The easiest thing to model is a user behavior that is directly observed and attributable to an action of the system:

- Was this ranked link clicked?
- Was this ranked object downloaded?
- Was this ranked object forwarded/replied to/emailed?
- Was this ranked object rated?
- Was this shown object marked as spam/pornography/offensive?

Avoid modeling indirect effects at first:

- Did the user visit the next day?
- How long did the user visit the site?
- What were the daily active users?

Indirect effects make great metrics, and can be used during A/B testing and during launch decisions.

Finally, don't try to get the machine learning to figure out:

- Is the user happy using the product?
- Is the user satisfied with the experience?
- Is the product improving the user's overall wellbeing?
- How will this affect the company's overall health?

These are all important, but also incredibly hard to measure. Instead, use proxies: if the user is happy, they will stay on the site longer. If the user is satisfied, they will visit again tomorrow. Insofar as well-being and company health is concerned, human judgement is required to connect any machine learned objective to the nature of the product you are selling and your business plan.

Rule #14: Starting with an interpretable model makes debugging easier.

Linear regression, logistic regression, and Poisson regression are directly motivated by a probabilistic model. Each prediction is interpretable as a probability or an expected value. This makes them easier to debug than models that use objectives (zero-one loss, various hinge losses, and so on) that try to directly optimize classification accuracy or ranking performance. For example, if probabilities in training deviate from probabilities predicted in side-by-sides or by inspecting the production system, this deviation could reveal a problem.

For example, in linear, logistic, or Poisson regression, **there are subsets of the data where the average predicted expectation equals the average label (1- moment calibrated, or just calibrated)**. This is true assuming that you have no regularization and that your algorithm has converged, and it is approximately true in general. If you have a feature which is either 1 or 0 for each example, then the set of 3 examples where that feature is 1 is calibrated. Also, if you have a feature that is 1 for every example, then the set of all examples is calibrated.

With simple models, it is easier to deal with feedback loops (see Rule #36 (#rule_36_avoid_feedback_loops_with_positional_features)). Often, we use these probabilistic predictions to make a decision: e.g. rank posts in decreasing expected value (i.e. probability of click/download/etc.). **However, remember when it comes time to choose which model to use, the decision matters more than the likelihood of the data given the model (see Rule #27 (#rule_27_try_to_quantify_observed_undesirable_behavior)).**

Rule #15: Separate Spam Filtering and Quality Ranking in a Policy Layer.

Quality ranking is a fine art, but spam filtering is a war. The signals that you use to determine high quality posts will become obvious to those who use your system, and they will tweak their posts to have these properties. Thus, your quality ranking should focus on ranking content that is posted in good faith. You should not discount the quality ranking

learner for ranking spam highly. **Similarly, "racy" content should be handled separately from Quality Ranking.** Spam filtering is a different story. You have to expect that the features that you need to generate will be constantly changing. Often, there will be obvious rules that you put into the system (if a post has more than three spam votes, don't retrieve it, et cetera). Any learned model will have to be updated daily, if not faster. The reputation of the creator of the content will play a great role.

At some level, the output of these two systems will have to be integrated. Keep in mind, filtering spam in search results should probably be more aggressive than filtering spam in email messages. This is true assuming that you have no regularization and that your algorithm has converged. It is approximately true in general. Also, it is a standard practice to remove spam from the training data for the quality classifier.

ML Phase II: Feature Engineering

In the first phase of the lifecycle of a machine learning system, the important issues are to get the training data into the learning system, get any metrics of interest instrumented, and create a serving infrastructure. **After you have a working end to end system with unit and system tests instrumented, Phase II begins.**

In the second phase, there is a lot of low-hanging fruit. There are a variety of obvious features that could be pulled into the system. Thus, the second phase of machine learning involves pulling in as many features as possible and combining them in intuitive ways. During this phase, all of the metrics should still be rising. There will be lots of launches, and it is a great time to pull in lots of engineers that can join up all the data that you need to create a truly awesome learning system.

