

# Machine Learning from Scratch

Hacking ML without Libraries

# Section I

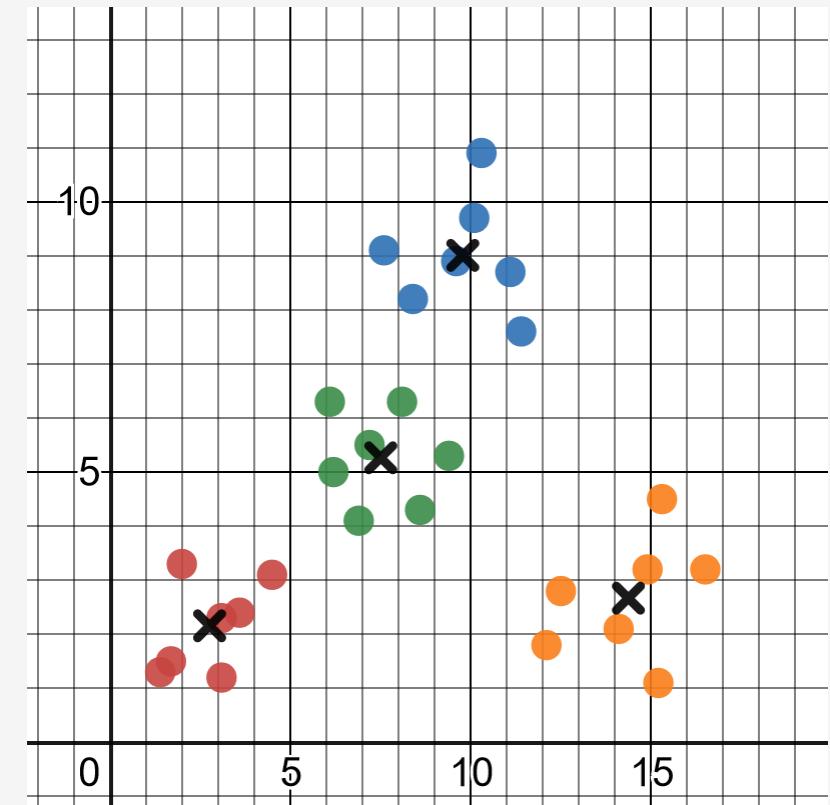
## Overview

# Agenda

---

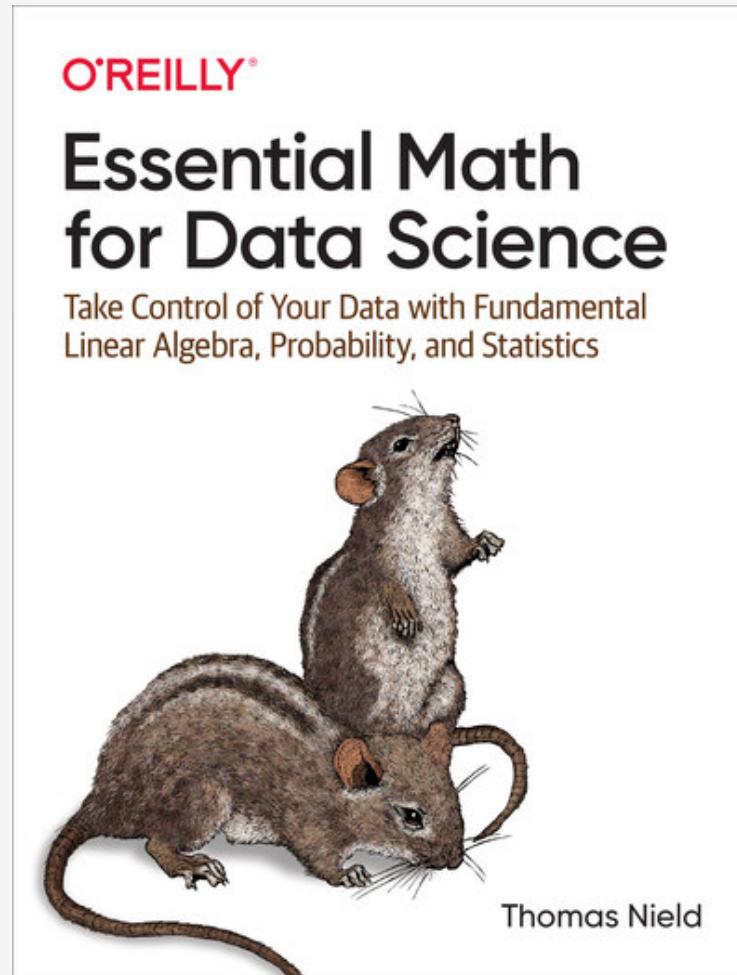
Here is what we will do for the next 4 hours:

- 1 **Introduction** and what to expect
- 2 **Linear Regression** and **K-Means Clustering**
- 3 Text classification with **Naive Bayes**
- 4 Binary classification with **Logistic Regression**
- 5 **Decision trees** and **Random Forests**
- 6 Brief Introduction to **Neural Networks**



# Supplemental Material

---



# What to Expect

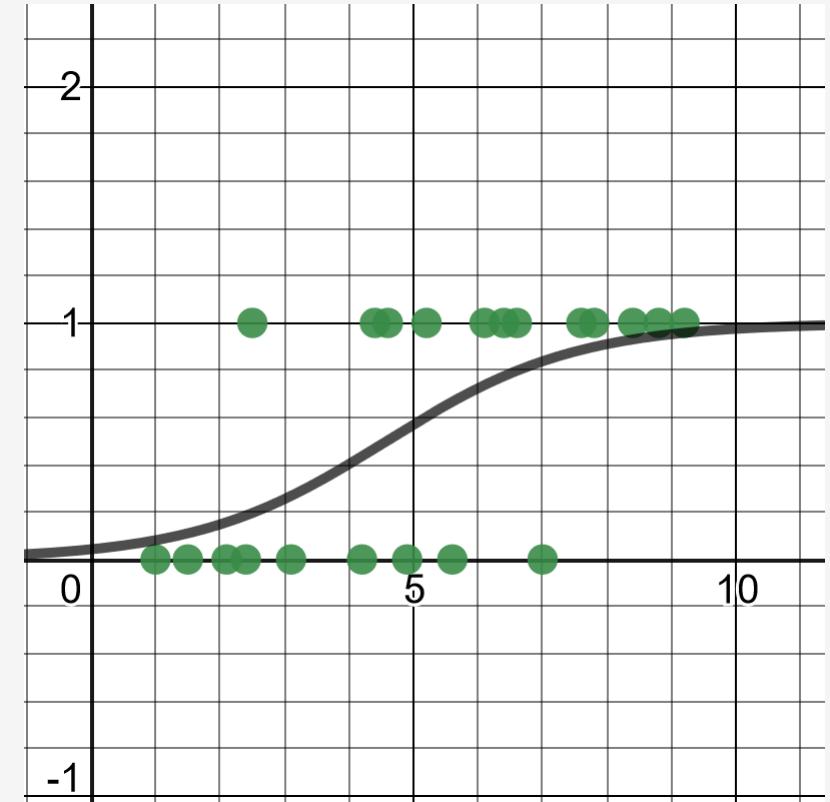
---

**How to build machine learning algorithms from scratch to develop intuition and understanding.**

**Remove some unnecessary barriers to get started with machine learning:**

- No calculus or partial derivatives
- Little to no linear algebra
- All Python code

**While we may take some unconventional approaches, we will be doing actual machine learning.**



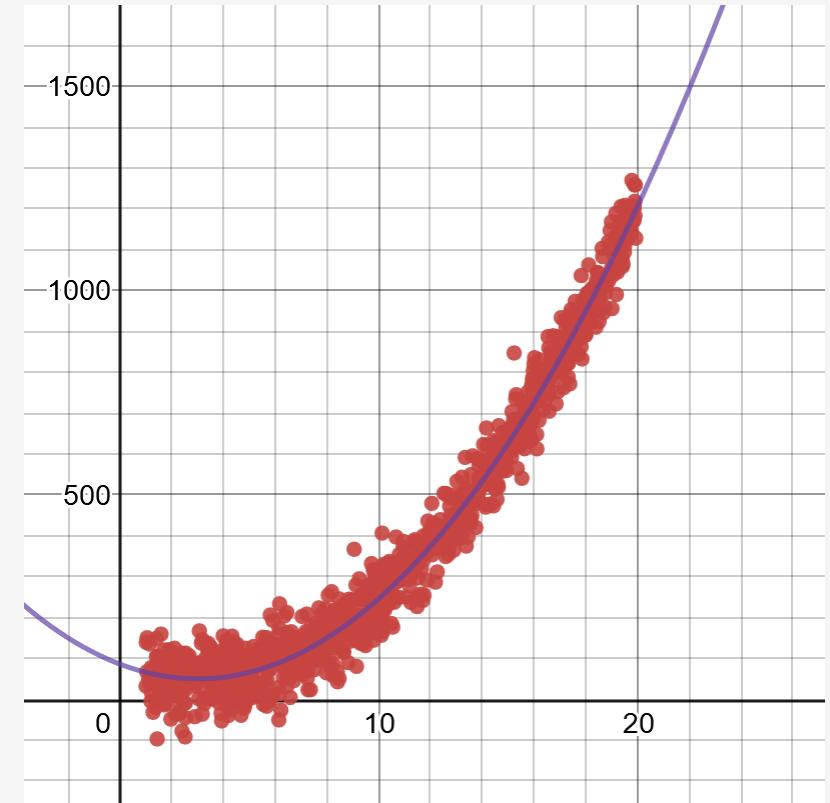
# What Not to Expect

---

**We will not be building state-of-the-art machine learning algorithms especially in regards to performance.**

- We will primarily be using vanilla Python to build these algorithms, which has a high performance cost.
- Linear algebra libraries like NumPy can be used to gain efficiencies, and I will strive to include these alternate implementations in the code files.

**We will not be validating and analyzing our machine learning models formally, and instead focus on building the ML algorithms.**

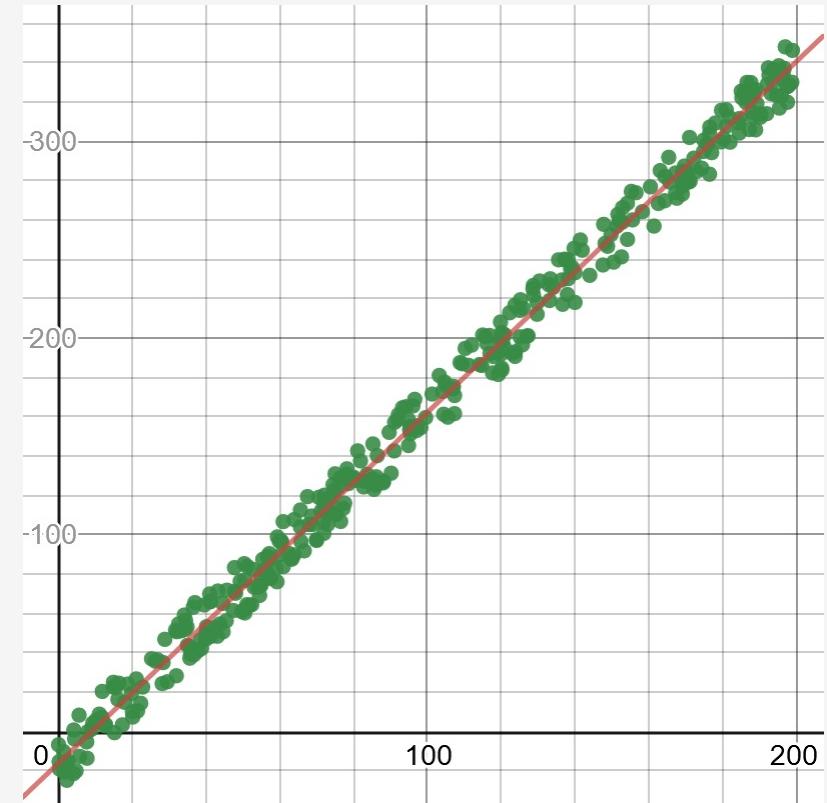


# What Not to Expect

---

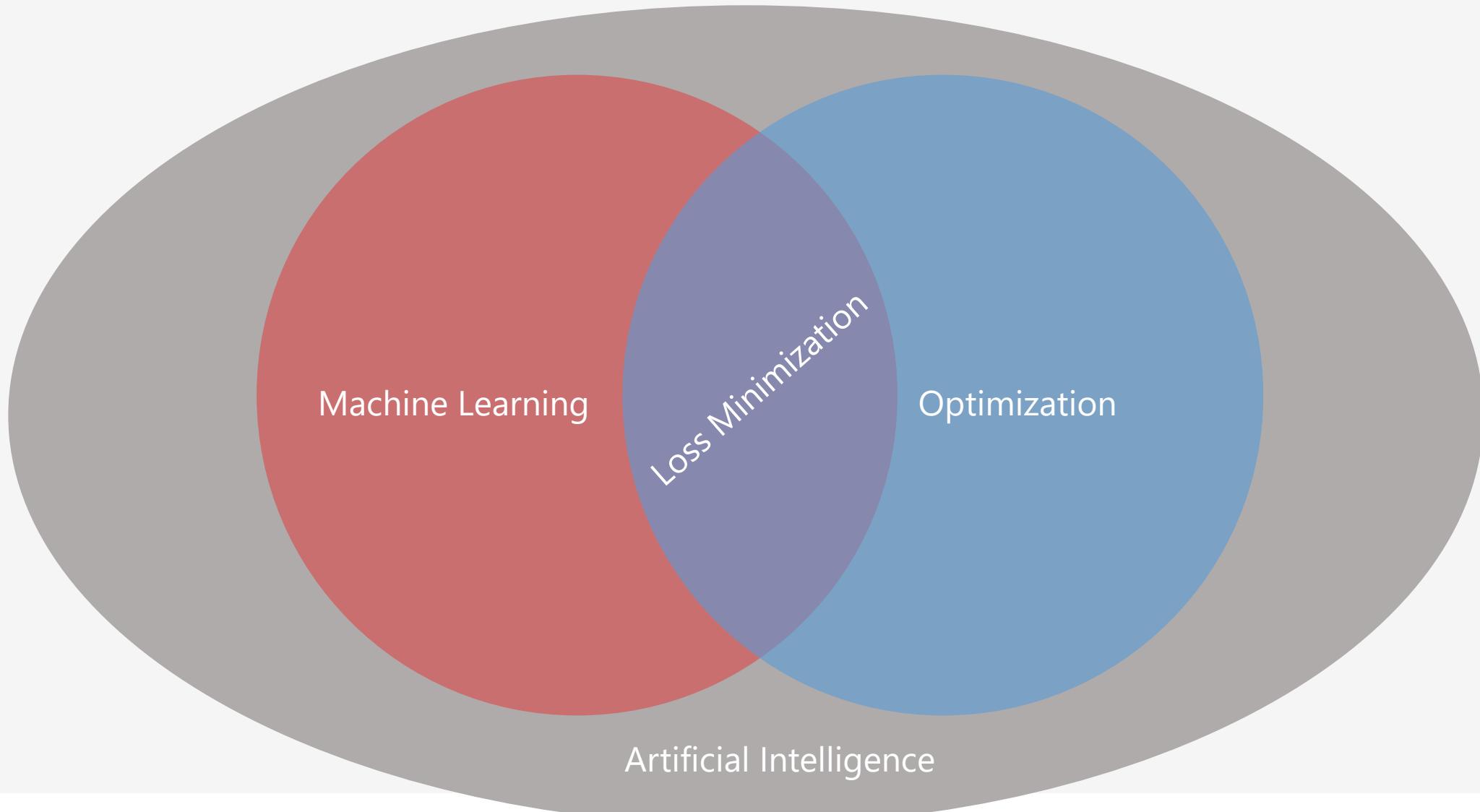
**For model fitting we will not be using conventional methods like gradient descent and calculus.**

- Instead we will use simple hill-climbing algorithms.
- Optimization is optimization regardless if it is hill-climbing, simulated annealing, genetic algorithms, or gradient descent.
- Hill climbing is the simplest optimization algorithm to implement, and works remarkably well for basic machine learning.



# Machine Learning vs Optimization vs AI

---



# Programming Languages and Performance

---

We will build some exciting from-scratch models in Python.

Python is a great, easy-to-learn language, but it is ***really*** slow because it is dynamically typed and interpreted.

- Pure Python models will struggle to scale, even with modestly sized data sets of thousands of records.
- This is going to be the cost of learning machine learning without libraries like NumPy.
- NumPy is necessary to scale Python because it is efficiently written in C, but it is also esoteric and distracting for first-time learners.



# Programming Languages and Performance

---

Java, Kotlin, Scala, Swift, C, C++, C#, Go, and other statically typed/compiled languages will perform much faster than Python.

- You will have more success scaling machine learning algorithms built from scratch with these platforms.
- If you truly enjoy building machine learning algorithms from scratch, please learn one of these platforms!

At this level with our toy examples, Python is going to work just fine.



# Section II

## Linear Regression and K-Means Clustering

# Simple Linear Regression

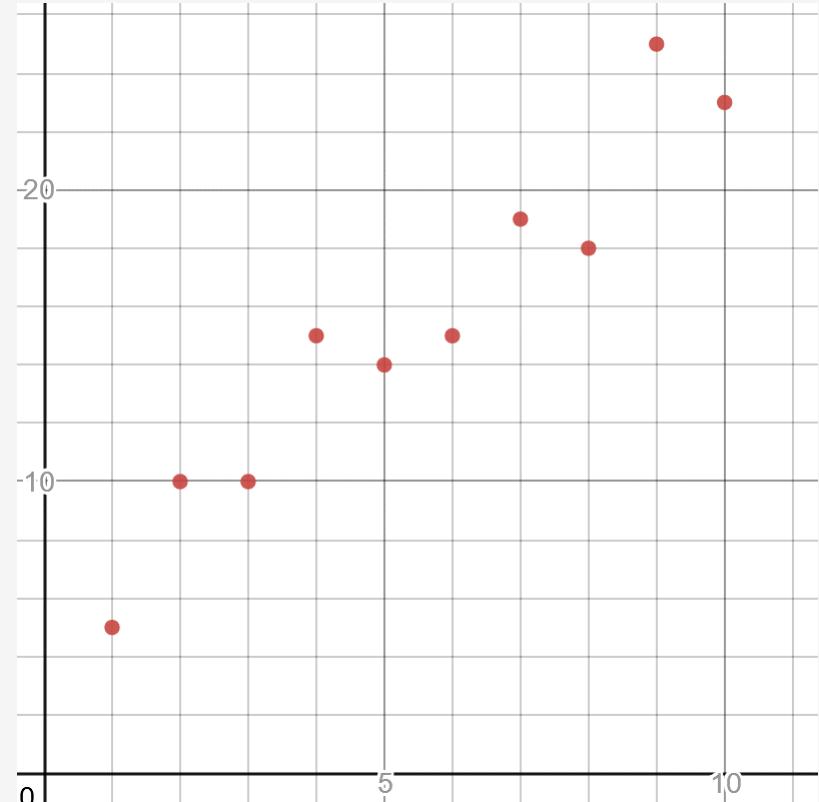
---

In practicality, machine learning is often about two tasks: regression and classification.

Let's start with regression, particularly the simplest one called **linear regression** which finds the best fit straight line through some points.

Here is a simple 2-dimensional plot of two variables, where x is independent and y is dependent.

- **Independent variables** are observed values that will serve as inputs into a function.
- **Dependent variables** are the outputted variables derived off the independent variables.



<https://www.desmos.com/calculator/fmhotfn3qm>

# Simple Linear Regression

---

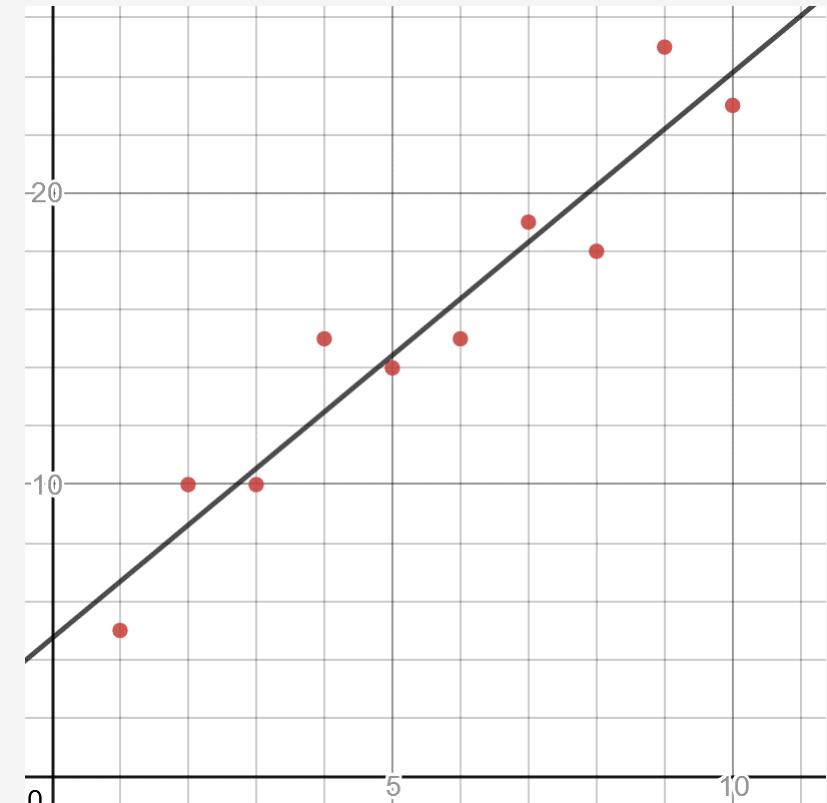
For a simple linear regression, we want to find a function  $y = mx + b$  that best fits to these points.

$$y = mx + b$$

We already know the  $x$  and  $y$  values from our existing data (the red points).

So the missing information is “what  $m$  and  $b$  values will create the best fit line”?

But before we solve for the  $m$  and  $b$  values, let’s first ask “what defines a best fit anyway?”



<https://www.desmos.com/calculator/fmhotfn3qm>

# Simple Linear Regression

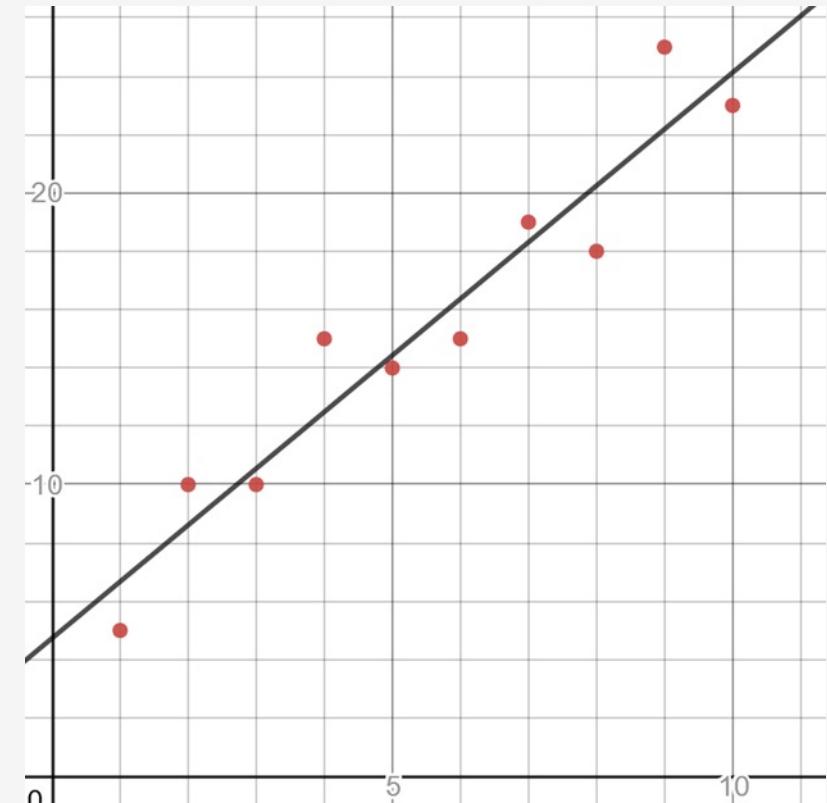
---

Pretend you drew any straight line through the points.

You will not be able to fit a perfect line, because the points do not exist on a straight line.

- Machine learning models are never perfect, as real-world data is never perfect.
- But you can fit a line to estimate a new  $y$  value for a given  $x$  value, even if there is an inevitable margin of error called **loss**.

Even if there is loss, it can be helpful in estimating predictions, such as how much  $y$  growth there will be at  $x$  time in the future.



<https://www.desmos.com/calculator/fmhotfn3qm>

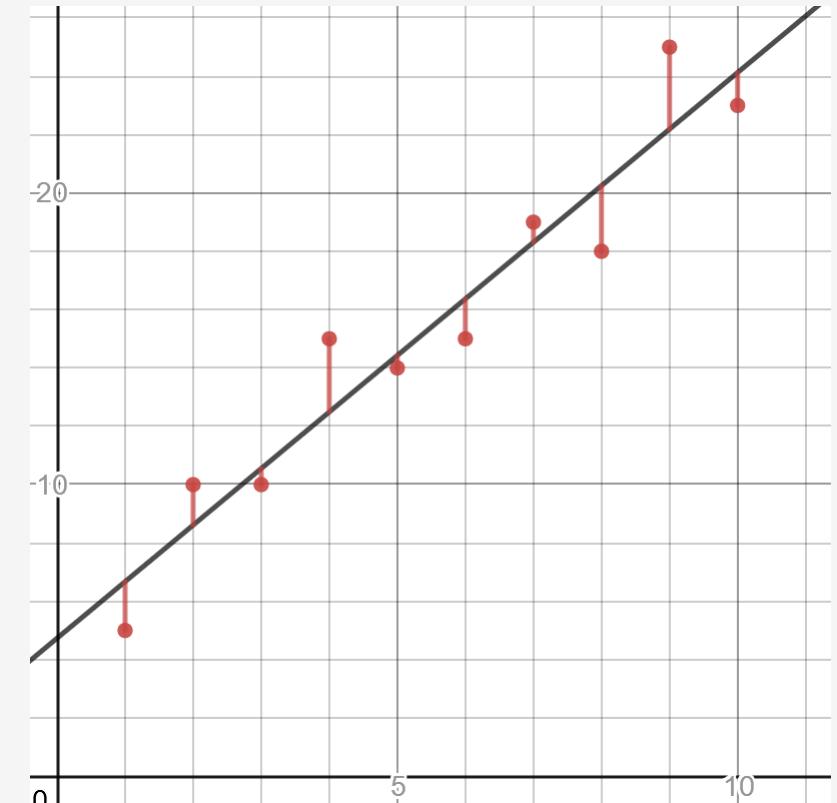
# Simple Linear Regression

---

When you plot a given line, notice that there is a difference between the predicted  $y$  values and actual  $y$  values with our observed data.

