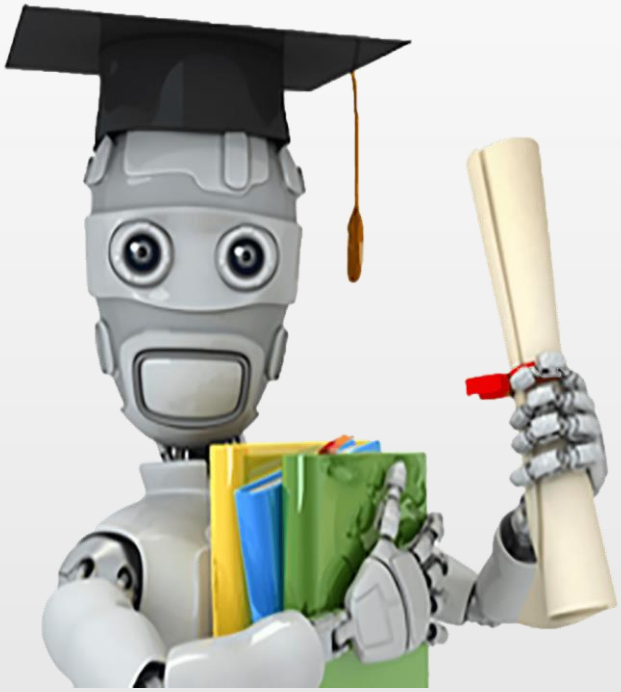
Welcome to the Google Machine Learning Crash Course!



Machine Learning Crash Course and Workshop

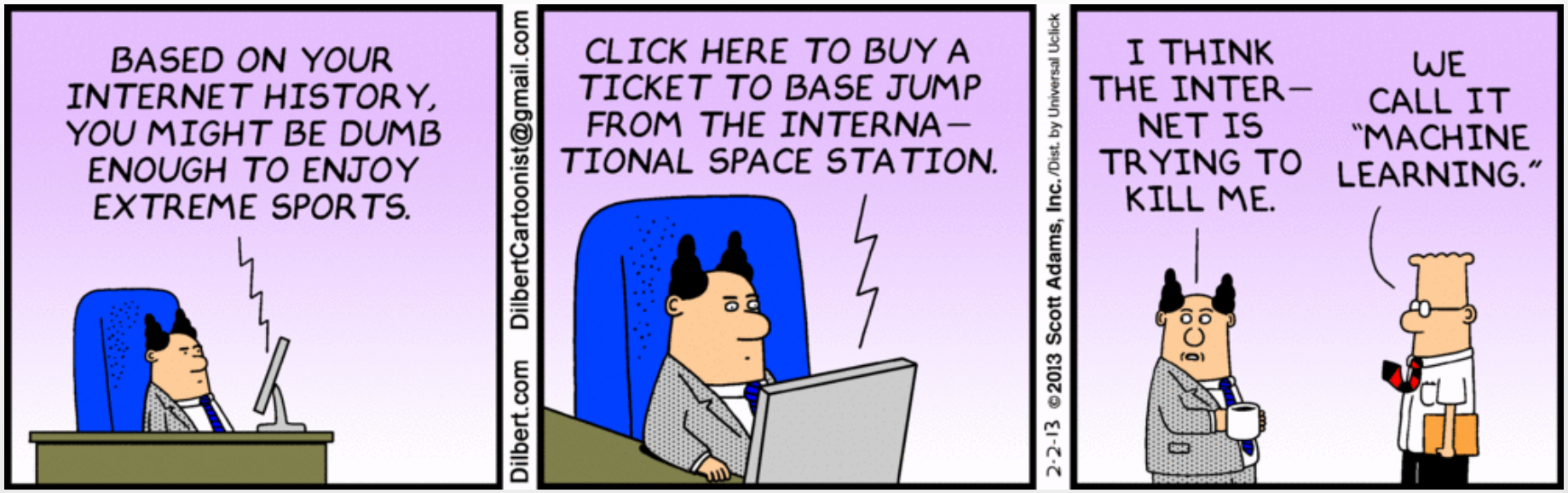
Charmi Chokshi | Facilitator
Maitrey Mehta | Facilitator
Mayank Jobanputra | Facilitator

Google™

What is Learning?

A closer look at how humans learn.

What is Machine Learning?



What is Machine Learning?

Learn from experience

Learn from ~~experience~~ data

Follow instructions

The Formal Definition

Machine Learning:

As given by Tom Mitchell, A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

In other words, it is the study of computer algorithms that improve automatically through experience.

Why Machine Learning?

The practicality of Machine Learning:

- Reduces Programming Time
- Customize and Scale Products
- Complete Seemingly ‘unprogrammable’ tasks

“A breakthrough in Machine Learning would be worth ten Microsofts.” – Bill Gates

“Machine Learning is the next Internet.” – Tony Tether, Director, DARPA

“Machine learning is today’s discontinuity” - Jerry Yang, former CEO, Yahoo

“Machine learning is the hot new thing” – John Hennessy, President, Stanford

Our Pasts

Aristotle invented syllogistic logic.

15th Century

1956 — John McCarthy coined the term “Artificial Intelligence"

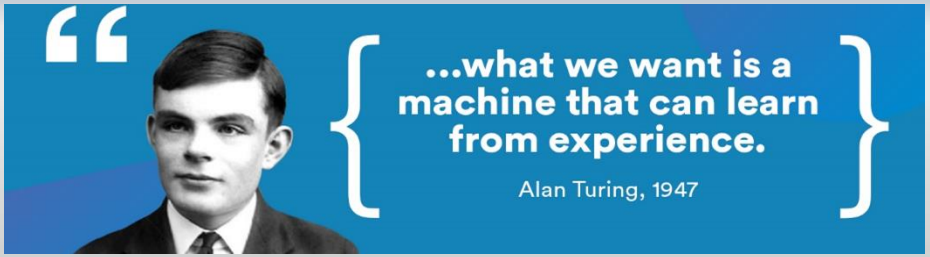
1959 — Arthur Samuel wrote the first game-playing program and coined the term “Machine Learning”

1979 — The Stanford Cart becomes the first computer-controlled, autonomous vehicle

1997 — Deep Blue beats a reigning world chess champion, Gary Kasparov

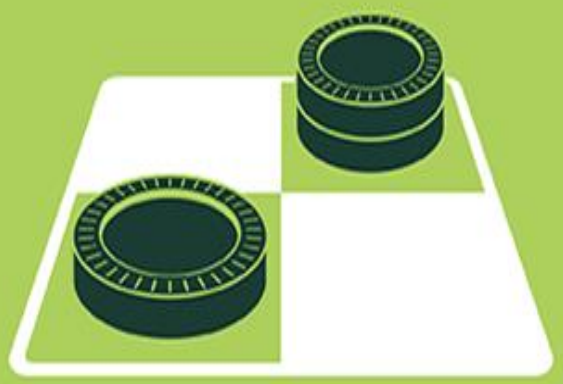
2011 — IBM's Watson beats two human champions in a Jeopardy! competition.

2011 — Google's AlphaGo program becomes the first Computer Go program to a professional human player



ARTIFICIAL INTELLIGENCE

Early artificial intelligence stirs excitement.



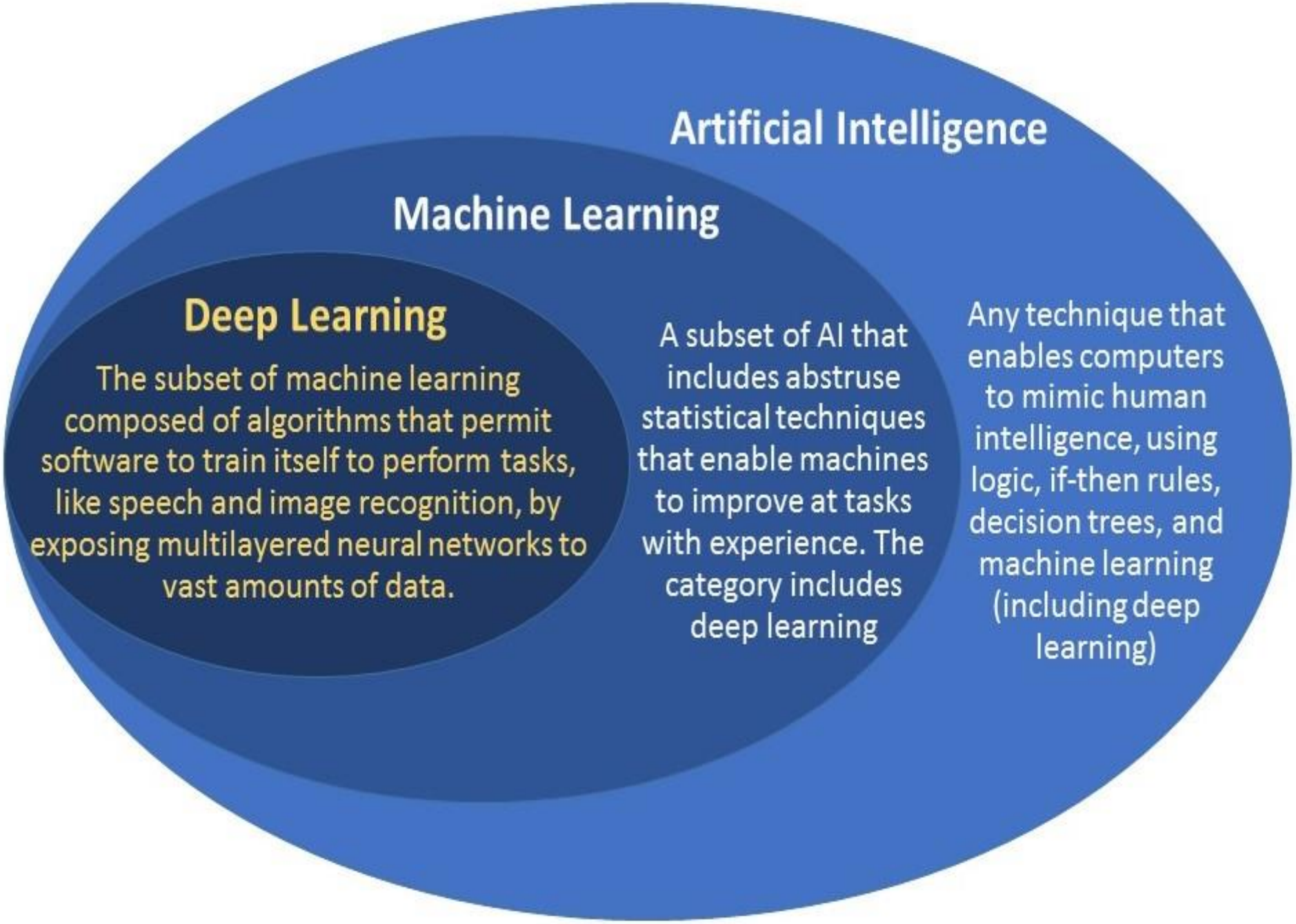
MACHINE LEARNING

Machine learning begins to flourish.

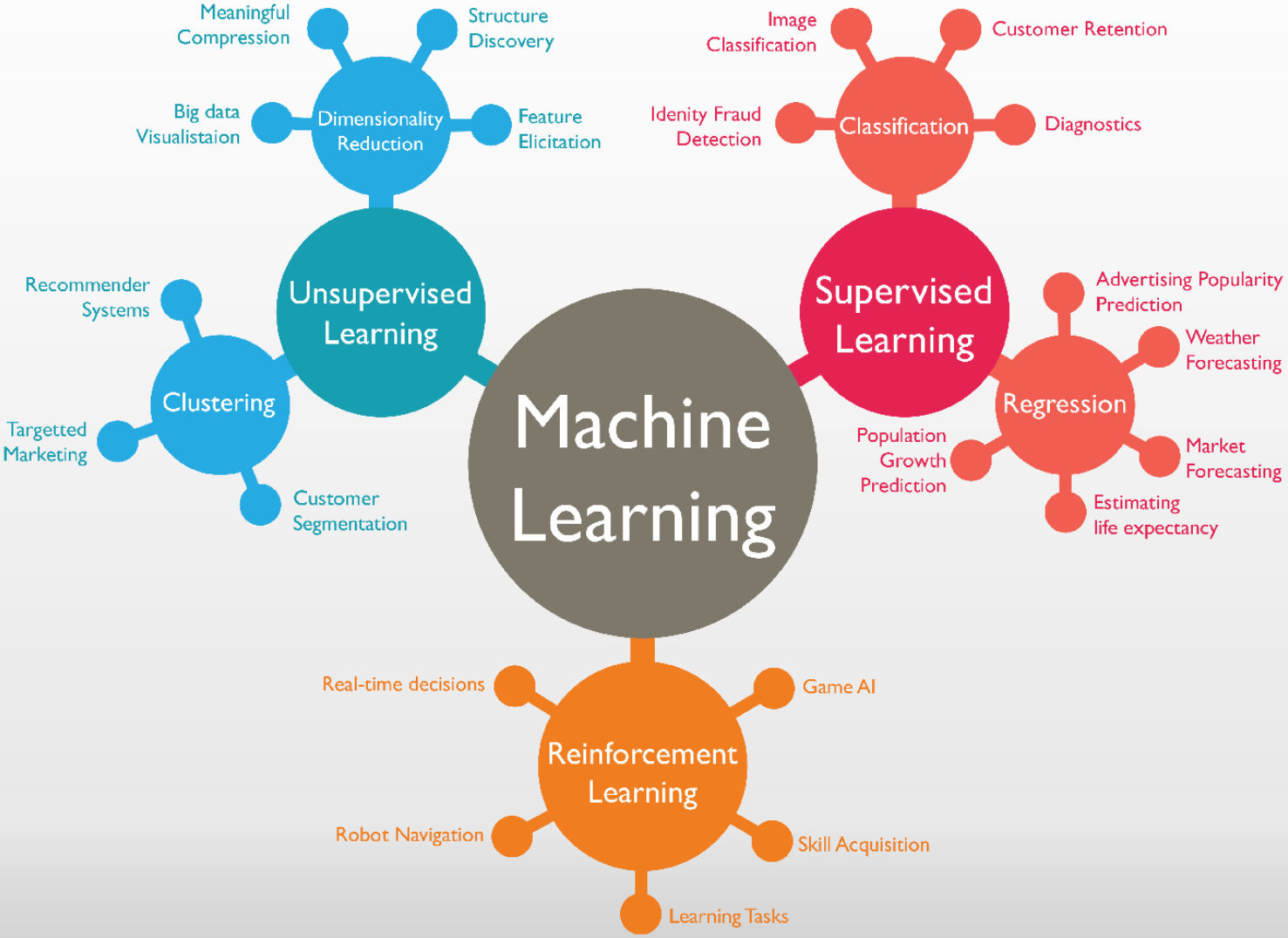
DEEP LEARNING

Deep learning breakthroughs drive AI boom.

1950's 1960's 1970's 1980's 1990's 2000's 2010's

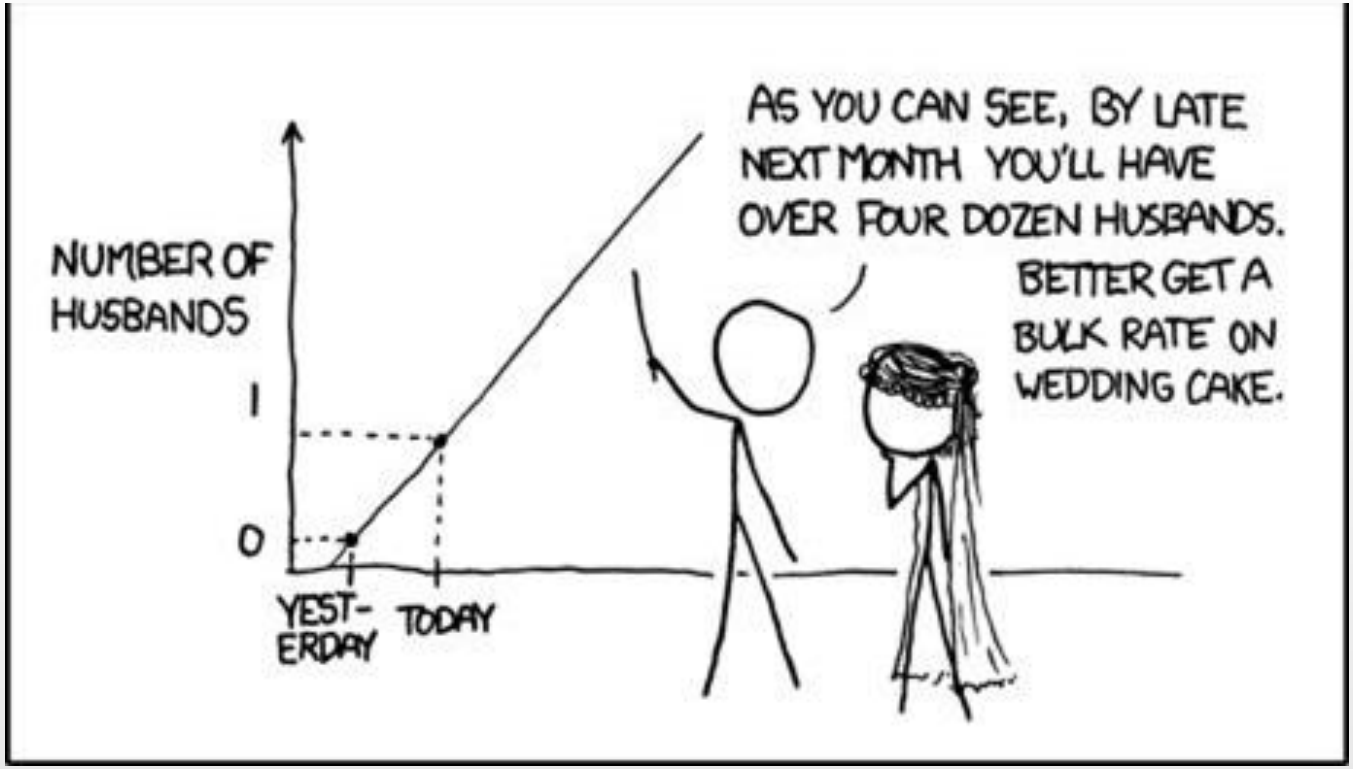

Artificial Intelligence
Any technique that enables computers to mimic human intelligence, using logic, if-then rules, decision trees, and machine learning (including deep learning)
Machine Learning
A subset of AI that includes abstruse statistical techniques that enable machines to improve at tasks with experience. The category includes deep learning
Deep Learning
The subset of machine learning composed of algorithms that permit software to train itself to perform tasks, like speech and image recognition, by exposing multilayered neural networks to vast amounts of data.

The Machine Learning Domain



Supervised Learning

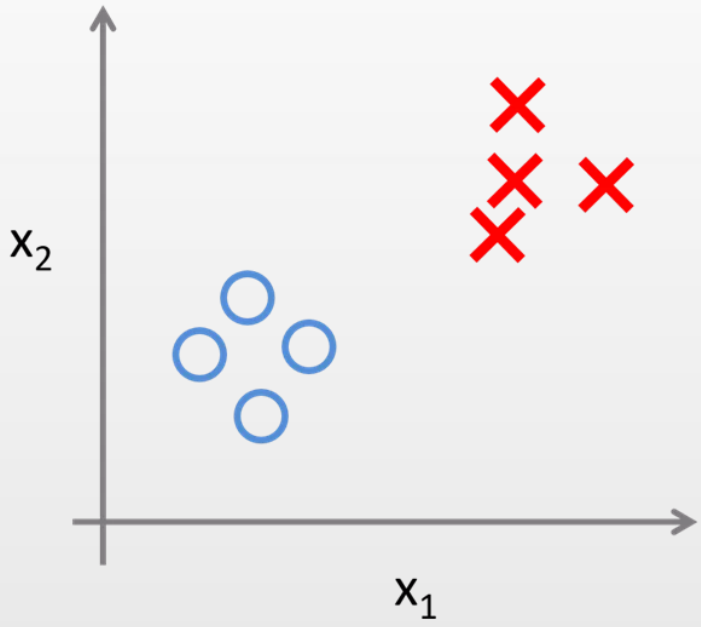
- Supervised Learning deals with prediction of values based on given combinations of data values given beforehand.
- ML systems learn how to combine input to produce useful predictions on never-before-seen data
- It is like learning with a teacher.
- Types - Regression, Classification


NUMBER OF HUSBANDS
1
0
YEST-ERDAY
TODAY
AS YOU CAN SEE, BY LATE NEXT MONTH YOU'LL HAVE OVER FOUR DOZEN HUSBANDS.
BETTER GET A BULK RATE ON WEDDING CAKE.

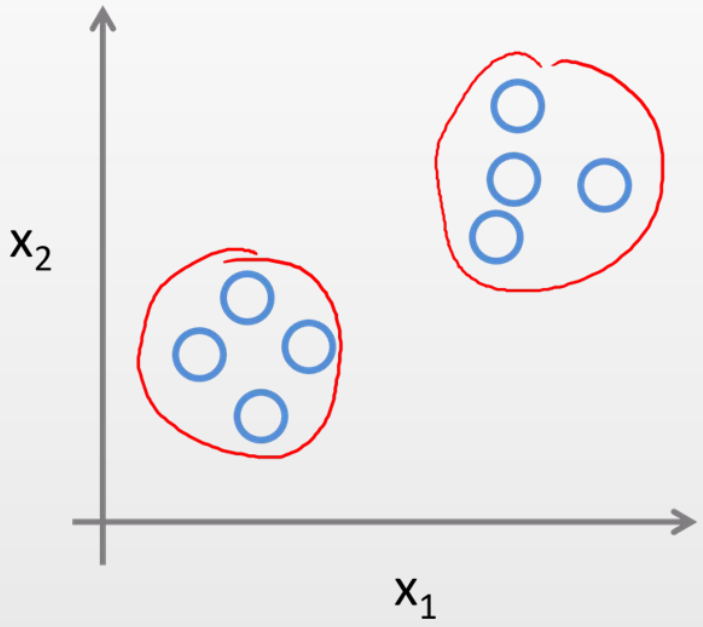
Unsupervised Learning

- Unsupervised Learning deals with clustering values or forming groups of values.
- One aims to infer patterns from the data rather than predicting values.
- It is like learning on your own.
- Types - Clustering, Dimensionality Reduction

Supervised Learning

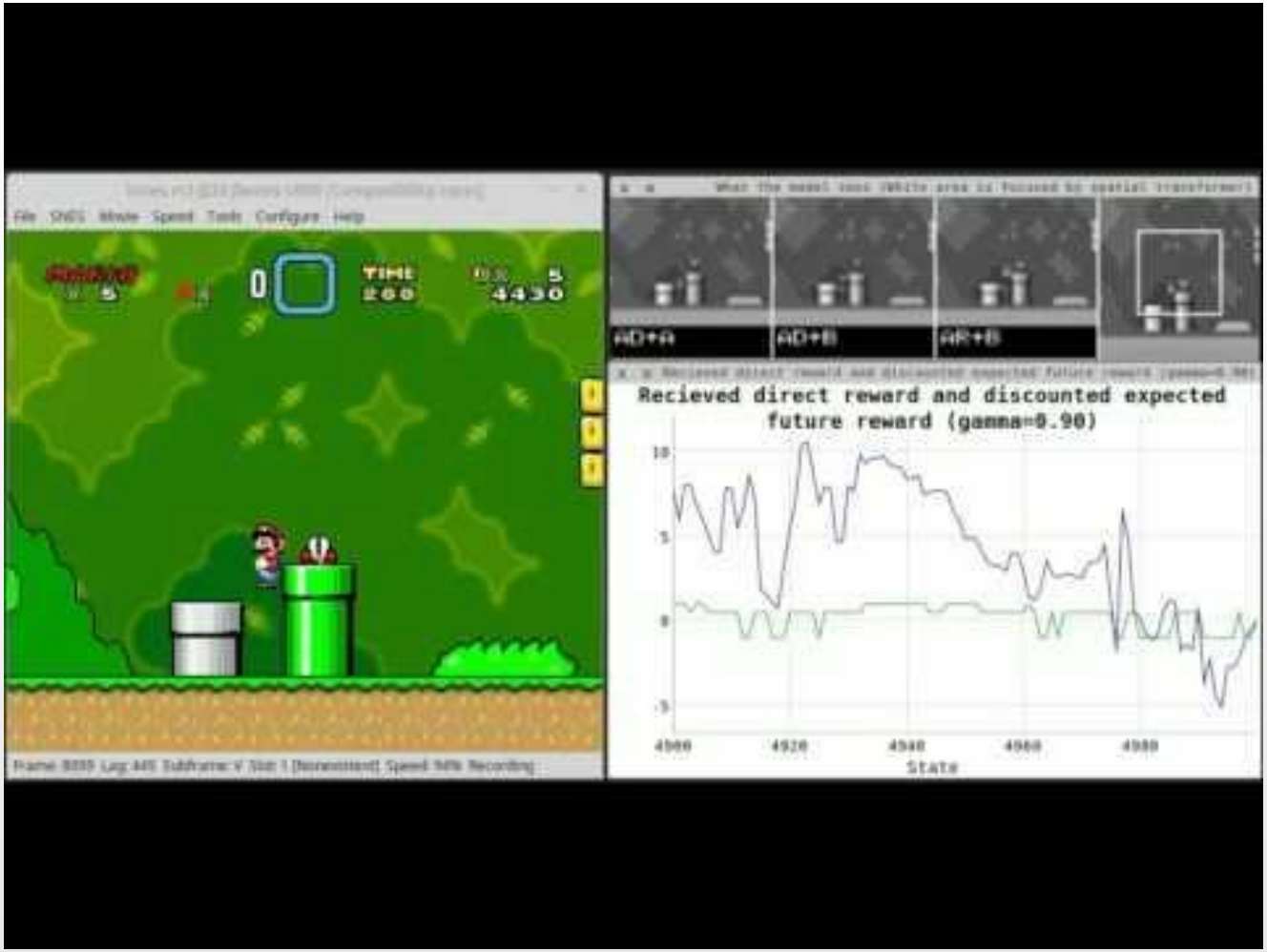


Unsupervised Learning



Reinforcement Learning

- It is a reward based training approach in which the model interacts with a dynamic environment and in turn collects rewards according to the action chosen.
- Widely used in automating games.
- Example- Shortest path finder


File SNES Movie Speed Tools Configure Help
TIME
4430
Recieved direct reward and discounted expected future reward (gamma=0.90)
10
5
0
-5
4900
4920
4940
4960
4980
State

Basic Terminologies

Features:

A **feature** is an input variable. A simple machine learning project might use a single feature, while a more sophisticated machine learning project could use millions of features, specified as {x1,x2,...xN}.