Rule #16: Plan to launch and iterate.

Don't expect that the model you are working on now will be the last one that you will launch, or even that you will ever stop launching models. Thus consider whether the complexity you are adding with this launch will slow down future launches. Many teams have launched a model per quarter or more for years. There are three basic reasons to launch new models:

- You are coming up with new features.
- You are tuning regularization and combining old features in new ways.
- You are tuning the objective.

Regardless, giving a model a bit of love can be good: looking over the data feeding into the example can help find new signals as well as old, broken ones. So, as you build your model,

think about how easy it is to add or remove or recombine features. Think about how easy it is to create a fresh copy of the pipeline and verify its correctness. Think about whether it is possible to have two or three copies running in parallel. Finally, don't worry about whether feature 16 of 35 makes it into this version of the pipeline. You'll get it next quarter.

Rule #17: Start with directly observed and reported features as opposed to learned features.

This might be a controversial point, but it avoids a lot of pitfalls. First of all, let's describe what a learned feature is. A learned feature is a feature generated either by an external system (such as an unsupervised clustering system) or by the learner itself (e.g. via a factored model or deep learning). Both of these can be useful, but they can have a lot of issues, so they should not be in the first model.

If you use an external system to create a feature, remember that the external system has its own objective. The external system's objective may be only weakly correlated with your current objective. If you grab a snapshot of the external system, then it can become out of date. If you update the features from the external system, then the meanings may change. If you use an external system to provide a feature, be aware that this approach requires a great deal of care.

The primary issue with factored models and deep models is that they are nonconvex. Thus, there is no guarantee that an optimal solution can be approximated or found, and the local minima found on each iteration can be different. This variation makes it hard to judge whether the impact of a change to your system is meaningful or random. By creating a model without deep features, you can get an excellent baseline performance. After this baseline is achieved, you can try more esoteric approaches.

Rule #18: Explore with features of content that generalize across contexts.

Often a machine learning system is a small part of a much bigger picture. For example, if you imagine a post that might be used in What's Hot, many people will plus-one, reshare, or comment on a post before it is ever shown in What's Hot. If you provide those statistics to the learner, it can promote new posts that it has no data for in the context it is optimizing. YouTube (#youtube_overview) Watch Next could use number of watches, or co- watches (counts of how many times one video was watched after another was watched) from YouTube (#youtube_overview) search. You can also use explicit user ratings. Finally, if you have a user action that you are using as a label, seeing that action on the document in a different context can be a great feature. All of these features allow you to bring new content into the context. Note that this is not about personalization: figure out if someone likes the content in this context first, then figure out who likes it more or less.

Rule #19: Use very specific features when you can.

With tons of data, it is simpler to learn millions of simple features than a few complex features. Identifiers of documents being retrieved and canonicalized queries do not provide much generalization, but align your ranking with your labels on head queries. Thus, don't be afraid of groups of features where each feature applies to a very small fraction of your data, but overall coverage is above 90%. You can use regularization to eliminate the features that apply to too few examples.

Rule #20: Combine and modify existing features to create new features in human-understandable ways.

There are a variety of ways to combine and modify features. Machine learning systems such as TensorFlow allow you to pre-process your data through transformations (https://www.tensorflow.org/tutorials/linear#feature-columns-and-transformations). The two most standard approaches are "discretizations" and "crosses".

Discretization consists of taking a continuous feature and creating many discrete features from it. Consider a continuous feature such as age. You can create a feature which is 1 when age is less than 18, another feature which is 1 when age is between 18 and 35, et cetera. Don't overthink the boundaries of these histograms: basic quantiles will give you most of the impact.

Crosses combine two or more feature columns. A feature column, in TensorFlow's terminology, is a set of homogenous features, (e.g. {male, female}, {US, Canada, Mexico}, et cetera). A cross is a new feature column with features in, for example, {male, female} × {US,Canada, Mexico}. This new feature column will contain the feature (male, Canada). If you are using TensorFlow and you tell TensorFlow to create this cross for you, this (male, Canada) feature will be present in examples representing male Canadians. Note that it takes massive amounts of data to learn models with crosses of three, four, or more base feature columns.