These are called **residuals**, and in machine learning we want to minimize them.

So how do we minimize residuals in aggregate?



<https://www.desmos.com/calculator/fmhotfn3qm>

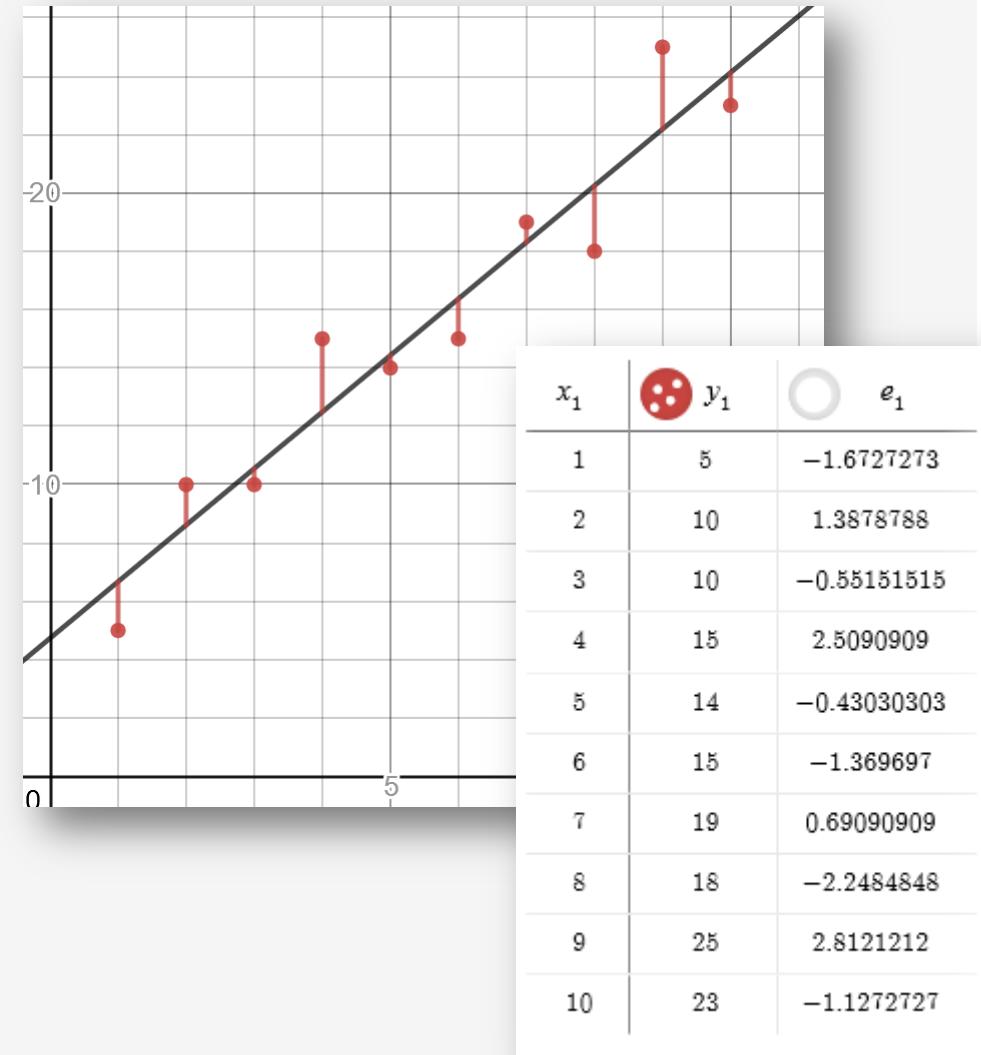
# Loss Function

---

When we solve for  $m$  and  $b$  values, we need an objective to minimize total residuals using a **loss function**.

- We do not necessarily want to sum up the residuals, as the negatives will cancel out the positives.
- We can sum the absolute values, but that does not amplify larger residuals, and absolute values are mathematically difficult to work with.

So what is the best approach to total all the residuals to evaluate the quality of the fit?



# Sum of Least Squares

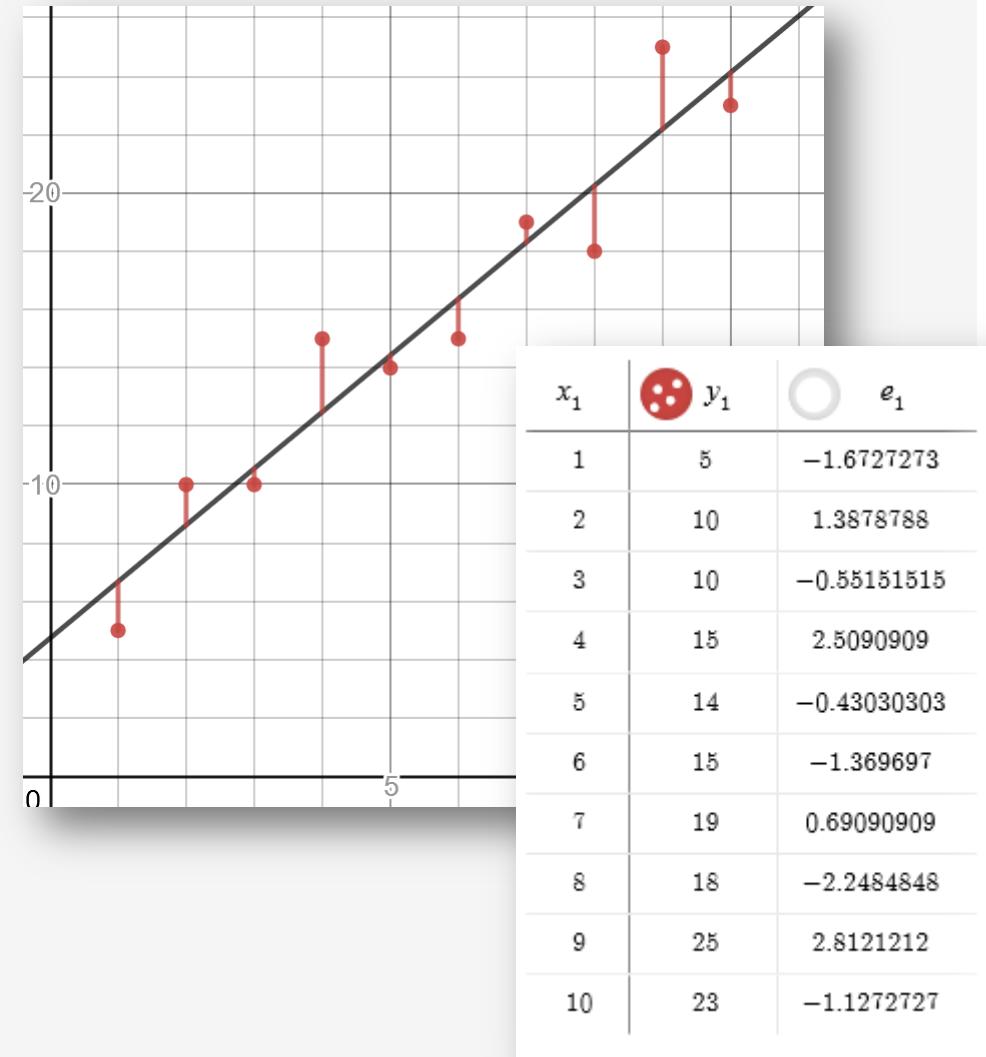
---

**The best approach:** we can square the residuals and sum them!

- Squaring will penalize large residuals by making them even larger!
- Squaring will also conveniently turn negatives into positives.
- Although we are not going to dive into calculus, squares are much easier to take the derivative of.

We can then find the  $m$  and  $b$  values that will find the sum of least squares.

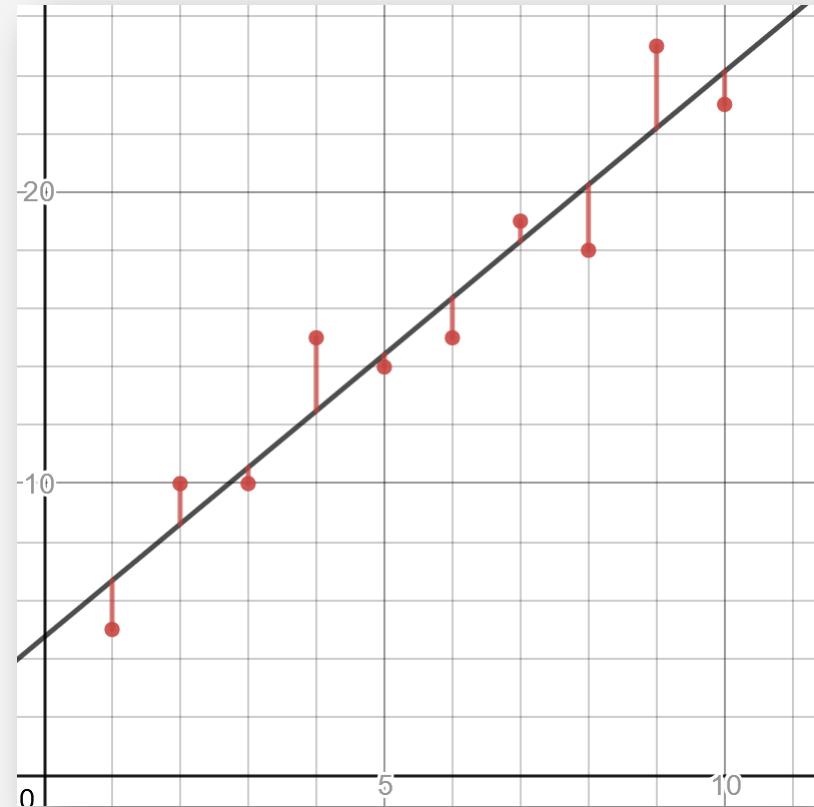
<https://www.desmos.com/calculator/fvrnuhw0hy>



# Calculating Sum of Squares

---

m	b			
1.93939	4.73333			
x	y	y_predict (mx+b)	residual	residual squared
1	5	6.67272	-1.67272	2.797992198
2	10	8.61211	1.38789	1.926238652
3	10	10.5515	-0.5515	0.30415225
4	15	12.49089	2.50911	6.295632992
5	14	14.43028	-0.43028	0.185140878
6	15	16.36967	-1.36967	1.875995909
7	19	18.30906	0.69094	0.477398084
8	18	20.24845	-2.24845	5.055527402
9	25	22.18784	2.81216	7.908243866
10	23	24.12723	-1.12723	1.270647473
Sum of Squares			28.0969697	



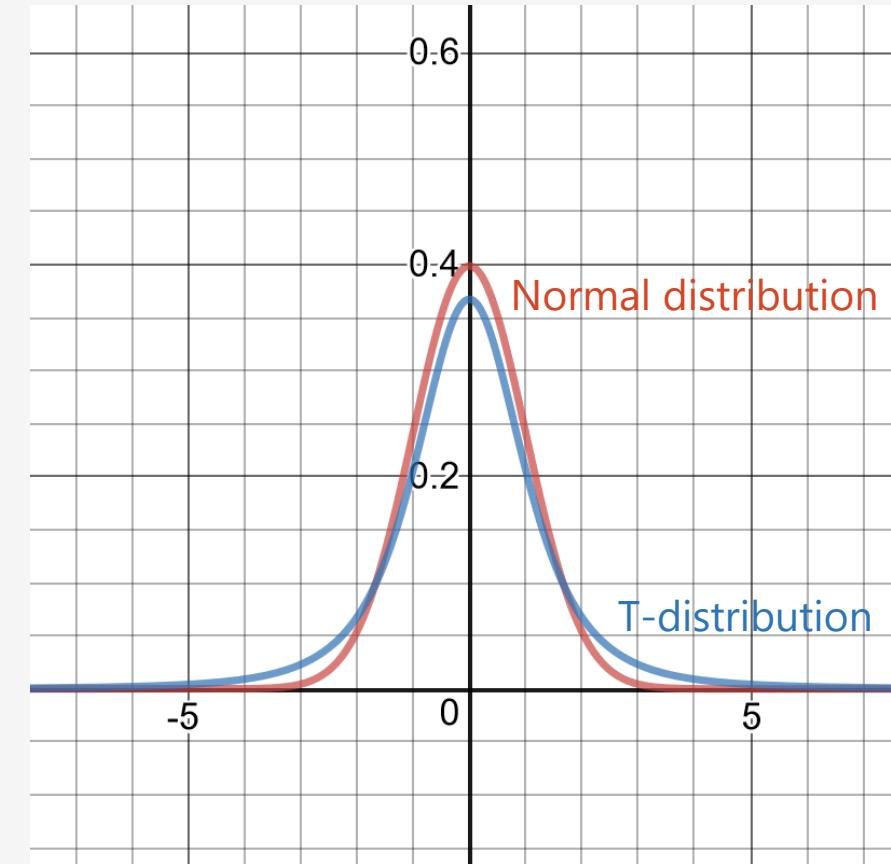
# Solving for a Fit

---

Now that we have a loss function (the sum of squared residuals), how do we find the **m** and **b** values to minimize it?

- We randomly increase/decrease **m** and **b** with random values from a standard normal distribution, or even better a T-Distribution which has fatter tails.
- Fatter tails = more smaller/larger values = more diverse moves.

We can use these random adjustments to **m** and **b** as our moves in **hill climbing**.



<https://www.desmos.com/calculator/xm56tvvalh>

# Hill Climbing – A Simple Optimization Algorithm

---

- 1 Start with a random or initial solution, even if it is poor quality.
- 2 Repeat the following steps for a number of iterations and/or until the solution cannot improve anymore.
  1. Select a random part of the solution and change it.
  2. If that results in an improvement, keep it.



# Hands-On: Simple Linear Regression

The screenshot shows a Python code editor interface with the following details:

- Project Tree:** The project is named "oreilly\_machine\_learning\_from\_scratch". It contains a "code" folder which includes several files: "k\_means\_two\_variables.py", "k\_means\_two\_variables\_hc.kt", "k\_means\_two\_variables\_hc.py", "linear\_regression\_hc.py", "linear\_regression\_numpy\_hc.py", "multiple\_linear\_regression\_hc.py", "quadratic\_quantile\_regression.py", "quadratic\_regression\_hc.py", "quantile\_regression\_hc.py", and "square\_root\_hc.py". There are also "section\_ii", "section\_iii", "section\_iv", "section\_v", and "playground.py" files.
- Code Editor:** The "linear\_regression\_hc.py" file is open. The code implements stochastic gradient descent for linear regression. It includes imports for numpy, defines a "Point" class, and contains a main loop that iterates over "epochs". The code uses random adjustments to the slope (*m*) and intercept (*b*) and calculates the total mean squared error for each point.
- Terminal:** A terminal window titled "Run: linear\_regression\_hc" shows the output of the script. It prints five different linear regression equations, each consisting of a slope and an intercept, separated by newlines. The equations are:

```
y = 1.9479609782812204x + 4.700427862797459
y = 1.939005588895919x + 4.71908143579035
y = 1.944800557041935x + 4.708139399718407
y = 1.9416140460943887x + 4.72531427596339
y = 1.9416140460943443x + 4.72531427596339
```
- Status:** Below the terminal, a message states "Process finished with exit code 0".

# Multiple Linear Regression

---

Solving for a function  $y = \mathbf{mx} + \mathbf{b}$  is elementary as it only has one independent variable  $x$ .

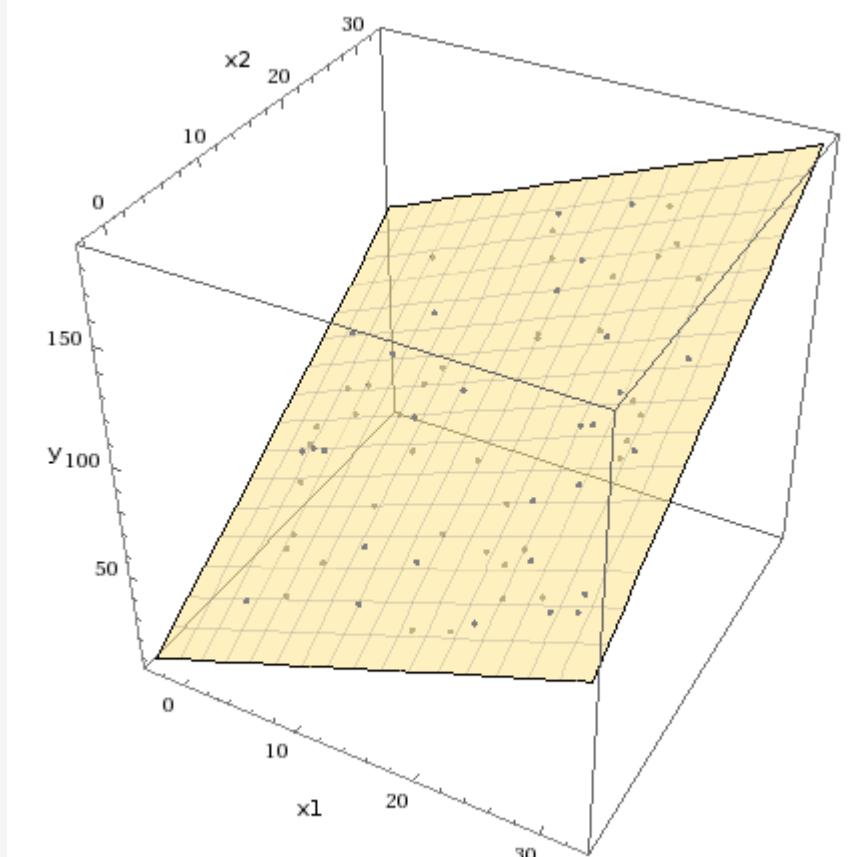
We can also solve for multiple independent variables, like  $x_1$ ,  $x_2$ ,  $x_3$ , and so on...

Let's say we have columns of independent variables  $x_1$  and  $x_2$ , and a dependent variable  $y$ .

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

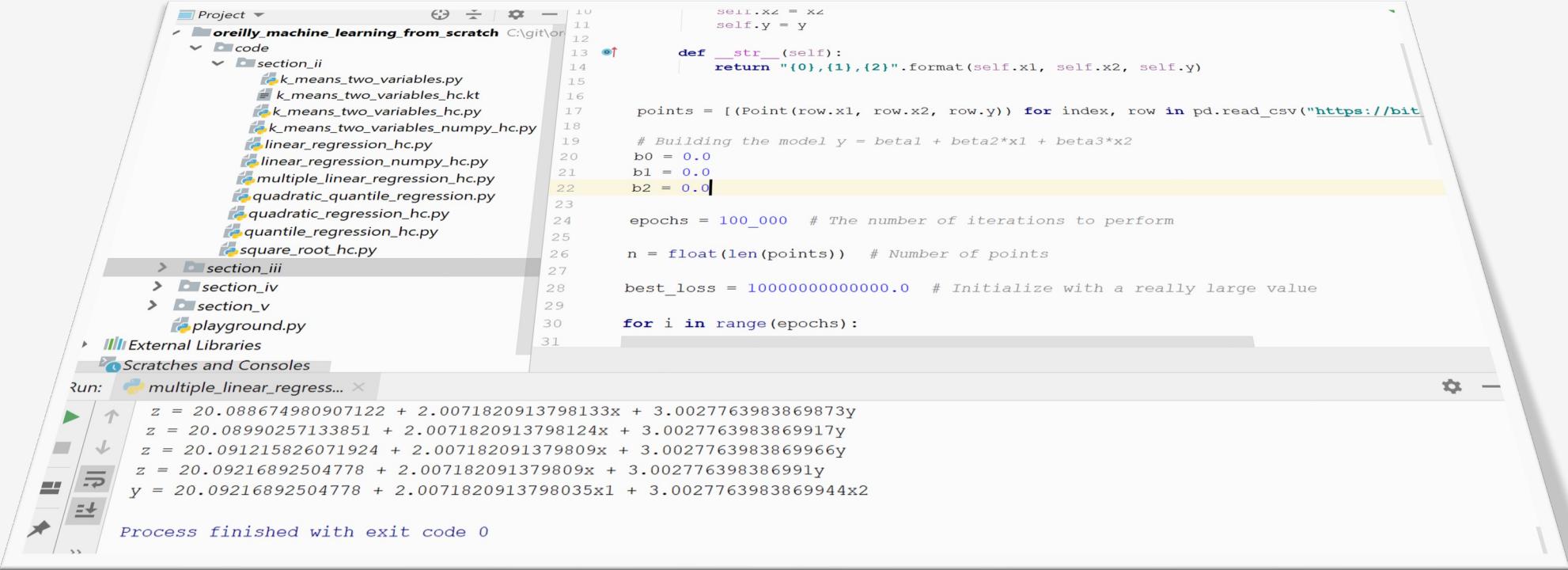
Instead of  $\mathbf{m}$  and  $\mathbf{b}$  constants, we now need to solve for each  $\beta$  parameter, where  $\beta_0$  is the y-intercept and  $\beta_1$  and  $\beta_2$  are slopes for the respective  $x$  variables.

We can use the same hill-climbing technique as before!



Computed by Wolfram|Alpha

# Hands-On: Multiple Linear Regression



The screenshot shows a code editor interface with a project structure on the left and a code editor window on the right.

**Project Structure:**

- oreilly\_machine\_learning\_from\_scratch
- code
  - section\_ii
    - k\_means\_two\_variables.py
    - k\_means\_two\_variables\_hc.kt
    - k\_means\_two\_variables\_hc.py
    - k\_means\_two\_variables\_numpy\_hc.py
    - linear\_regression\_hc.py
    - linear\_regression\_numpy\_hc.py
    - multiple\_linear\_regression\_hc.py
    - quadratic\_quantile\_regression.py
    - quadratic\_regression\_hc.py
    - quantile\_regression\_hc.py
    - square\_root\_hc.py
  - section\_iii
  - section\_iv
  - section\_v
  - playground.py

**External Libraries**

**Scratches and Consoles**

**Run:** multiple\_linear\_regress... (running)

**Code Editor Content:**

```
10
11
12
13 self.x2 = x2
14 self.y = y
15
16
17 def __str__(self):
18     return "{0},{1},{2}".format(self.x1, self.x2, self.y)
19
20 points = [(Point(row.x1, row.x2, row.y)) for index, row in pd.read_csv("https://bit.ly/2QrJfDl").iterrows()]
21 # Building the model y = betal + beta2*x1 + beta3*x2
22 b0 = 0.0
23 b1 = 0.0
24 b2 = 0.0
25
26 epochs = 100_000 # The number of iterations to perform
27
28 n = float(len(points)) # Number of points
29 best_loss = 1000000000000.0 # Initialize with a really large value
30
31 for i in range(epochs):
```

**Output Console:**

```
z = 20.088674980907122 + 2.0071820913798133x + 3.0027763983869873y
z = 20.08990257133851 + 2.0071820913798124x + 3.0027763983869917y
z = 20.091215826071924 + 2.007182091379809x + 3.0027763983869966y
z = 20.09216892504778 + 2.007182091379809x + 3.002776398386991y
y = 20.09216892504778 + 2.0071820913798035x1 + 3.0027763983869944x2

Process finished with exit code 0
```

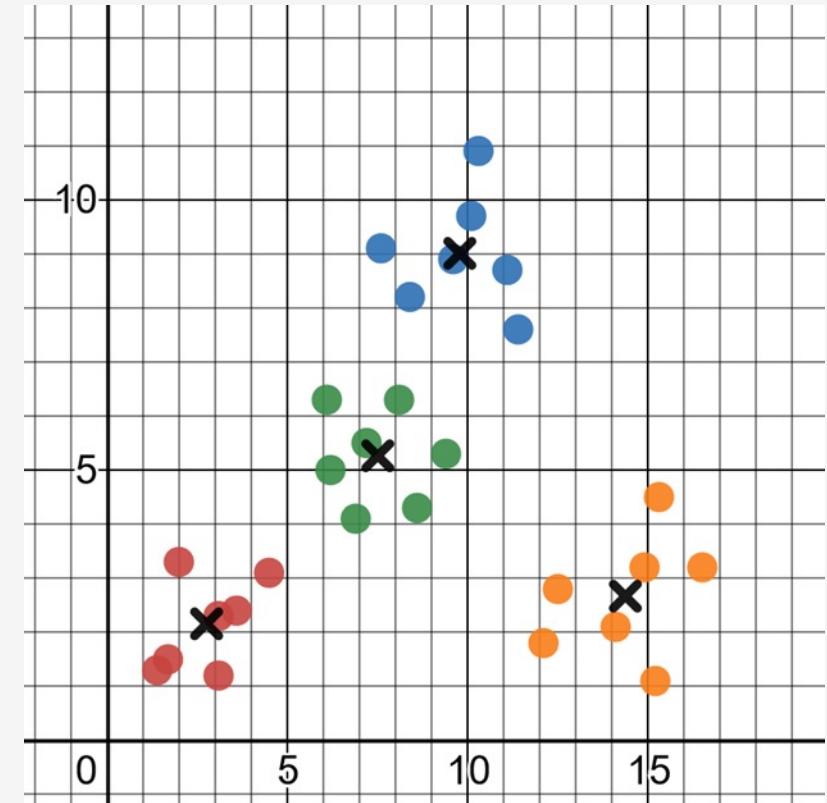
# Clustering

---

Taking our exploration of machine learning with hill-climbing further, let's talk about k-means clustering!

**Clustering** groups up points based on a specified number of centroids.

- A **centroid** is a best-fit center for a cluster of points.
- To the right is some data, and we can easily eyeball there are approximately 4 groups of points.