What are the Features in spam detector example?

words in the email text

sender's address

time of day the email was sent

email contains the phrase "one weird trick."

Labels:

A **label** is the thing we're predicting denoted by y. The label could be the future price of wheat, the kind of animal shown in a picture etc.

Basic Terminologies

Examples:

An example is a particular instance of data, x. It can be of two types-

- **Labelled** Example: In this case label y for corresponding x is given alongside x.
- **Unlabelled** Example: In this case only features x are given, label y is missing

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/ Testing** means applying the trained model to unlabeled examples. That is, you use the trained model to make useful predictions.

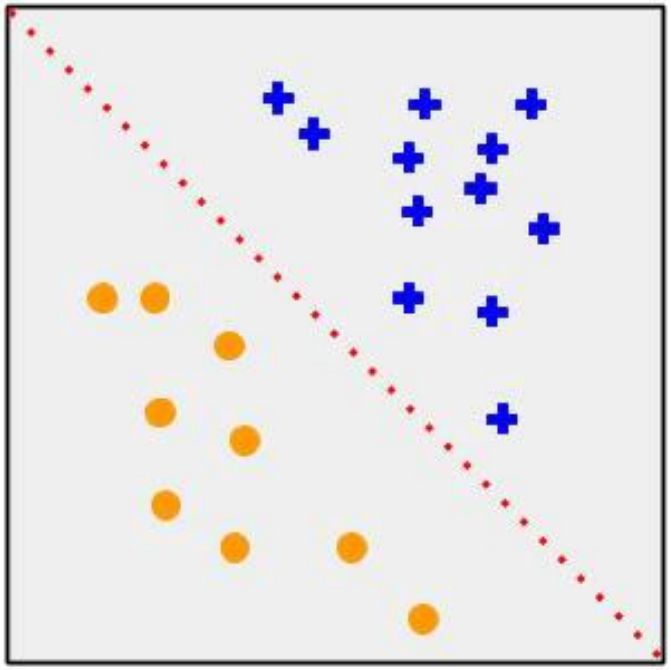
Regression and Classification

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

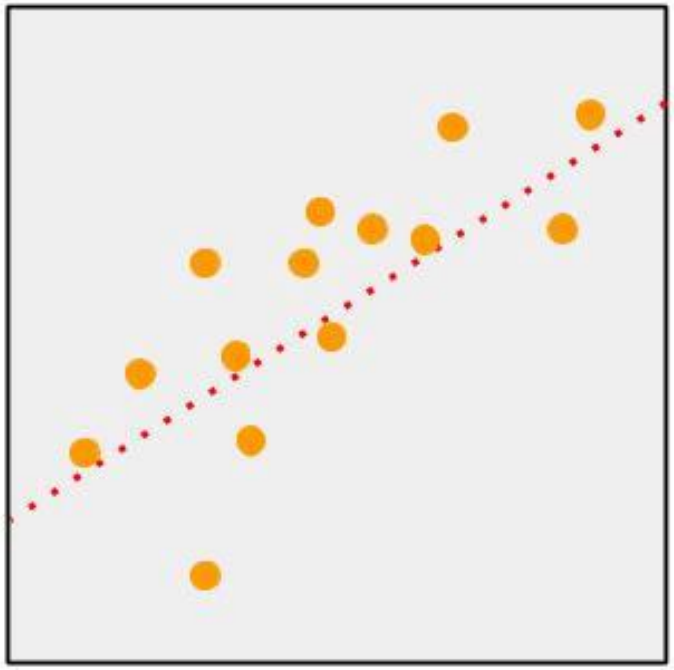
- What is the value of a house in California?
- What is the probability that a user will click on this ad?

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?



Classification



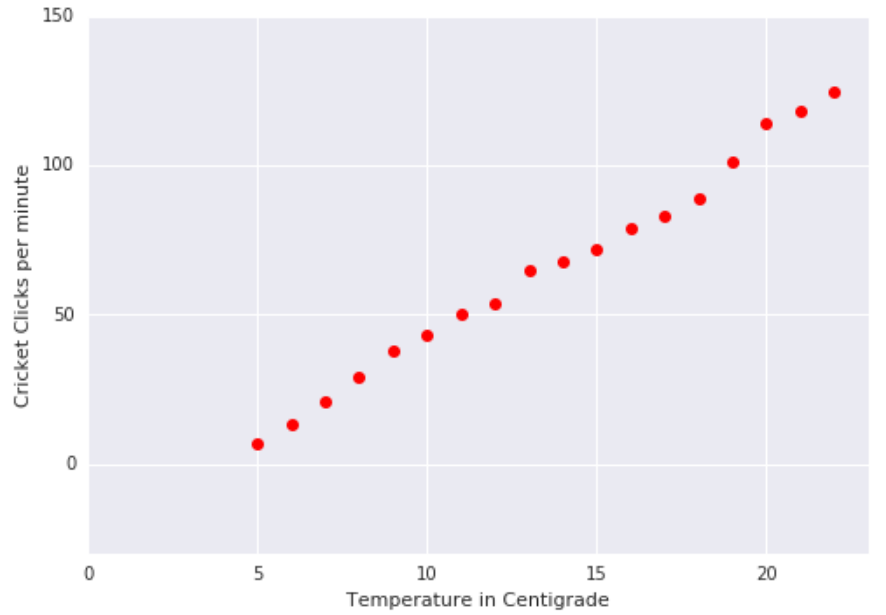
Regression


Course Introduction

Linear Regression

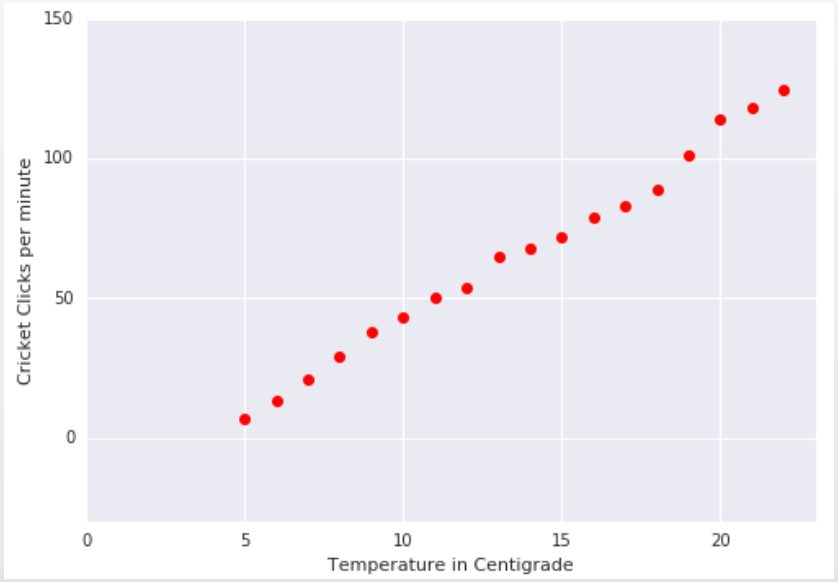
We start with an example, It has long been known that crickets chirp more frequently on hotter days than on cooler days. For decades, professional and amateur entomologists have cataloged data on chirps-per-minute and temperature.

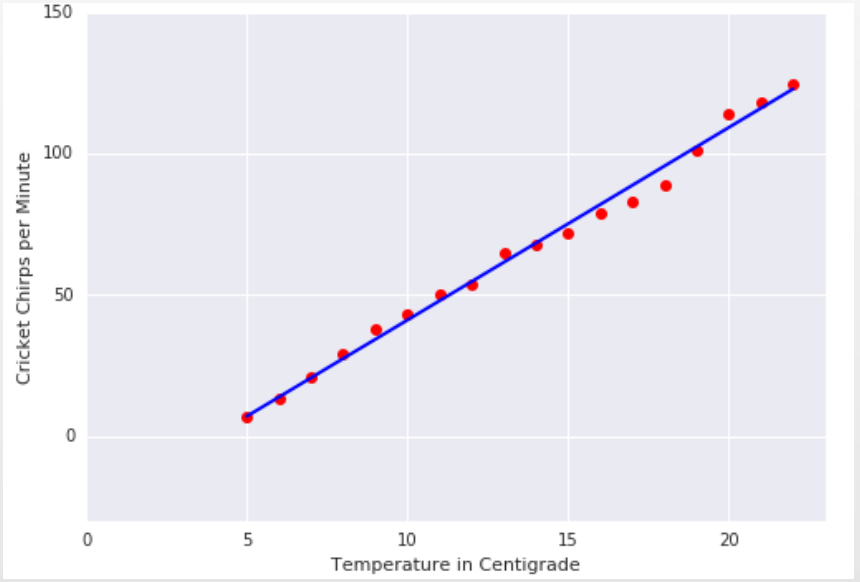
A nice first step is to examine your data by plotting it-



Linear Regression

The plot shows the number of chirps rising with the temperature. We see that the relationship between chirps and temperature looks ‘almost’ linear. So, we draw a straight line to approximate this relationship.



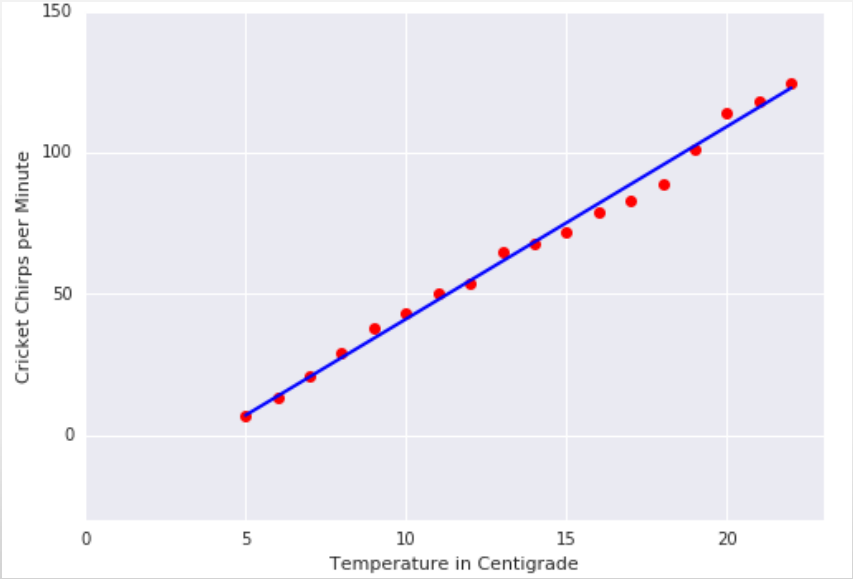


Linear Regression

Note that the line doesn't pass perfectly through every dot. However, the line clearly shows the relationship between chirps and the temperature. We can describe the line as:

$$y = mx + c$$

where y - number of chirps/minute
m - slope of the line
x - Temperature
b - y-intercept



Linear Regression

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

$$y' = b + w_1x_1$$

where:

y' is predicted label (a desired output).
b is the bias (the y-intercept). Also referred to as w_0.
w_1 is the weight of feature x_1
x_1 is a feature (a known input).

To predict the number of chirps per minute y' on a new value of temperature x_1, just plug the new value of x_1 into this model.

Multiple Linear Regression: Contains multiple features and weights the equation would be:

$$y' = b + w_1x_1 + w_2x_2 + w_3x_3$$

Price of a house

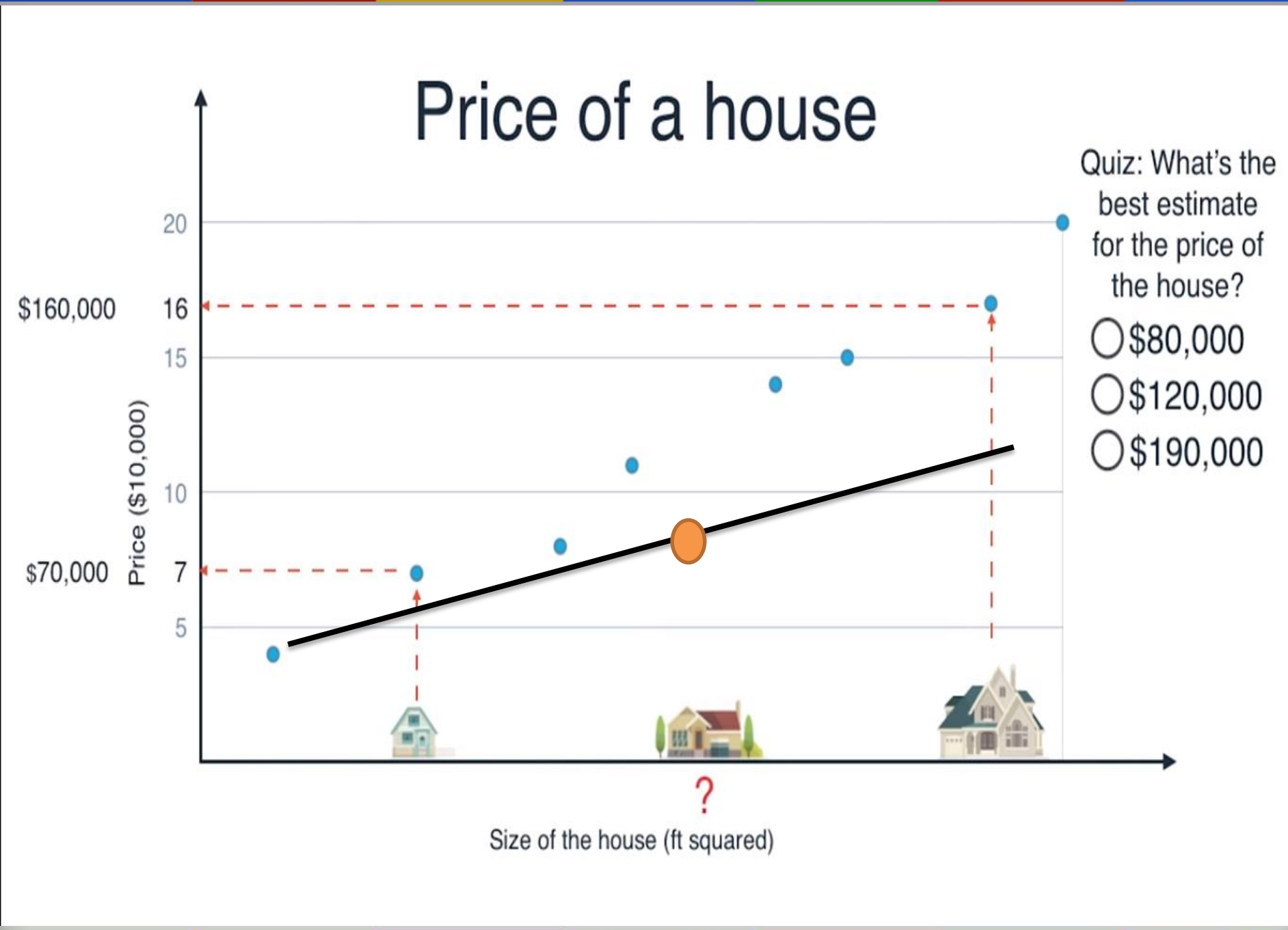
$70,000

?

$160,000


Price of a house
20
15
10
7
5
$70,000
Price ($10,000)
?
Size of the house (ft squared)

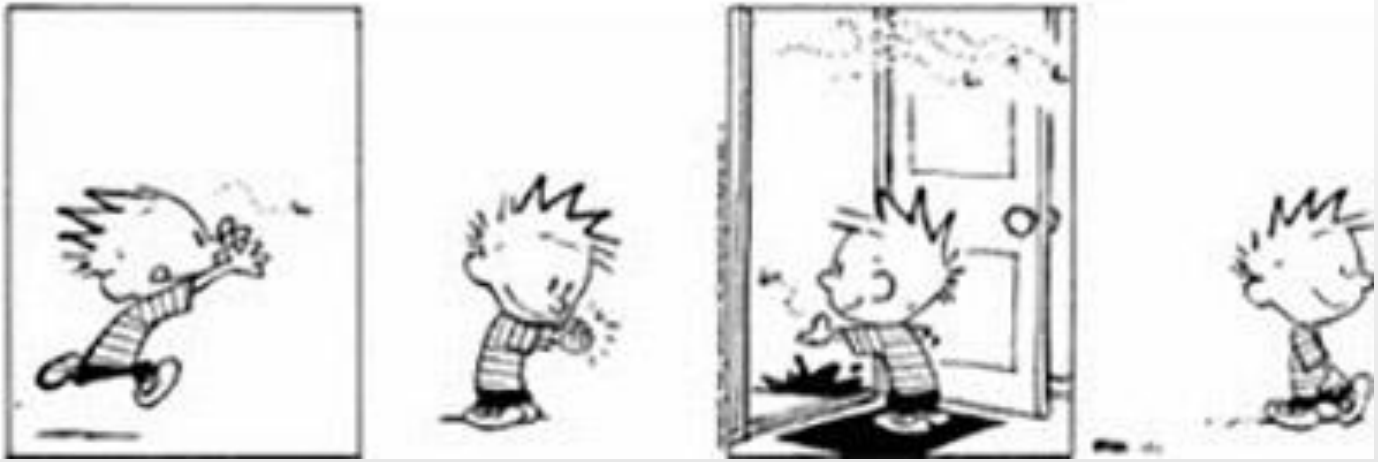

Price of a house
Quiz: What's the best estimate for the price of the house?
$80,000
$120,000
$190,000
20
16
15
10
7
5
$160,000
$70,000
Price ($10,000)
?
Size of the house (ft squared)


Price of a house
Quiz: What's the best estimate for the price of the house?
$80,000
$120,000
$190,000
20
16
15
10
7
5
$160,000
$70,000
Price ($10,000)
?
Size of the house (ft squared)

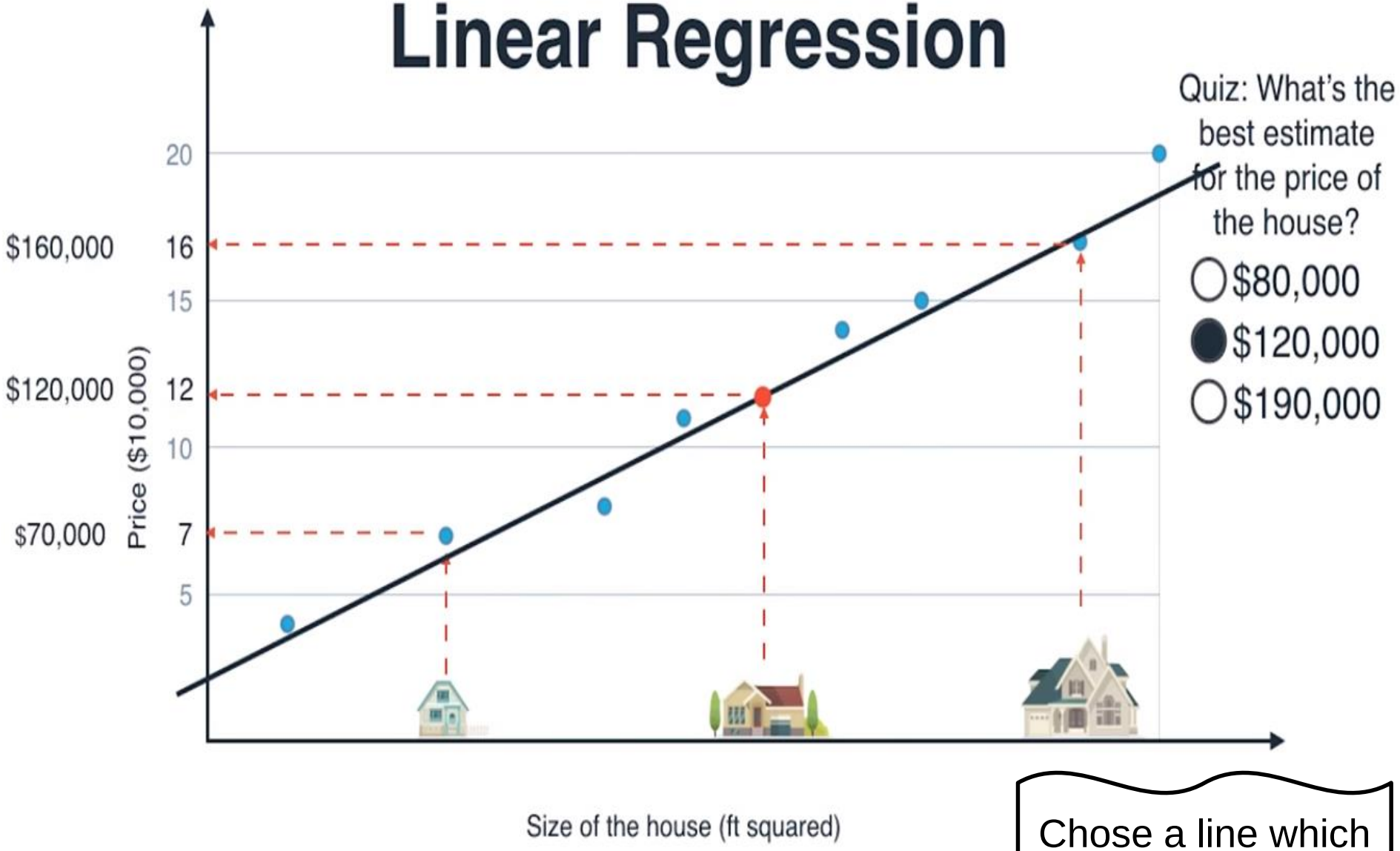

Price of a house
Quiz: What's the best estimate for the price of the house?
$80,000
$120,000
$190,000
20
16
15
10
7
5
$160,000
$70,000
Price ($10,000)
?
Size of the house (ft squared)


Price of a house
Quiz: What's the best estimate for the price of the house?
$80,000
$120,000
$190,000
20
16
15
10
7
5
$160,000
$70,000
Price ($10,000)
?
Size of the house (ft squared)

Regression:
"when you fix one bug, you introduce several newer bugs."



Linear Regression



Chose a line which best fits the data

Training a Model

We see from the equation of the linear model $y' = b + w_1x_1$ that we would just be given x's and y's. However, w_1 and b would have to be determined.

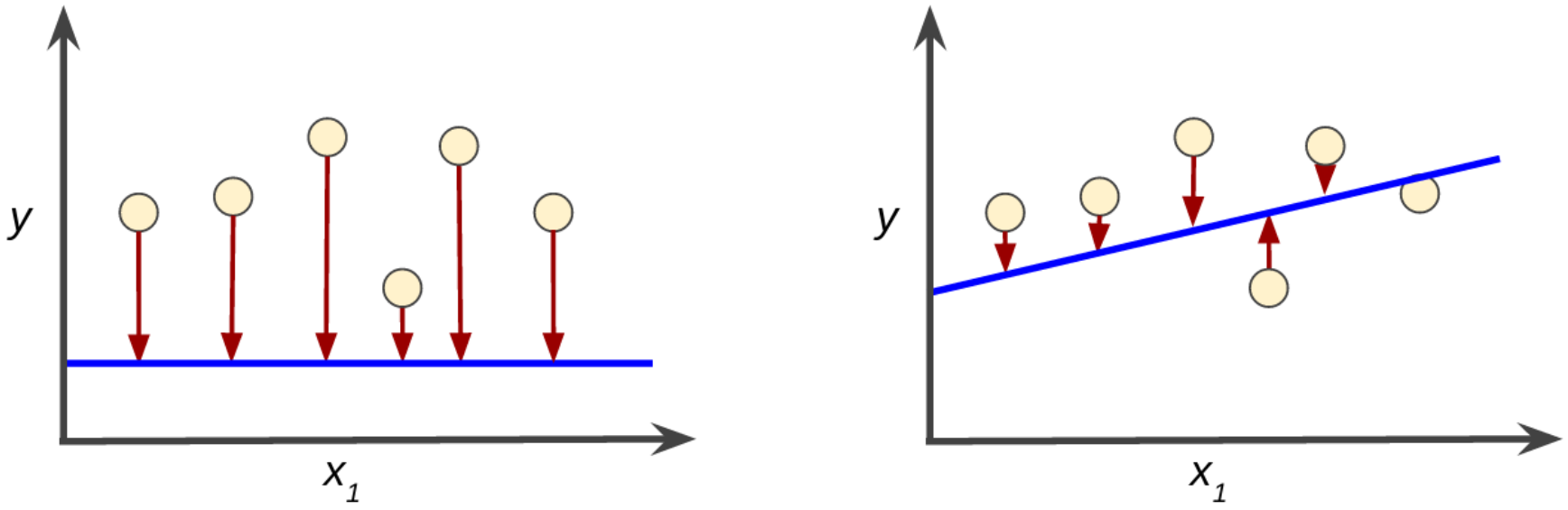
Training a model simply means **learning good values for all the weights and the bias from labeled examples**. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that **minimizes loss**; this process is called **empirical risk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


Tomorrow's forecast calls for cloudy skies and light rain throughout the day
Aw shucks.
HUDLOW RD

High Loss vs Low Loss Model



The blue line is the linear model followed while the red arrows denote the loss.