Crosses that produce very large feature columns may overfit. For instance, imagine that you are doing some sort of search, and you have a feature column with words in the query, and you have a feature column with words in the document. You can combine these with a cross, but you will end up with a lot of features (see Rule #21 (#rule_21_the_number_of_feature_weights_you_can_learn_in_a_linear_model_is_roughly_proportional_to_the_amount_of_data_you_have)
).

When working with text there are two alternatives. The most draconian is a dot product. A dot product in its simplest form simply counts the number of words in common between

the query and the document. This feature can then be discretized. Another approach is an intersection: thus, we will have a feature which is present if and only if the word "pony" is in both the document and the query, and another feature which is present if and only if the word "the" is in both the document and the query.

Rule #21: The number of feature weights you can learn in a linear model is roughly proportional to the amount of data you have.

There are fascinating statistical learning theory results concerning the appropriate level of complexity for a model, but this rule is basically all you need to know. I have had conversations in which people were doubtful that anything can be learned from one thousand examples, or that you would ever need more than one million examples, because they get stuck in a certain method of learning. The key is to scale your learning to the size of your data:

1. If you are working on a search ranking system, and there are millions of different words in the documents and the query and you have 1000 labeled examples, then you should use a dot product between document and query features, TF-IDF (https://wikipedia.org/wiki/Tf%E2%80%93idf), and a half-dozen other highly human-engineered features. 1000 examples, a dozen features.
2. If you have a million examples, then intersect the document and query feature columns, using regularization and possibly feature selection. This will give you millions of features, but with regularization you will have fewer. Ten million examples, maybe a hundred thousand features.
3. If you have billions or hundreds of billions of examples, you can cross the feature columns with document and query tokens, using feature selection and regularization. You will have a billion examples, and 10 million features. Statistical learning theory rarely gives tight bounds, but gives great guidance for a starting point.

In the end, use Rule #28 (#rule_28_be_aware_that_identical_short_term_behavior_does_not_imply_identical_long_term_behavior) to decide what features to use.

Rule #22: Clean up features you are no longer using.

Unused features create technical debt. If you find that you are not using a feature, and that combining it with other features is not working, then drop it out of your infrastructure. You want to keep your infrastructure clean so that the most promising features can be tried as fast as possible. If necessary, someone can always add back your feature.

Keep coverage in mind when considering what features to add or keep. How many examples are covered by the feature? For example, if you have some personalization features, but only 8% of your users have any personalization features, it is not going to be very effective.

At the same time, some features may punch above their weight. For example, if you have a feature which covers only 1% of the data, but 90% of the examples that have the feature are positive, then it will be a great feature to add.

Human Analysis of the System

Before going on to the third phase of machine learning, it is important to focus on something that is not taught in any machine learning class: how to look at an existing model, and improve it. This is more of an art than a science, and yet there are several anti-patterns that it helps to avoid.

Rule #23: You are not a typical end user.

This is perhaps the easiest way for a team to get bogged down. While there are a lot of benefits to fishfooding (using a prototype within your team) and dogfooding (using a prototype within your company), employees should look at whether the performance is correct. While a change which is obviously bad should not be used, anything that looks reasonably near production should be tested further, either by paying laypeople to answer questions on a crowdsourcing platform, or through a live experiment on real users.

There are two reasons for this. The first is that you are too close to the code. You may be looking for a particular aspect of the posts, or you are simply too emotionally involved (e.g. confirmation bias). The second is that your time is too valuable. Consider the cost of nine engineers sitting in a one hour meeting, and think of how many contracted human labels that buys on a crowdsourcing platform.