# Clustering

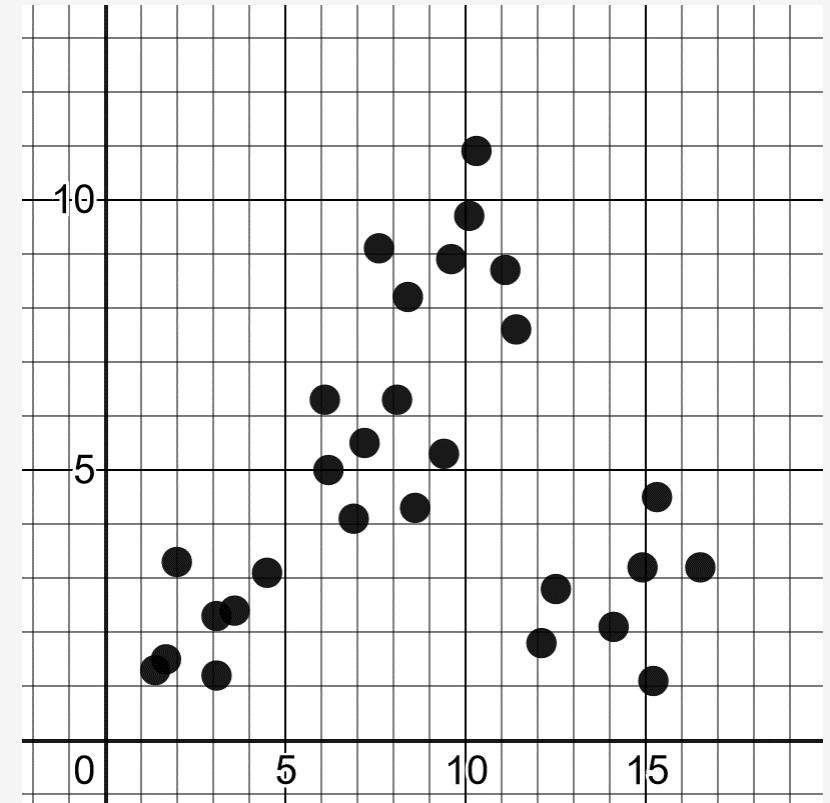
---

Let's start with four centroids ( $k = 4$ ) and use hill-climbing to center them on four clusters.

Each point will belong to the centroid it is closest to, and distance between a point and centroid is measured by:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Our loss function will be the sum of squared distances between the centroids and their closest points.



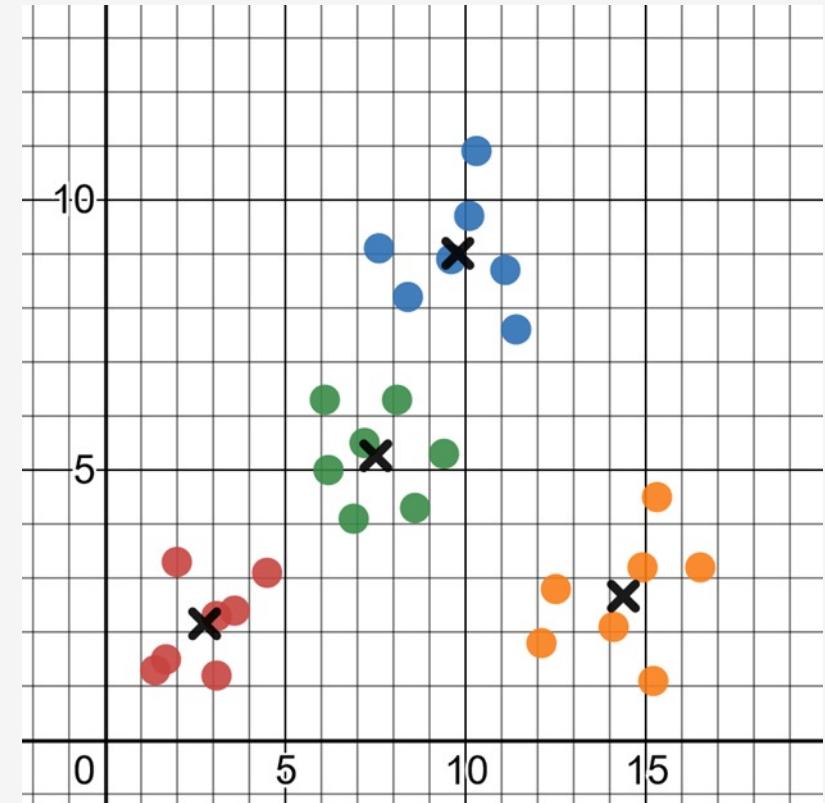
# Clustering

---

We will start each centroid at (0,0).

For each iteration, we will:

- 1 Select a centroid and randomly adjust its  $x$  and  $y$  value.
- 2 Calculate the distances between each point and their closest centroid, square them, and sum them.
- 3 If the sum of squared distances is reduced, keep the change.
- 4 Otherwise, revert the centroid back to its previous position.

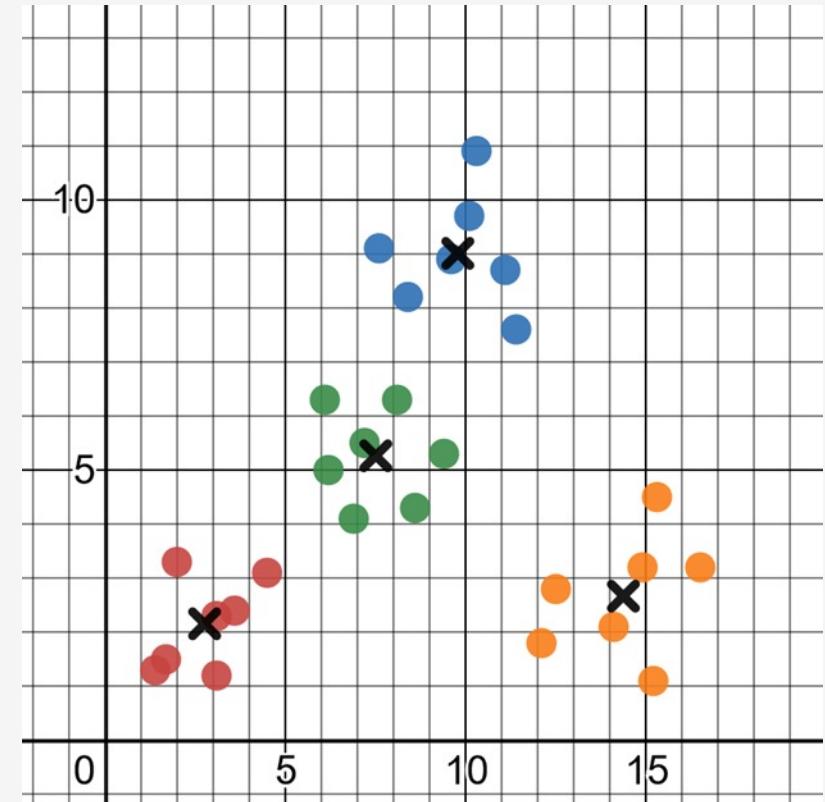


# K-Means Clustering

---

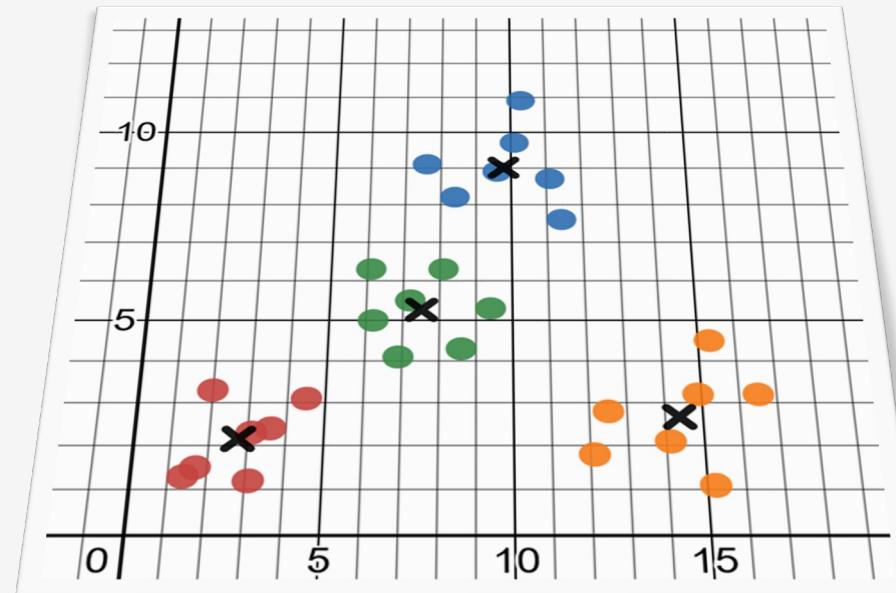
Of course, there's a more efficient heuristic (search strategy) we can use for k-means clustering.

- 1 Take the average  $x$  and  $y$  values for the points nearest each centroid (hence, "k-means"!)
- 2 Set each centroid to that average  $x$  and  $y$ .
- 3 Repeat until the centroids do not move anymore and are at the average of all their points.



# Hands-On: K-Means Clustering

---



# A Note About Overfitting

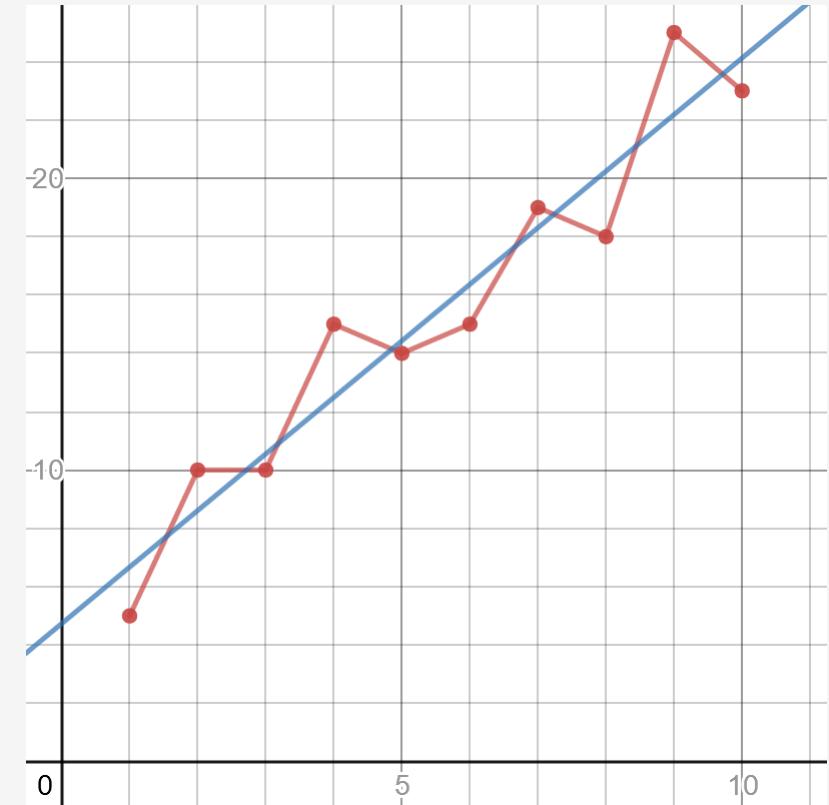
---

We are not going to focus too much on validating and analyzing machine learning models, but we should at least mention overfitting.

**Overfitting** means that our ML model works well with the data it was trained on but fails to predict correctly with new data.

- This can be due to many factors, but a common cause is the sampled data does not represent the larger population and more data is needed.
- The red line has high **variance**, meaning its predictions are sensitive to outliers and therefore can vary greatly.
- The blue line has high **bias**, meaning the model is less sensitive to outliers because it prioritizes a method (maintaining a straight line) rather than bend and respond to variance.

The red line to the right is likely overfitted (high variance, low bias), but the blue linear regression line (low variance, high bias) is less likely to be overfit.



<https://www.desmos.com/calculator/wmwfolbvdk>

# A Note About Overfitting

---

Linear regression is a highly biased method and is resilient to overfitting.

There are other remedies to mitigate overfitting, the most basic being separating **training data** and **test data**.

- The model is fit to the training data, and then is tested with the test data.
- If the test data performs poorly compared to the training data, there is a possibility of overfit (or just no correlation altogether).

You can also try to train with more data as well as utilize cross-validation, regularization, bagging, boosting, and other techniques.

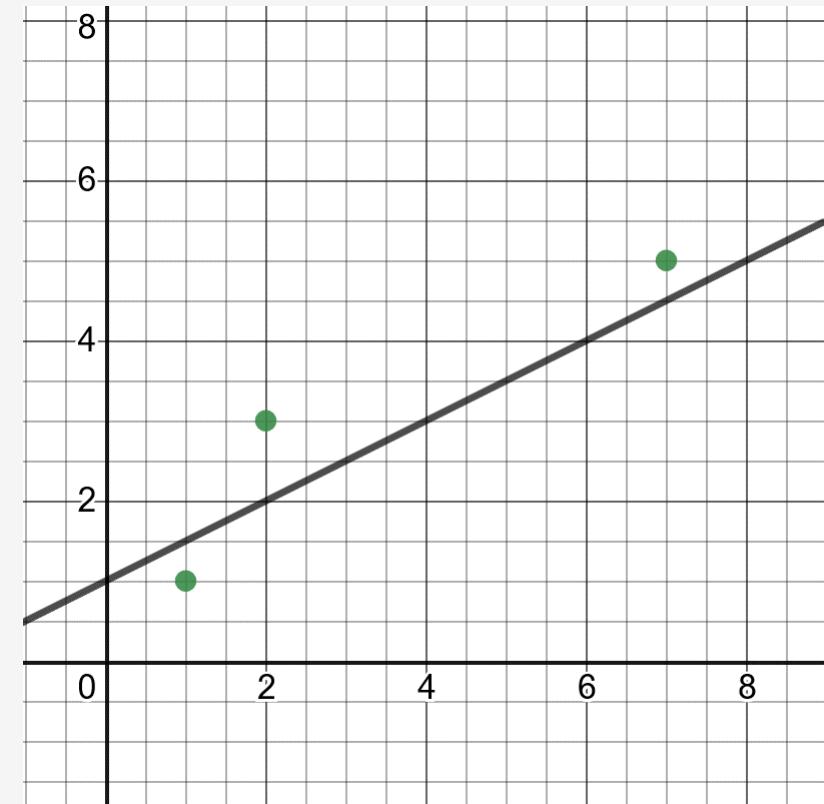


# Quiz Time!

---

You have three points  $(1,1)$ ,  $(2,3)$ , and  $(7,5)$ . You are testing a fit for a line  $y = \frac{1}{2}x + 1$ . What is the loss calculated by sum of squares?

- A) 1.5
- B) 1.75
- C) 1.0
- D) -1.5

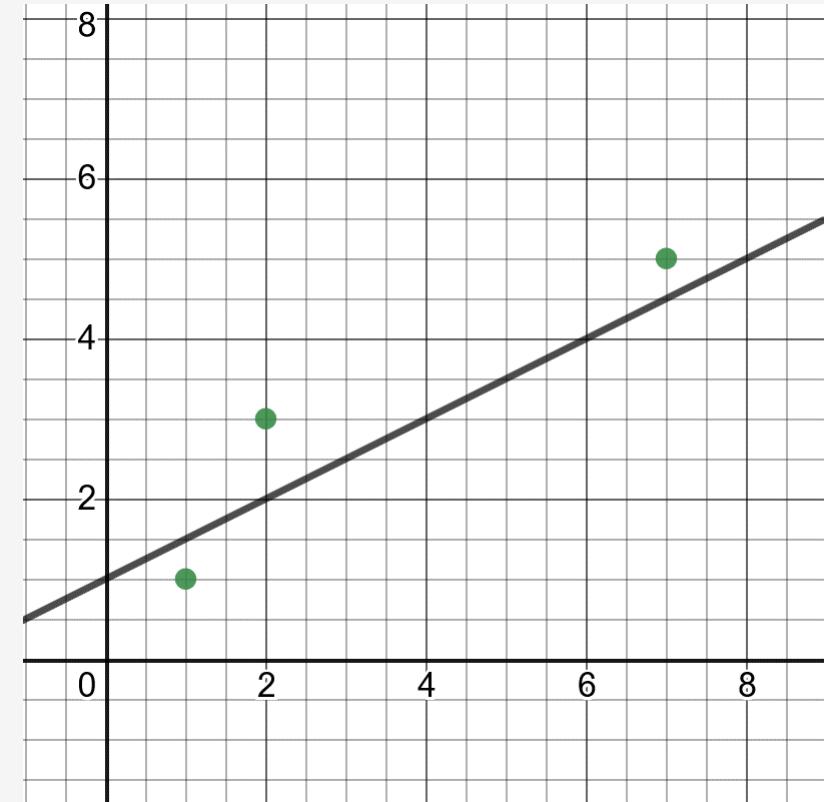


# Quiz Time!

---

You have three points  $(1,1)$ ,  $(2,3)$ , and  $(7,5)$ . You are testing a fit for a line  $y = \frac{1}{2}x + 1$ . What is the loss calculated by sum of squares?

- A) 1.5  $= (1 - (\frac{1}{2}1 + 1))^2 + (3 - (\frac{1}{2}2 + 1))^2 + 5 - (\frac{1}{2}7 + 1))^2$
- B) 1.75
- C) 1.0
- D) -1.5

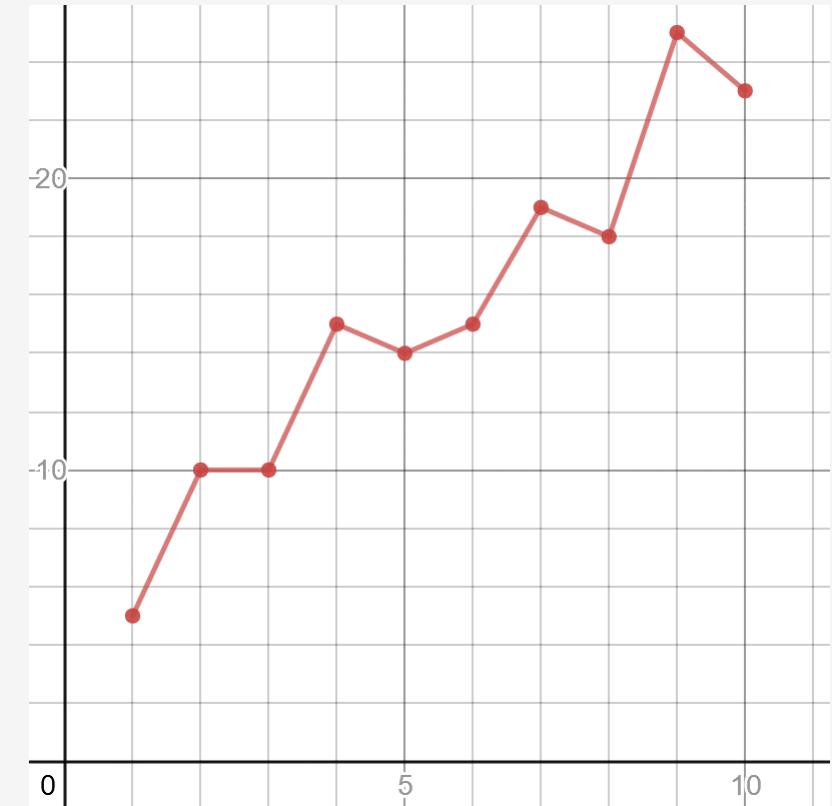


# Quiz Time!

---

It is good to fit a regression as close to the points as possible,  
even if you just connect the points as shown to the right.

- A) True
- B) False



# Quiz Time!

---

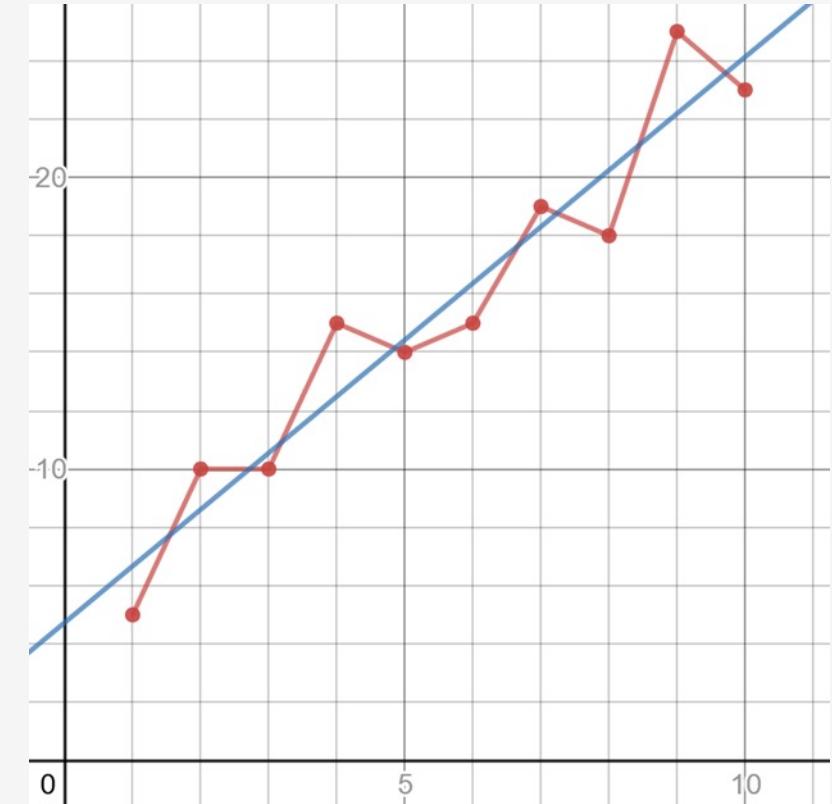
It is good to fit a regression as close to the points as possible, even if you just connect the points as shown to the right.

A) True

B) False

Doing a regression is a balancing act between fitting exactly to your data (variance) versus leaving some wiggle room (bias) to predict new data that will be different. Otherwise you will risk overfitting even if you have zero loss.

You might be better doing a linear regression (blue line) as shown to the right, which has a loss but is more likely to predict new data accurately.



# Section III

# Logistic Regression

# Classifying Things

---

**So far we just did regressions via linear regression.**

**However, linear regressions are awkward to use for classification.**

- Lines extend in a straight direction for positive infinity and negative infinity, well outside a range of acceptable values.
- Lines do not do a good job representing a probability and staying within the limits of 0.0 and 1.0.
- When doing classification, probability is a critical tool.



Hot dog: 0.92

# Classifying Things

---

**Classification tasks are pretty common in machine learning:**

- How do I classify images of *dogs* versus *cats*?
- Will a shipment be most likely be *late*, *early*, or *on-time*?
- Is this email *spam* or *not spam*?
- Will this movie get *1 star*, *2 stars*... or *5 stars*?



Hot dog: 0.92

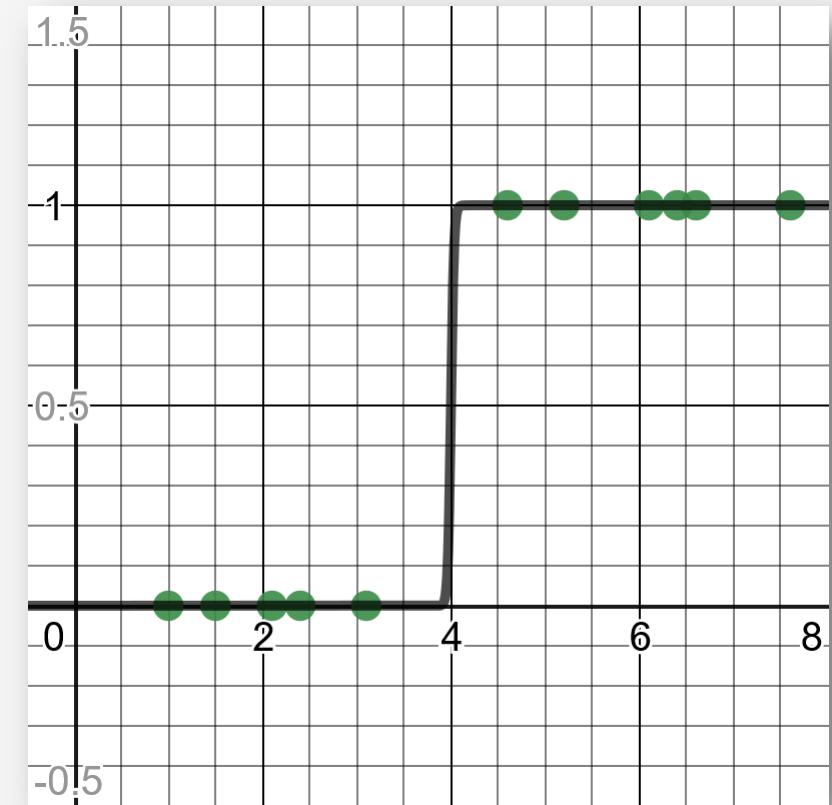
There are several machine learning algorithms that work well for classification, and we are going to learn first about Logistic Regression.

# Logistic Regression Intuition

---

Imagine you have 11 patients exposed to a chemical for  $x$  hours, and you plot whether they exhibited symptoms (1) or not (0).

Plotting our patient data (right), we can easily eyeball a clear cutoff at 4 hours where patients transition from ***not showing symptoms (0)*** to ***showing symptoms (1)***.