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

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

= the square of the difference between the label and the prediction
= (observation - prediction(**x**))^2
= (y - y')^2

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

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

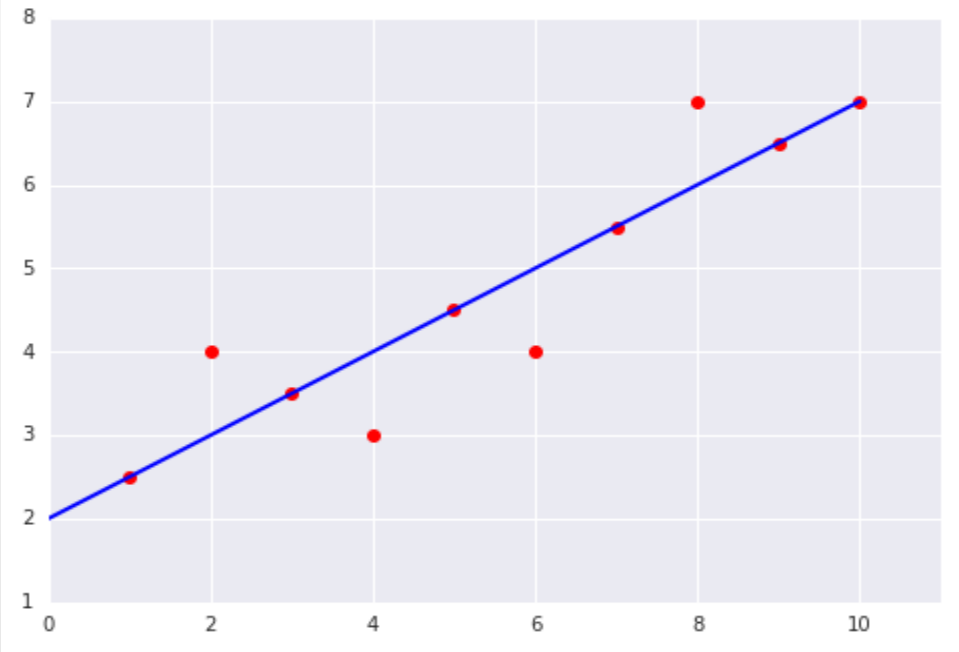
Mean Squared Error

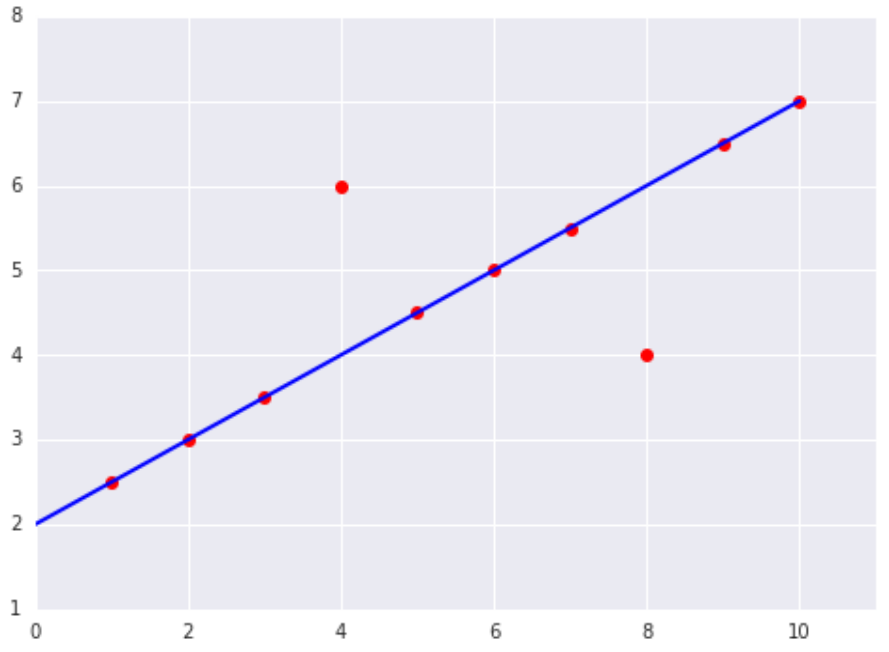
$$MSE = \frac{1}{N} \sum_{(x,y) \in D} (y - prediction(x))^2$$

where:

- *x,y* is an example in which
  - *x* is the set of features (for example, temperature, age etc) that the model uses to make predictions.
  - *y* is the example's label (for example, chirps/minute).
- *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.





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

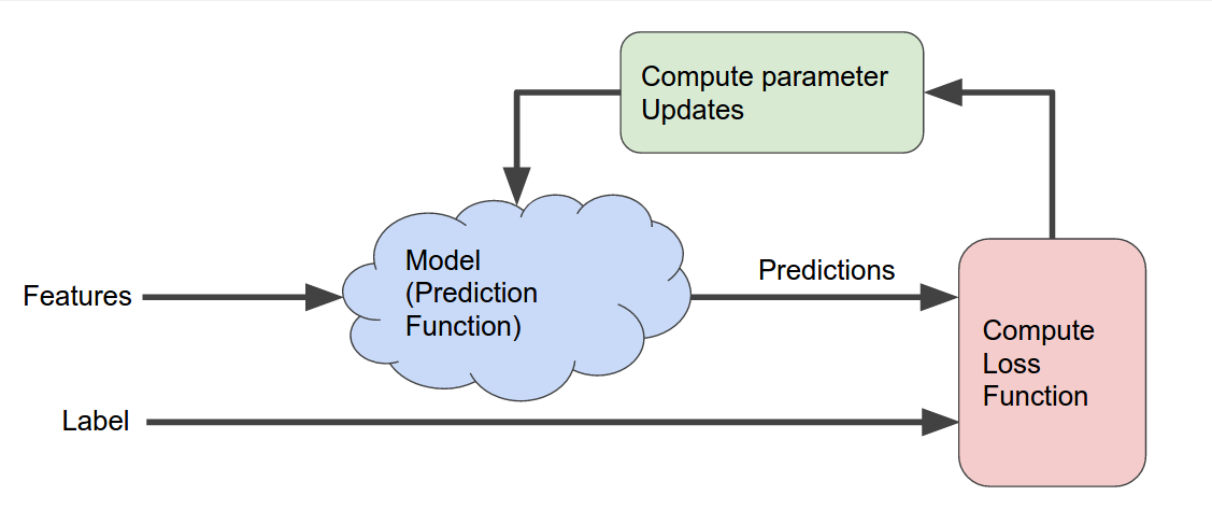
☐ Left ☑ Right

The Game of Hot and Cold

Iterative Learning

Iterative learning is like the "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. 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:



Steps for Reducing Loss

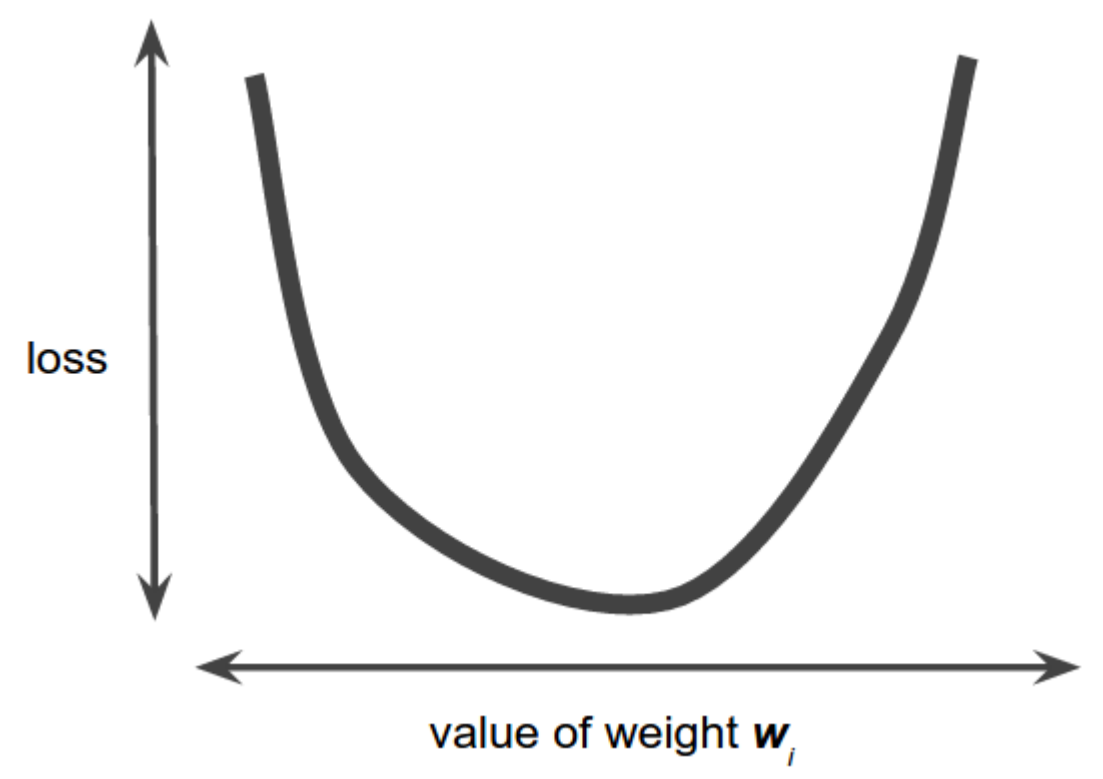
We have two unknowns b and w_1.

1. We initialize b and w_1 with random values. Initializing with 0 would also be a good choice.
2. We will calculate the prediction with theses values by plugging in values of x.
3. Loss is then calculated and new values of b and w_1 . For now, just assume that the mysterious green 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**.

Opening the Green Box

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

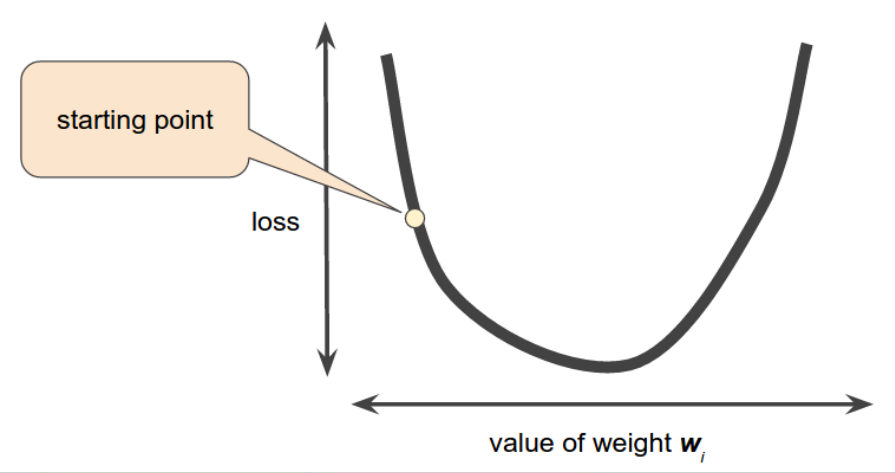


Gradient Descent

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

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

The first stage in gradient descent is to pick a starting value (a starting point) for w1. The starting point doesn't matter much; therefore, many algorithms simply set w1 to 0 or pick a random value.

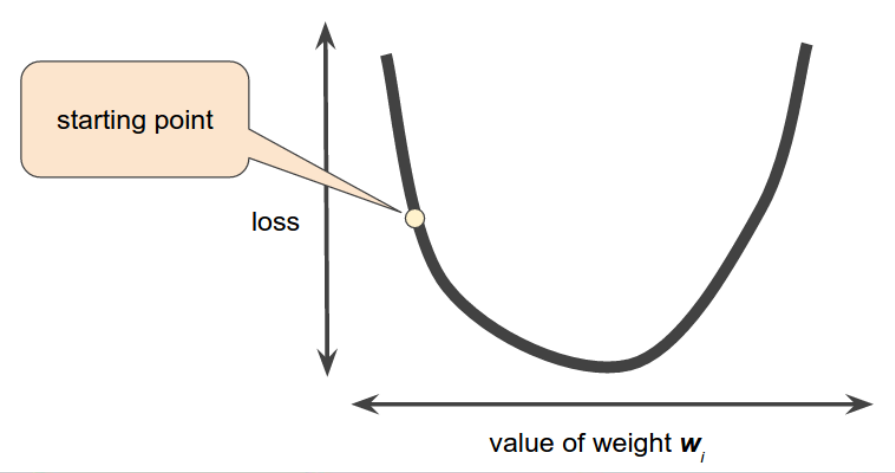


Gradient Descent

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

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

The first stage in gradient descent is to pick a starting value (a starting point) for w1. The starting point doesn't matter much; therefore, many algorithms simply set w1 to 0 or pick a random value.



Gradient Descent

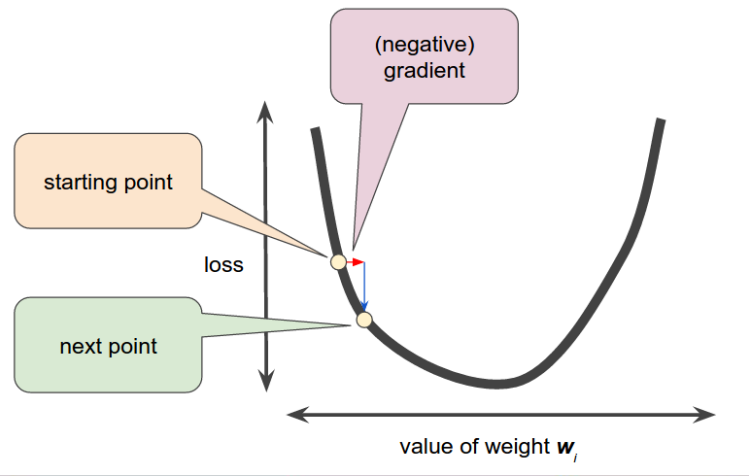
The gradient descent algorithm then calculates the gradient of the loss curve at the starting point. In brief, a gradient is a vector of partial derivatives.

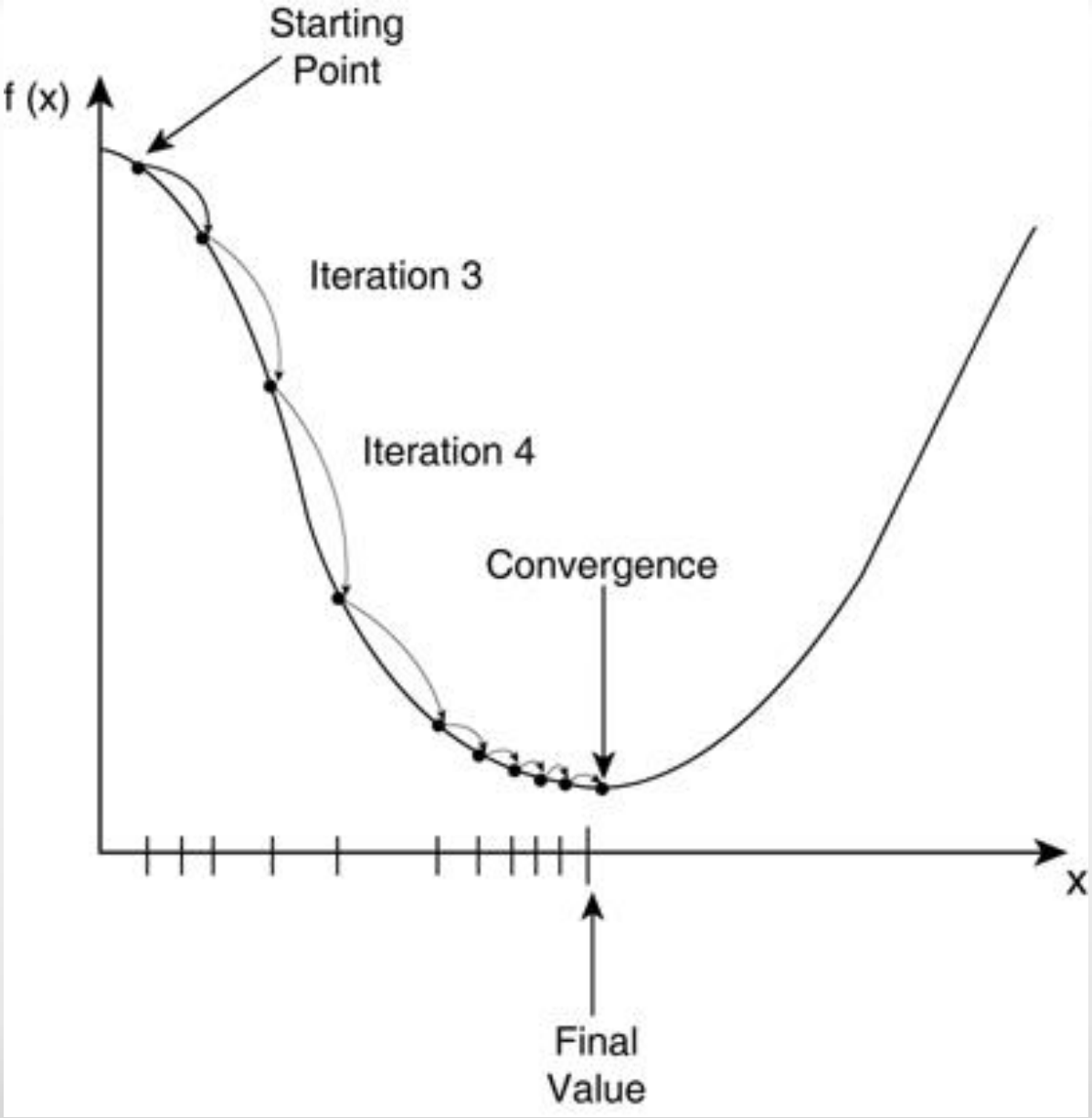
A gradient is a vector and hence has magnitude and direction.

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

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.

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



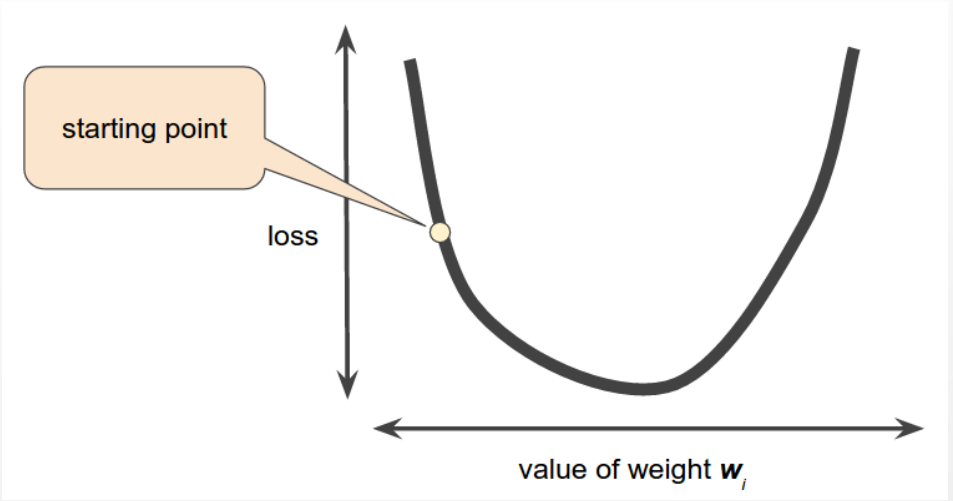

Starting
Point
f (x)
Iteration 3
Iteration 4
Convergence
x
Final
Value

Mathematical Significance

Repeat until convergence {

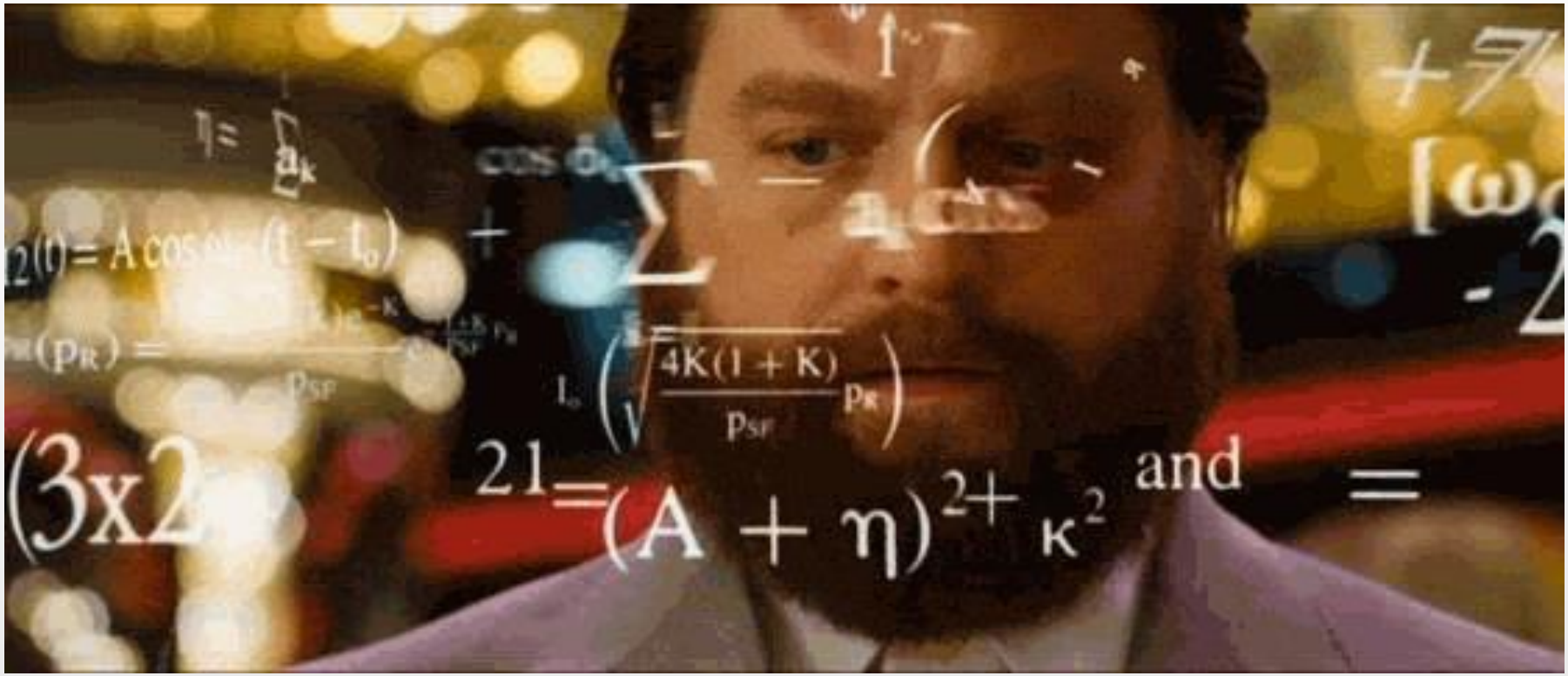
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}



The algorithm on the left signifies Gradient Descent algorithm.
In our case,

- Θ_j will be w_i
- $\boldsymbol{\alpha}$ is the learning rate
- $J(\Theta)$ is the cost function


(3x2
21=(A + η)^2+ κ^2 and =

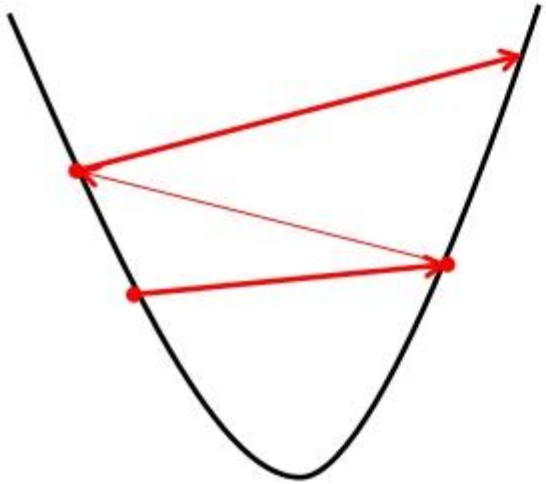
The Learning Rate

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

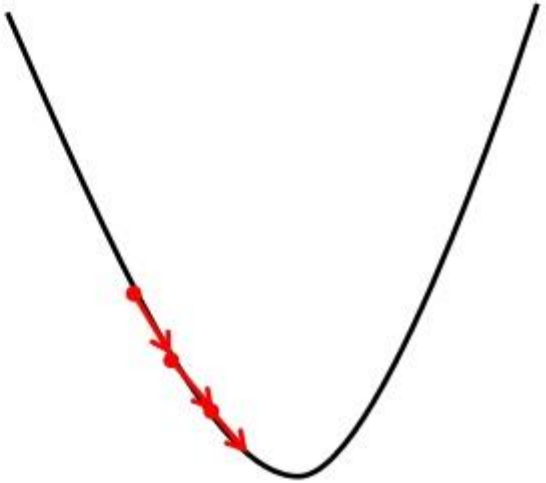
Hyperparameters are the knobs that programmers tweak in machine learning algorithms. Most machine learning programmers spend a fair amount of time tuning the learning rate. If you pick a learning rate that is too small, learning will take too long. Conversely, if you specify a learning rate that is too large, the next point will perpetually bounce haphazardly across the bottom of the well