If you really want to have user feedback, **use user experience methodologies**. Create user personas (one description is in Bill Buxton's Sketching User Experiences (https://play.google.com/store/books/details/Bill_Buxton_Sketching_User_Experiences_Getting_the?id=2vfPxocmLh0C)) early in a process and do usability testing (one description is in Steve Krug's Don't Make Me Think (https://play.google.com/store/books/details/Steve_Krug_Don_t_Make_Me_Think_Revisited?id=QlduAgAAQBAJ)) later. User personas involve creating a hypothetical user. For instance, if your team is all male, it might help to design a 35-year-old female user persona (complete with user

features), and look at the results it generates rather than 10 results for 25-to-40 year old males. Bringing in actual people to watch their reaction to your site (locally or remotely) in usability testing can also get you a fresh perspective.

Rule #24: Measure the delta between models.

One of the easiest and sometimes most useful measurements you can make before any users have looked at your new model is to calculate just how different the new results are from production. For instance, if you have a ranking problem, run both models on a sample of queries through the entire system, and look at the size of the symmetric difference of the results (weighted by ranking position). If the difference is very small, then you can tell without running an experiment that there will be little change. If the difference is very large, then you want to make sure that the change is good. Looking over queries where the symmetric difference is high can help you to understand qualitatively what the change was like. Make sure, however, that the system is stable. Make sure that a model when compared with itself has a low (ideally zero) symmetric difference.

Rule #25: When choosing models, utilitarian performance trumps predictive power.

Your model may try to predict click-through rate. However, in the end, the key question is what you do with that prediction. If you are using it to rank documents, then the quality of the final ranking matters more than the prediction itself. If you predict the probability that a document is spam and then have a cutoff on what is blocked, then the precision of what is allowed through matters more. Most of the time, these two things should be in agreement: when they do not agree, it will likely be on a small gain. Thus, if there is some change that improves log loss but degrades the performance of the system, look for another feature. When this starts happening more often, it is time to revisit the objective of your model.

Rule #26: Look for patterns in the measured errors, and create new features.

Suppose that you see a training example that the model got "wrong". In a classification task, this error could be a false positive or a false negative. In a ranking task, the error could be a pair where a positive was ranked lower than a negative. The most important point is that this is an example that the machine learning system knows it got wrong and would like to fix if given the opportunity. If you give the model a feature that allows it to fix the error, the model will try to use it.

On the other hand, if you try to create a feature based upon examples the system doesn't see as mistakes, the feature will be ignored. For instance, suppose that in Play Apps Search, someone searches for "free games". Suppose one of the top results is a less relevant gag app. So you create a feature for "gag apps". However, if you are maximizing number of

installs, and people install a gag app when they search for free games, the "gag apps" feature won't have the effect you want.

Once you have examples that the model got wrong, look for trends that are outside your current feature set. For instance, if the system seems to be demoting longer posts, then add post length. Don't be too specific about the features you add. If you are going to add post length, don't try to guess what long means, just add a dozen features and the let model figure out what to do with them (see Rule #21 (#rule_21_the_number_of_feature_weights_you_can_learn_in_a_linear_model_is_roughly_proportional_to_the_amount_of_data_you_have)). That is the easiest way to get what you want.

Rule #27: Try to quantify observed undesirable behavior.

Some members of your team will start to be frustrated with properties of the system they don't like which aren't captured by the existing loss function. At this point, they should do whatever it takes to turn their gripes into solid numbers. For example, if they think that too many "gag apps" are being shown in Play Search, they could have human raters identify gag apps. (You can feasibly use humanlabelled data in this case because a relatively small fraction of the queries account for a large fraction of the traffic.) If your issues are measurable, then you can start using them as features, objectives, or metrics. The general rule is "**measure first, optimize second**".

Rule #28: Be aware that identical short-term behavior does not imply identical long-term behavior.

Imagine that you have a new system that looks at every doc_id and exact_query, and then calculates the probability of click for every doc for every query. You find that its behavior is nearly identical to your current system in both side by sides and A/B testing, so given its simplicity, you launch it. However, you notice that no new apps are being shown. Why? Well, since your system only shows a doc based on its own history with that query, there is no way to learn that a new doc should be shown.

The only way to understand how such a system would work long-term is to have it train only on data acquired when the model was live. This is very difficult.