<https://www.desmos.com/calculator/prs2p0sofc>

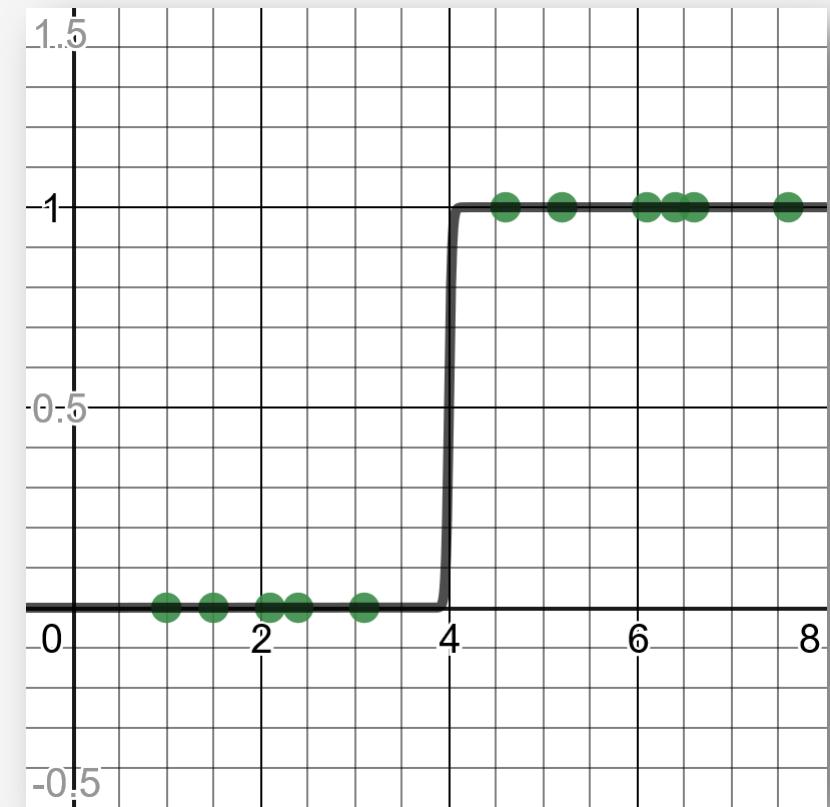
# Logistic Regression Intuition

---

This indicates any patient exposed for less than 4 hours will have a 0% chance of showing symptoms, but greater than 4 will have a 100% chance of showing symptoms.

Because there is a distinct separation at 4 hours, a logistic regression is going to “jump” from 0% to 100% at that boundary.

Of course, real life rarely works out this way...



<https://www.desmos.com/calculator/prs2p0sofc>

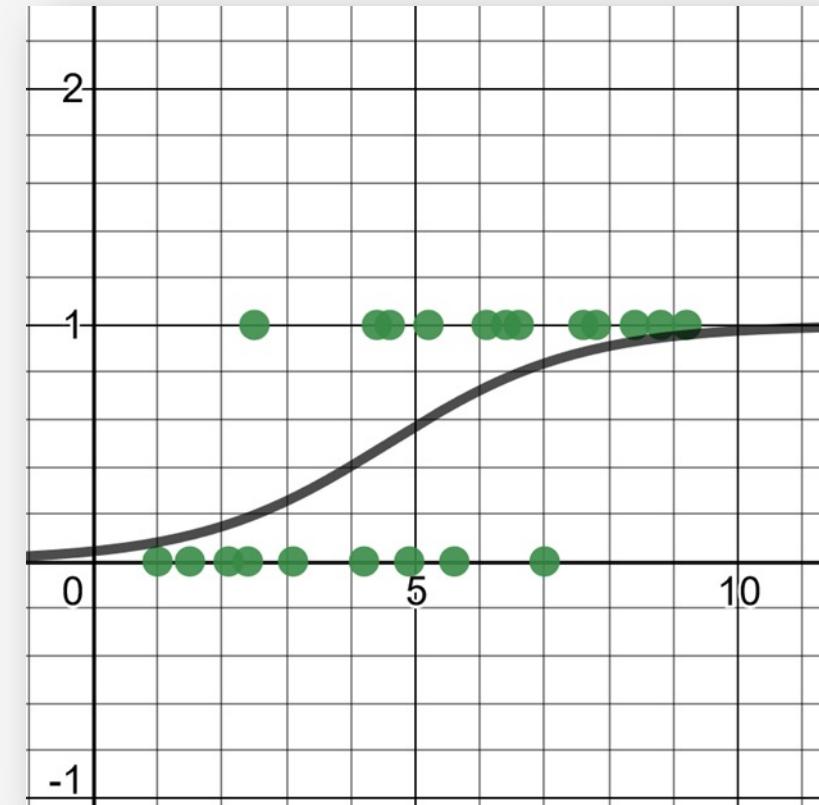
# Logistic Regression Intuition

---

Now let's say you gathered more data and got a realistic picture, where the middle of the range has a mix of patients showing symptoms and not showing symptoms.

The way to interpret this is the probability of patients showing symptoms gradually increases with each hour of exposure.

Because of this overlap of points in the middle, there is no distinct cutoff when patients show symptoms, but rather a gradual transition from 0% probability to 100% probability ("0" and "1").



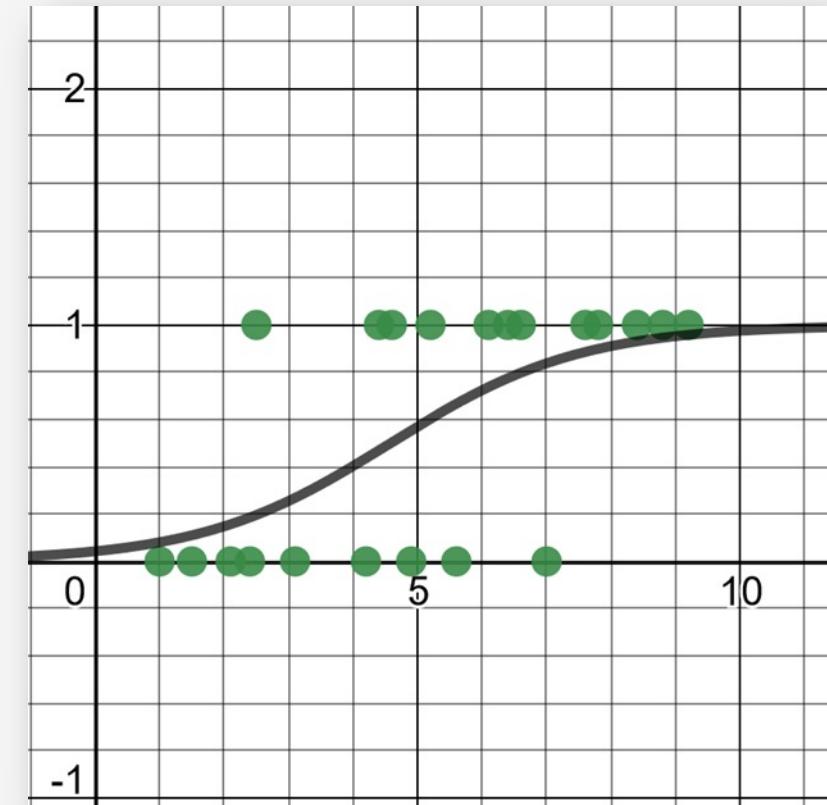
<https://www.desmos.com/calculator/bsqtqfians>

# Logistic Regression Intuition

---

More technically, a logistic regression results in a curve indicating a probability of belonging to the **true** (1) category, which in this case means ***a patient showed symptoms***.

As the hours of chemical exposure increases, the number of patients showing symptoms increases, and thus the probability of showing symptoms increases.



<https://www.desmos.com/calculator/bsqtqfians>

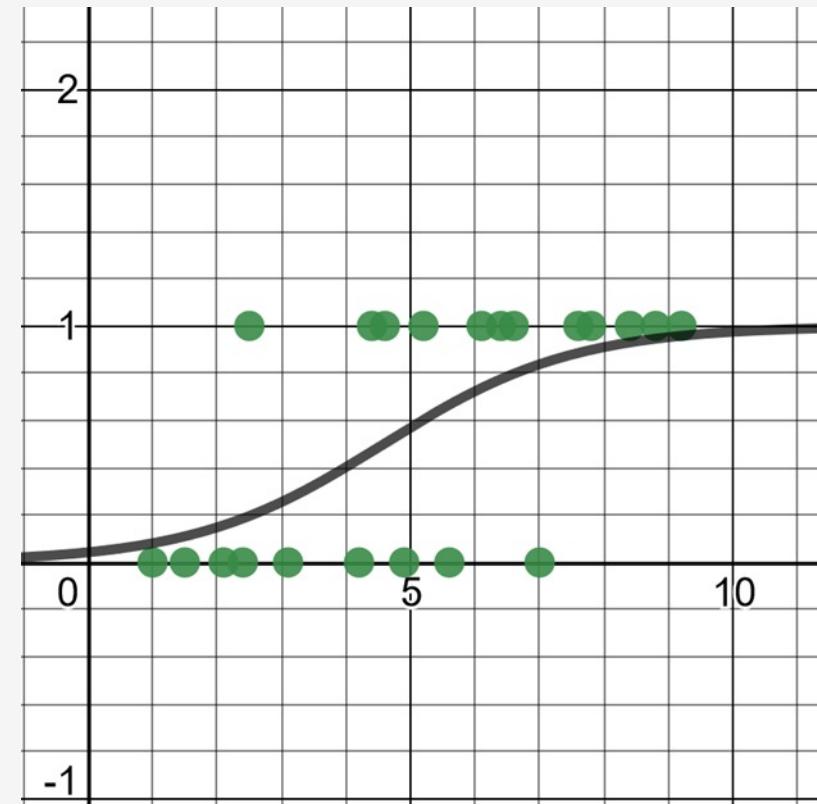
# Logistic Regression

---

**Logistic Regression** is a classification tool that predicts a **true** or **false** value for one or more variables.

Training data must have outcomes of 0 (false) or 1 (true), but the regression will output a probability value between 0 and 1.

- An S-shaped curve (a **logistic function**) is fit to the points and then used to predict probability.
- If a predicted value (the y-axis) is less than .5 it is typically categorized as false (0), and if the predicted value is greater than/equal to .5 it is typically categorized as true (1).

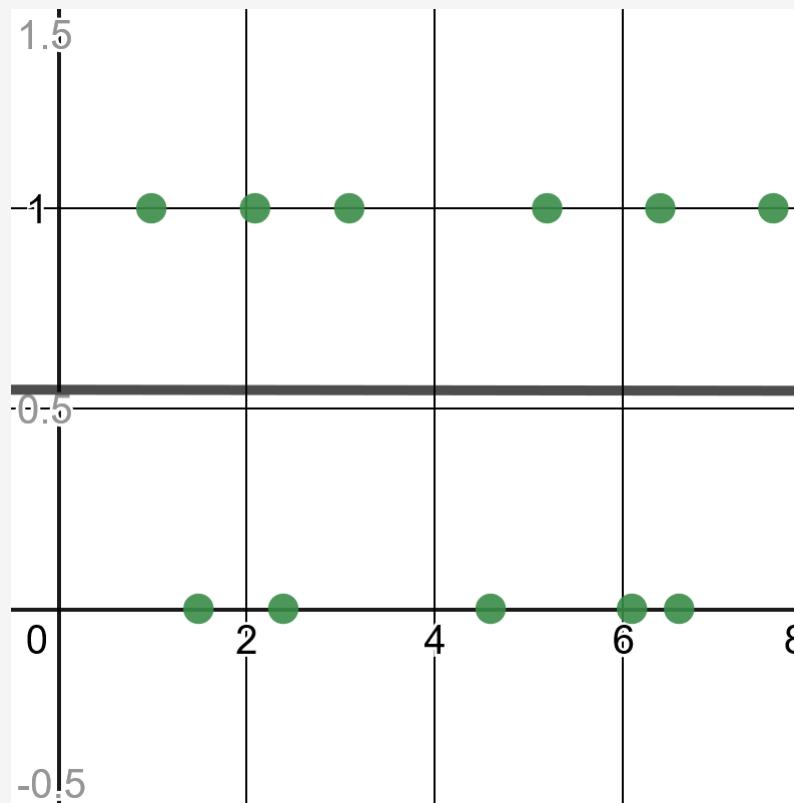


<https://www.desmos.com/calculator/bsqtqfians>

# When Not To Use Logistic Regression

---

Of course, if there is no transitional trend in your data then you should not use logistic regression.



<https://www.desmos.com/calculator/cmeksxk5rk>

# Just Show Me the Math!

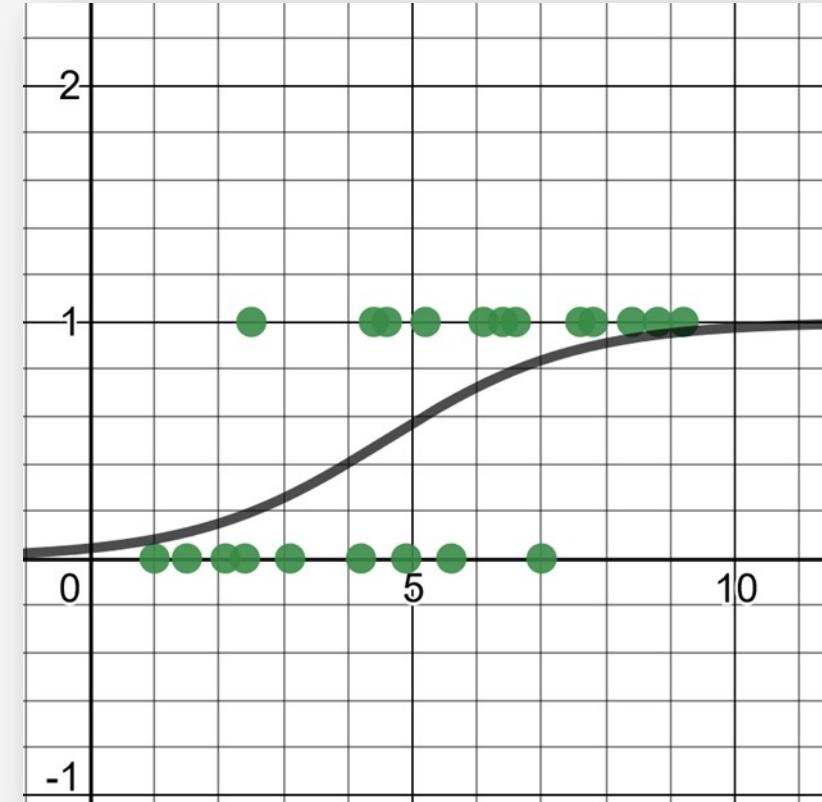
---

For a single independent variable  $x$  to predict a dependent probability variable  $y$ , you need to fit the data to the logistic function:

$$y = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x)}}$$

You can express this in Python as:

```
def predict_probability(x):
    p = 1.0 / (1.0 + math.exp(-(b0 + b1 * x)))
    return p
```



<https://www.desmos.com/calculator/blfcwrlnuw>

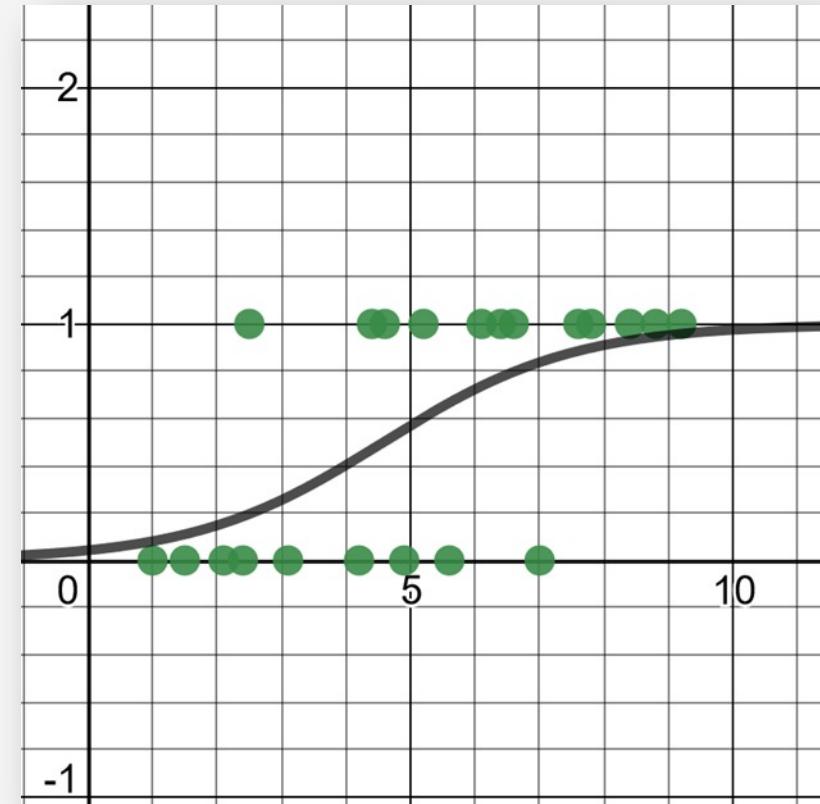
# Just Show Me the Math!

---

You may also see this logistic function expressed as:

$$y = \frac{e^{\beta_0 + \beta_1 x}}{1.0 + e^{\beta_0 + \beta_1 x}}$$

However it is the same and just algebraically expressed differently.



<https://www.desmos.com/calculator/blfcwrlnuw>

# Just Show Me the Math!

---

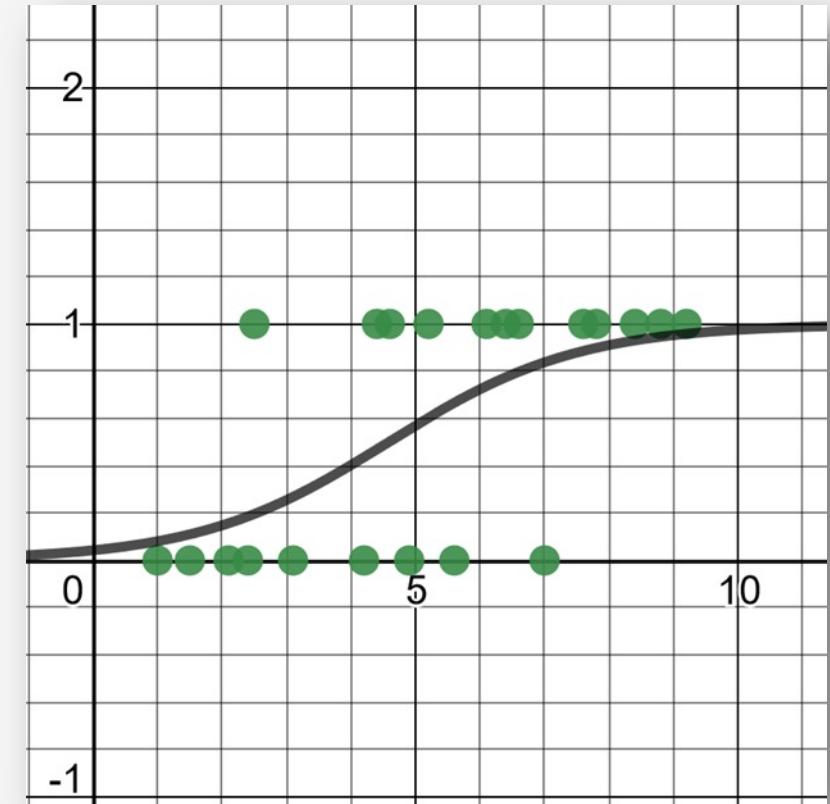
$$y = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x)}}$$

```
def predict_probability(x):
    p = 1.0 / (1.0 + math.exp(-(b0 + b1 * x)))
    return p
```

Notice the expression  $\beta_0 + \beta_1 x$  is linear, and this known as the **log odds function** which is translated logarithmically into a probability.

In the interest of time, we will avoid going into proofs and mathematical details about how this function works.

Just know it produces this S-shaped curve we need to output a probability between 0 and 1.



<https://www.desmos.com/calculator/blfcwrlnuw>

# Just Show Me the Math!

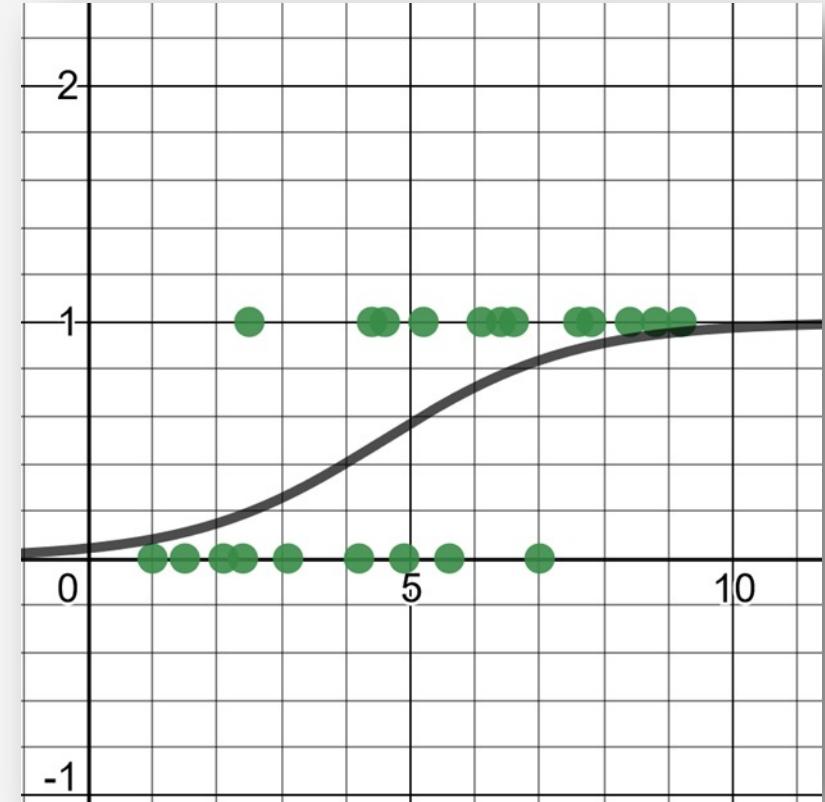
---

$$y = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x)}}$$

```
def predict_probability(x):
    p = 1.0 / (1.0 + math.exp(-(b0 + b1 * x)))
    return p
```

We need to solve for  $\beta_0$  and  $\beta_1$ , but we cannot use least squares like in linear regression.

- We are trying to maximize probability of the curve predicting correctly, not finding the best fit.
- Using hill climbing, we need to find  $\beta_0$  and  $\beta_1$  that produces the maximum likelihood.



<https://www.desmos.com/calculator/blfcwrlnuw>

# Maximum Likelihood

---

**Maximum likelihood** is a technique to estimate parameters that have the highest probability of outputting the observed data.

In our case, we need to find values for  $\beta_0$  and  $\beta_1$  that will yield the highest likelihood of outputting the correct true/false values.

Remember that our logistic function outputs a probability  $y$  for a given value  $x$ .

$$y = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x)}}$$

# Maximum Likelihood

---

Let's say during our hill-climbing, we test parameters  $\beta_0 = -3.17$  and  $\beta_1 = 0.69$

$$y = \frac{1.0}{1.0 + e^{-(3.17+0.69x)}}$$

**To calculate the total likelihood for these parameters...**

1. Get the true (1) data, calculate the probability  $y$  for each  $x$  value, and multiply them together.
2. Get the false (0) data, calculate the probability  $(1.0 - y)$  for each  $x$  value, and multiply them together.
3. Multiply the two products above together, and that is your total likelihood.

## Maximum Likelihood – Avoiding Floating Point Underflow

---

However, multiplying this many decimals together can cause floating point underflow.

With a clever mathematical hack, we can remedy this by using logarithmic addition instead of multiplication.

$$y = \log\left(\frac{1.0}{1.0 + e^{-(3.17+0.69x)}}\right)$$

**If you need to learn about logarithms, YouTube is the best place to get crash coured.**

PatrickJMT: <https://youtu.be/AAW7WRFBKdw>

Don't Memorize: <https://youtu.be/4UNkQcBrLaQ>

# Maximum Likelihood – Avoiding Floating Point Underflow

---

$$y = \log\left(\frac{1.0}{1.0 + e^{-(3.17+0.69x)}}\right)$$

**To calculate the total likelihood for these parameters, but avoid floating point underflow...**

1. Get the true (1) data, calculate the probability  $y$  for each  $x$  value, pass it to a  $\log()$  function, then sum the values.
2. Get the false (0) data, calculate the probability  $(1.0 - y)$  for each  $x$  value, pass it to a  $\log()$  function, then sum the values.
3. Sum the two values above together, pass it to the  $\exp()$  function to undo the logarithm, and that is your total likelihood.

# Maximum Likelihood

---

**We now know how to calculate the likelihood for a given set of parameters.**

**To calculate the maximum likelihood...**

1. Randomly adjust the  $\beta_0$  and  $\beta_1$  values (using hill-climbing, gradient descent, or other optimization)
2. Calculate the likelihood (as shown in the previous slide)
3. If the likelihood improves, keep the changes to  $\beta_0$  and  $\beta_1$ , otherwise revert.
4. Do this for as many iterations as necessary, until the likelihood stops improving.

# Hands-On: Logistic Regression

The screenshot shows the PyCharm IDE interface with the following details:

- Project:** oreilly\_machine\_learning\_from\_scratch
- File:** simple\_logistic\_regression\_hc.py
- Code Content:**

```
def __str__(self):
    return "(0), (1)".format(self.x, self.y)

points = [(Point(row.x, row.y)) for index, row in pd.read_csv("https://tinyurl.com/y2coceo7").iterrows()]
best_likelihood = -10_000_000
b0 = .01
b1 = .01

# calculate maximum likelihood
def predict_probability(x):
    p = 1.0 / (1.0001 + math.exp(-(b0 + b1 * x)))
    return p

for i in range(1_000_000):
    # Select b0 or b1 randomly, and adjust it randomly
    random_b = random.choice(range(2))
    random_adjust = np.random.normal()

    if random_b == 0:
        b0 += random_adjust
    elif random_b == 1:
        b1 += random_adjust

    true_estimates = sum(math.log(predict_probability(p.x)) for p in points if p.y == 1.0)
    false_estimates = sum(math.log(1.0 - predict_probability(p.x)) for p in points if p.y == 0.0)
```
- Run Tab:** simple\_logistic\_regression\_hc
- Output:**

```
1.0 / (1 + exp(-(-3.1748004158250778 + 0.6925413073093338*x))
BEST LIKELIHOOD: -9.946330353188507
```

Process finished with exit code 0
- Bottom Navigation:** Run, Debug, TODO, Version Control, Terminal, Python Console, Event Log

# Multivariable Logistic Regression

---

We can easily extend logistic regression to handle multiple independent variables, simply by adding more  $\beta_x$  variables for each additional variable.

We then solve for those  $\beta_x$  variables the same way as before.

$$y = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \dots + \beta_n x_n)}}$$

# Multivariable Logistic Regression

---

We have some historical employee data (<https://tinyurl.com/y6r7qjrp>) and want to use it to predict whether an employee will quit or not.