Gradient Descent

Big learning rate



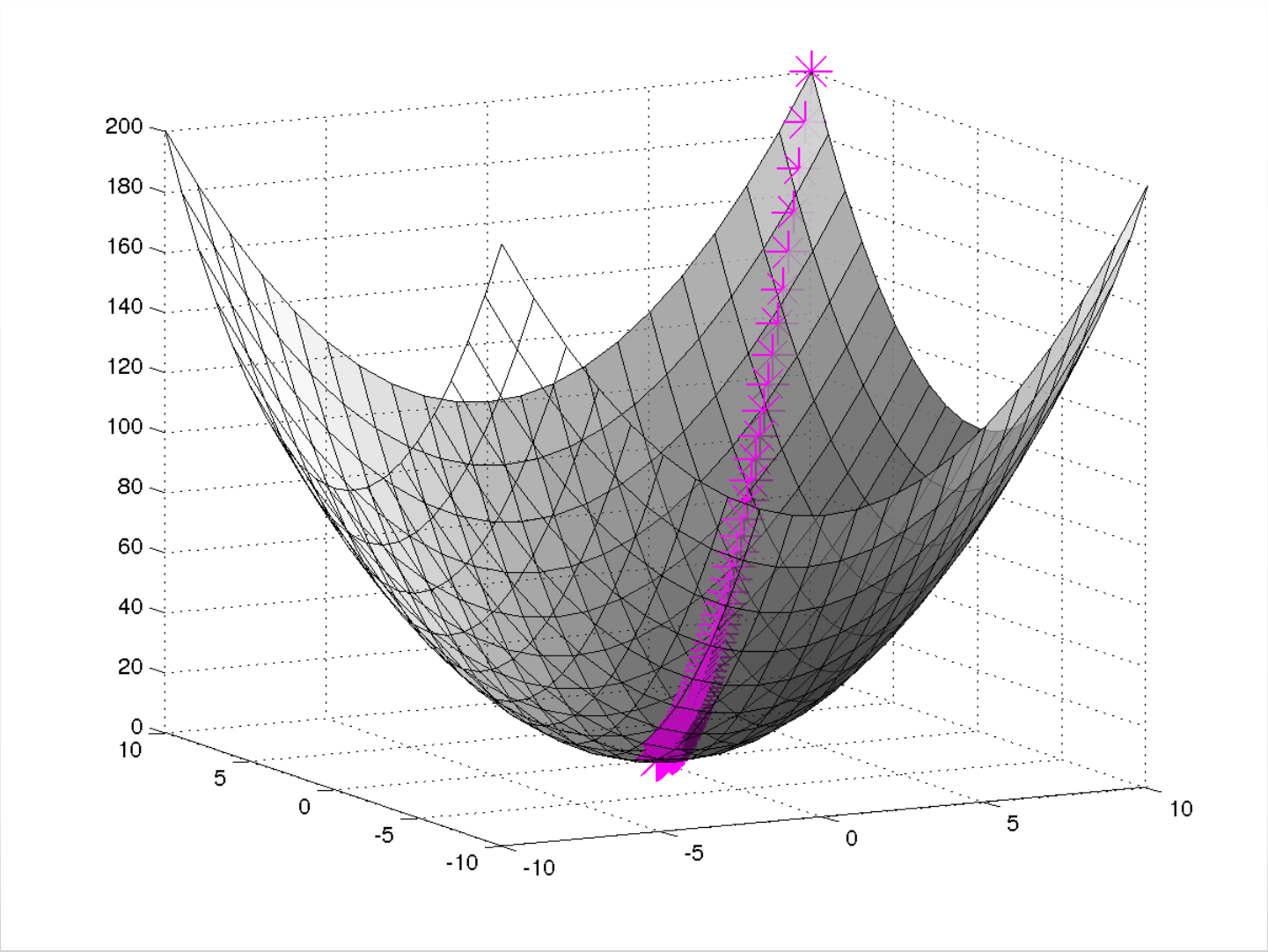
Small learning rate

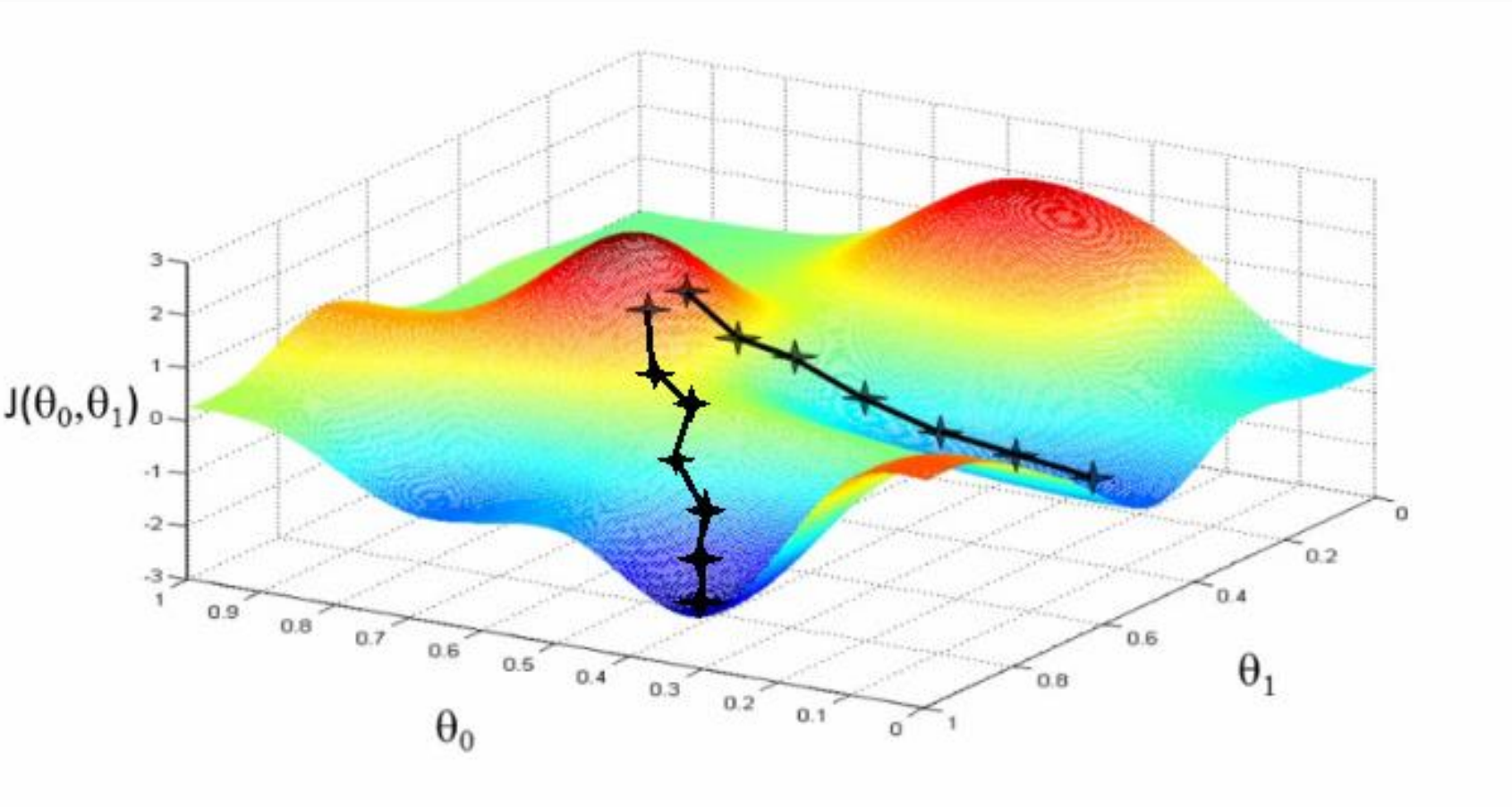


The Goldilocks Learning Rate

There's a Goldilocks learning rate for every regression problem. The Goldilocks value is related to how flat the loss function is. The flatter the loss function, the bigger a step you can safely take.

Gradient Descent on Multiple Features




J(θ0,θ1)
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
θ0
0
0.2
0.4
0.6
0.8
1
θ1

Batches

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. Enormous batches tend not to carry much more predictive value than large batches.

By choosing examples at random from our data set, we could estimate (albeit, noisily) a big average from a much smaller one.

Types of Gradient Descent

- **Stochastic gradient descent** (SGD) takes the idea of picking a dataset average 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.

TensorFlow

- TensorFlow is an open-source library for Machine Intelligence
- TensorFlow was developed by the Google Brain and released in 2015
- It provides high-level APIs to help implement many machine learning algorithms and develop complex models in a simpler manner.

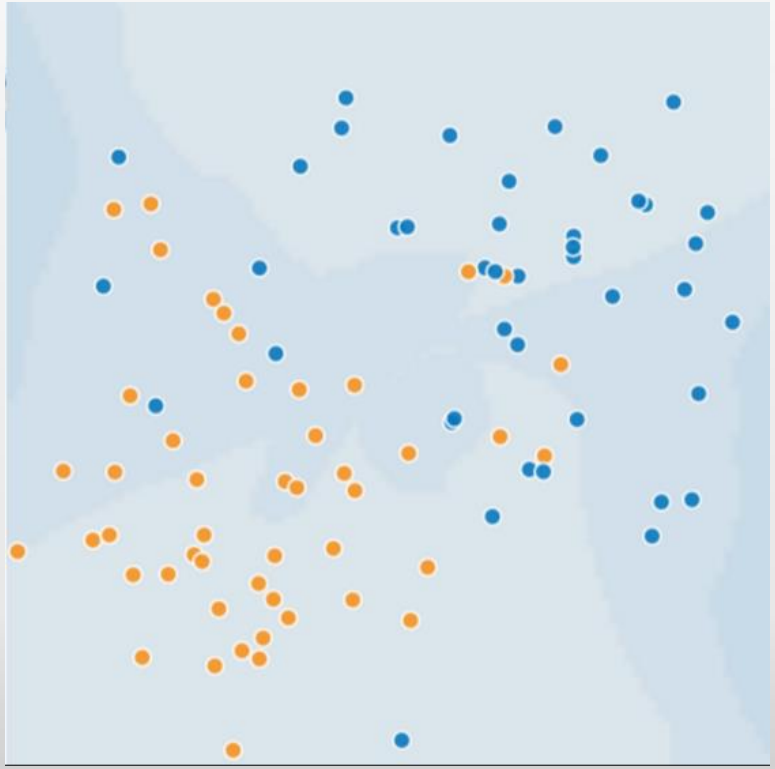
- What is a tensor?

    - A mathematical object, analogous to but more general than a vector, represented by an array of components that are functions of the coordinates of a space.
    - TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays known as ‘tensors’.

Generalization

Model's **ability to perform well on previously unseen data**, drawn from the same distribution as the one used to create the model.

Take a look at Figure

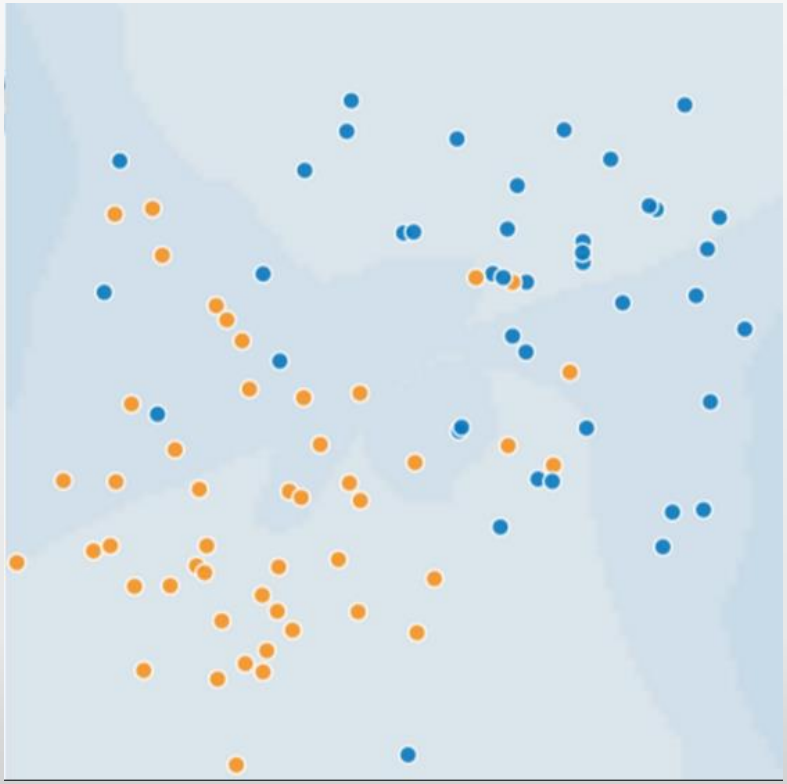


Assume that each dot in these figures represents a tree's position in a forest. The two colors have the following meanings:

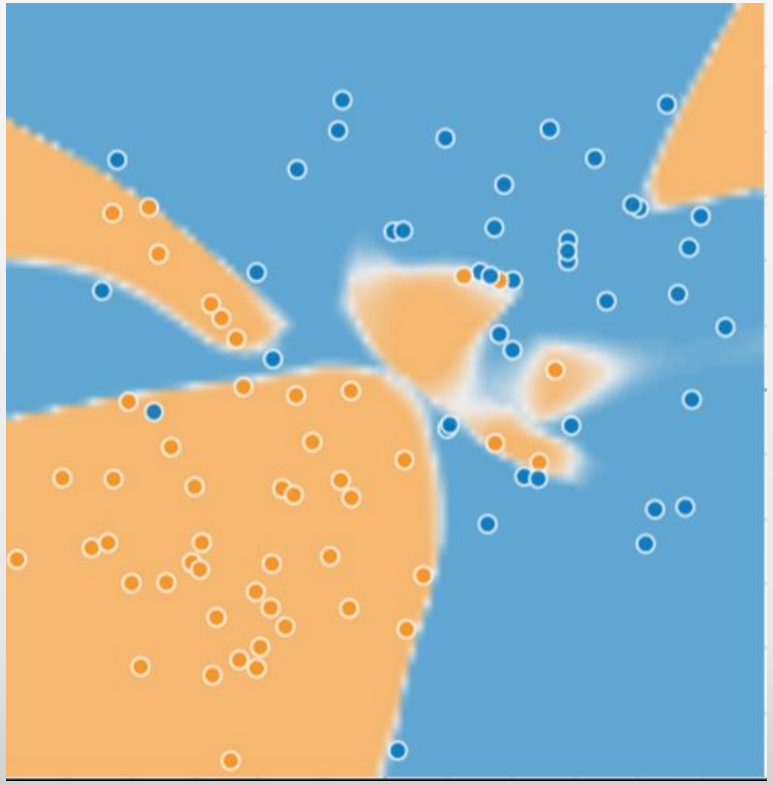
- The **blue** dots represent sick trees.
- The **orange** dots represent healthy trees.

Generalization

Imagine a good model for predicting subsequent sick or healthy trees.

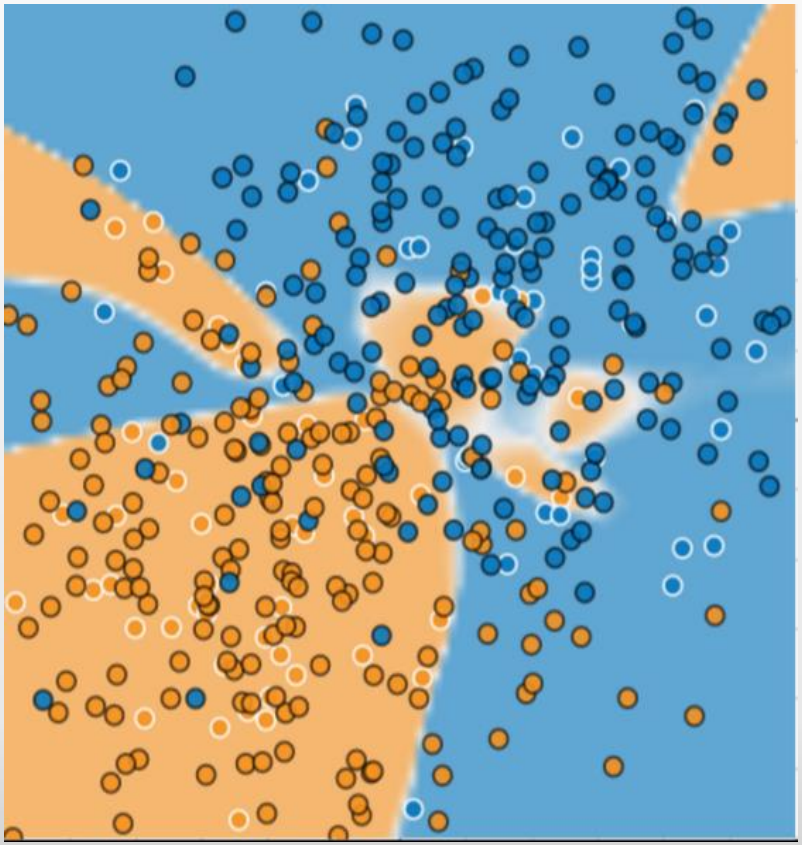


An excellent job by a certain (complex) machine learning model which produced a **very low loss**!



Generalization: Peril of Overfitting

Low loss, but still a bad model?
Let's check by adding new data to the model.

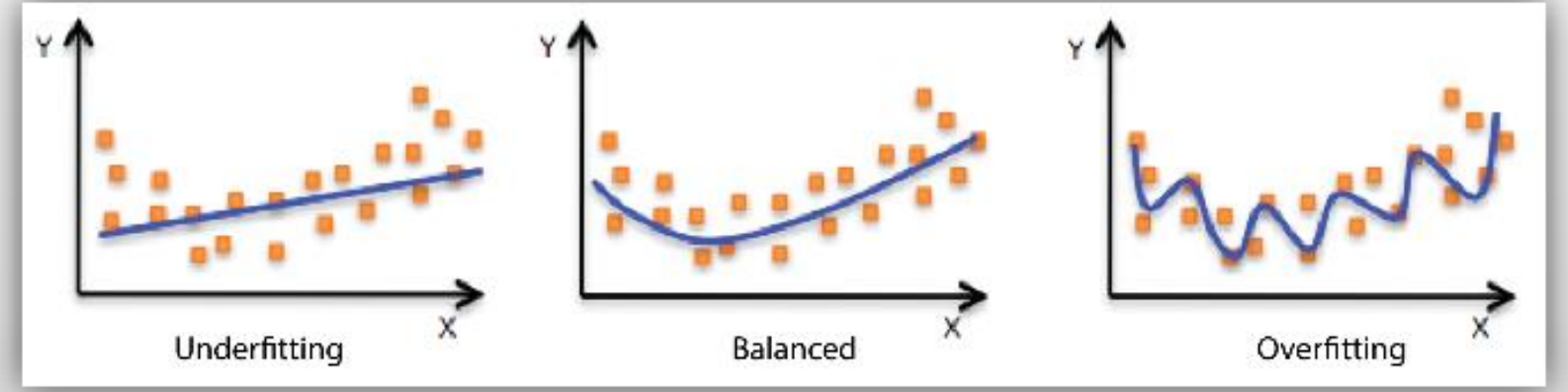


- It turned out that the model **adapted very poorly to the new data**.
- Notice that the model miscategorized much of the new data.
- Model **overfits** the peculiarities of the data it trained on.

अति सर्वत्र वर्जयेत् ।

Oooooverfitting = Game Over

- An overfit model gets a **low loss during training but does a poor job predicting new data**.
- 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.



"All things being equal, the simplest solution tends to be the best one."

William of Ockham

To put Occam's razor in machine learning terms

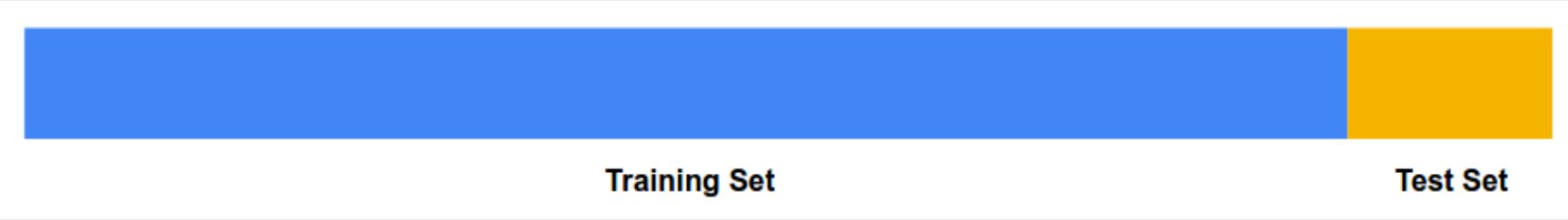
- **“ The less complex an ML model, the more likely that a good empirical result is not just due to the peculiarities of the sample. ”**
- In modern times, we've formalized Occam's razor into the fields of **statistical learning theory** and **computational learning theory**.
- These fields have developed **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

Splitting 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.




divide and
conquer....
GOOD
quickmeme.com

Make sure...

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

- 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.**

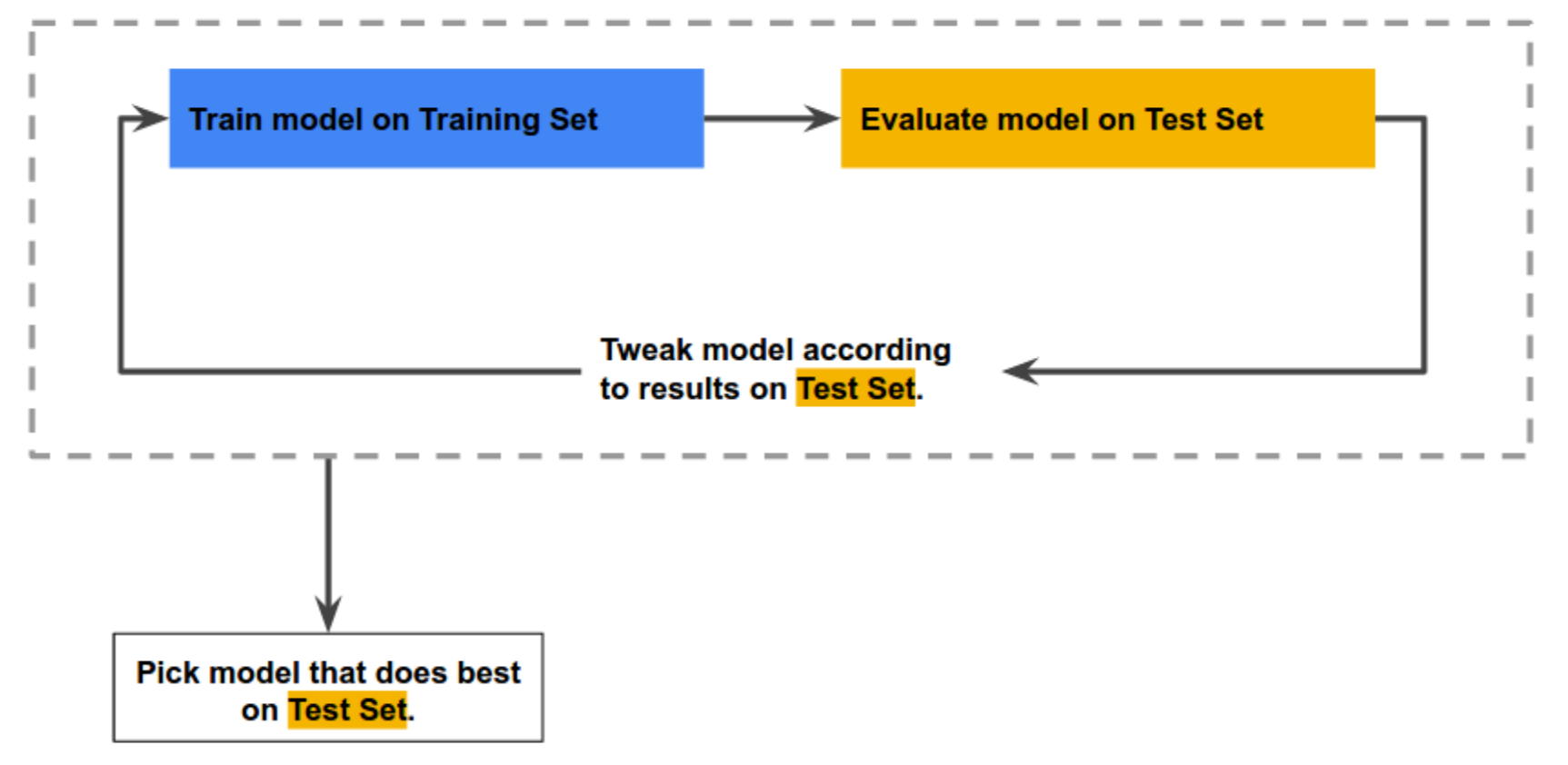
Warning!!

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 i.e. high accuracy might indicate that **test data has leaked** into the training set. So, **Never train on test data.**

Giving an 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'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**.

A possible Workflow



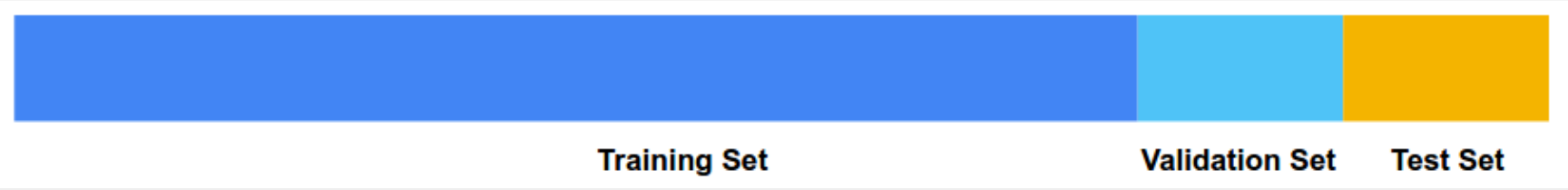
"**Tweak model**" means adjusting anything about the model you can dream up-from changing the learning rate, to adding or removing features, to designing a completely new model from scratch.

At the end of this workflow, you pick the model that does best on the *test set*.

Can we do better?