Training-Serving Skew

Training-serving skew is a difference between performance during training and performance during serving. This skew can be caused by:

- A discrepancy between how you handle data in the training and serving pipelines.
- A change in the data between when you train and when you serve.
- A feedback loop between your model and your algorithm.

We have observed production machine learning systems at Google with training- serving skew that negatively impacts performance. The best solution is to explicitly monitor it so that system and data changes don't introduce skew unnoticed.

Rule #29: The best way to make sure that you train like you serve is to save the set of features used at serving time, and then pipe those features to a log to use them at training time.

Even if you can't do this for every example, do it for a small fraction, such that you can verify the consistency between serving and training (see Rule #37 (#rule_37_measure_training_serving_skew)). Teams that have made this measurement at Google were sometimes surprised by the results. YouTube (#youtube_overview) home page switched to logging features at serving time with significant quality improvements and a reduction in code complexity, and many teams are switching their infrastructure as we speak.

Rule #30: Importance-weight sampled data, don't arbitrarily drop it!

When you have too much data, there is a temptation to take files 1-12, and ignore files 13-99. This is a mistake. Although data that was never shown to the user can be dropped, importance weighting is best for the rest. Importance weighting means that if you decide that you are going to sample example X with a 30% probability, then give it a weight of 10/3. **With importance weighting, all of the calibration properties discussed in Rule #14 (#rule_14_starting_with_an_interpretable_model_makes_debugging_easier) still hold.**

Rule #31: Beware that if you join data from a table at training and serving time, the data in the table may change.

Say you join doc ids with a table containing features for those docs (such as number of comments or clicks). Between training and serving time, features in the table may be changed. Your model's prediction for the same document may then differ between training and serving. The easiest way to avoid this sort of problem is to log features at serving time (see Rule #32 (#rule_32_re_use_code_between_your_training_pipeline_and_your_serving_pipeline_whenever_possible)). If the table is changing only slowly, you can also snapshot the table hourly or daily to get reasonably close data. Note that this still doesn't completely resolve the issue.

Rule #32: Re-use code between your training pipeline and your serving pipeline whenever possible.

Batch processing is different than online processing. In online processing, you must handle each request as it arrives (e.g. you must do a separate lookup for each query), whereas in batch processing, you can combine tasks (e.g. making a join). At serving time, you are doing online processing, whereas training is a batch processing task. However, there are some things that you can do to re-use code. For example, you can create an object that is particular to your system where the result of any queries or joins can be stored in a very human readable way, and errors can be tested easily. Then, once you have gathered all the information, during serving or training, you run a common method to bridge between the human-readable object that is specific to your system, and whatever format the machine learning system expects. **This eliminates a source of training-serving skew**. As a corollary, try not to use two different programming languages between training and serving. That decision will make it nearly impossible for you to share code.

Rule #33: If you produce a model based on the data until January 5th, test the model on the data from January 6th and after.

In general, measure performance of a model on the data gathered after the data you trained the model on, as this better reflects what your system will do in production. If you produce a model based on the data until January 5th, test the model on the data from January 6th. You will expect that the performance will not be as good on the new data, but it shouldn't be radically worse. Since there might be daily effects, you might not predict the average click rate or conversion rate, but the area under the curve, which represents the likelihood of giving the positive example a score higher than a negative example, should be reasonably close.

Rule #34: In binary classification for filtering (such as spam detection or determining interesting emails), make small short-term sacrifices in performance for very clean data.

In a filtering task, examples which are marked as negative are not shown to the user. Suppose you have a filter that blocks 75% of the negative examples at serving. You might be tempted to draw additional training data from the instances shown to users. For example, if a user marks an email as spam that your filter let through, you might want to learn from that.

But this approach introduces sampling bias. You can gather cleaner data if instead during serving you label 1% of all traffic as "held out", and send all held out examples to the user. Now your filter is blocking at least 74% of the negative examples. These held out examples can become your training data.