*SEX*, *AGE*, *PROMOTIONS*, and *YEARS\_EMPLOYED* are the predictor variables, and *DID\_QUIT* is the outcome variable where 1 = true and 0 = false.

SEX	AGE	PROMOTIONS	YEARS_EMPLOYED	DID_QUIT
1	43	4	10	0
1	38	3	8	0
1	44	4	11	1
0	41	2	6	0
1	45	2	6	1
0	36	3	9	0
1	33	1	3	0
1	44	3	10	0
...				

# Hands-On: Multivariable Logistic Regression

The screenshot shows the PyCharm IDE interface with the following details:

- Project Structure:** The project is named "oreilly\_machine\_learning\_from\_scratch". It contains several sub-directories like "code", "section\_ii", "section\_iii", and "section\_iv", each with various Python files such as "employee\_retention\_logistic\_regression.py", "simple\_logistic\_regression\_hc.kt", and "simple\_logistic\_regression\_hc\_numpy.py".
- Code Editor:** The main editor window displays the "employee\_retention\_logistic\_regression.py" file. The code implements a multivariable logistic regression model. It includes a function to calculate maximum likelihood and a loop for iterative random adjustment of coefficients.

```
# calculate maximum likelihood
def predict_probability(sex, age, promotions, years_employed):
    x = b0 + (b1 * sex) + (b2 * age) + (b3 * promotions) + (b4 * years_employed)
    odds = exp(-x)
    p = 1.0 / (1.0 + odds)
    return p

for i in range(iterations):
    # Select b0, b1, b2, b3, or b4 randomly, and adjust it by a random amount
    random_b = random.choice(range(5))

    random_adjust = np.random.standard_normal()

    if random_b == 0:
        b0 += random_adjust
    elif random_b == 1:
        b1 += random_adjust
    elif random_b == 2:
        b2 += random_adjust
    elif random_b == 3:
        b3 += random_adjust
    elif random_b == 4:
        b4 += random_adjust
```

- Run Tab:** The "Run" tab shows the output of the script. It prints the calculated probability and a prediction for an employee's retention status based on their sex, age, promotions, and years employed.

```
1.0 / (1 + exp(-(1.4531090636325825 + 0.168083973158077*s + -0.138222546057723*a + -2.4369251029348566*p + 1.2741701131437995*y))
BEST LIKELIHOOD: 0.887004090343420e-13
Predict employee will stay or leave (sex), (age), (promotions), (years employed): 0,32,1,4
WILL STAY, 42.23% chance of leaving
Predict employee will stay or leave (sex), (age), (promotions), (years employed): 0,32,0,4
WILL LEAVE, 89.32% chance of leaving
Predict employee will stay or leave (sex), (age), (promotions), (years employed):
```

- Bottom Navigation:** The navigation bar includes tabs for Run, TODO, Version Control, Terminal, and Python Console.

# Using Logistic Regression for Classification

---

Logistic regression may seem limited in that it only supports two categories: true (1) or false (0).

But you can make it support any number of categories!

## **To use logistic regression for more than two categories:**

1. Build a separate logistic regression for each category, where a given item belongs to that category (1) or doesn't belong to that category (0).
2. To predict which category an item belongs to, pass it through each category's logistic regression, and choose the one with the highest probability.

# Exercise: “There’s No Correlation”

---

**It is one day from a manned space launch.**

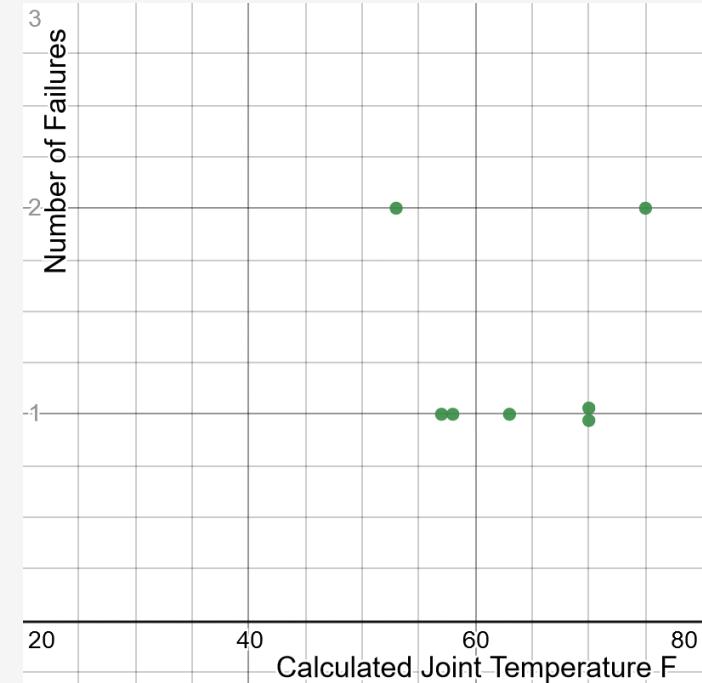
**However, your engineering team has been expressing concern about the O-Rings that seal rocket gases from releasing, and whether they perform in colder temperatures.**

You share this concern with other parties and are provided data of all 7 O-Ring failures from 24 launches and the temperature (shown to the right).

The consensus from other parties is there is no correlation between number of failures and temperature.

Is this assessment correct?

temperature	o_ring_failures
53	2
57	1
58	1
63	1
70	1
70	1
75	2



# Exercise: “There’s No Correlation”

---

**It is one day from a manned space launch.**

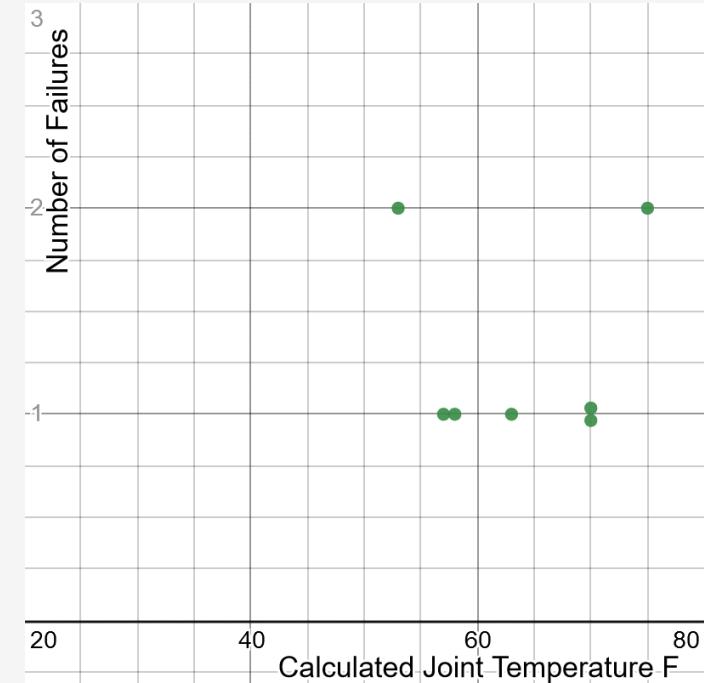
**However, your engineering team has been expressing concern about the O-Rings that seal rocket gases from releasing, and whether they perform in colder temperatures.**

You share this concern with other parties and are provided data of all **7 O-Ring failures** from **24 launches** and the temperature (shown to the right).

The consensus from other parties is there is no correlation between number of failures and temperature.

Is this assessment correct? **Is anything missing?**

temperature	o_ring_failures
53	2
57	1
58	1
63	1
70	1
70	1
75	2



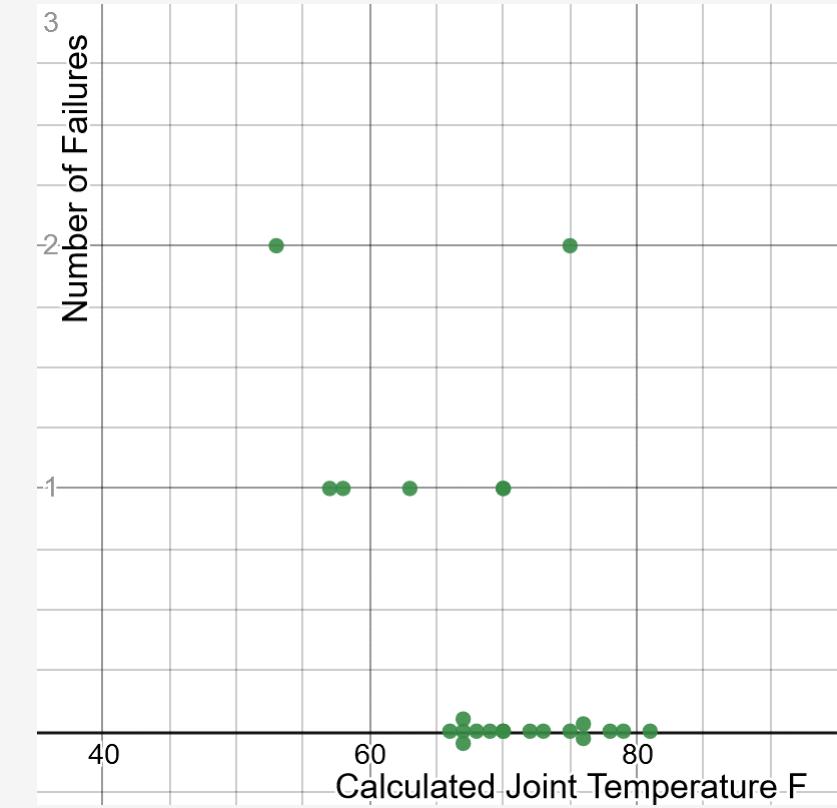
# Exercise: “There’s No Correlation”

---

Notice that data from successful launches were not included, which may tell an entirely different story.

What are your thoughts now? Is there a correlation? Is there a model that can be used to predict risk?

temperature	o_ring_failures
53	2
57	1
58	1
63	1
66	0
67	0
67	0
67	0
68	0
69	0
70	1
70	0
70	1
70	0
72	0
73	0
75	0
75	2
76	0
76	0
78	0
79	0
81	0



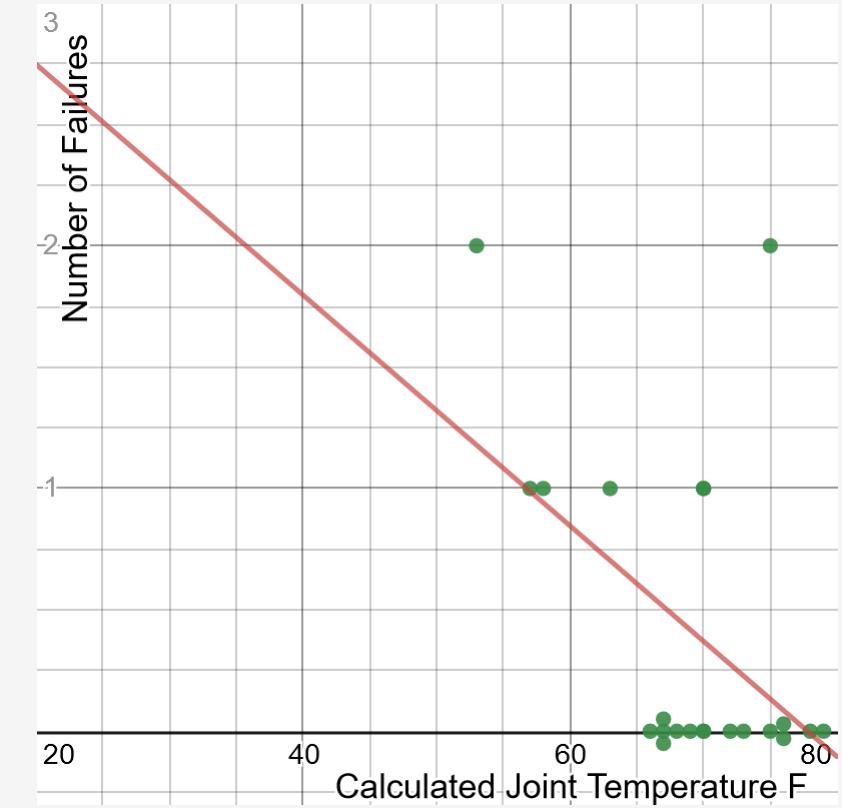
# Exercise: “There’s No Correlation”

---

You can try to apply a linear regression here, and while it does show a trend it is a little awkward especially since our data is sparse.

Should we transform our data somehow?  
Are there any other models we can try?

temperature	o_ring_failures
53	2
57	1
58	1
63	1
66	0
67	0
67	0
67	0
68	0
69	0
70	1
70	0
70	1
70	0
72	0
73	0
75	0
75	2
76	0
76	0
78	0
79	0
81	0



# Linear Regression Source Code

---

```
import pandas as pd
from sklearn.linear_model import LinearRegression

# Learn more: https://scikit-
Learn.org/stable/modules/generated/skLearn.Linear_model.LinearRegression.html

# Import points

df = pd.read_csv('https://bit.ly/2DgjTk5', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Plain ordinary least squares
fit = LinearRegression().fit(X, Y)

# Print "m" and "b" coefficients
print("m = {0}".format(fit.coef_.flatten()))
print("b = {0}".format(fit.intercept_.flatten()))
```

# Exercise: “There’s No Correlation”

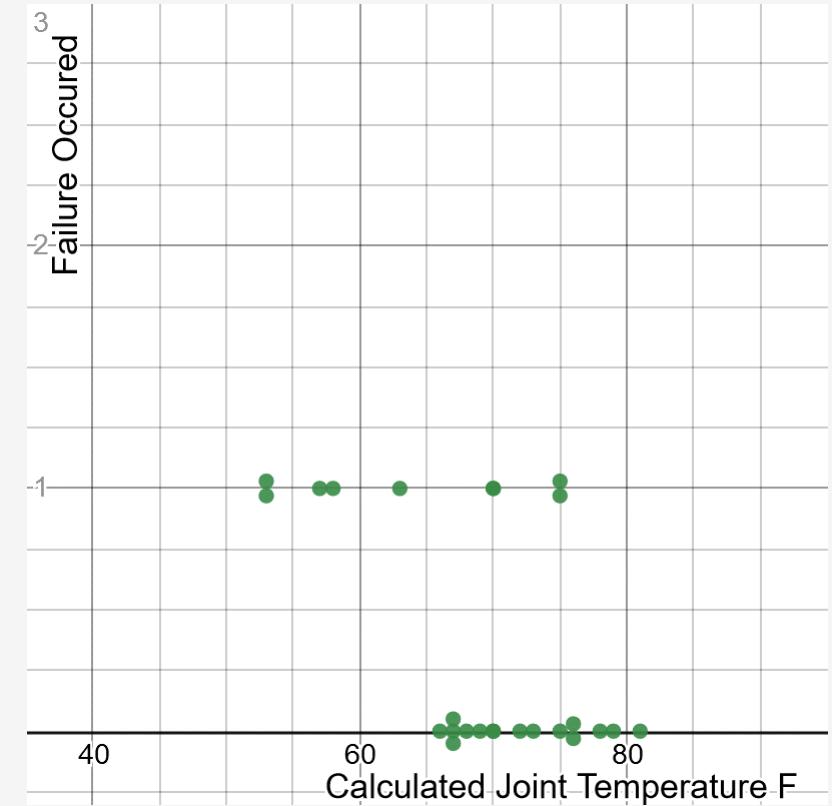
---

What if we converted the data to be binary, showing whether a failure occurred or not occurred, by separating each instance into its own record?

This reduces the domain of output variables to “0” and “1” creating a binary model.

Is a story now becoming clear? What model can we use to predict probability of failure at a given temperature?

temperature	o_ring_failures
53	1
53	1
57	1
58	1
63	1
66	0
67	0
67	0
67	0
68	0
69	0
70	1
70	0
70	1
70	0
72	0
73	0
75	0
75	1
75	1
76	0
76	0
78	0
79	0
81	0



# Exercise: “There’s No Correlation”

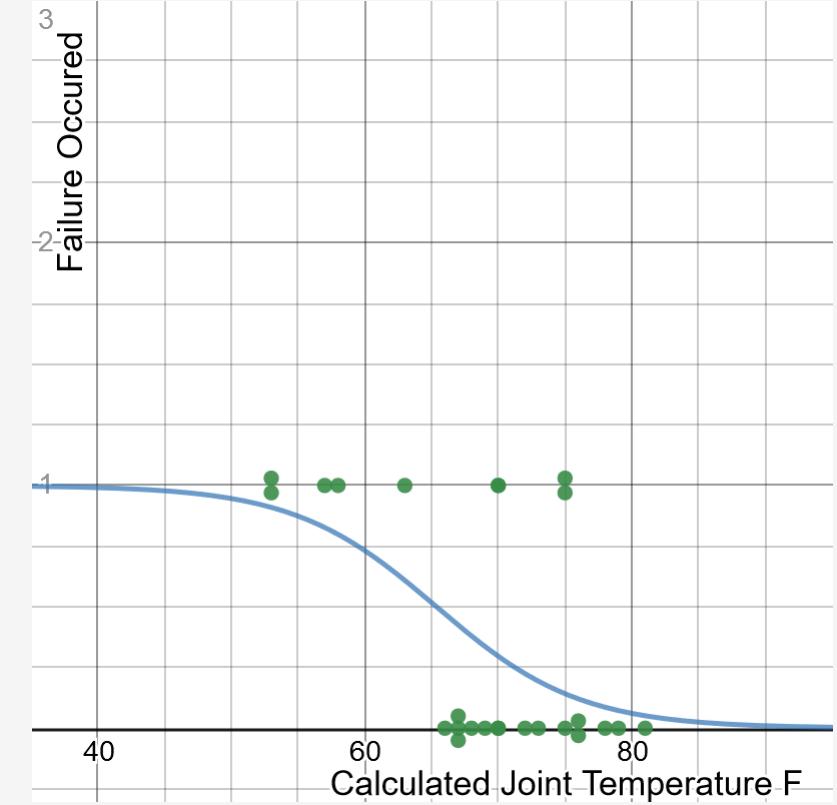
---

Logistic regression might be the best way to model this risk.

Even though we lack freezing temperature data, the logistic regression points to a high probability of risk for O-ring failure.

If our launch is going to happen in freezing temperatures, this does not bode well.

temperature	o_ring_failures
53	1
53	1
57	1
58	1
63	1
66	0
67	0
67	0
67	0
68	0
69	0
70	1
70	0
70	1
70	0
72	0
73	0
75	0
75	1
75	1
76	0
76	0
78	0
79	0
81	0



## Exercise: “There’s No Correlation”

---

This is exactly what happened to the space shuttle Challenger on January 28, 1986, and if you already have not figured out already, we are doing the analysis.

Through a series of unfortunate events, only partial data was accessible and omitted non-failure data, which showed a correlation with temperature and O-ring failure.

The analysis we just did should have happened before the accident, but unfortunately it occurred afterwards.

We will learn later about how biases, class imbalance, and broken data can derail machine learning and statistical models.



*The space shuttle Challenger just moments before disaster on January 28, 1986 (above) and Richard Feynman famously demonstrating O-ring failure with a glass of ice water (left).*

# Space Shuttle Challenger: Logistic Regression Source Code

---

```
import numpy as np
import pandas as pd
import plotly.graph_objects as go
from sklearn.linear_model import LogisticRegression

# Load the data
df = pd.read_csv("https://bit.ly/2DgjTk5", delimiter=",")

# Duplicate and convert records to have binary outcomes
df = df.loc[df.index.repeat(df["o_ring_failures"])] .append(df[df["o_ring_failures"] == 0])
df.loc[(df["o_ring_failures"] > 0), 'o_ring_failures'] = 1

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column) \
Y = df.values[:, -1]

model = LogisticRegression()
model.fit(X, Y)

# Print "m" and "b" coefficients
print("m = {0}".format(model.coef_.flatten()))
print("b = {0}".format(model.intercept_.flatten()))

# Plot results
fig = go.Figure()
fig.add_trace(go.Scatter(x=df["temperature"], y = df["o_ring_failures"], mode='markers', name="Observations"))
fig.add_trace(go.Scatter(x=np.arange(0.0,90.0,.1),
                        y= model.predict_proba(np.arange(0.0,90.0,.1).reshape(-1, 1))[:, -1],
                        mode='lines',
                        name='Logistic Regression'))
fig.show()
```

# Quiz Time!

---

Logistic regression will only output a probability between 0 and 1 that an event will happen.

- 1) True
- 2) False

# Quiz Time!

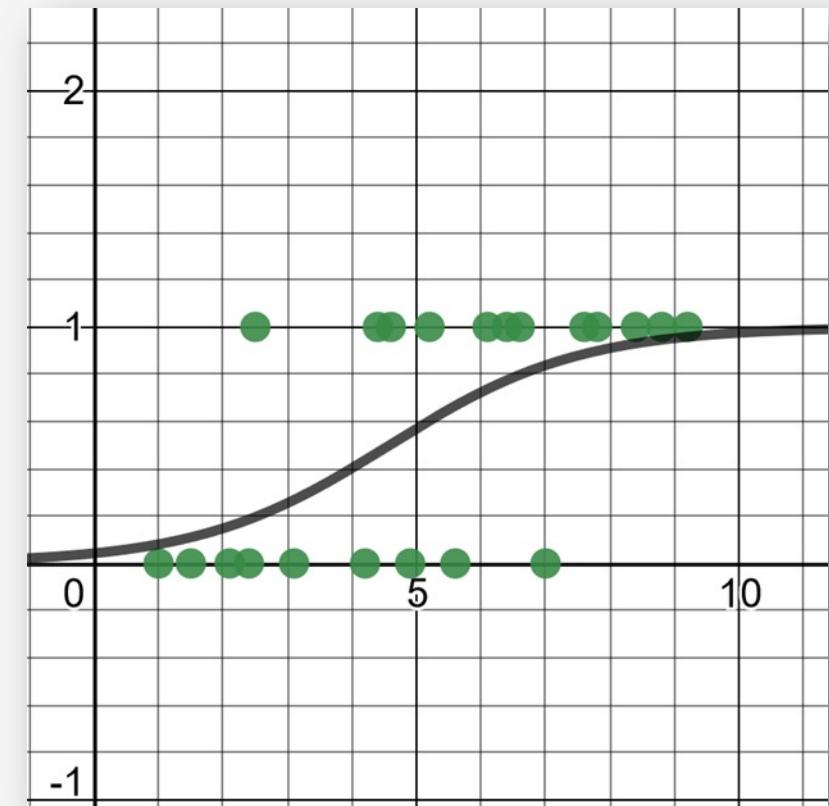
---

Logistic regression will only output a probability between 0 and 1 that an event will happen.

1) True

Like any probability model, logistic regression will output a percentage between 0 and 1

2) False



## Quiz Time!

---

Logistic regression cannot be used for more than two categories.

- A) True
- B) False

# Quiz Time!

---

Logistic regression cannot be used for more than two categories.

- A) True      Logistic regression can support more than one category by doing a separate logistic regression for each category, and predicting the one that yields the highest probability.
- B) False

# Section IV

# Naïve Bayes

# What is Naïve Bayes?

---

**Naïve Bayes** is a machine learning application of Bayes Theorem that merges probabilities of multiple features to predict a given category.

It is often used to classify text (e.g. email spam/not spam), which is the exercise we will be doing today.

- Naïve Bayes is effective because it learns quickly, even with little data.
- Naïve Bayes works by mapping probabilities of each individual feature occurring/not occurring for a given category (e.g. a word occurring in spam/not spam).
- While commonly used for discrete variables like words, it can also be used with continuous variables using statistical distributions (we will not cover this today).



Thomas Bayes, the inventor of Bayes Theorem

# Demo: Categorizing Bank Transactions



# How to Build a Naïve Bayes Text Classifier

---

To predict a category for a new set of features:

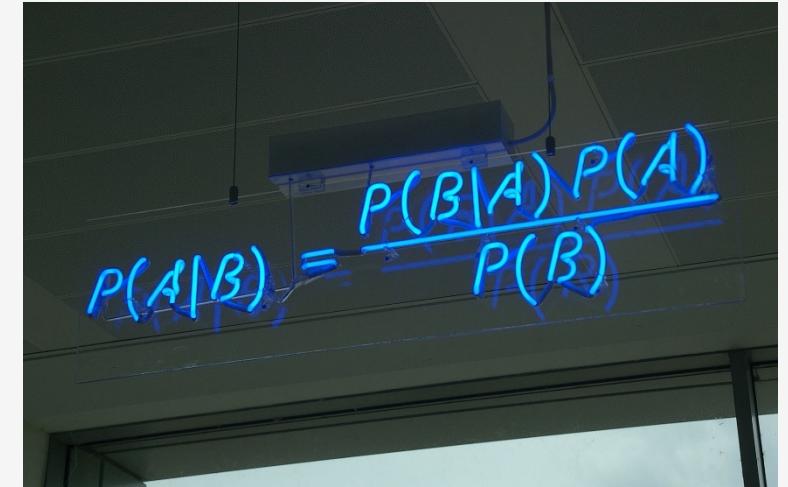
- 1) For a given category (like **spam**, **grocery**, etc), combine the probabilities of each feature **occurring** and **not occurring** by multiplying them.

$$\text{Occur Product} = P_{f1} * P_{f2} * P_{f3} * \dots * P_{fn}$$

$$\text{Not Occur Product} = (1 - P_{f1}) * (1 - P_{f2}) * (1 - P_{f3}) * \dots * (1 - P_{fn})$$

- 2) Combine the probabilities above with this division operation:

$$\text{Combined Probability} = \frac{(\text{Occur Product})}{(\text{Occur Product}) + (\text{Not Occur Product})}$$

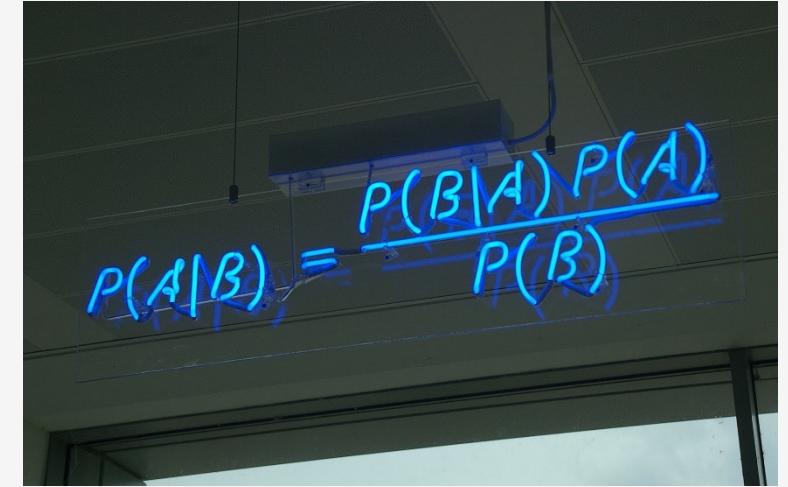


# How to Build a Naïve Bayes Text Classifier

---

3) Calculate the combined probability for every category, and the one that is the highest is the category you predict.