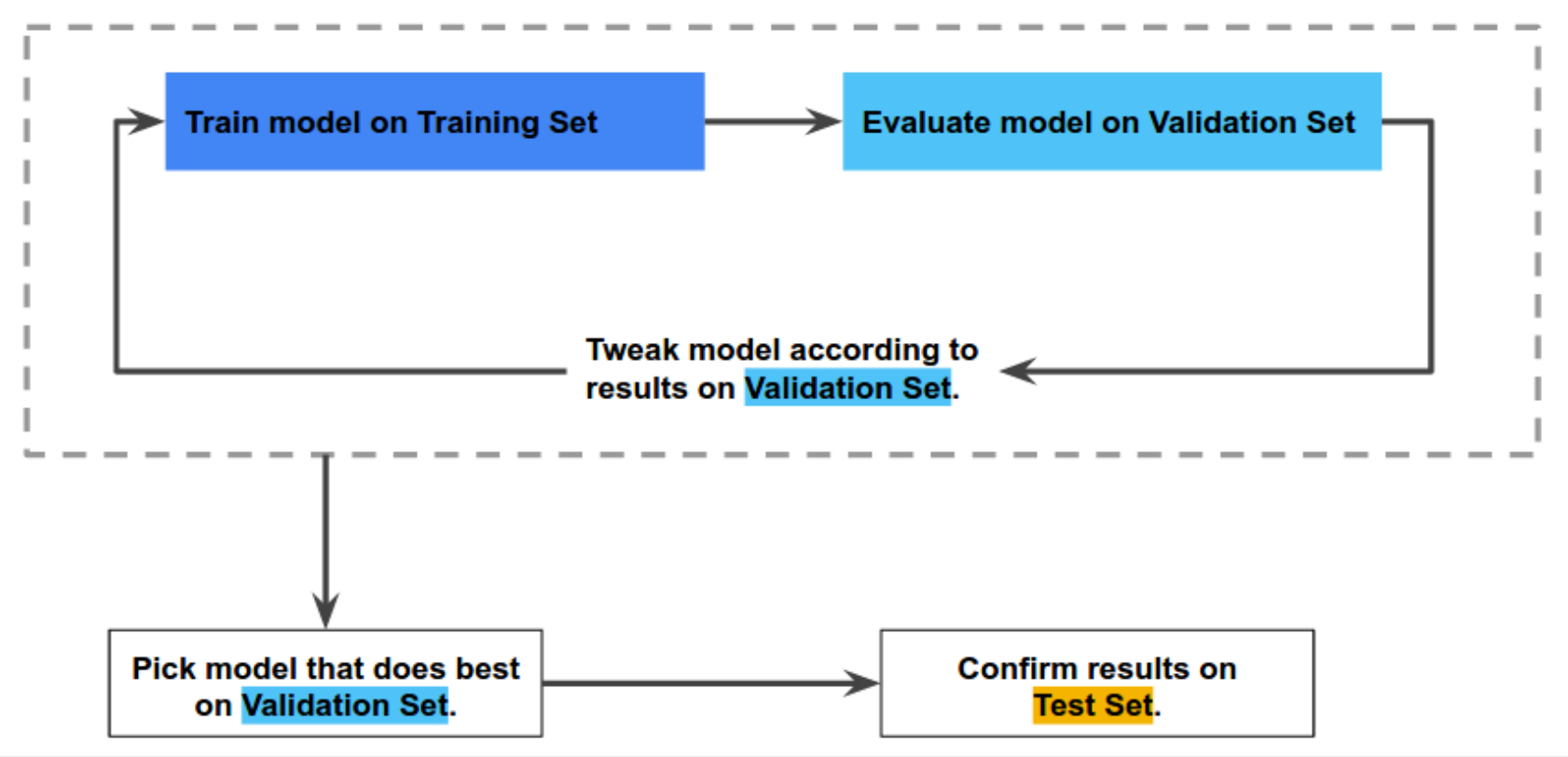
How about having an another Partition?

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:



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.

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.

Can we perform still better?

- How about having an another Partition?
  - Well, you can have **N** partitions!

- **The more you Divide, the more you Conquer!!**

Representation

A machine learning **model can't directly see, hear, or sense input** examples. Instead, you must create a representation of the data to provide the model with a useful vantage point into the data's key qualities.

In **traditional programming**:

- The focus is on code.

In **machine learning** projects:

- The focus shifts to representation.

That is, developers hone a model by addin es.

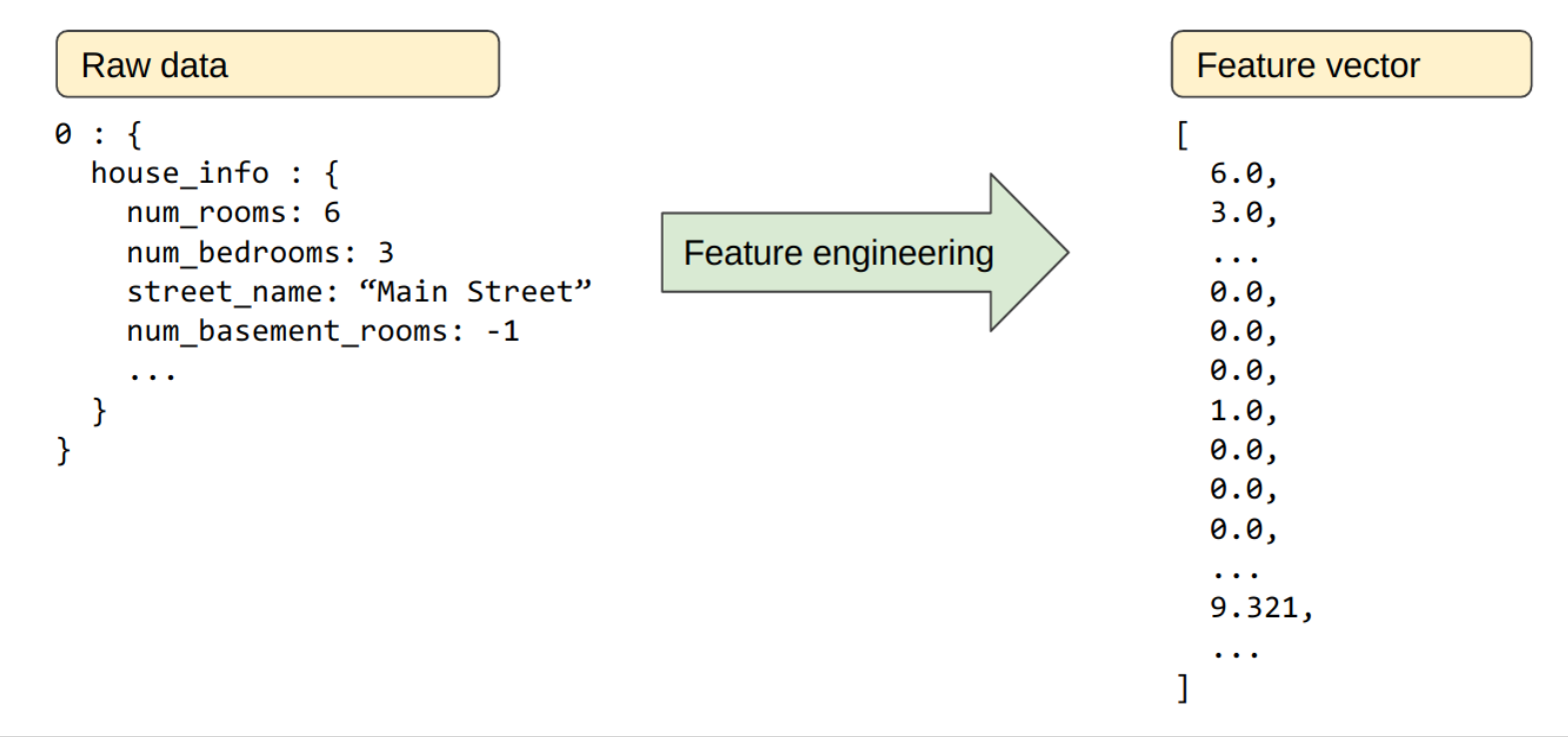
Feature **Engineering**

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

- **Feature engineering** means transforming raw data into a feature vector.
- Expect to spend significant time doing feature engineering.
- Machine learning models typically expect examples to be represented as **real-numbered vectors**.
- The vector is constructed by deriving features for each field, then concatenating them all together.

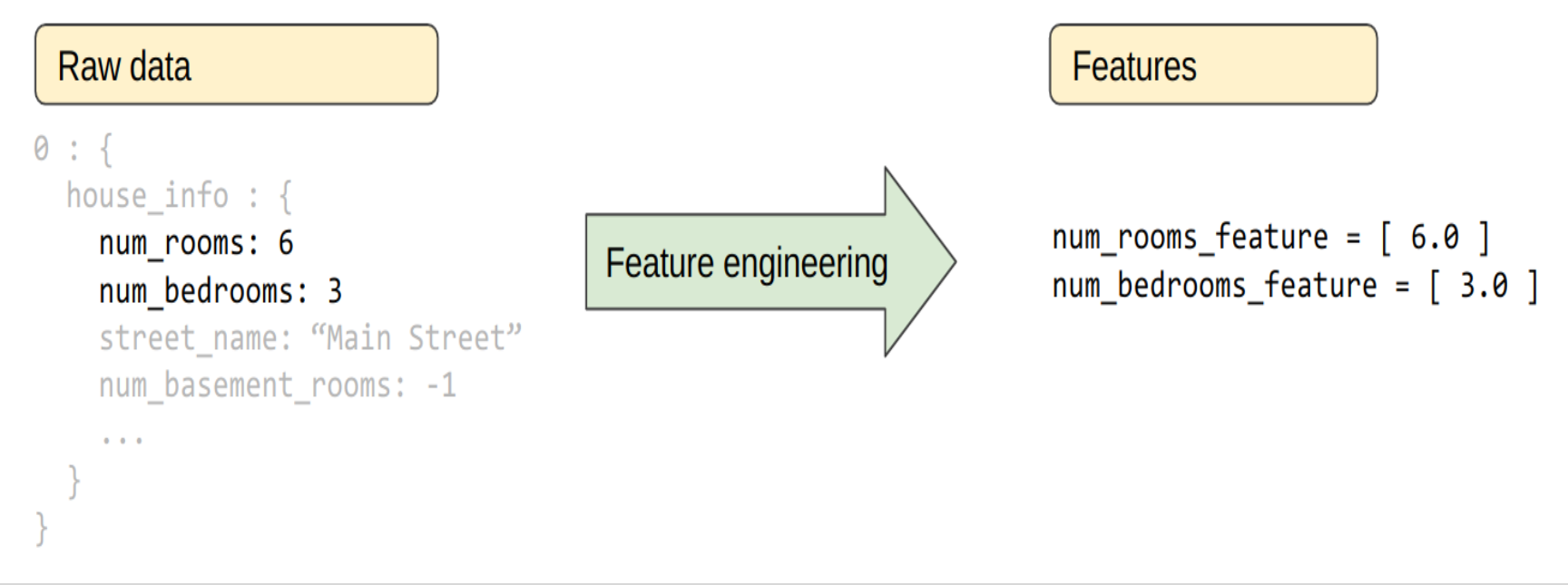
Mapping Raw Data to Features

Figure 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.



Mapping numeric values

ML models train on floating-point values, so integer and floating-point raw data don't need a special encoding. As suggested in Figure, converting the raw integer value 6 to the feature value 6.0 is trivial.



Mapping string values

Models can't learn from string values, so you'll have to perform some feature engineering to convert those values to something numeric.

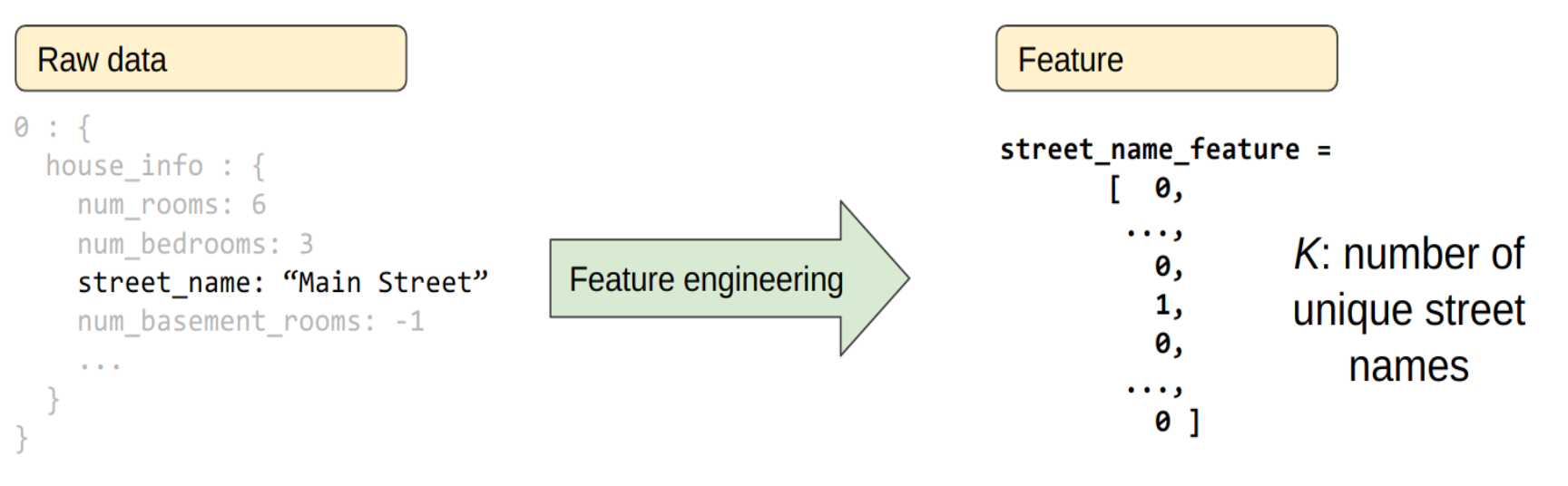
One-hot encoding is one popular way to represent string values as a floating-point vector. In a one-hot encoding:

- Only one element is set to 1
- All other elements are set to 0

Mapping string values

A one-hot encoding is a **sparse vector**; that is, a vector that is mostly zeroes. The dimensionality of the vector is the total number of possible values for that field.

For example, the sparse vector representing streets must be large enough to represent all possible street names. The particular street (Main Street in our example) will be stored as a 1, while all other street names will be represented as a 0.



Mapping categorical (enumerated) values

Categorical features have a discrete set of possible values. For example, a feature called `**Lowland Countries**` would consist of only three possible values:

{'Netherlands', 'Belgium', 'Luxembourg'}

You might be tempted to **encode** categorical features like `Lowland Countries` as an enumerated type or as a discrete set of integers representing different values. For example:

- represent Netherlands as **0**
- represent Belgium as **1**
- represent Luxembourg as **2**

Mapping categorical (enumerated) values

However, machine learning models typically represent each categorical feature as a separate Boolean value.

For example, `Lowland Countries` would be represented in a model as **three separate Boolean features**:

- x_1: is it Netherlands?
- x_2: is it Belgium?
- x_3: is it Luxembourg?

Encoding this way also simplifies situations in which a **value can belong to more than one category**.
For example, "borders France" is True for both Belgium and Luxembourg.

Representation: Cleaning Data

Apple trees produce some mixture of great fruit and wormy messes. Yet the apples in high-end grocery stores display 100% perfect fruit. Between orchard and grocery, someone spends significant time **removing the bad** apples or throwing a little wax on the salvageable ones.

As an ML engineer, you'll spend enormous amounts of your time tossing out bad examples and cleaning up the salvageable ones. **Even a few "bad apples" can spoil a large data set.**

Scaling feature values

Scaling means converting floating-point feature values from their natural range into a standard range.
For example, **100 to 900 -> 0 to 1 or -1 to +1**

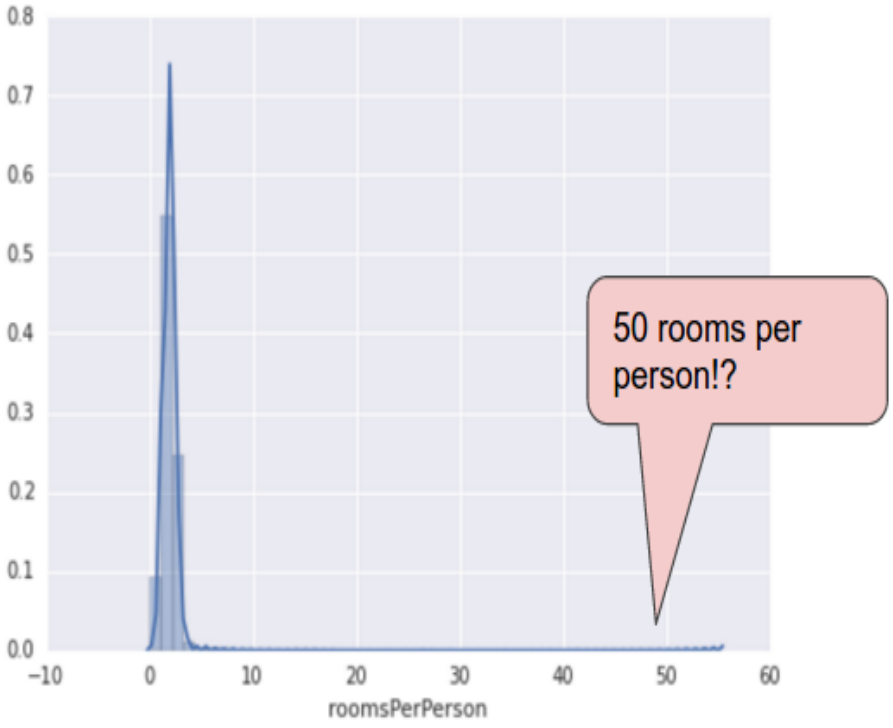
Feature scaling provides the following benefits:

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

You don't have to give every floating-point feature 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.

OUTLIER

Handling extreme outliers




```
roomsPerPerson = totalRooms / population
```

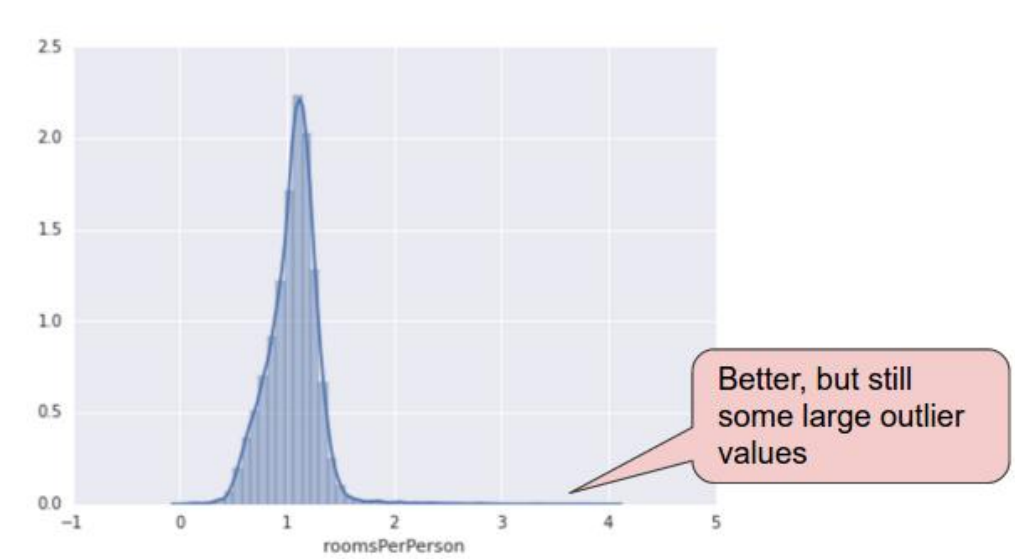
The following plot represents a feature called roomsPerPerson from the California Housing data set.

The value of **roomsPerPerson** was calculated by dividing the total number of rooms for an area by the population for that area.

The plot shows that the vast majority of areas in California have **one or two rooms per person**.

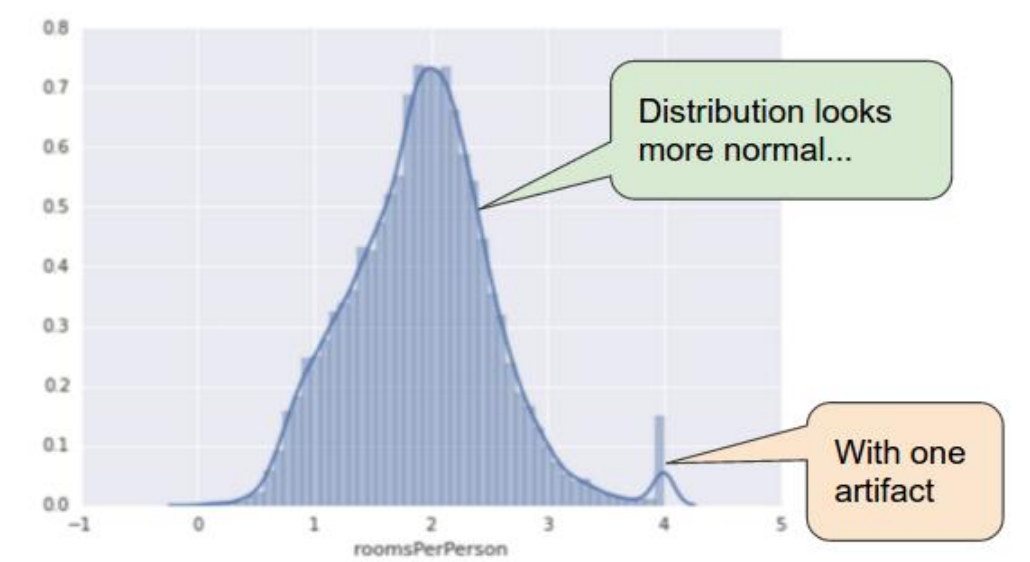
A verrrrry lonnnnnnng tail

Take a look along the x-axis.
How could we minimize the influence of those extreme outliers? Well, one way would be to take the log of every value:
Log scaling does a slightly better job, but there's still a significant tail of outlier values.



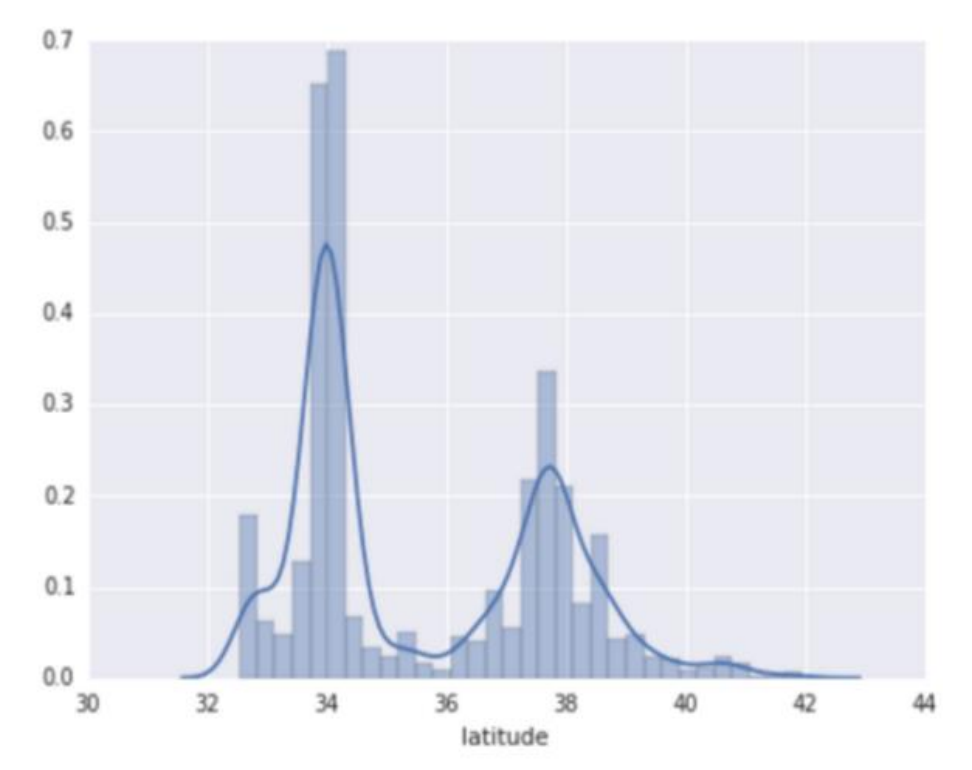
Clipping feature values

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?
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 at latitude 34** and **San Francisco is roughly at latitude 38**.



In the data set, latitude is a floating-point value. However, **it doesn't make sense to represent latitude as a floating-point feature in our model**.

That's because no linear relationship exists between latitude and housing values.