Note that if your filter is blocking 95% of the negative examples or more, this approach becomes less viable. Even so, if you wish to measure serving performance, you can make an even tinier sample (say 0.1% or 0.001%). Ten thousand examples is enough to estimate performance quite accurately.

Rule #35: Beware of the inherent skew in ranking problems.

When you switch your ranking algorithm radically enough that different results show up, you have effectively changed the data that your algorithm is going to see in the future. This kind of skew will show up, and you should design your model around it. There are multiple different approaches. These approaches are all ways to favor data that your model has already seen.

1. Have higher regularization on features that cover more queries as opposed to those features that are on for only one query. This way, the model will favor features that are specific to one or a few queries over features that generalize to all queries. This approach can help prevent very popular results from leaking into irrelevant queries. Note that this is opposite the more conventional advice of having more regularization on feature columns with more unique values.
2. Only allow features to have positive weights. Thus, any good feature will be better than a feature that is "unknown".
3. Don’t have document-only features. This is an extreme version of #1. For example, even if a given app is a popular download regardless of what the query was, you don’t want to show it everywhere. Not having document-only features keeps that simple. The reason you don’t want to show a specific popular app everywhere has to do with the importance of making all the desired apps reachable. For instance, if someone searches for "bird watching app", they might download "angry birds", but that certainly wasn’t their intent. Showing such an app might improve download rate, but leave the user’s needs ultimately unsatisfied.

Rule #36: Avoid feedback loops with positional features.

The position of content dramatically affects how likely the user is to interact with it. If you put an app in the first position it will be clicked more often, and you will be convinced it is more likely to be clicked. One way to deal with this is to add positional features, i.e. features about the position of the content in the page. You train your model with positional features, and it learns to weight, for example, the feature "1stposition" heavily. Your model thus gives less weight to other factors for examples with "1stposition=true". Then at serving you don't give any instances the positional feature, or you give them all the same default feature,

because you are scoring candidates before you have decided the order in which to display them.

Note that it is important to keep any positional features somewhat separate from the rest of the model because of this asymmetry between training and testing. Having the model be the sum of a function of the positional features and a function of the rest of the features is ideal. For example, don't cross the positional features with any document feature.

Rule #37: Measure Training/Serving Skew.

There are several things that can cause skew in the most general sense. Moreover, you can divide it into several parts:

- The difference between the performance on the training data and the holdout data. In general, this will always exist, and it is not always bad.
- The difference between the performance on the holdout data and the "nextday" data. Again, this will always exist. You should tune your regularization to maximize the next-day performance. However, large drops in performance between holdout and next-day data may indicate that some features are time-sensitive and possibly degrading model performance.
- The difference between the performance on the "next-day" data and the live data. If you apply a model to an example in the training data and the same example at serving, it should give you exactly the same result (see Rule #5 (https://developers.google.com/machine-learning/guides/rules-of-ml/rule_5_test_the_infrastructure_independently_from_the_machine_learning)). Thus, a discrepancy here probably indicates an engineering error.

ML Phase III: Slowed Growth, Optimization Refinement, and Complex Models

There will be certain indications that the second phase is reaching a close. First of all, your monthly gains will start to diminish. You will start to have tradeoffs between metrics: you will see some rise and others fall in some experiments. This is where it gets interesting. Since the gains are harder to achieve, the machine learning has to get more sophisticated. A caveat: this section has more blue-sky rules than earlier sections. We have seen many teams go through the happy times of Phase I and Phase II machine learning. Once Phase III has been reached, teams have to find their own path.

Rule #38: Don't waste time on new features if unaligned objectives have become the issue.

As your measurements plateau, your team will start to look at issues that are outside the scope of the objectives of your current machine learning system. As stated before, if the product goals are not covered by the existing algorithmic objective, you need to change either your objective or your product goals. For instance, you may optimize clicks, plus-ones, or downloads, but make launch decisions based in part on human raters.

Rule #39: Launch decisions are a proxy for long-term product goals.