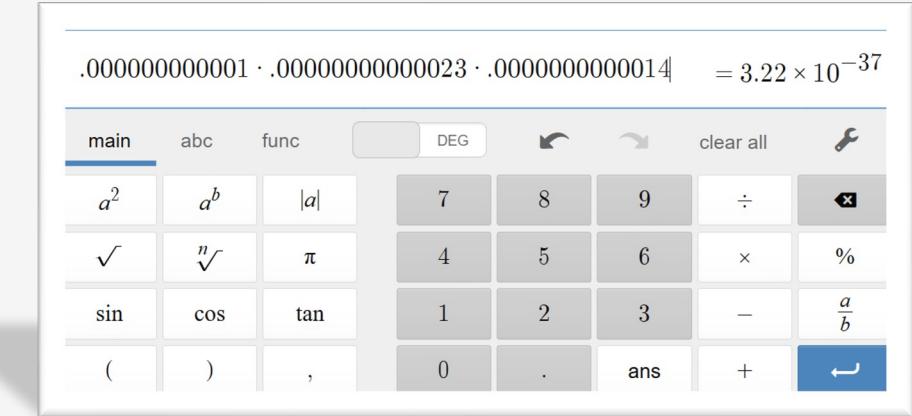
Easy right? But there is one complication...


$$P(A|B) = \frac{P(B|A) P(A)}{P(B)}$$

# Dealing with Floating Point Underflow

Remember in logistic regression where we had floating point underflow issues, where multiplying small decimals gets so small the computer cannot handle them?

- We can remedy the issue here too by using logarithmic addition rather than multiplication.
- Transform each probability with a `log()` or `ln()` function and then sum them, then call `exp()` to convert the result back!



# Dealing with Floating Point Underflow

---

$P_{fx}$  = Probability of feature x

$$\text{Occur Product} = \exp(\log(P_{f1}) + \log(P_{f2}) + \log(P_{f3}) + \dots + \log(P_{fn}))$$

$$\text{Not Occur Product} = \exp(\log(1 - P_{f1}) + \log(1 - P_{f2}) + \log(1 - P_{f3}) + \dots + \log(1 - P_{fn}))$$

$$\text{Combined Probability} = \frac{(\text{Occur Product})}{(\text{Occur Product}) + (\text{Not Occur Product})}$$

## One More Thing...

---

Never let a feature have a zero probability, so add some small constants to the numerator and denominator like 0.1 and 0.2 to make a default probability.

“Fudging” the numbers in machine learning can be acceptable since machine learning deals with estimates anyway.

You can make these numbers smaller if you like.

$$\text{Feature Probability} = \frac{0.1 + (\text{Occur Probability})}{0.2 + (\text{Occur Probability}) + (\text{Not Occur Probability})}$$

## Final Naïve Bayes Formulation

---

$P_{fx}$  = Probability of feature x, with a fudged constant in numerator and denominator

Occur Product =  $\exp(\log(P_{f1}) + \log(P_{f2}) + \log(P_{f3}) + \dots + \log(P_{fn}))$

Not Occur Product =  $\exp(\log(1 - P_{f1}) + \log(1 - P_{f2}) + \log(1 - P_{f3}) + \dots + \log(1 - P_{fn}))$

$$\text{Combined Probability} = \frac{(\text{Occur Product})}{(\text{Occur Product}) + (\text{Not Occur Product})}$$

# Section V

## Decision Trees and Random Forests

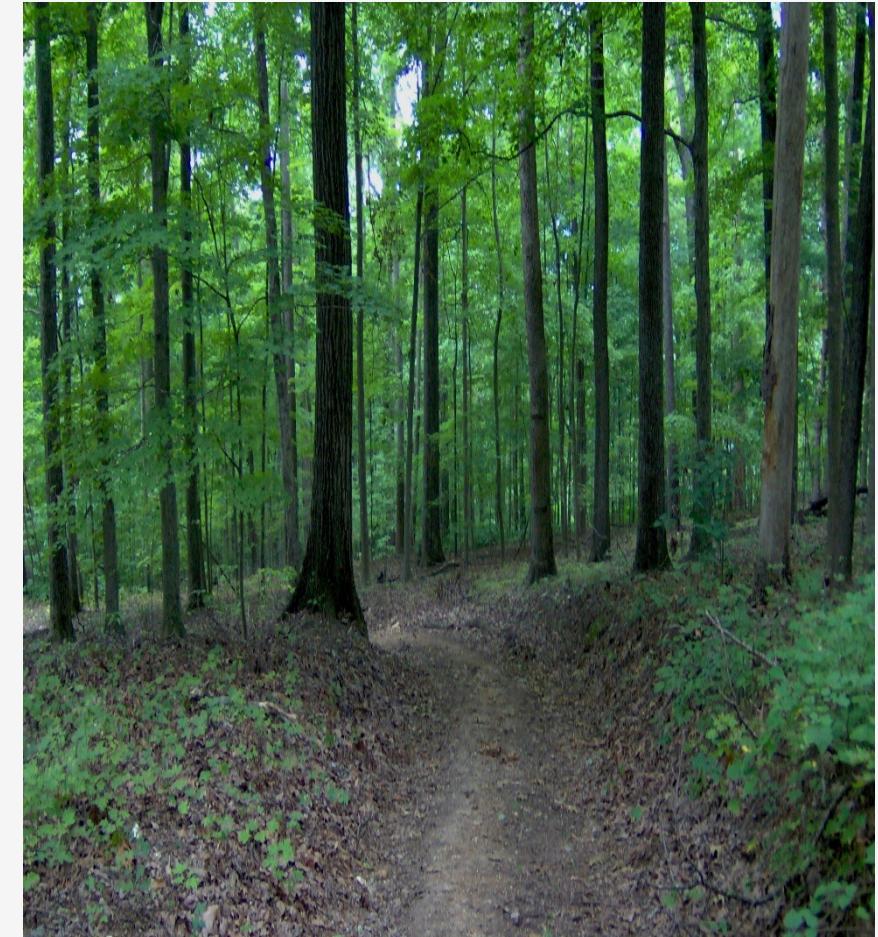
# What Are Decision Trees?

---

**Decision trees are a powerful machine learning tool and work well for a lot of machine learning problems.**

**As a matter of fact, decisions trees work so well they are notorious for overfitting.**

- This can be remedied with random forests which generates hundreds of decision trees with randomly sampled data.
- Decision trees can also be improved with gradient boosting and other techniques.
- Other flavors of decision trees exist, like regression trees.



# Decision Tree Intuition

---

We have some weather data and labeled each record as being “1” (good weather) or “0” (bad weather).

We could solve this using a logistic regression but let us try using decision trees instead.



RAIN	LIGHTNING	CLOUDY	TEMPERATURE	GOOD_WEATHER_IND
0	1	1	74	0
0	0	0	69	1
1	0	1	58	0
0	0	0	71	1
0	0	0	73	1
0	1	1	80	0
0	1	1	74	0
0	0	0	73	1
...				

# Decision Tree Intuition

---

Start with the variable that is most likely to separate good weather and bad weather as best as possible.

For reasons we will discuss later, you determine **RAIN** is good at separating good weather and bad weather, so you bucket records like this:



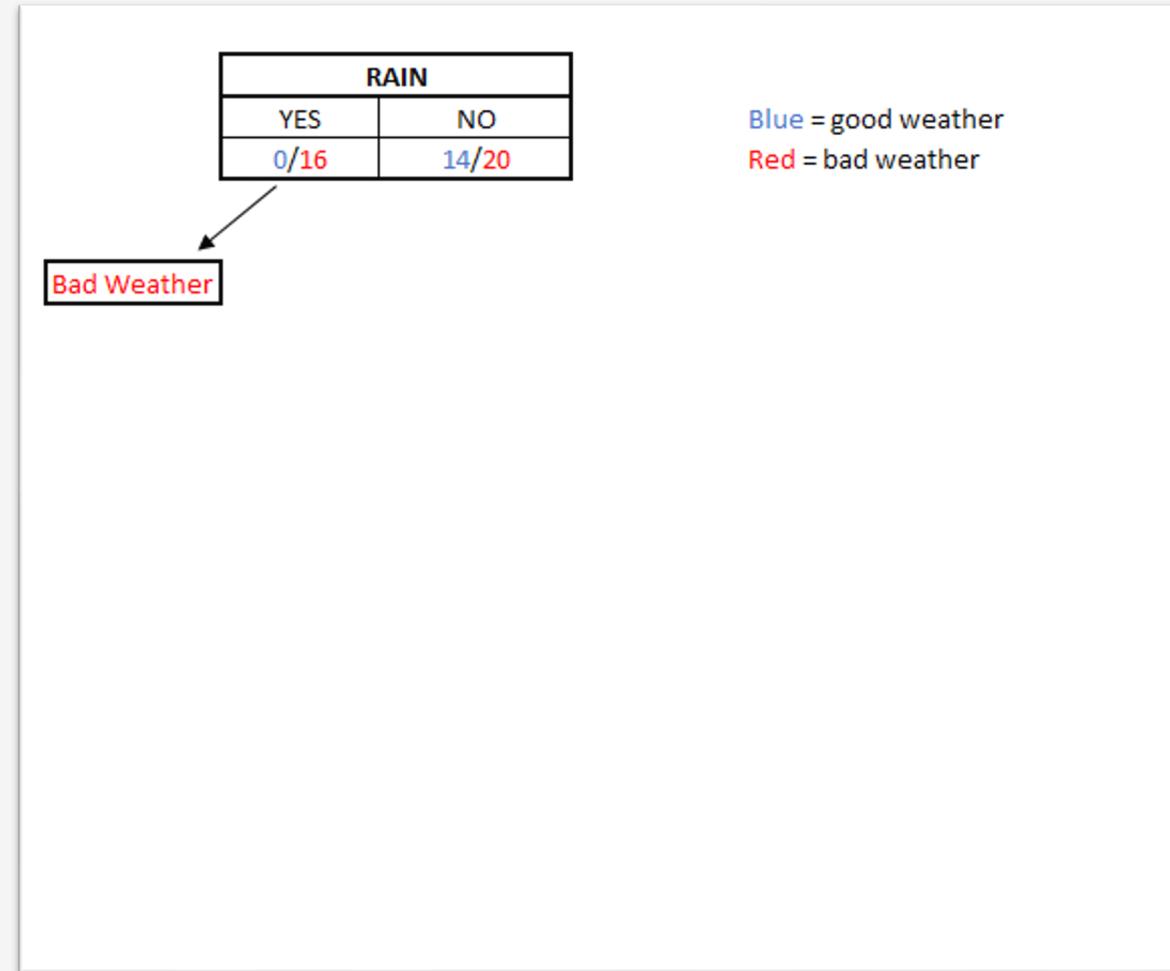
RAIN			
YES	NO	Blue = good weather	
0/16	14/20	Red = bad weather	

# Decision Tree Intuition

---

Since all records where **RAIN** was present always yielded **bad weather**, we will simply predict any new record with **RAIN** as **bad weather**.

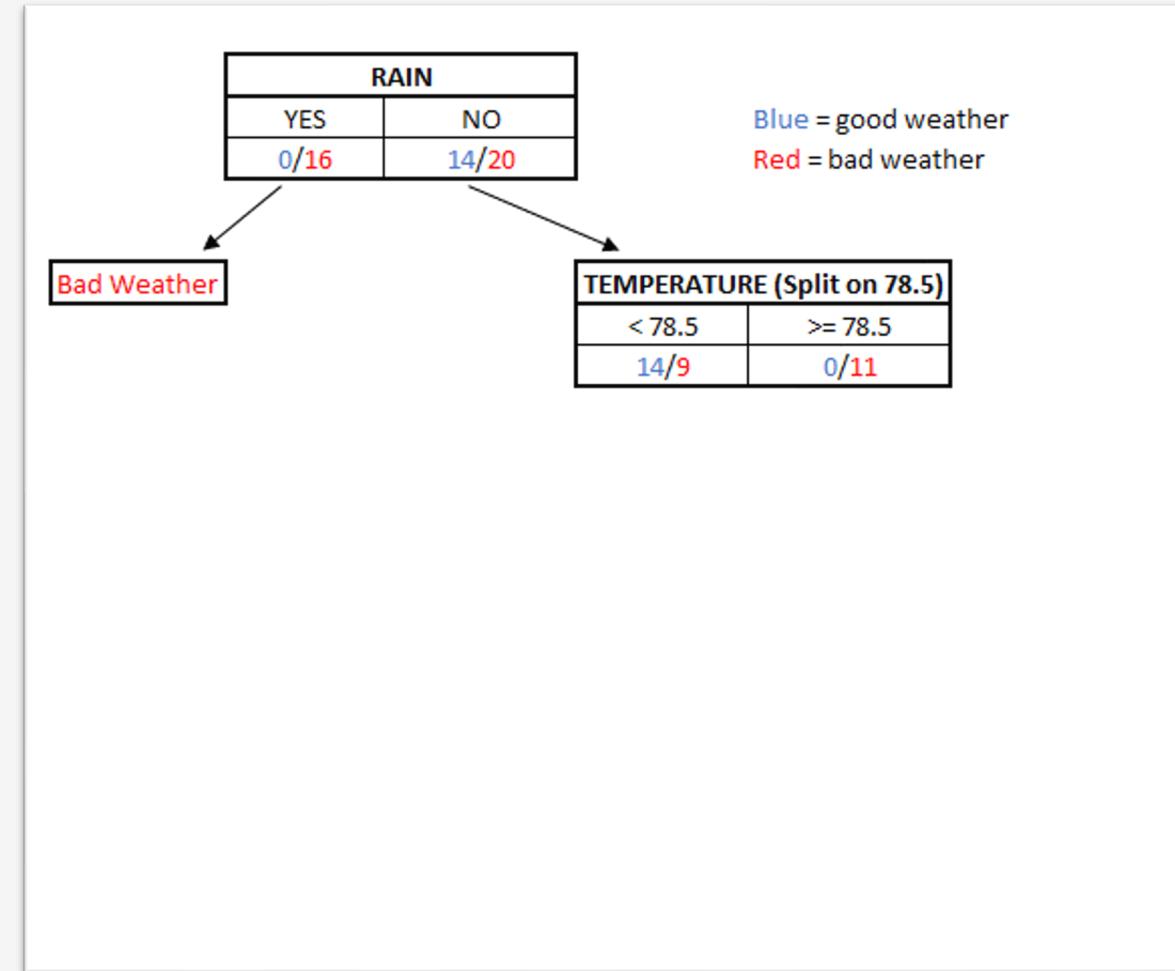
However, records that did not have **RAIN** are mixed, where some are **good weather**, and others are **bad weather**. These will need to be split again.



# Decision Tree Intuition

We take records that did not have **RAIN** and now split them on **TEMPERATURE**.

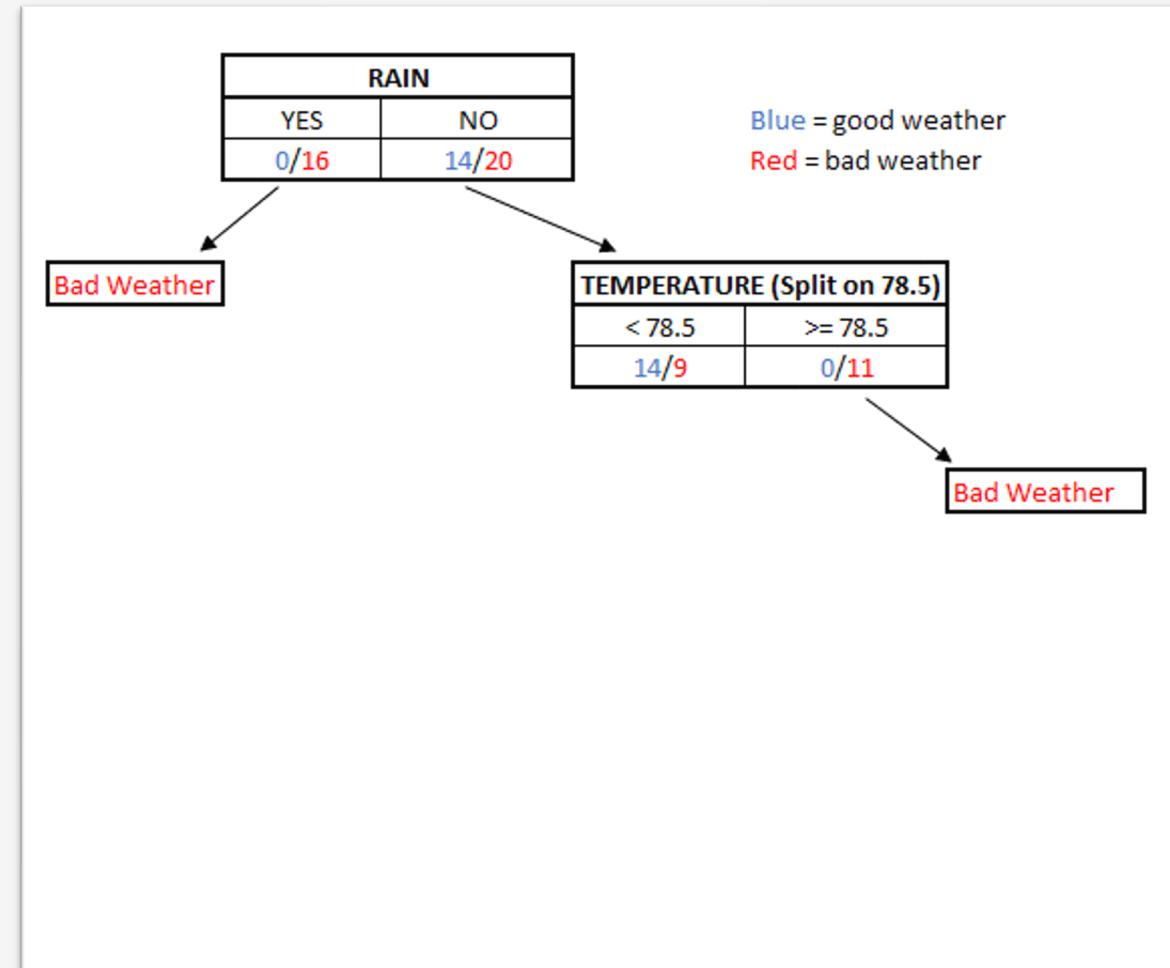
Since **TEMPERATURE** is continuous and not a simple yes/no binary, we split on a value 78.5 that optimizes the split (more on this later).



# Decision Tree Intuition

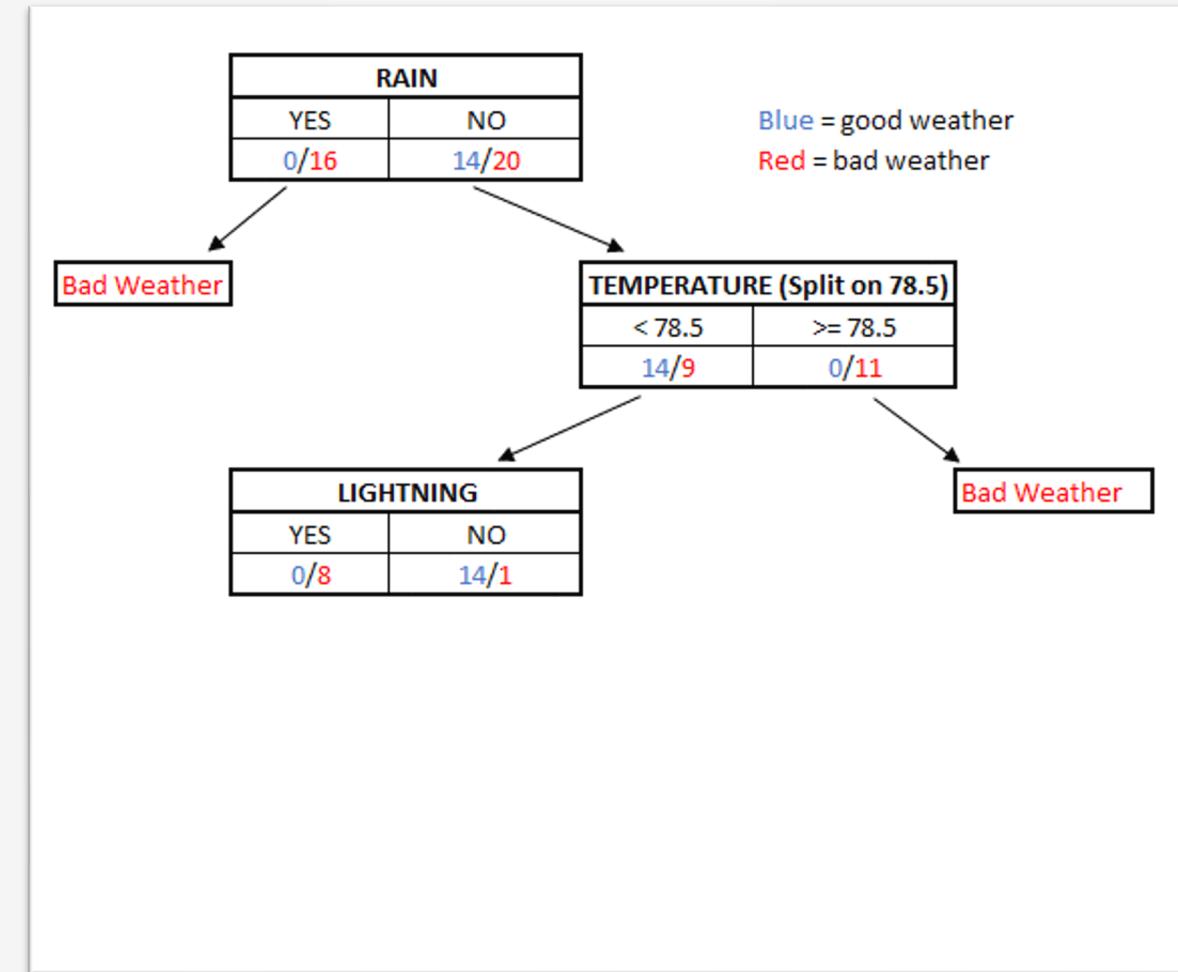
Notice how all records with **TEMPERATURE** greater than/equal to 78.5 are **bad weather**, so we will predict any records with a **TEMPERATURE** at least 78.5 as **bad weather**.

But records with a **TEMPERATURE** less than 78.5 have a mix of **good weather** and **bad weather**. So we need to split those again.



# Decision Tree Intuition

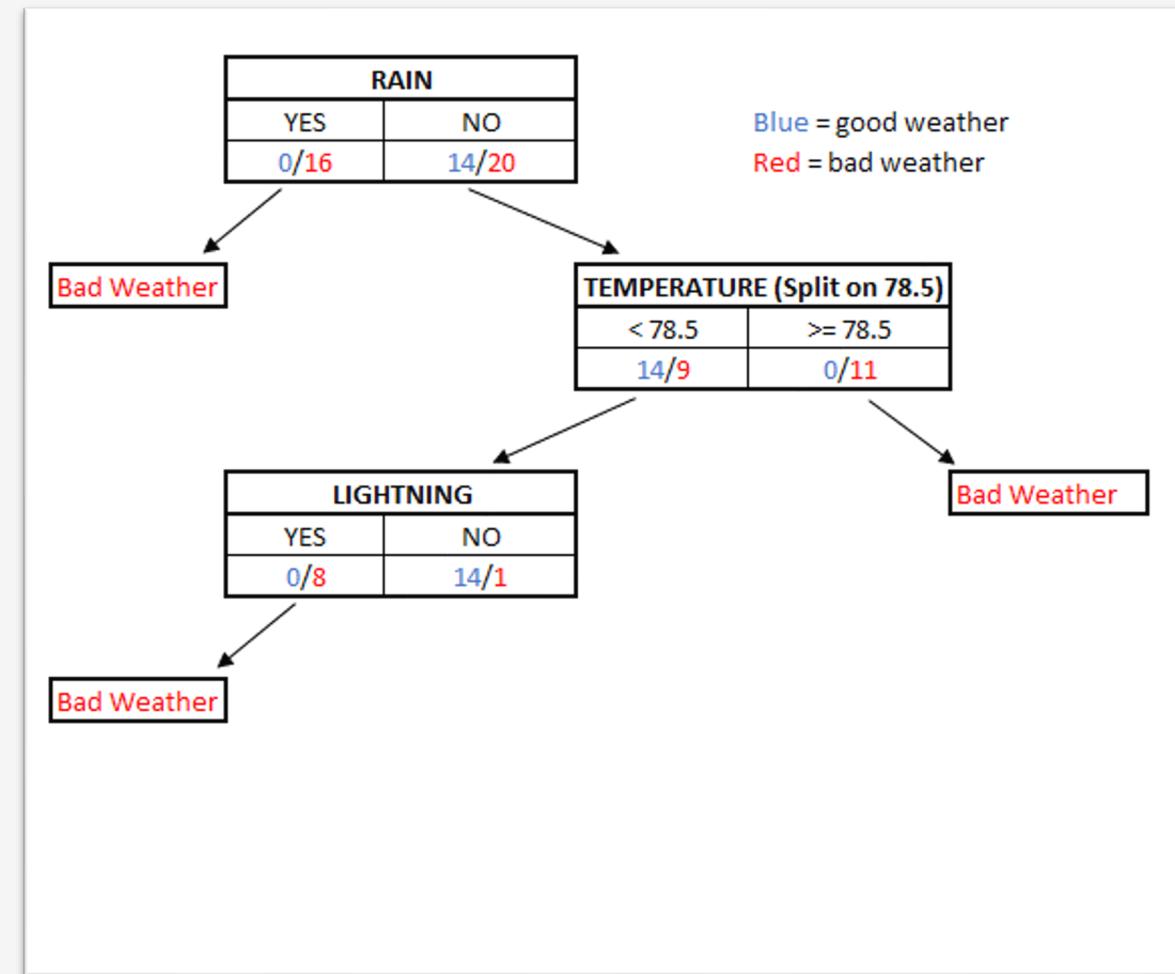
We take those records with **TEMPERATURE** less than 78.5 and now split them on whether **LIGHTNING** was present.