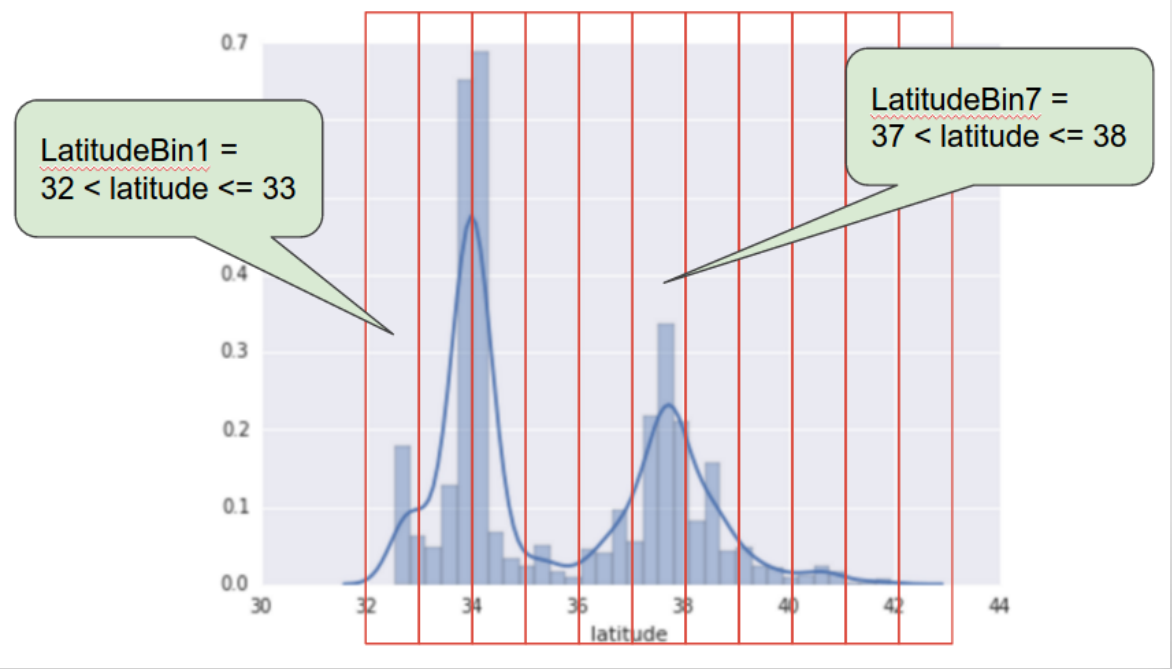
For example, houses in latitude 35 are not 35/34 more expensive (or less expensive) than houses at latitude 34.

Binning

To make latitude a helpful predictor, let's **divide latitudes into "bins"** as suggested by the following figure.
Instead of having one floating-point feature, we now have **11 distinct boolean features** (LatitudeBin1, LatitudeBin2, ..., LatitudeBin11).

Doing so will enable us to represent latitude 37.4 (San Francisco) as follows:
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

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



Scrubbing

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

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

Once detected, you typically **"fix" bad examples by removing them from the data set**.

Scrubbing

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

Know your data

Follow these rules:

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

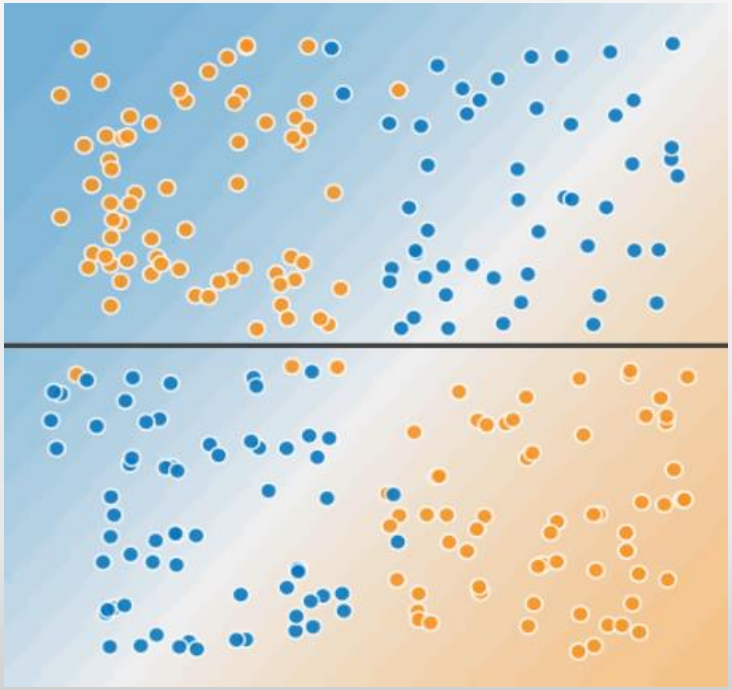
Treat your data with all the care that you would treat any mission-critical code.
Good ML relies on good data.

Feature Crosses: Encoding Nonlinearity

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

Can you draw a line that neatly separates the sick trees from the healthy trees? Sure.
This is a linear problem.

Can you draw a single straight line that neatly separates trees? No, you can't.
This is a nonlinear problem.



Feature Crosses

To solve the nonlinear problem shown in Figure, 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**.)

Let's create a feature cross named x3 by crossing x1 and x2:

$$x_3 = x_1 x_2$$

We treat this newly minted x3 feature cross just like any other feature. The linear formula becomes:

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3$$

A linear algorithm can learn a weight for w3 just as it would for w1 and w2.

In other words, **although w3 encodes nonlinear information, you don't need to change how the linear model trains to determine the value of w3.**

Kinds of feature crosses

We can create many 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, 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.

Crossing One-Hot Vectors

So far, we've focused on feature-crossing two individual floating-point features. In practice, machine learning models seldom cross continuous features. However, machine learning models do frequently cross one-hot feature vectors.

Think of feature crosses of one-hot feature vectors as **logical conjunctions**.

For example, suppose we have two features: **country and language**.
A one-hot encoding of each generates vectors with binary features that can be interpreted as

country=USA, country=France or language=English, language=Spanish.

Then, if you do a feature cross of these one-hot encodings, you get binary features that can be interpreted as logical conjunctions, such as:

country:usa **AND** language:spanish

Crossing One-Hot Vectors

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.