Alice has an idea about reducing the logistic loss of predicting installs. She adds a feature. The logistic loss drops. When she does a live experiment, she sees the install rate increase. However, when she goes to a launch review meeting, someone points out that the number of daily active users drops by 5%. The team decides not to launch the model. Alice is disappointed, but now realizes that launch decisions depend on multiple criteria, only some of which can be directly optimized using ML.

The truth is that the real world is not dungeons and dragons: there are no "hit points" identifying the health of your product. The team has to use the statistics it gathers to try to effectively predict how good the system will be in the future. They need to care about engagement, 1 day active users (DAU), 30 DAU, revenue, and advertiser's return on investment. These metrics that are measureable in A/B tests in themselves are only a proxy for more longterm goals: satisfying users, increasing users, satisfying partners, and profit, which even then you could consider proxies for having a useful, high quality product and a thriving company five years from now.

The only easy launch decisions are when all metrics get better (or at least do not get worse). If the team has a choice between a sophisticated machine learning algorithm, and a simple heuristic, if the simple heuristic does a better job on all these metrics, it should choose the heuristic. Moreover, there is no explicit ranking of all possible metric values. Specifically, consider the following two scenarios:

Experiment	Daily Active Users	Revenue/Day
A	1 million	$4 million
B	2 million	$2 million

If the current system is A, then the team would be unlikely to switch to B. If the current system is B, then the team would be unlikely to switch to A. This seems in conflict with rational behavior; however, predictions of changing metrics may or may not pan out, and thus there is a large risk involved with either change. Each metric covers some risk with which the team is concerned.

Moreover, no metric covers the team's ultimate concern, "where is my product going to be five years from now"?

Individuals, on the other hand, tend to favor one objective that they can directly optimize. Most machine learning tools favor such an environment. An engineer banging out new features can get a steady stream of launches in such an environment. There is a type of machine learning, multi-objective learning, which starts to address this problem. For instance, one can formulate a constraint satisfaction problem that has lower bounds on each metric, and optimizes some linear combination of metrics. However, even then, not all metrics are easily framed as machine learning objectives: if a document is clicked on or an app is installed, it is because that the content was shown. But it is far harder to figure out why a user visits your site. How to predict the future success of a site as a whole is AI-complete (https://wikipedia.org/wiki/AI-complete): as hard as computer vision or natural language processing.

Rule #40: Keep ensembles simple.

Unified models that take in raw features and directly rank content are the easiest models to debug and understand. However, an ensemble of models (a "model" which combines the scores of other models) can work better. **To keep things simple, each model should either be an ensemble only taking the input of other models, or a base model taking many features, but not both.** If you have models on top of other models that are trained separately, then combining them can result in bad behavior.

Use a simple model for ensembling that takes only the output of your "base" models as inputs. You also want to enforce properties on these ensemble models. For example, an increase in the score produced by a base model should not decrease the score of the ensemble. Also, it is best if the incoming models are semantically interpretable (for example, calibrated) so that changes of the underlying models do not confuse the ensemble model. Also, **enforce that an increase in the predicted probability of an underlying classifier does not decrease the predicted probability of the ensemble**.

Rule #41: When performance plateaus, look for qualitatively new sources of information to add rather than refining existing signals.

You've added some demographic information about the user. You've added some information about the words in the document. You have gone through template exploration, and tuned the regularization. You haven't seen a launch with more than a 1% improvement in your key metrics in a few quarters. Now what?

It is time to start building the infrastructure for radically different features, such as the history of documents that this user has accessed in the last day, week, or year, or data from a different property. Use wikidata (https://wikipedia.org/wiki/Wikidata) entities or something internal to your company (such as Google's knowledge graph (https://wikipedia.org/wiki/Knowledge_Graph)). Use deep learning. Start to adjust your expectations on how much return you expect on investment, and expand your efforts accordingly. As in any engineering project, you have to weigh the benefit of adding new features against the cost of increased complexity.

Rule #42: Don't expect diversity, personalization, or relevance to be as correlated with popularity as you think they are.