# Decision Tree Intuition

We take those records with **TEMPERATURE** less than 78.5 and now split them on whether **LIGHTNING** was present.

Notice how all records with **LIGHTNING** are labelled as **bad weather**, so we can predict any new record with **LIGHTNING** as bad weather.

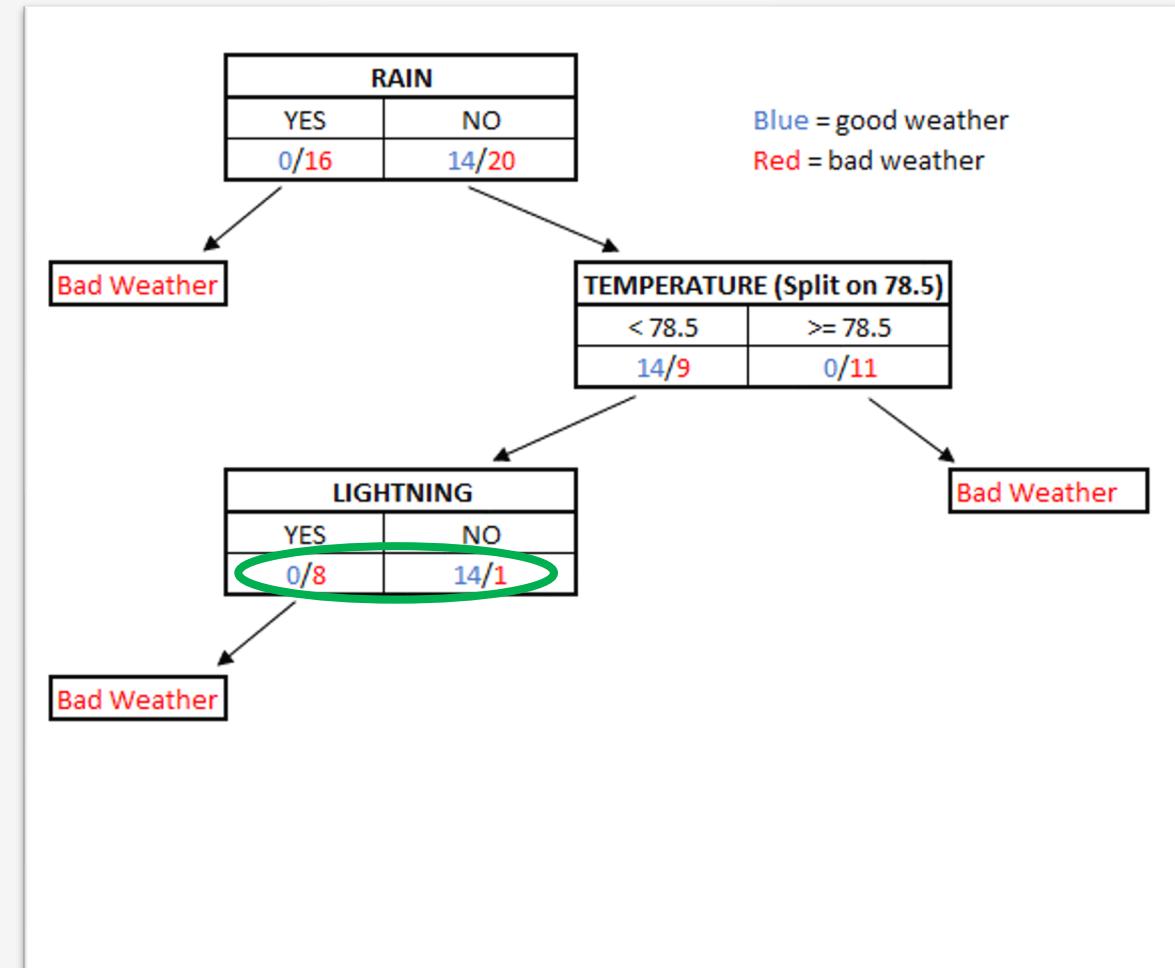


# Decision Tree Intuition

We take those records with **TEMPERATURE** less than 78.5 and now split them on whether **LIGHTNING** was present.

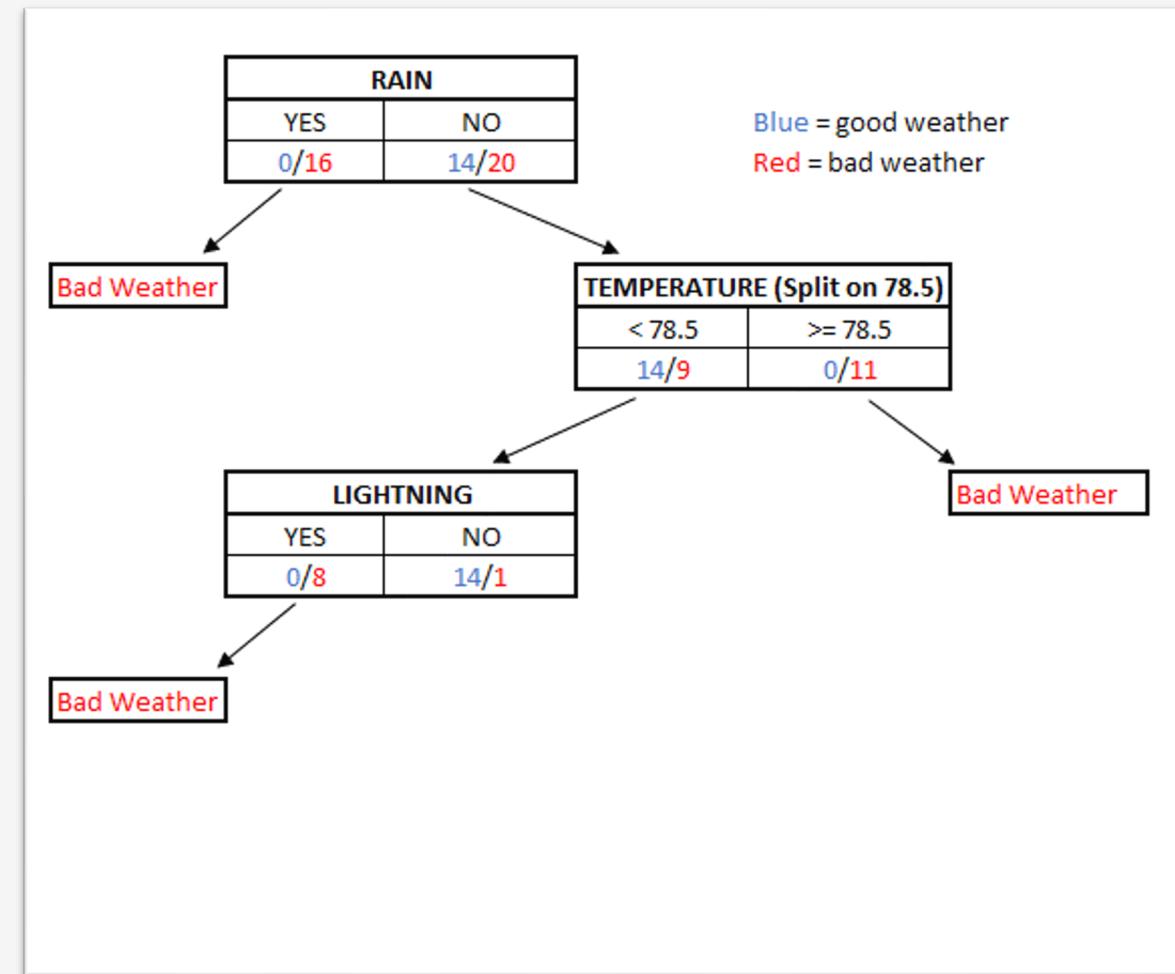
Notice how all records with **LIGHTNING** are labelled as **bad weather**, so we can predict any new record with **LIGHTNING** as bad weather.

Notice how close we are to a perfect clean split now!



# Decision Tree Intuition

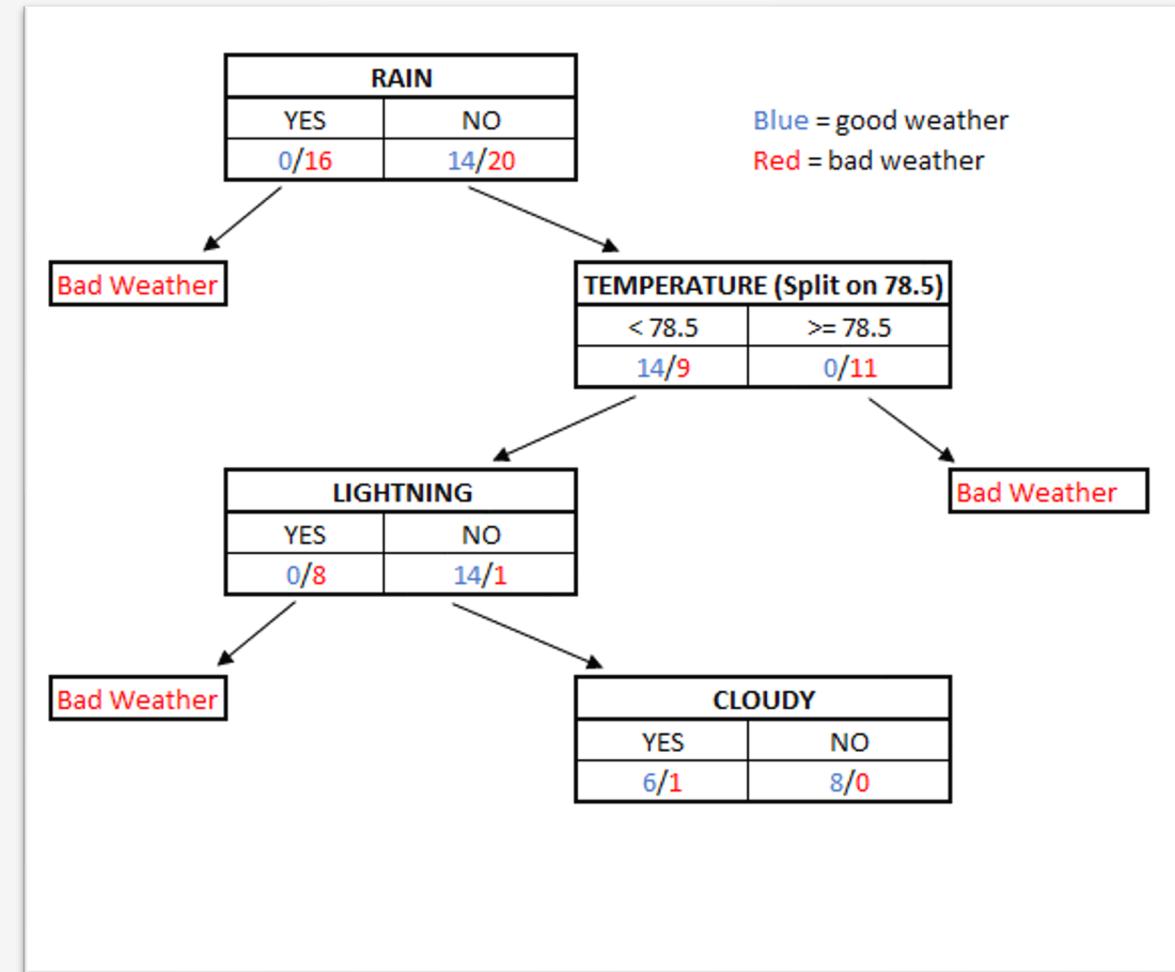
**LIGHTNING** almost gave us a perfect split, where lightning being present would predict bad weather and no lightning *almost* always meant good weather.



# Decision Tree Intuition

**LIGHTNING** almost gave us a perfect split, where lightning being present would predict bad weather and no lightning *almost* always meant good weather.

We could probably stop here, but let's split these remaining records with no lightning. Finally we split on whether it was **CLOUDY** or not.

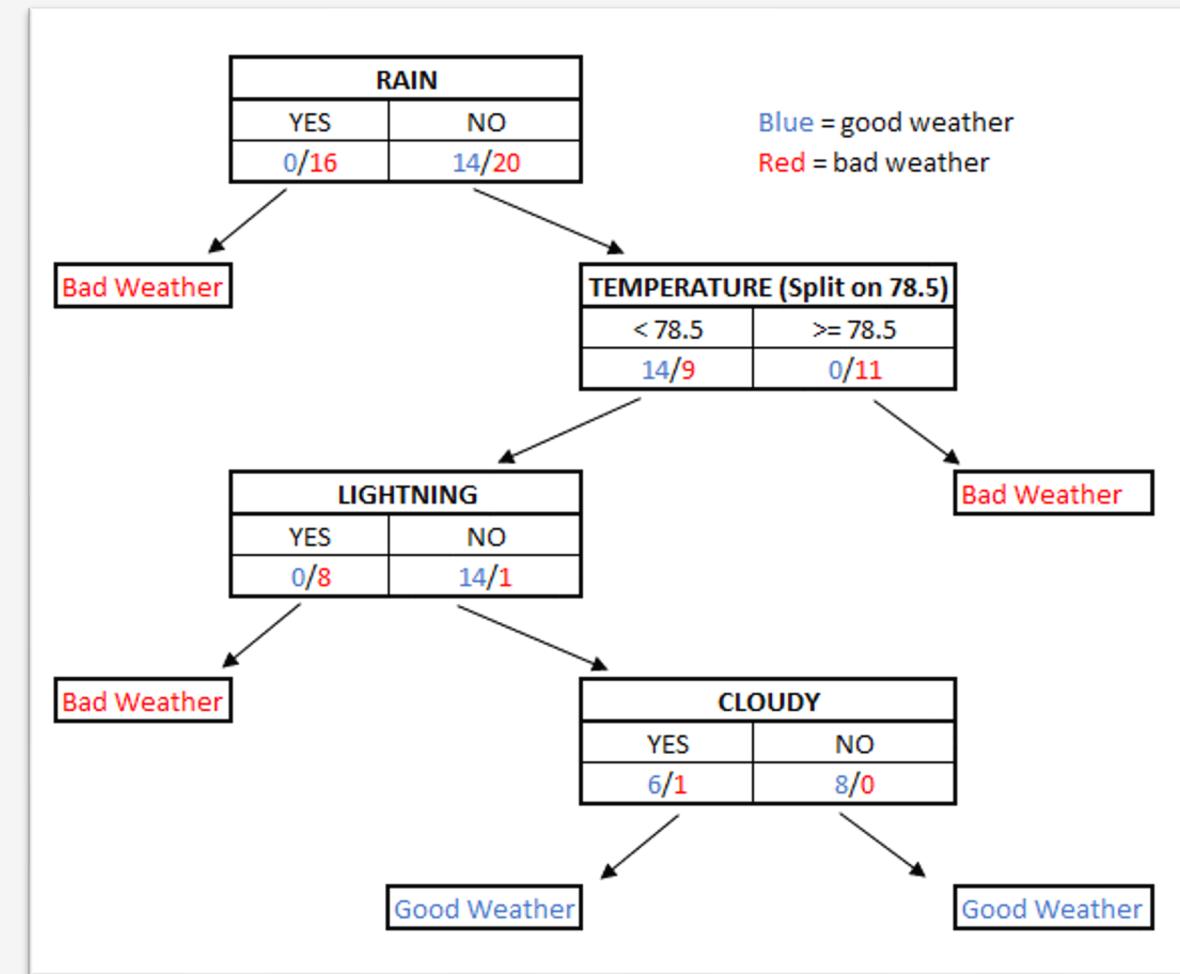


# Decision Tree Intuition

At this point, we establish we cannot separate productively anymore and **CLOUDY** does not have any impact to our prediction.

It will always be **good weather**, given we already established there was no **LIGHTNING**, the **TEMPERATURE** is less than 78.5, and there is no **RAIN**.

Congratulations! We just built our first decision tree. But you might have a few questions...

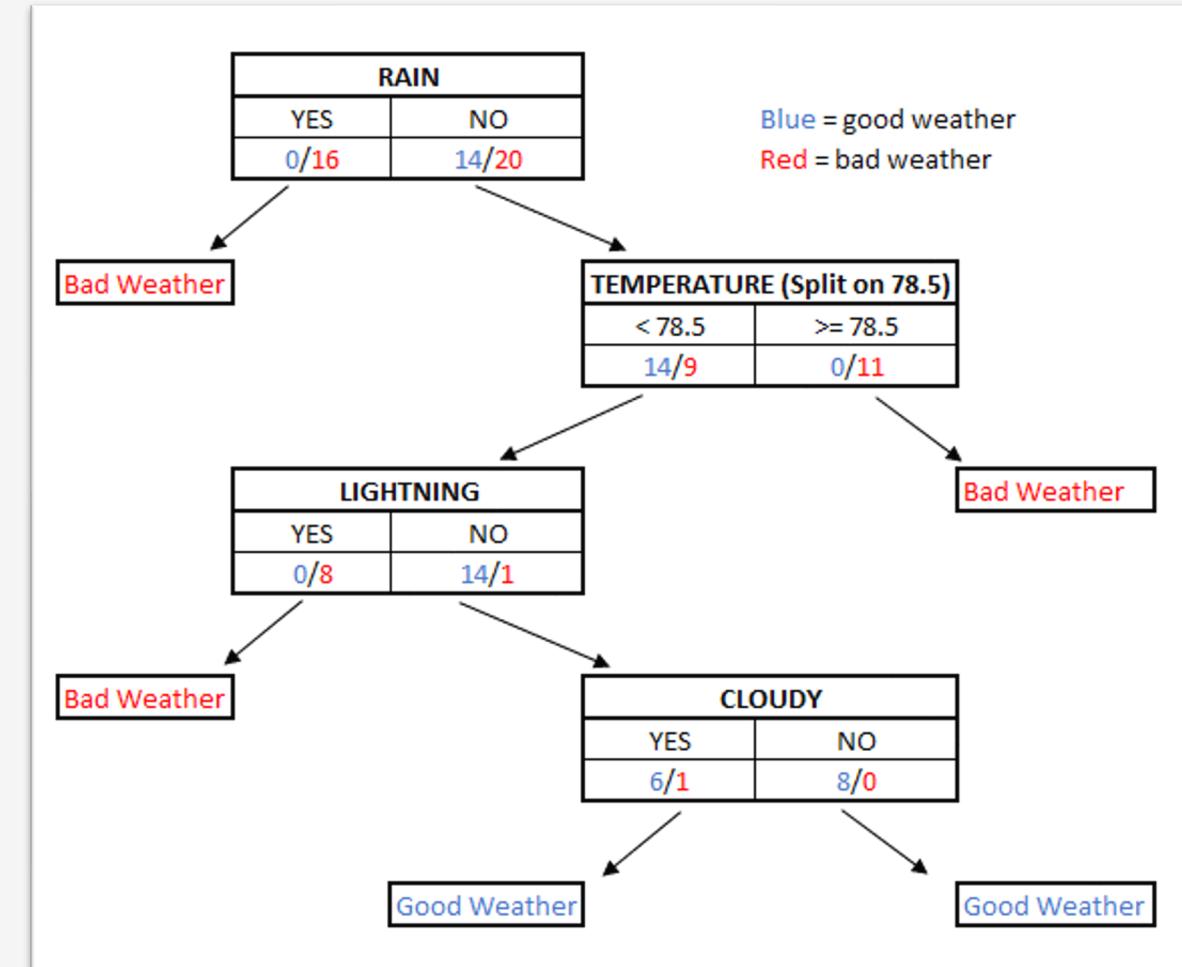


# Decision Trees – Getting to the “Decision” Part

Hopefully by now, you have a strong intuition on what decision trees are trying to accomplish.

However, you may have some lingering questions on **HOW** to build a decision tree.

- At a given step, how do I determine which property is the best to split on?
- Where do I split continuous variables like **TEMPERATURE**?
- When do I stop splitting and end my decision tree?



# Gini Impurity

---

To build a decision tree, we must discuss the concept of **impurity** which describes how mixed something is.

*EXAMPLE: If we have 6 dogs and 3 cats in a kennel, we do not purely have dogs or cats.*

**Gini Impurity** is a common way to measure impurity using this function (for events A and B):

$$1 - (\text{Probability of } A)^2 - (\text{Probability of } B)^2$$

So the Gini impurity of dogs versus cats in the kennel:

$$1 - \left(\frac{6}{6+3}\right)^2 - \left(\frac{3}{6+3}\right)^2 = .44444$$

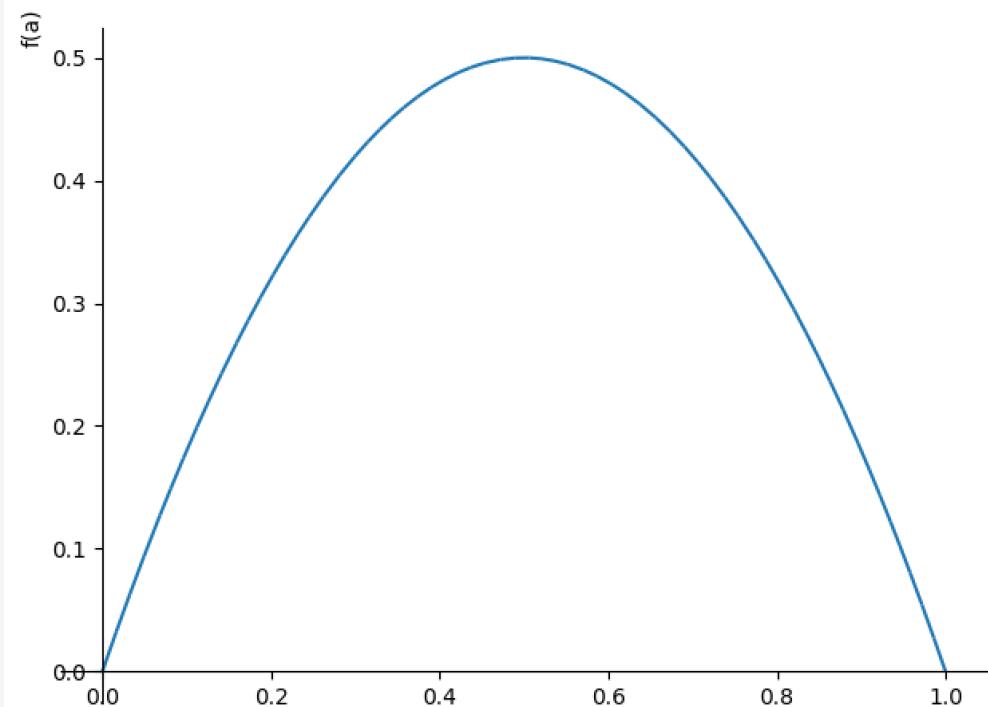
# Gini Impurity – Plotting for the Curious

---

The Gini impurity can never exceed 0.5, and this makes sense because having something 100% mixed (1.0) doesn't make sense.

To the right we use SymPy to plot the gini impurity function.

```
from sympy import *
a = symbols('a')
b = 1.0 - a
gini_impurity_f = 1 - a**2 - b**2
plot(gini_impurity_f, (a, 0, 1))
```



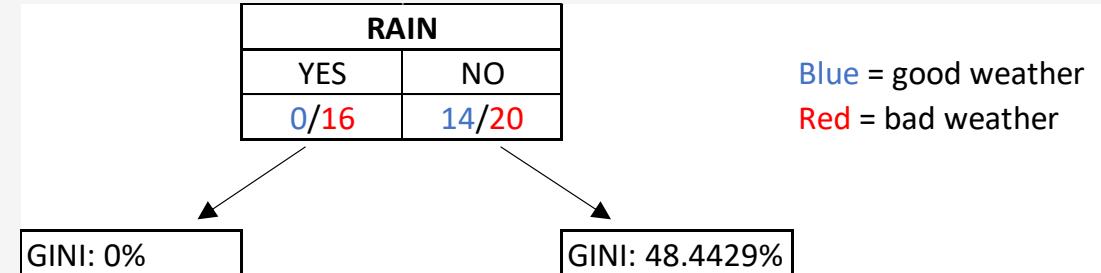
# Gini Impurity

---

Going back to the weather example, here's how we look at Gini impurity for **RAIN**.

Notice how the impurity for “good/bad weather” on the YES side is 0% meaning it is pure.

$$1 - \left( \frac{0}{0 + 16} \right)^2 - \left( \frac{16}{0 + 16} \right)^2 = 0$$



# Gini Impurity

---

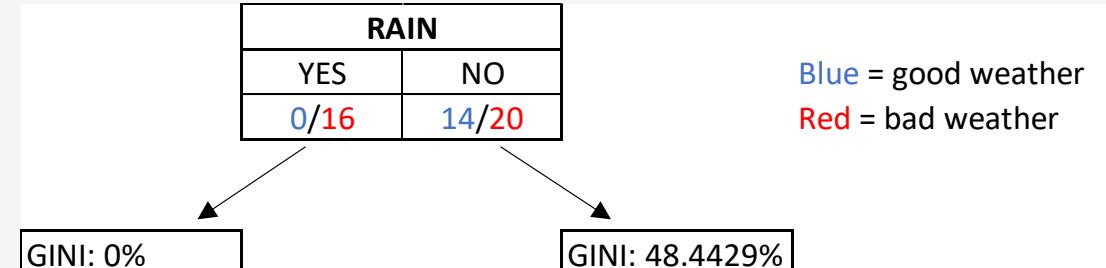
Going back to the weather example, here's how we look at Gini impurity for **RAIN**.