Crossing One-Hot Vectors

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   < longitude <= 15
    15 < longitude <= 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
  ]
```

Crossing One-Hot Vectors

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:

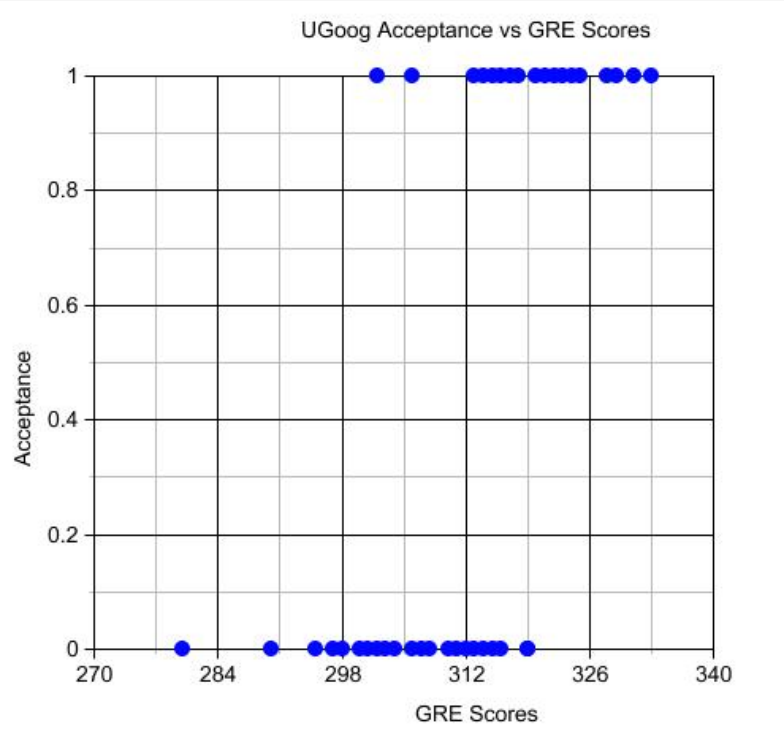
[behavior type X time of day]

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**.

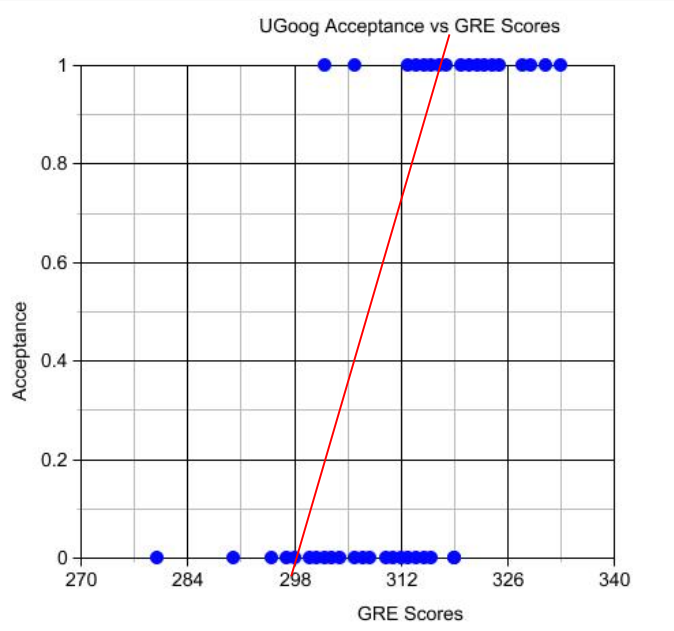
Linear learners scale well to massive data. Using feature crosses on massive data sets is one efficient strategy for learning highly complex models.

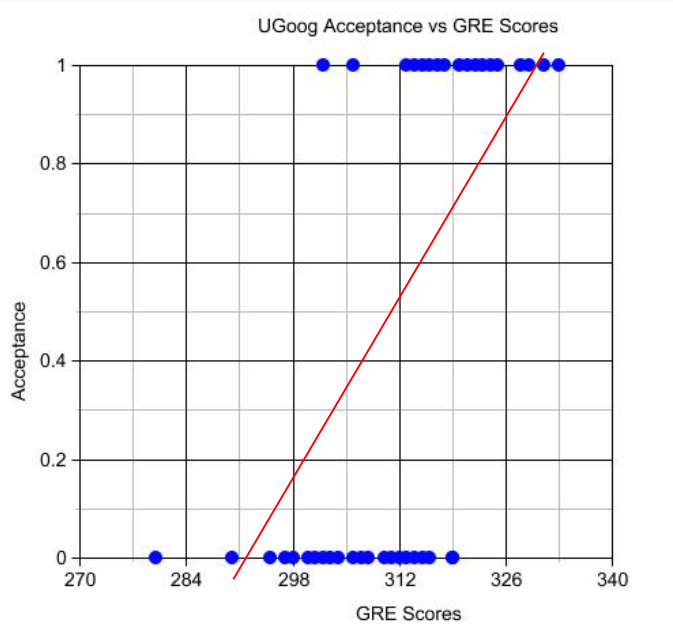
The Acceptance Dilemma

My Story: I am highly interested in pursuing M.S. in Artificial Intelligence at UGoog. My GRE Score is 315 and I fancy my chances of getting in. As a prospective graduate student, I start browsing through the GRE scores of students that had been admitted to UGoog and those which were rejected. Thanks to MLCC, I could predict my chances through regression model. Unless the graph looks……



I tried to fit a line





but all gave high losses

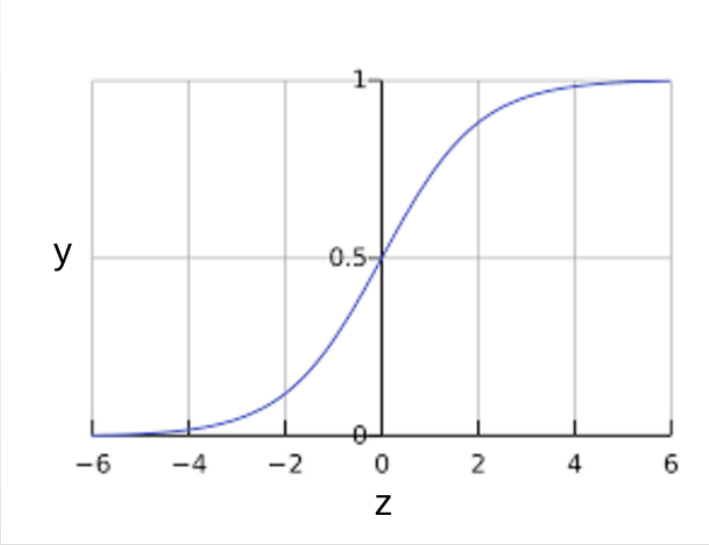
and I was like……

The Solution: Logistic Regression

Many problems require a probability estimate as output. Logistic regression is an extremely efficient mechanism for calculating probabilities.

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:

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



Log Loss

Now, $z = w_0 + x_1 w_1$, and we take a log loss function-

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

The weights are then updated by the log loss until the the log loss converges to almost 0.

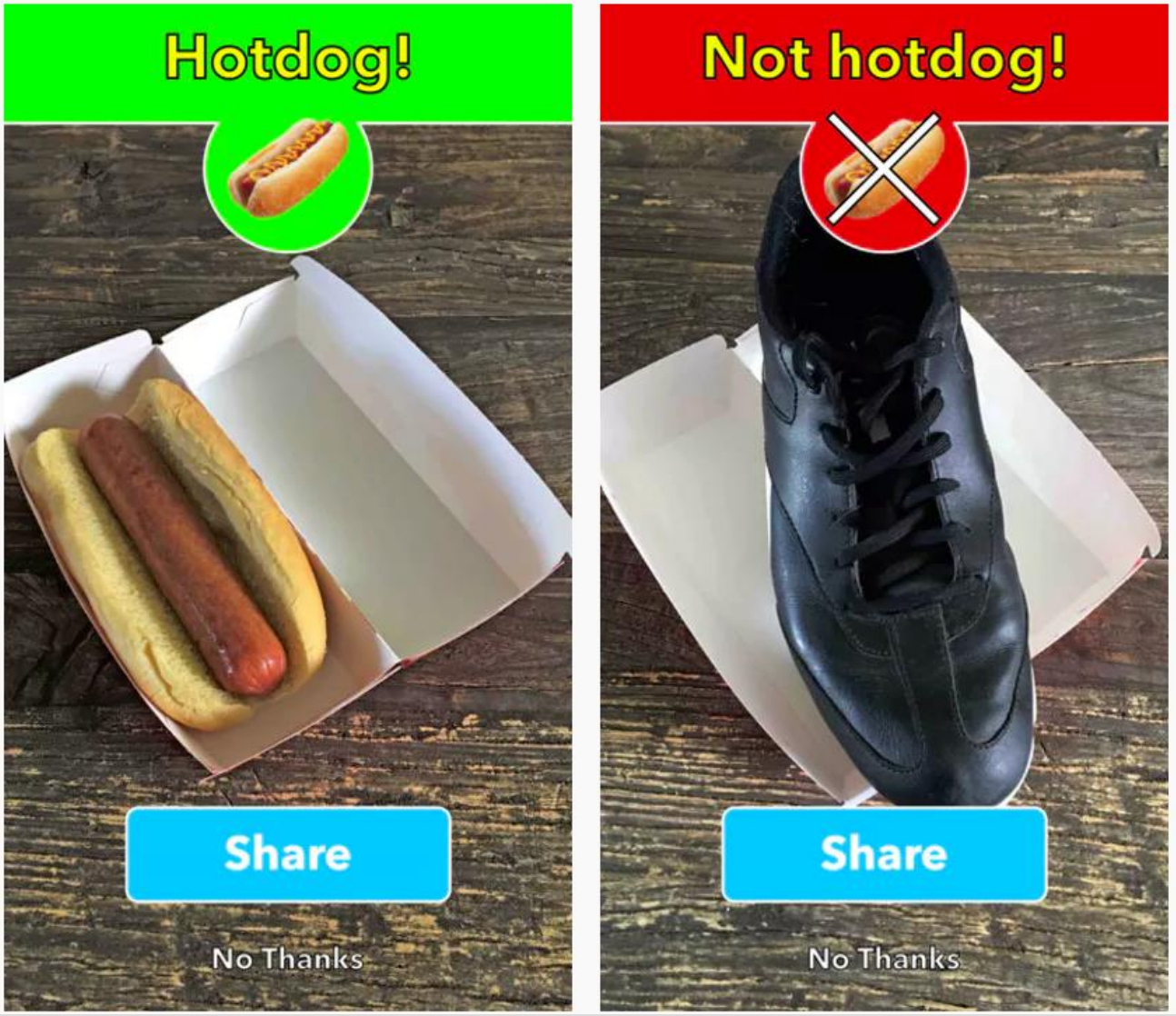
My regression model now gives a probability of my acceptance at UGoog as an output.

Regression, Thresholding and Classification

Logistic regression returns a probability. You can use the returned probability as it is or convert the returned probability to a binary value. How do we convert a regression model to a classification model?

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 class A; a value below indicates class B. 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.

How can one determine a good decision threshold?


Hotdog!
Share
No Thanks
Not hotdog!
Share
No Thanks

Classification: True vs False, Positive vs Negative

A famous tale,

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. Moral ensues.

Consider,

- ‘Wolf’ as a positive class
- ‘No Wolf’ as a negative class

Classification: True vs False, Positive vs Negative

Our "wolf-prediction" model, there are 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.

Metrics for Classification

Accuracy:

Accuracy is the fraction of predictions our model got right.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{TP + TN}{TP + TN + FP + FN}$$

Let's take an example,

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**

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

Metrics for Classification

The accuracy of the model was 91% which seems great!!

Let’s take a closer look.

- 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!
- 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.

Hence we need better metrics.

Towards Better Metrics

Precision:

Precision attempts to answer: “Was the model correct when it predicted a positive class?”

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

Recall:

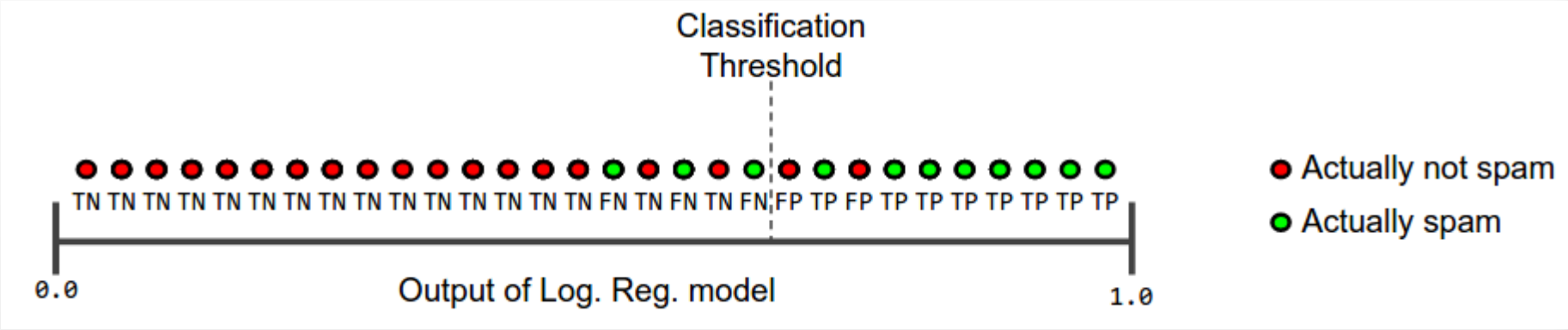
Recall attempts to answer: “Out of all the possible positive class outcomes, how many did the model correctly identify?”

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

Precision and recall are often in tension. That is, improving precision typically reduces recall and vice versa.

Precision and Recall

We look at 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."



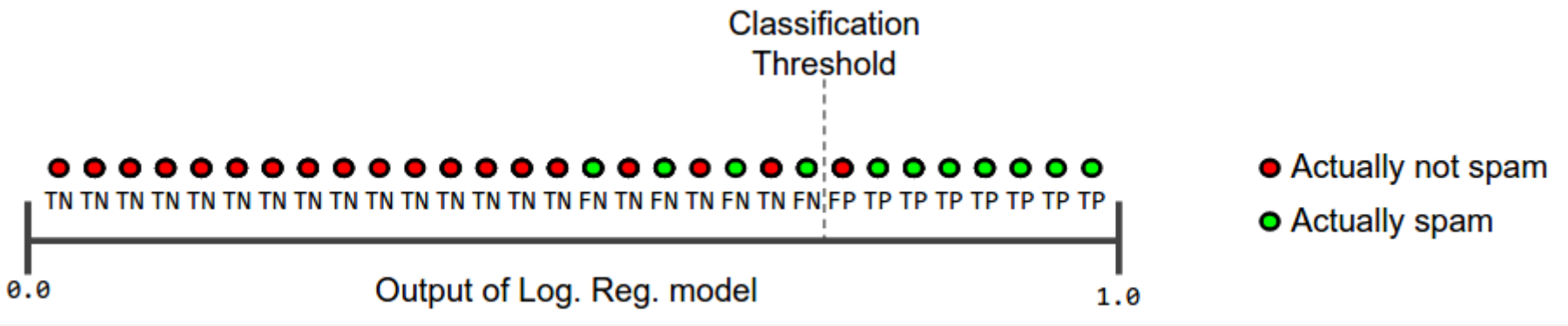
True Positives (TP): 8	False Positives (FP): 2
False Negatives (FN): 3	True Negatives (TN): 17

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

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

Precision and Recall

We shift the classification threshold to the right.



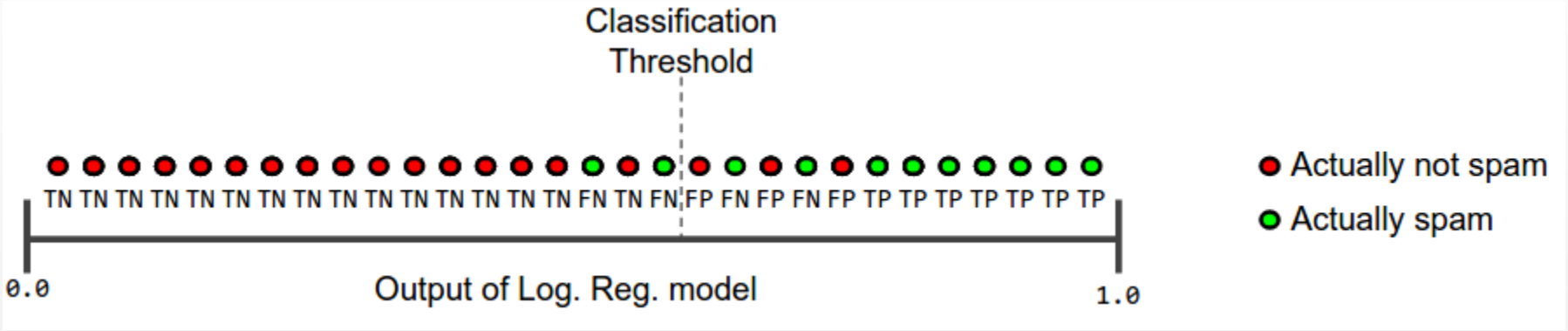
True Positives (TP): 7	False Positives (FP): 1
False Negatives (FN): 4	True Negatives (TN): 18

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

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

Precision and Recall

We shift the classification threshold to the left.



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

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

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

Hence we see that when precision increases recall decreases and vice versa.

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-

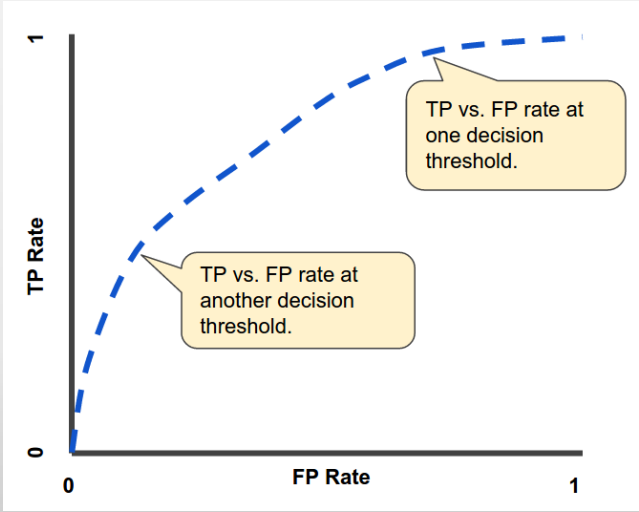
True Positive Rate (TPR) is a synonym for recall and is therefore defined as follows:

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

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 increases False Positives and True Positives, though typically not to the same degree.



Area Under ROC Curve (AUC)

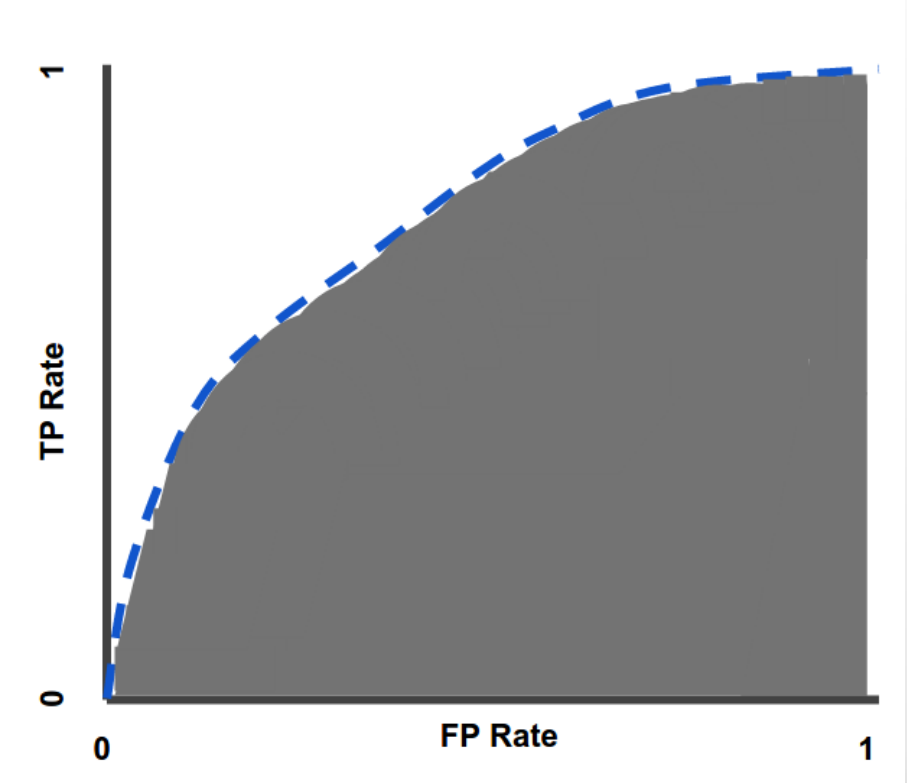
To compute the points in an ROC curve, we could evaluate a logistic regression model many times with different classification thresholds, but this would be inefficient. Fortunately, there's an efficient, sorting-based algorithm that can provide this information for us, called **AUC (Area under the ROC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

AUC

- AUC measures the entire two-dimensional area underneath the entire ROC curve from (0,0) to (1,1).

One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative 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.



Advantages and Disadvantages

Advantages:

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.

Disadvantages:

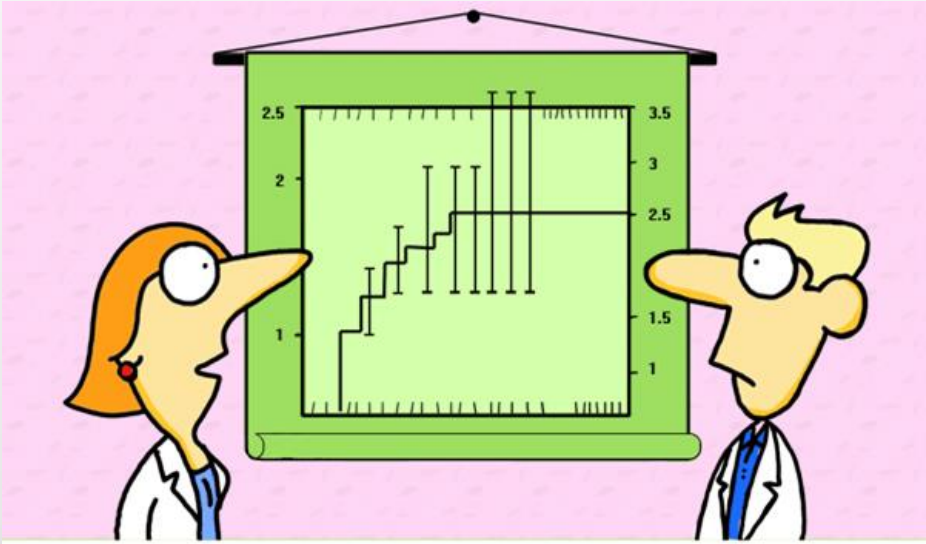
- **Scale invariance is not always desirable.** For example, sometimes we really do need well calibrated probability outputs, and AUC won't tell us about that.
- **Classification-threshold invariance is not always desirable**. In cases where there are wide disparities in the cost of false negatives vs. 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.

Prediction Bias

Prediction bias is a quantity that measures how far apart the average of predictions and input labels are. That is:

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

A significant nonzero prediction bias tells you there is a bug somewhere in your model.



"Did you really have to show the error bars?"

Prediction Bias

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. 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:

1. **Incomplete feature set** - Some critical features are not taken into account while modelling.
2. **Noisy data set -** There are some errors in the training data
3. **Buggy pipeline -** There is some error in the model itself
4. **Biased training sample -** Let’s say only spam emails are given for training
5. **Overly strong regularization -** The regularizer overpowers the effective parameter updation.

Calibration Layers

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 and ultimately become a nightmare to maintain.

Prediction Bias

Consider a case when the training data has a 1:1 ratio of spam(1) and non spam (0) emails. Hence the average label would be 0.5. Now let's suppose that our model predicts 5 out of 10 emails as spam(1) and the other 5 as non-spam(0). The prediction average would also be 0.5 and we don't have a prediction bias. The story ends well unless, on further inquiry, we observe that the results in each case were quite the opposite than predicted.

Bucketing

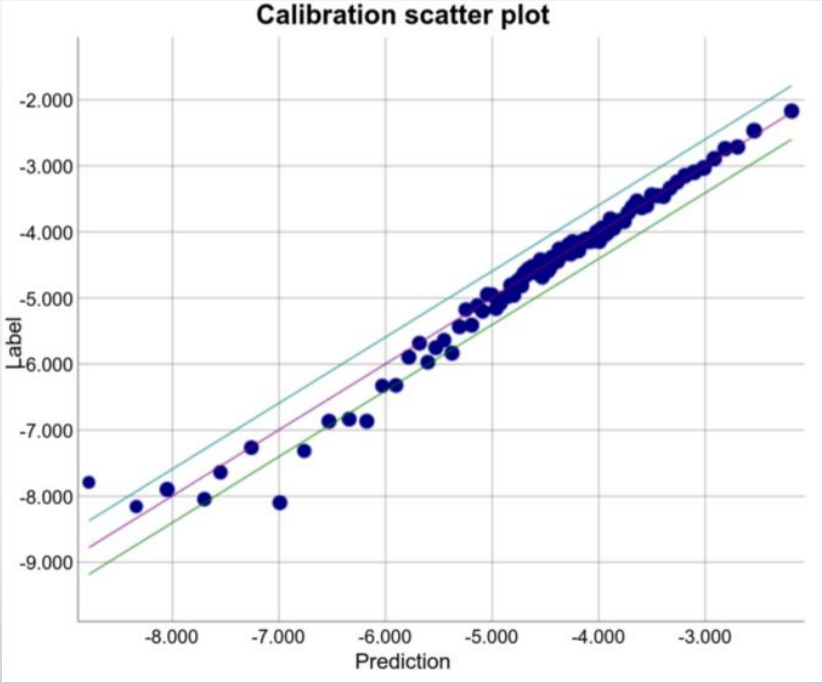
This is tackled by a concept known as **bucketing.**

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 and average of each bucket is calculated.

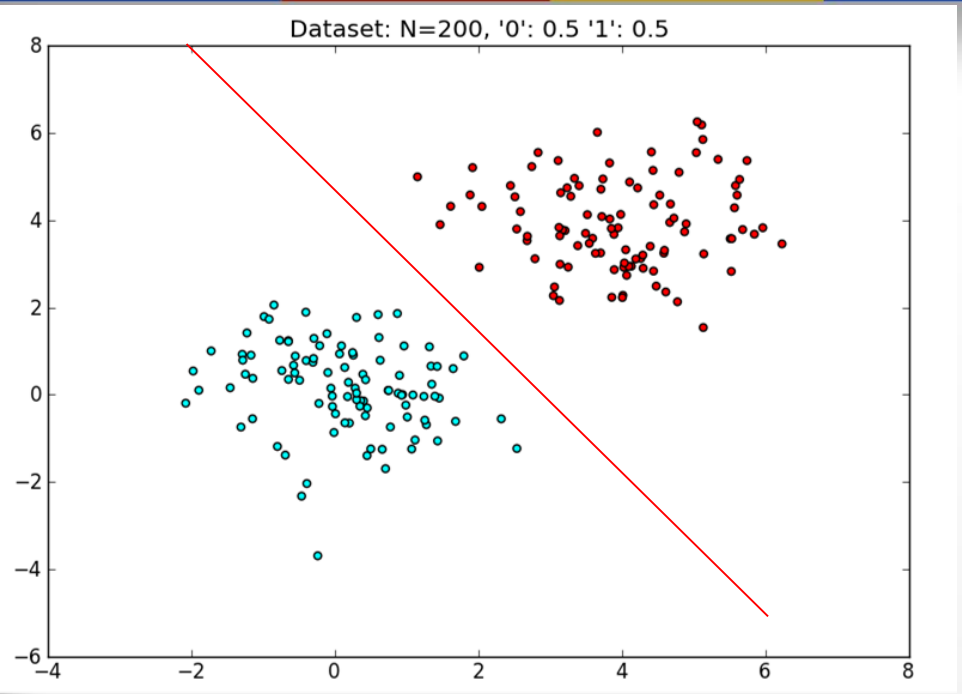
In the graph, each blue dot represents a bucket average.