Diversity in a set of content can mean many things, with the diversity of the source of the content being one of the most common. Personalization implies each user gets their own results. Relevance implies that the results for a particular query are more appropriate for that query than any other. Thus all three of these properties are defined as being different from the ordinary.

The problem is that the ordinary tends to be hard to beat.

Note that if your system is measuring clicks, time spent, watches, +1s, reshares, et cetera, you are measuring the **popularity** of the content. Teams sometimes try to learn a personal model with diversity. To personalize, they add features that would allow the system to personalize (some features representing the user's interest) or diversify (features indicating if this document has any features in common with other documents returned, such as author or content), and find that those features get less weight (or sometimes a different sign) than they expect.

This doesn't mean that diversity, personalization, or relevance aren't valuable. As pointed out in the previous rule, you can do postprocessing to increase diversity or relevance. If you see longer term objectives increase, then you can declare that diversity/relevance is valuable, aside from popularity. You can then either continue to use your postprocessing, or directly modify the objective based upon diversity or relevance.

Rule #43: Your friends tend to be the same across different products. Your interests tend not to be.

Teams at Google have gotten a lot of traction from taking a model predicting the closeness of a connection in one product, and having it work well on another. Your friends are who they are. On the other hand, I have watched several teams struggle with personalization features across product divides. Yes, it seems like it should work. For now, it doesn't seem

like it does. What has sometimes worked is using raw data from one property to predict behavior on another. Also, keep in mind that even knowing that a user has a history on another property can help. For instance, the presence of user activity on two products may be indicative in and of itself.

Related Work

There are many documents on machine learning at Google as well as externally.

- Machine Learning Crash Course (https://developers.google.com/machine-learning/crash-course/): an introduction to applied machine learning.
- Machine Learning: A Probabilistic Approach (https://www.cs.ubc.ca/%7Emurphyk/MLbook/) by Kevin Murphy for an understanding of the field of machine learning.
- Practical Advice for the Analysis of Large, Complex Data Sets (http://www.unofficialgoogledatascience.com/2016/10/practical-advice-for-analysis-of-large.html)
  : a data science approach to thinking about data sets.
- Deep Learning (http://www.iro.umontreal.ca/%7Ebengioy/dlbook/) by Ian Goodfellow et al for learning nonlinear models.
- Google paper on technical debt (http://research.google.com/pubs/pub43146.html), which has a lot of general advice.
- Tensorflow Documentation (https://www.tensorflow.org/).

Acknowledgements

Thanks to David Westbrook, Peter Brandt, Samuel Ieong, Chenyu Zhao, Li Wei, Michalis Potamias, Evan Rosen, Barry Rosenberg, Christine Robson, James Pine, Tal Shaked, Tushar Chandra, Mustafa Ispir, Jeremiah Harmsen, Konstantinos Katsiapis, Glen Anderson, Dan Duckworth, Shishir Birmiwal, Gal Elidan, Su Lin Wu, Jaihui Liu, Fernando Pereira, and Hrishikesh Aradhye for many corrections, suggestions, and helpful examples for this document. Also, thanks to Kristen Lefevre, Suddha Basu, and Chris Berg who helped with an earlier version. Any errors, omissions, or (gasp!) unpopular opinions are my own.

Appendix

There are a variety of references to Google products in this document. To provide more context, I give a short description of the most common examples below.

YouTube Overview

YouTube is a streaming video service. Both YouTube Watch Next and YouTube Home Page teams use ML models to rank video recommendations. Watch Next recommends videos to watch after the currently playing one, while Home Page recommends videos to users browsing the home page.

Google Play Overview

Google Play has many models solving a variety of problems. Play Search, Play Home Page Personalized Recommendations, and 'Users Also Installed' apps all use machine learning.

Google Plus Overview

Google Plus uses machine learning in a variety of situations: ranking posts in the "stream" of posts being seen by the user, ranking "What's Hot" posts (posts that are very popular now), ranking people you know, et cetera.

Last updated May 16, 2019.