A good predictor will have almost all the blue dots on the line y=x.

The bucket size has to be optimally chosen. There will be a trade-off between computation and accuracy. (Sounds Familiar!!!)

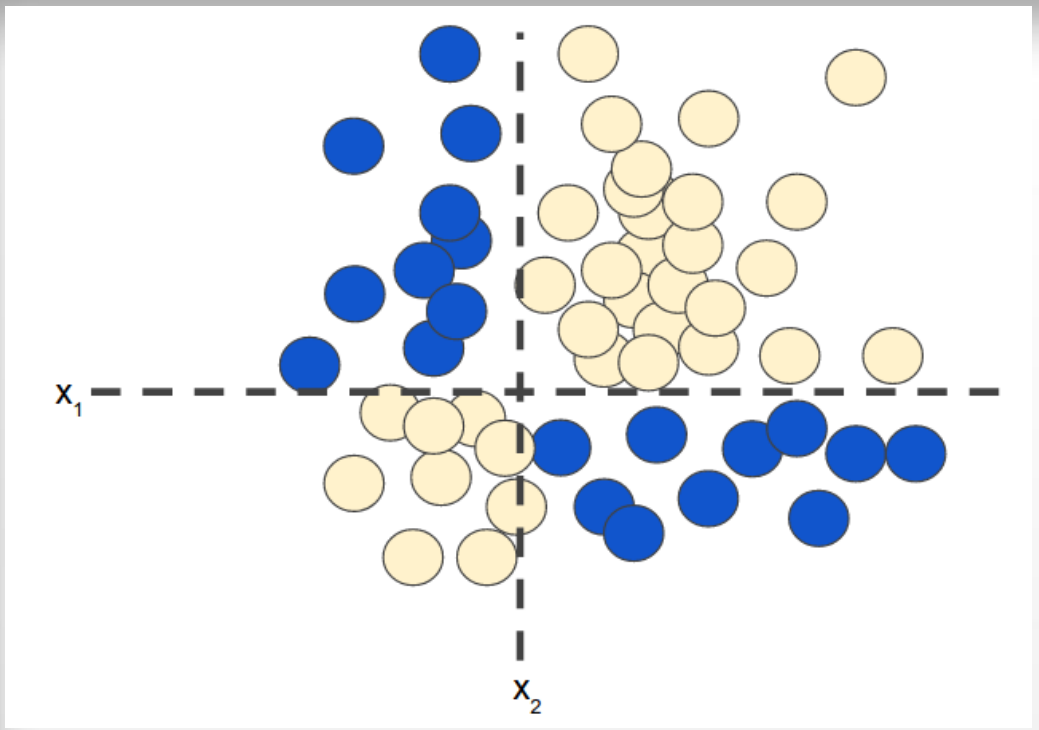


NEURAL NETWORKS



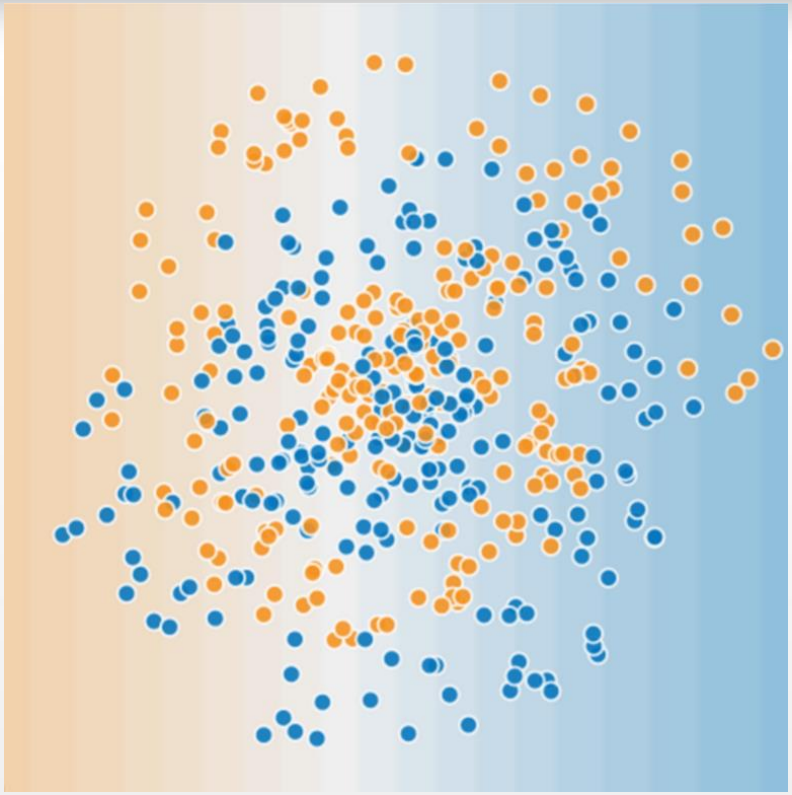
How do you classify these points?

$$b + w_1 x_1$$



How do you classify these points?

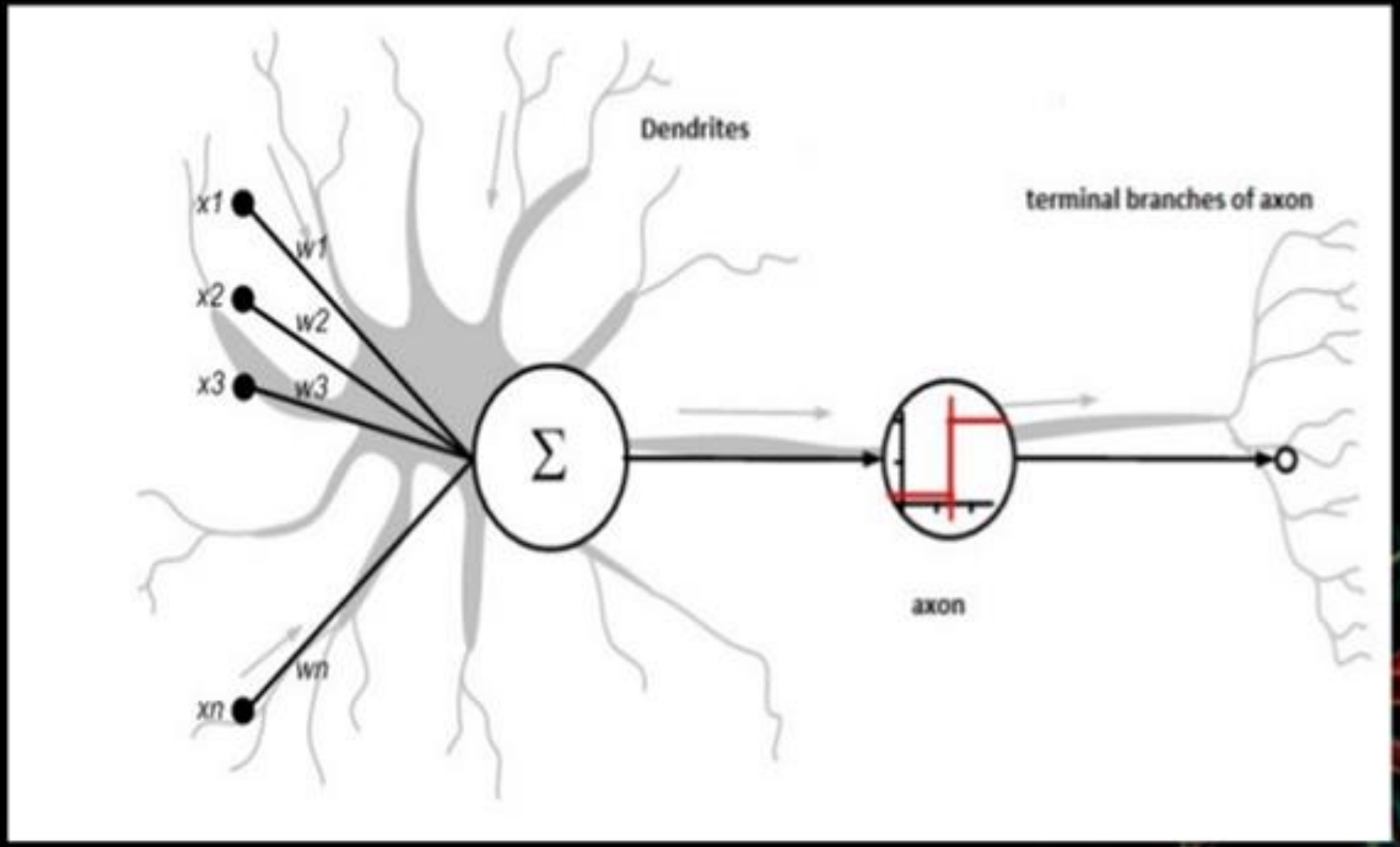
Feature Crosses!!!!!



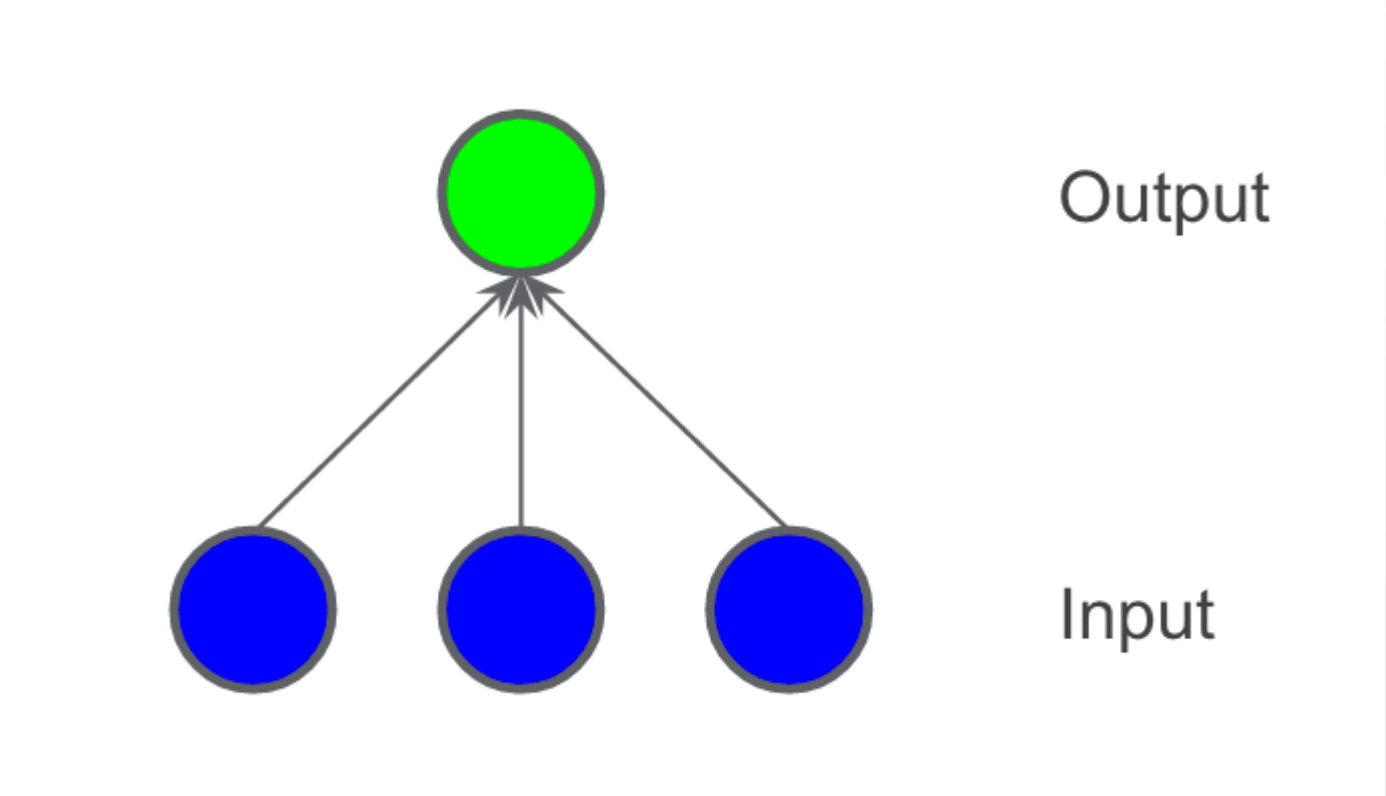
Non-linearities are tough to model. In complex datasets, the task becomes very cumbersome. What is the solution?

NEURAL NETS

Biologically Inspired Neuron

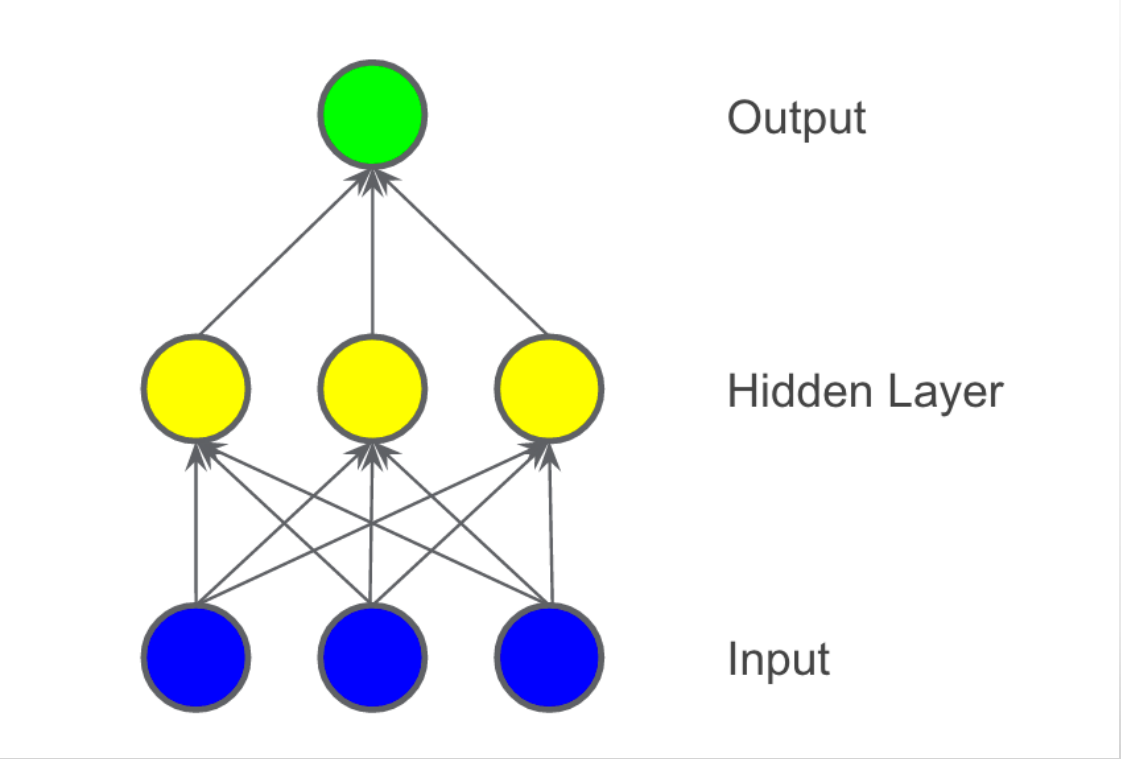


Modeling a Linear Equation

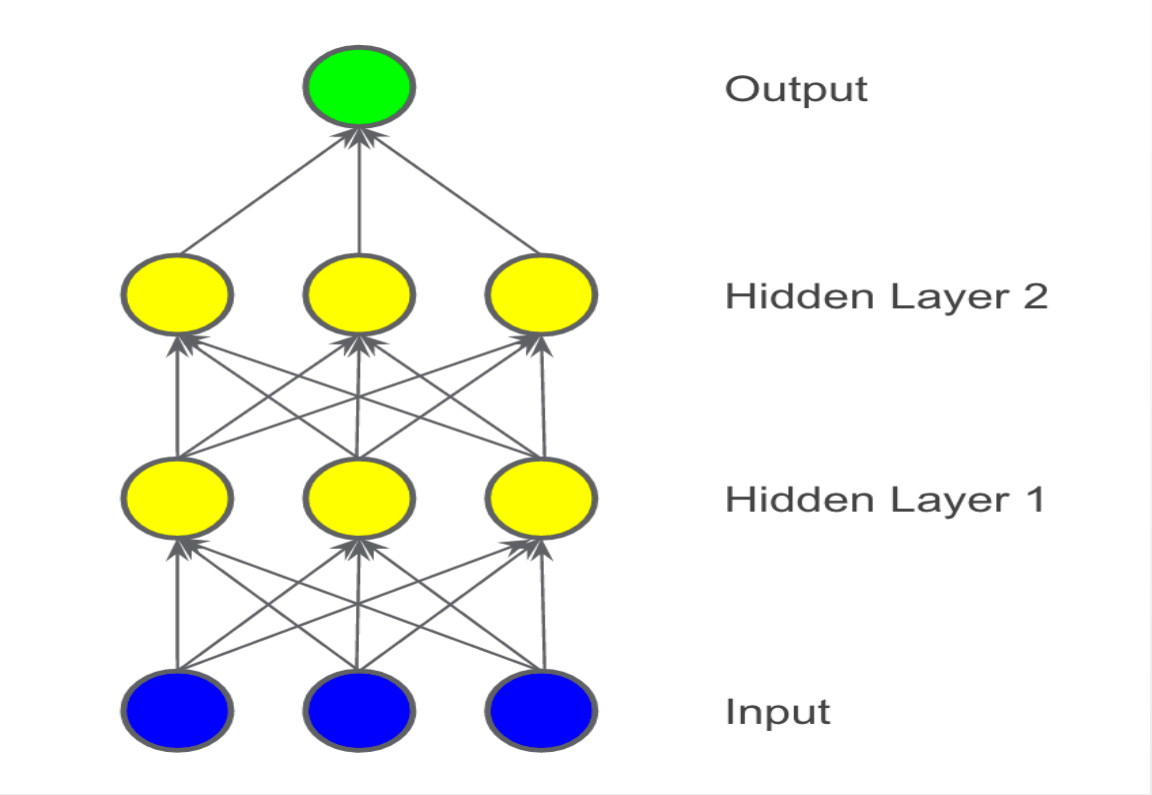


How to Deal with Nonlinear Problems

We 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.



Getting more complex

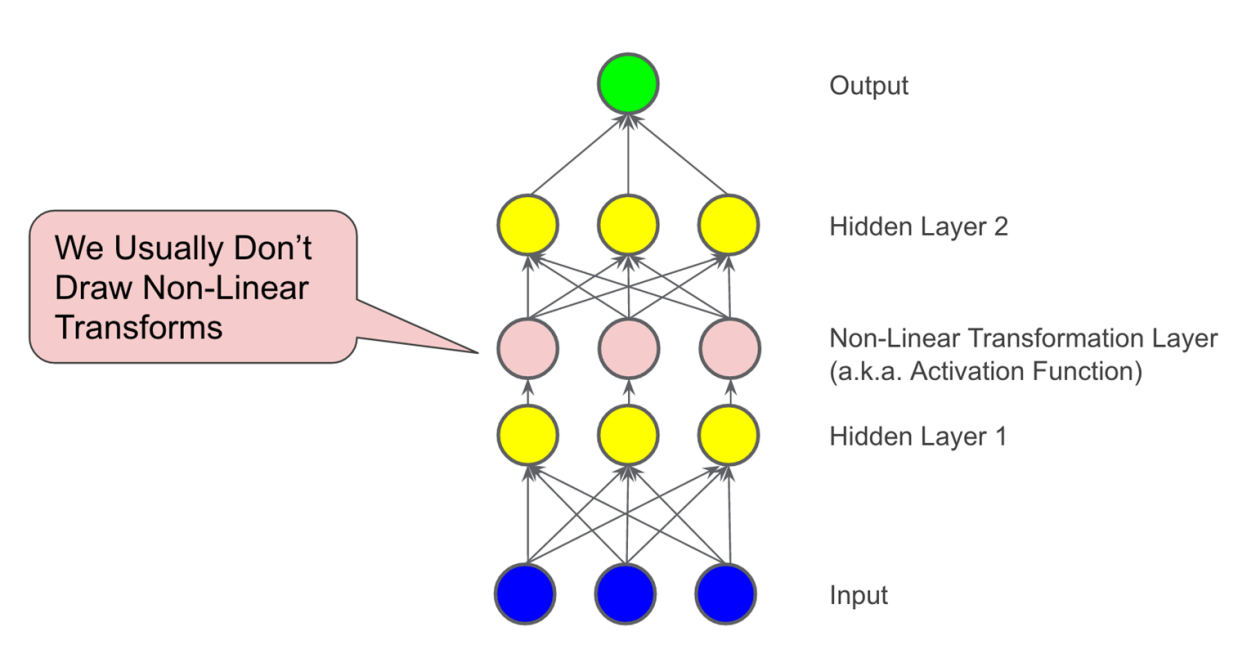


Is it still linear? What are we missing?

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.



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.

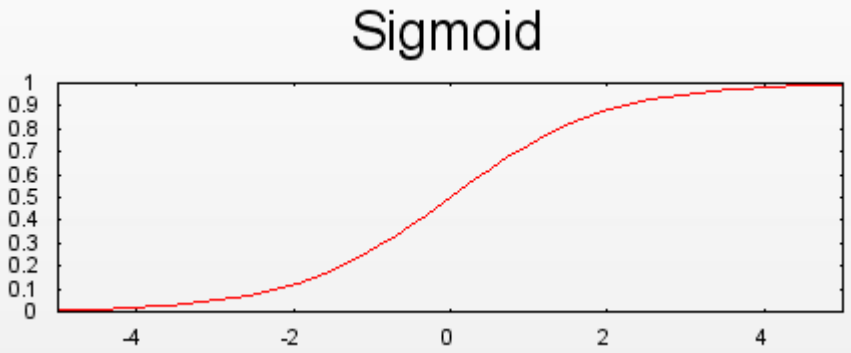

"Kaun hain ye log? Kaha se ate hain yeh?"
gifs.com

Common Activation Functions

Sigmoid:

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

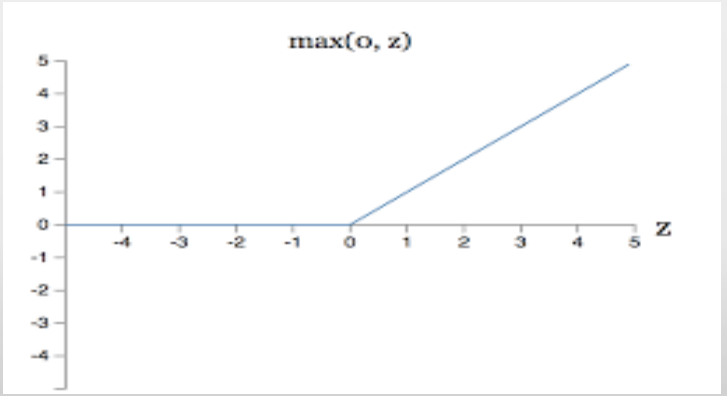
$$F(x) = \frac{1}{1 + e^{-x}}$$



ReLU:

It returns 0 if the value is negative and the input itself if positive.

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



Many More….

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:

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

There are many more popular activation functions like tanh, leaky- ReLU, randomized-leaky-ReLU etc.

How Do I Train My Model?

BACKPROPAGATION


AND, A SHORT DISTANCE AWAY...
MY FAULT--ALL MY FAULT! IF ONLY I HAD STOPPED HIM WHEN I COULD HAVE! BUT I DIDN'T--AND NOW --UNCLE BEN-- IS DEAD...


AND A LEAN, SILENT FIGURE SLOWLY FADES INTO THE GATHERING DARKNESS, AWARE AT LAST THAT IN THIS WORLD, WITH GREAT POWER THERE MUST ALSO COME--GREAT RESPONSIBILITY!

NOT SO PERFECT

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.

NOT SO PERFECT

Dead ReLU Units:

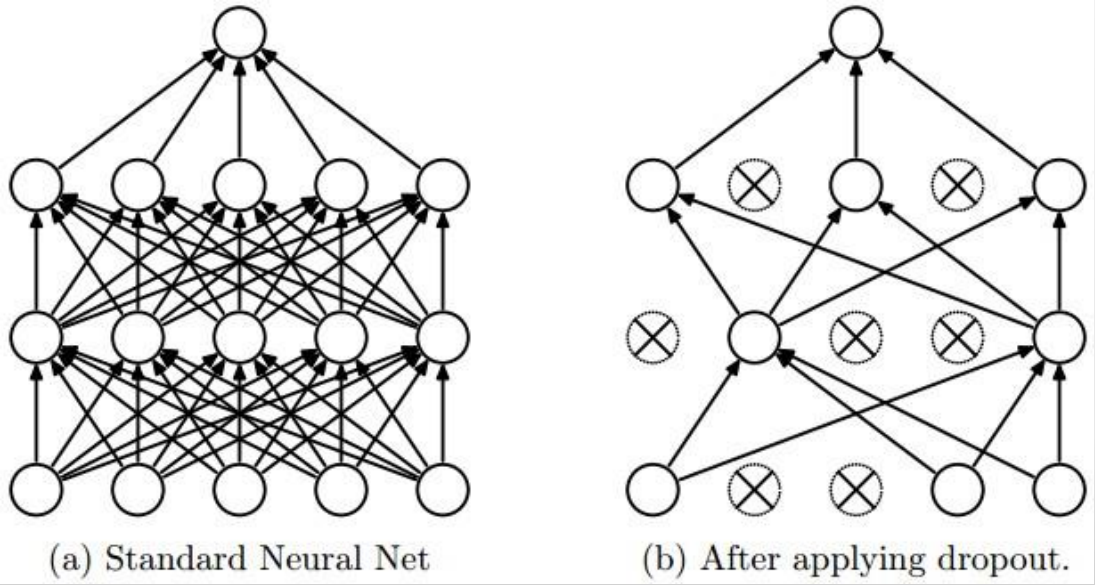
Once the weighted sum for a ReLU unit falls below 0, the ReLU unit can get stuck. It outputs 0 activation, contributing nothing to the network's output, and gradients can no longer flow through it during backpropagation. With a source of gradients cut off, the input to the ReLU may not ever change enough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



Multi- Class Neural Networks

Till now we have looked at problems pertaining to binary classification (spam/ not spam).

Consider the example that I have the facial database of all the participants. Now, a model has to be created that detects whether a person is Mayank or not. This is a binary classification problem and can be modeled by simple neural nets.

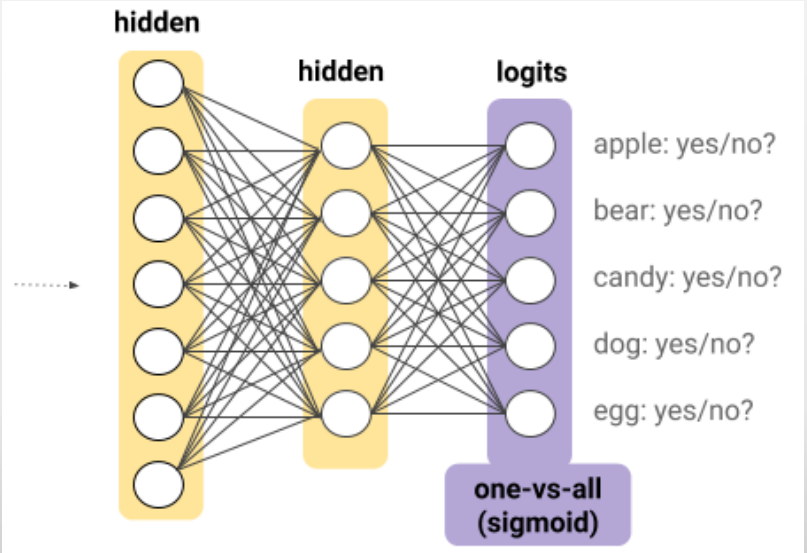
Now what if I want my model to recognize who the person is? How do I model this using neural nets?

One vs All

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).

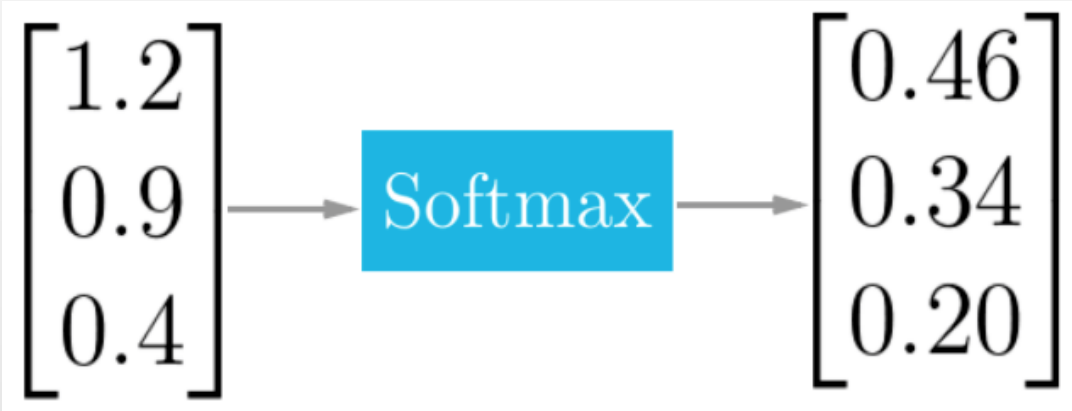
This approach is fairly reasonable when the total number of classes is small, but becomes increasingly inefficient as the number of classes rises.



Softmax

Now the problem with sigmoid function in multi-class classification is that the values calculated on each of the output nodes may not necessarily sum up to one.

The softmax function used for multi-classification model returns the probabilities of each class.

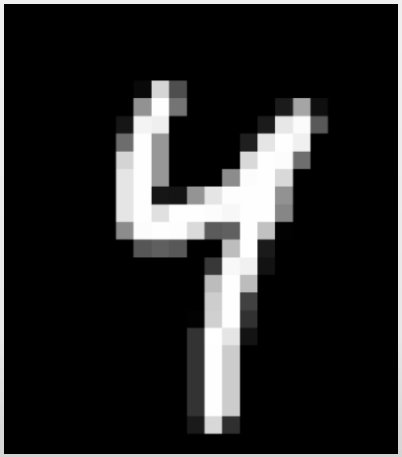


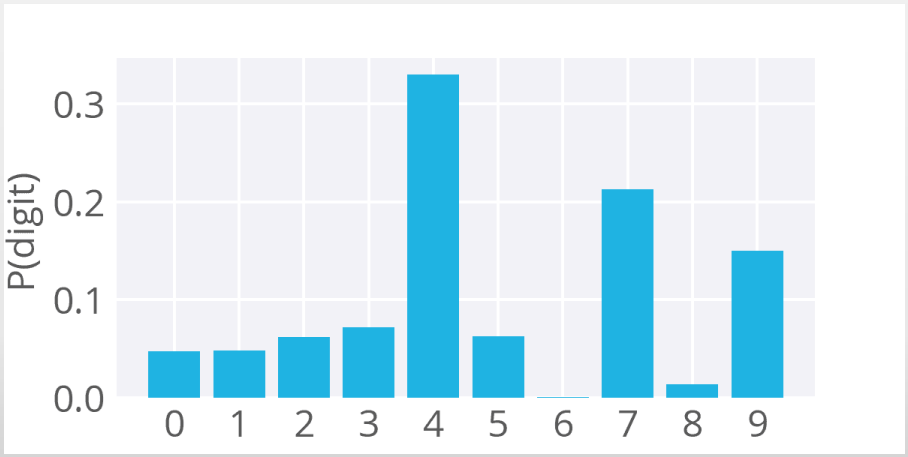
$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Softmax

To understand this better, think about training a network to recognize and classify handwritten digits from images. The network would have ten output units, one for each digit 0 to 9. Then if you fed it an image of a number 4 , the output unit corresponding to the digit 4 would be activated.

Building a network like this requires 10 output units, one for each digit. Each training image is labeled with the true digit and the goal of the network is to predict the correct label. So, if the input is an image of the digit 4, the output unit corresponding to 4 would be activated, and so on for the rest of the units.





Variants of Softmax

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. This can also be used for multi-label classification

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.

Candidate Sampling

Say we have a multiclass or multilabel problem where each training example (x_i,T_i) consists of a context x_i a small (multi)set of target classes T_i out of a large universe L of possible classes.

We wish to learn a compatibility function F(x, y) which says something about the compatibility of a class y with a context x . “Exhaustive” training methods such as softmax and logistic regression require us to compute F(x, y) for every class $y \in L$ for every training example. When |L| is very large, this can be prohibitively expensive.

“Candidate Sampling” training methods involve constructing a training task in which for each training example (x_i,T_i) , we only need to evaluate for a small set of candidate classes F(x, y) for a small set of candidate classes $C_i \subset L$. Typically, the set of candidates C_i is the union of the target classes with a randomly chosen sample of other classes $S_i \subset L$.

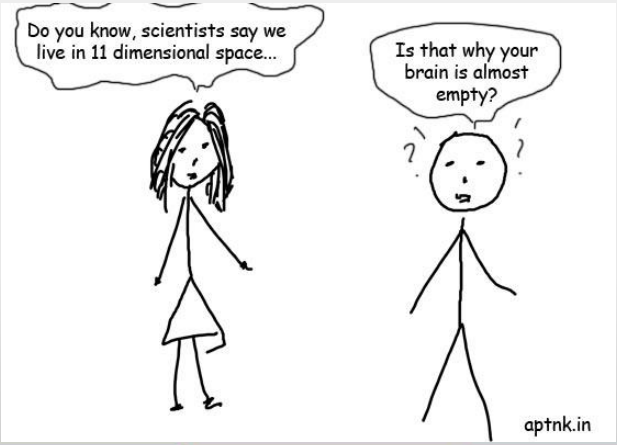
The training algorithm takes the form of a neural network, where the layer representing F(x, y) is trained by backpropagation from a loss function.

The Curse of Dimensionality

Till now we have been talking about problems that were on a very small scale. The input data had limited amount of features and the labels too were few in number.

Consider a data with 10000 dimensions. Train models on data of high number of dimensions would be computationally expensive and may even take days to train. This is what machine learning engineers call "**The Curse of Dimensionality**"

Moreover, there may be a dataset which is represented in a high dimensional form when it doesn't even require it...**Sparse Data.**



Embeddings

To tackle the curse of dimensionality and sparsity, we find a solution in **embeddings**. Embeddings are low dimensional representation of values that originally were in much higher dimension.

Yet another problem is that of discrete data. Neural Nets or any other model for that matter rely on numbers for training. Textual data, on the other hand, is not numerical. One possible solution would be to represent each word as a dimension and do **one-hot encoding**.

" Oxford English Dictionary contains full entries for 171,476 words in current use, and 47,156 obsolete words."

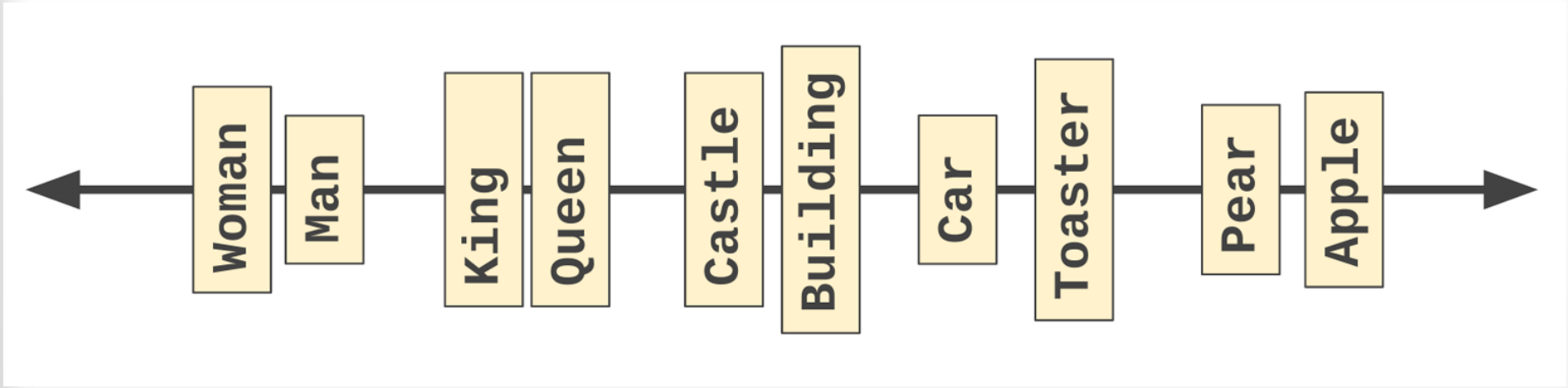
which is not feasible.

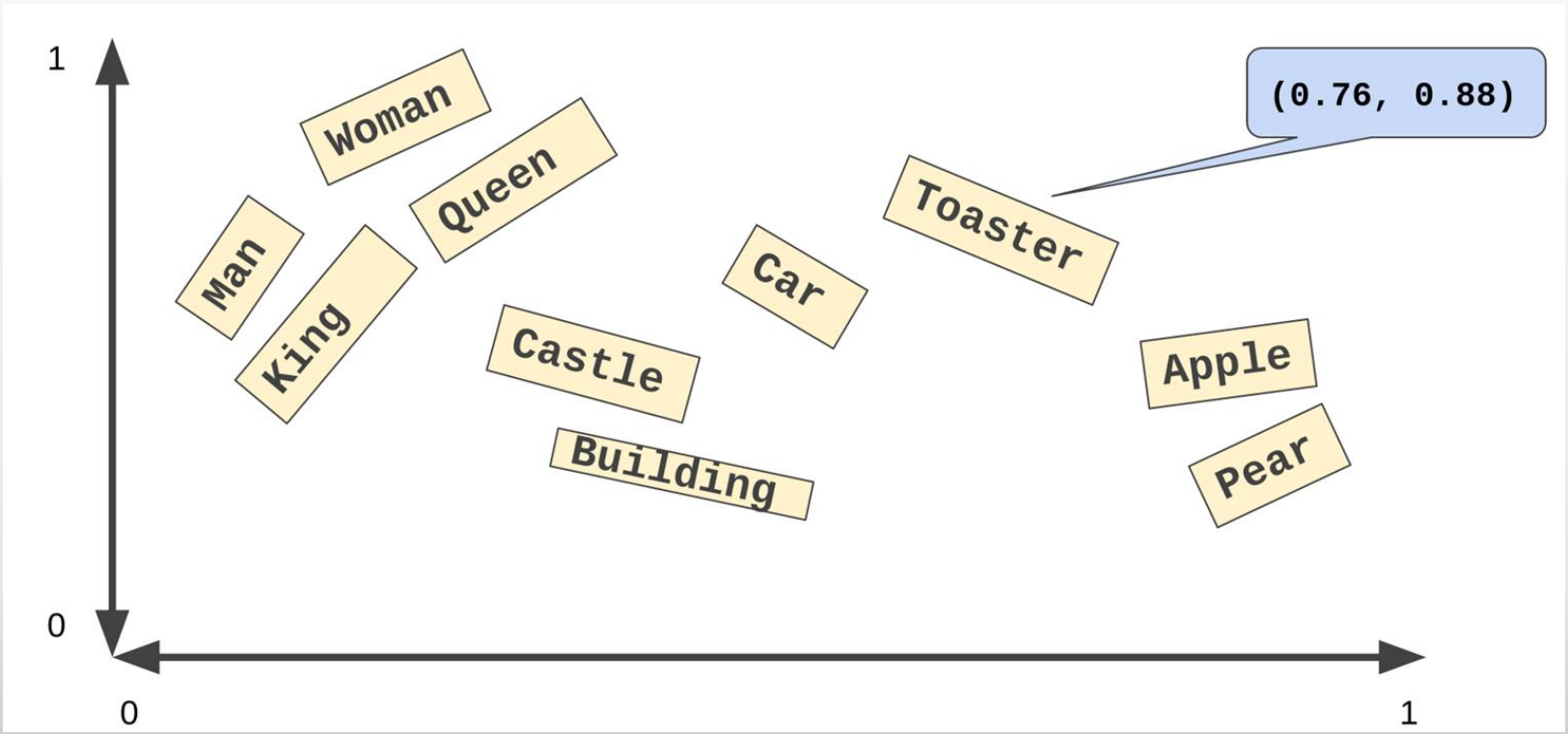
So, we use something known as **word embeddings.**

Exercise

On a piece of paper, try to arrange the following words on a one-dimensional number line and then on a two-dimensional plane so that the words nearest each other are the most closely related:

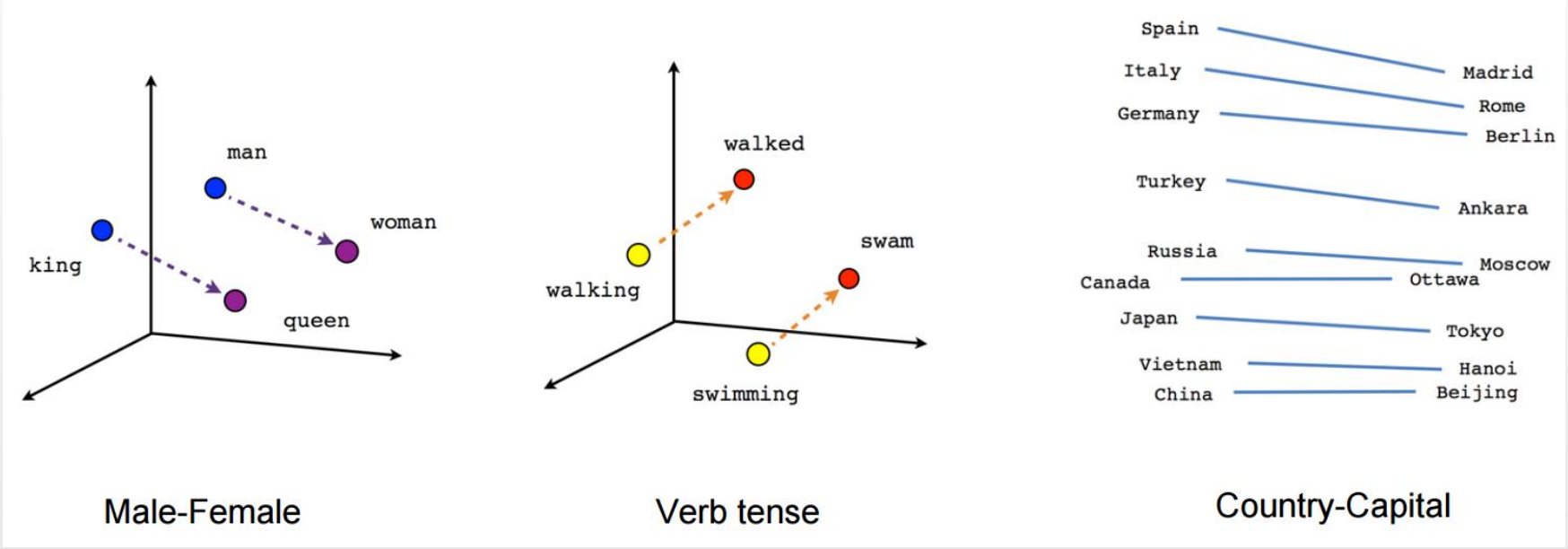
Apple
Building
Car
Castle
King
Man
Pear
Queen
Woman
Toaster


Woman
Man
King
Queen
Castle
Building
Car
Toaster
Pear
Apple


1
0
0
1
Woman
Queen
Man
King
Castle
Building
Car
Toaster
(0.76, 0.88)
Apple
Pear

Word Embeddings

As you can see from the exercise, even a small multidimensional space provides the freedom to group semantically similar instances together and keep dissimilar instances far apart. Position in the vector space can encode meaning.



The Mathematics

The operation of converting data from higher number of dimensions to lower number of dimensions can be interpreted as 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.

How do we get E?

The Origins of E

Principal Component Analysis:
Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

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