Notice how the impurity for “good/bad weather” on the YES side is 0% meaning it is pure.

$$1 - \left( \frac{0}{0 + 16} \right)^2 - \left( \frac{16}{0 + 16} \right)^2 = 0$$

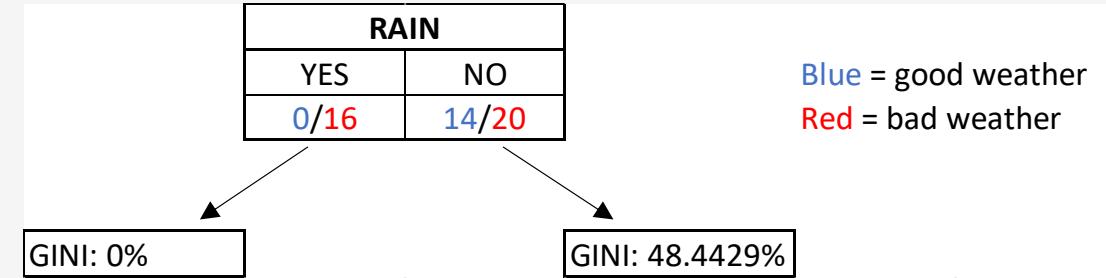
On the NO side we got an impurity of 48.4429%

$$1 - \left( \frac{14}{14 + 20} \right)^2 - \left( \frac{20}{14 + 20} \right)^2 = .484429$$



# Weighing Gini Impurities

To calculate the entire impurity of using the **RAIN** property for a split, we weight these two impurities together.



$$0 \frac{0+16}{0+16+14+20} + 0.484429 \frac{14+20}{0+16+14+20} = .32941172$$

This is known as the **weighted average Gini impurity**, and it can be used as a measure of quality for splitting with that property. *Lowest impurity is best!*

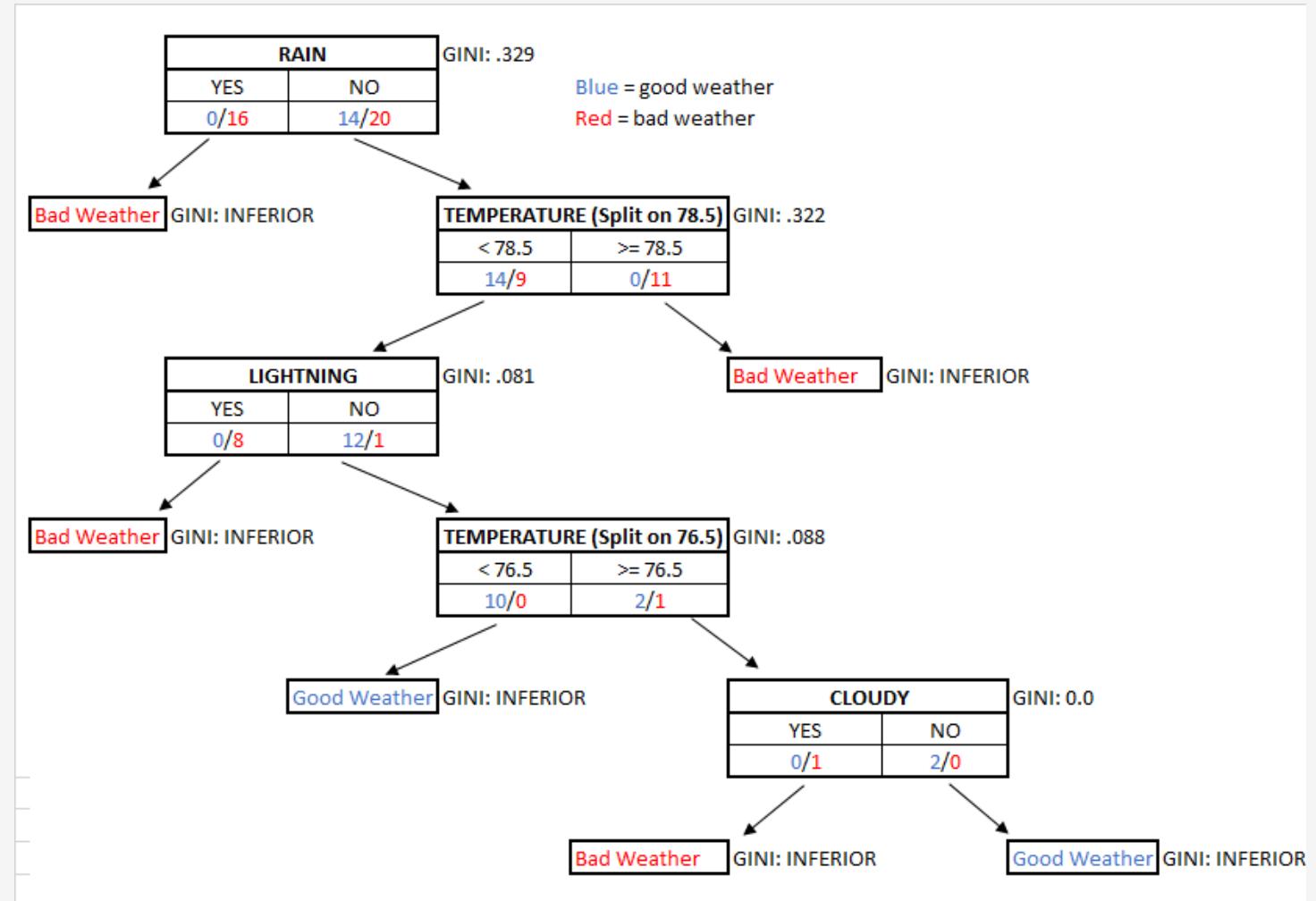
At every step we choose the property that provides the least weighted average Gini impurity.

# Building a Decision Tree with GINI

At each decision, we choose the next property with the best GINI impurity and split on it

EXAMPLE: After **RAIN**, the property **TEMPERATURE** has the next best GINI for those 34 items on the "NO" side.

**When the next property's weighted GINI is inferior to the previous GINI, we stop the branch there.**



# How to Split Continuous Variables?

---

Once you got the GINI and weighted GINI down, implementing a decision tree becomes relatively simple.

The goal always is to do a split that reduces the GINI impurity until it cannot be reduced anymore.

The only remaining question is how to split on continuous variables, which is not as straightforward as binary (true/false) or discrete (dog/cat/bird) variables.

TEMERATURE
74
76
77
80
81
83
84

Observe the following temperatures on the right. Any guesses on what is the best way to find the optimal split?

TEMPERATURE (Split on 78.5)	
< 78.5	$\geq 78.5$
14/9	0/11

# How to Split Continuous Variables?

---

Continuous variables extend the GINI concept further.

When you evaluate a continuous variable for its GINI, first sort the data for that variable then produce the rolling 2-value averages, each of which is a candidate for the split value.

Then choose the 2-value average that produces the best GINI impurity that splits on that value, which for the Temperature node we saw previously was 78.5.

TEMPERATURE	ROLLING 2-VALUE AVG
74	
76	75
77	76.5
<b>80</b>	<b>78.5</b>
81	80.5
83	82
84	83.5

TEMPERATURE (Split on 78.5)	
< 78.5	$\geq 78.5$
14/9	0/11

# Hands On: Decision Trees

---

The screenshot shows a code editor and a terminal window. The code editor displays Python code for calculating Gini impurity and splitting data based on weather features. The terminal window shows the execution of the code and the resulting decision tree structure.

```
file Edit View Navigate Code Refactor Run Tools VCS Window Help
31     # get impurity for provided samples
32     def gini_impurity(samples):
33         good_weather_item_ct = sum(1 for weather_item in samples if weather_item.good_weather_ind == 1)
34         bad_weather_item_ct = sum(1 for weather_item in samples if weather_item.good_weather_ind == 0)
35         sample_ct = len(samples)
36
37         return 1.0 - (good_weather_item_ct / sample_ct) ** 2 - (bad_weather_item_ct / sample_ct) ** 2
38
39
40     # get weighted impurity for entire
41     def gini_impurity_for_split(feature, split_value, samples):
42         feature_positive_items = [weather_item for weather_item in samples if feature.value_extractor(weather_item) >= split_value]
43         feature_negative_items = [weather_item for weather_item in samples if feature.value_extractor(weather_item) < split_value]
44
45         return (gini_impurity(feature_positive_items) * (len(feature_positive_items) / len(samples))) + (
46             gini_impurity(feature_negative_items) * (len(feature_negative_items) / len(samples)))
47
gini_impurity()
run: good_weather_classification
C:\Users\thoma\AppData\Local\Programs\Python\Python37\python.exe
C:/git/oreilly_machine_learning_from_scratch/code/section_v/good_weather_classification.py
(0) Rain split on 0.5, 34|16, Impurity: 0.3294117647058824
    (1) Temperature split on 78.5, 23|11, Impurity: 0.32225063938618925
        (2) Lightning split on 0.5, 15|8, Impurity: 0.08115942028985504
            (3) Temperature split on 76.5, 12|3, Impurity: 0.0888888888888888
                (4) Cloudy split on 0.5, 2|1, Impurity: 0.0
Predict if weather is good {rain}, {lightning}, {cloudy}, {temperature}:
```

# Decision Trees and Overfitting

---

Decision trees work well, so much they are notorious for overfitting.

**Overfitting** again means the model fits to the training data too exactly, and therefore becomes unreliable for predicting new data.

One way to adapt decision trees to not overfit is to utilize random forests, which generates hundreds of decision trees with randomly sampled data and features.

This forces the model to use different subsets of data and properties, and therefore not overfit to the data set.



# Random Forests

---

**Random Forests** are a machine learning technique that generates hundreds of decision trees, where each one builds off partial random data and properties, rather than all the data.

- Typically each decision tree will train with only 2/3 of the randomly sampled data, which is known as **bootstrapping**.
- You should also only consider a subset of variables when evaluating each node, forcing your decision tree to utilize other variables.
- You can use the other 1/3 of the data (known as the **"out-of-bag"** **data**) as the test data to evaluate prediction performance.

With these hundreds of decision trees built, you then have each tree "vote" on a prediction. The prediction with the highest votes wins.



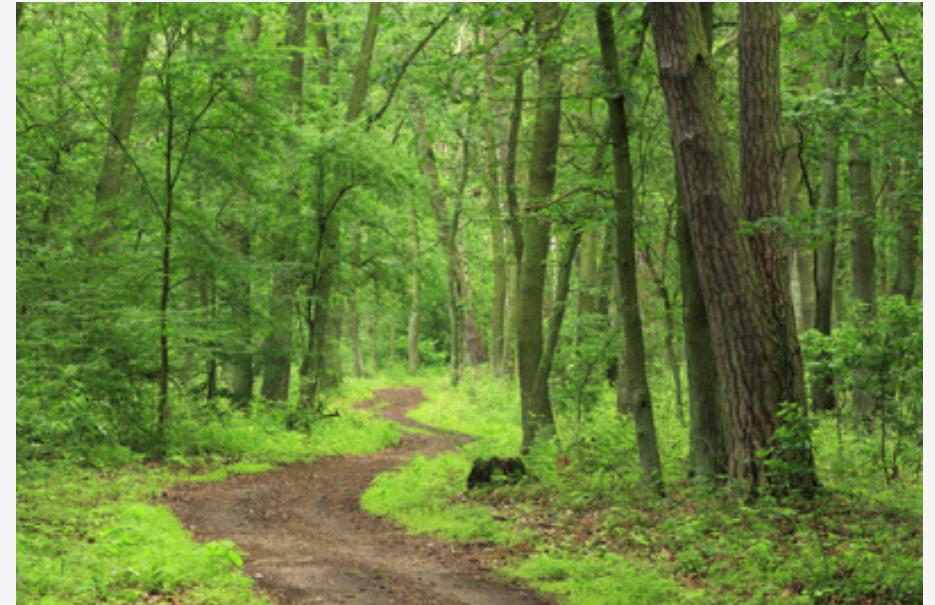
# Fine-Tuning Random Forests

---

Because each decision tree in a random forest uses only 2/3 of the data for training, you can use the other 1/3 as test data.

You can choose the number of variables that yields the best accuracy on that test data.

For this exercise, I'm just going to randomly sample 2-3 properties for each node and call it a day.



# Hands-On: Random Forests

---

The screenshot shows a Python code editor with a file named `random_forest.py`. The code implements a function `build_leaf` for a decision tree. The code is annotated with line numbers from 109 to 124. The editor has a dark theme with syntax highlighting. Below the code editor is a terminal window showing the output of a prediction. The terminal window title bar says "Run: employment\_retention\_random\_forest". The output text is:

```
Predict if weather is good {rain},{lightning},{cloudy},{temperature}: 1,0,0,75
Good weather vote: 90/299
Weather is bad, 30.1% confident it is good
```

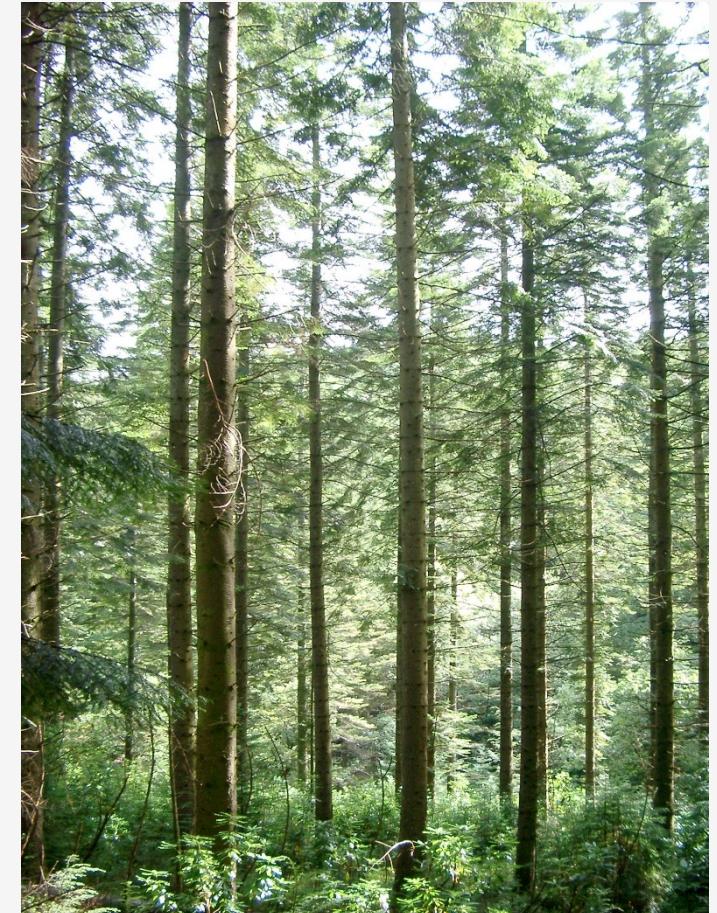
# Going Forward

---

Decision trees and random forests are a powerful and flexible tool in machine learning, and together work quite well for a lot of problems.

Decision trees can also be adapted to do regression with regression trees and enhanced with techniques like gradient boosting.

If I were to take one machine learning algorithm with me, it would be decision trees and all of its variants.

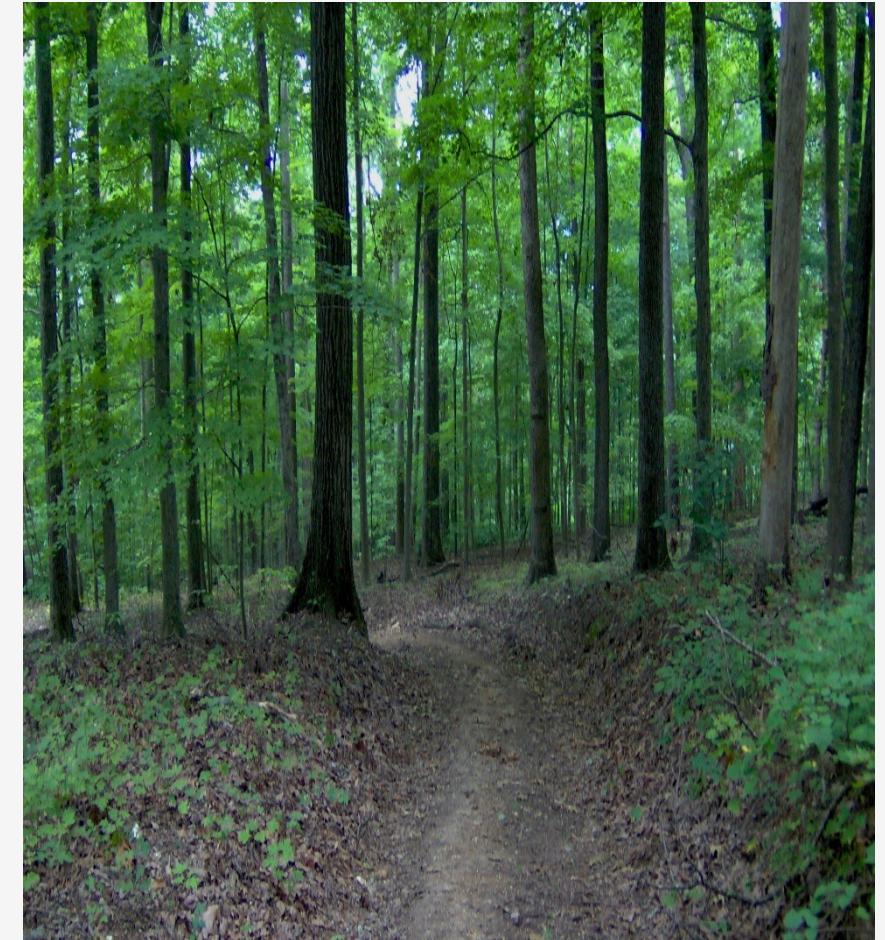


# Quiz Time!

---

Decision trees are effective because they do not overfit.

- A) True
- B) False



# Quiz Time!

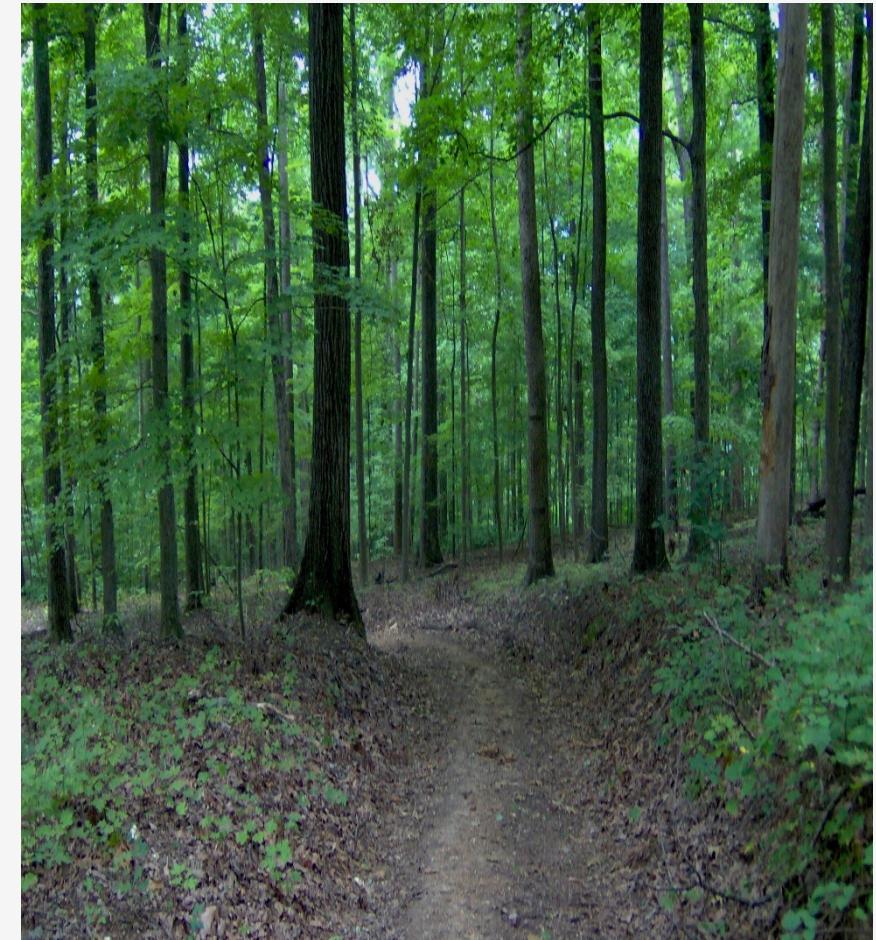
---

Decision trees are effective because they do not overfit.

A) True

B) False

Decision trees are so effective at fitting to data that they often overfit. Random forests and other techniques can be used to prevent overfitting



# Quiz Time!

---

You have 100 patients. 44 have Celiac disease, 56 do not.

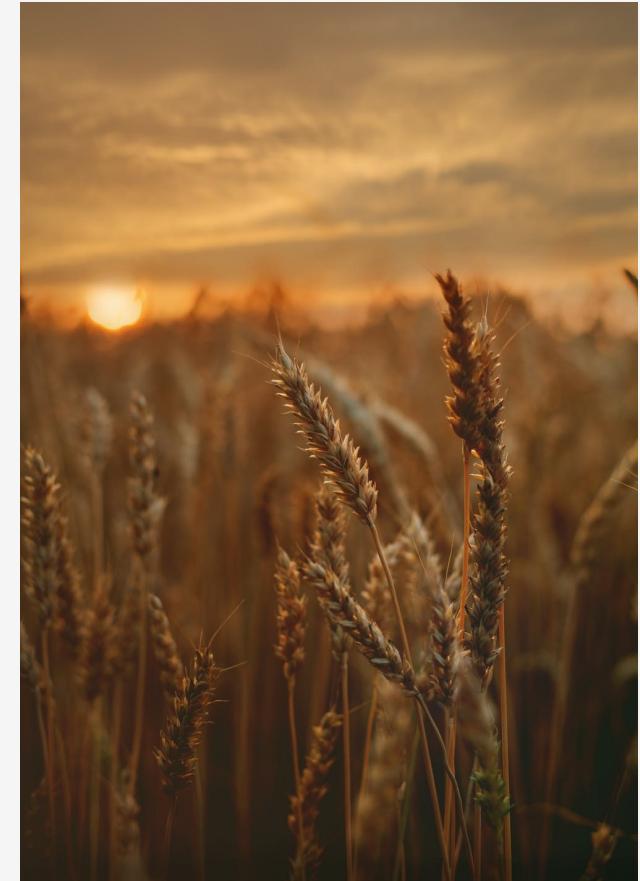
What is the GINI impurity of Celiac to non-Celiac patients?

A) .4928

B) .44

C) .56

D) .4720



# Quiz Time!

---

You have 100 patients. 44 have Celiac disease, 56 do not.

What is the GINI impurity of Celiac to non-Celiac patients?

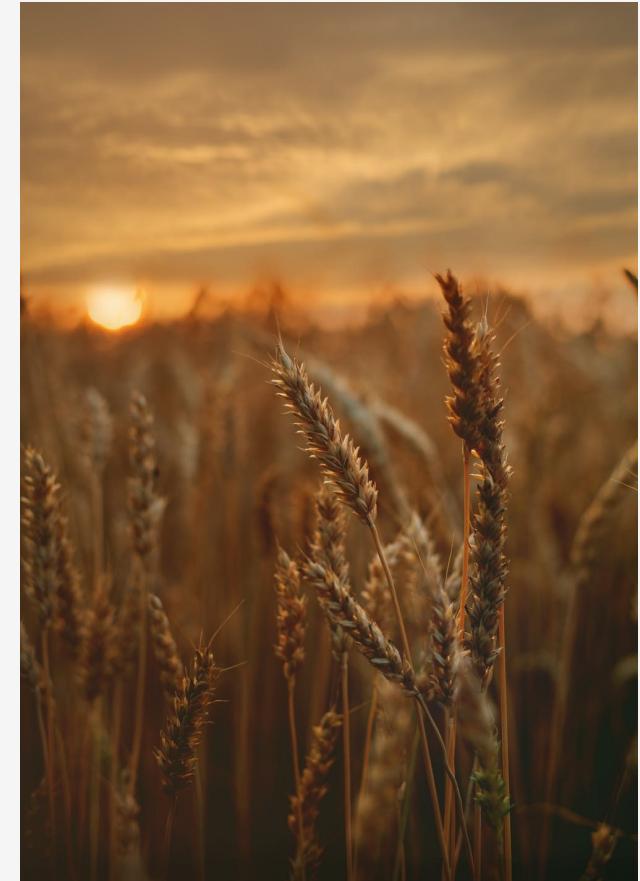
A) .4928

$$1 - \left(\frac{44}{100}\right)^2 - \left(\frac{56}{100}\right)^2 = .4928$$

B) .44

C) .56

D) .4720



# Quiz Time!

---

You have 100 patients. 45 complained of stomach pain after drinking beer, the other 55 did not.

Of the 45 patients with stomach pain, 37 have Celiac disease.

Of the 55 patients without stomach pain, 7 have Celiac disease.

What is the weighted GINI impurity for using post-beer stomach pain to determine whether a patient has Celiac disease?

- A) .50
- B) .2537
- C) .4506
- D) .45



## Quiz Time!

---

You have 100 patients. 45 complained of stomach pain after drinking beer, the other 55 did not.

Of the 45 patients with stomach pain, 37 have Celiac disease.

Of the 55 patients without stomach pain, 7 have Celiac disease.

What is the weighted GINI impurity for using post-beer stomach pain to determine whether a patient has Celiac disease?

- A) .50
- B) .2537
- C) .4506
- D) .45



$$\left(\frac{45}{100}\right)\left(1 - \left(\frac{37}{45}\right)^2 - \left(\frac{8}{45}\right)^2\right) + \left(\frac{55}{100}\right)\left(1 - \left(\frac{7}{55}\right)^2 - \left(\frac{48}{55}\right)^2\right) = .2537$$

# Section VI

# Neural Networks

# What Are Neural Networks?

---

**Neural networks** are multi-layered regressions that accept numeric inputs and produce numeric outputs, through a series of “middle layers” of multiplication and addition.

- A series of weights are multiplied and added against one input layer which feeds into another.
- Practical applications of neural networks include recognition and generation of image, audio, video, and language.

Neural networks are utilized to employ **deep learning**, which utilizes middle layers hence the name “deep”.



# How Much Can We Cover?

---

Unfortunately, we do not have a lot of time in this class, so we cannot cover the topic of building a neural network in great depth which can take several hours in itself!

But we will cover the key concepts and then show the code needed to build the entire neural network (including backpropagation).

For a complete and detailed walkthrough on the code and building the neural network from scratch, read Chapter 7 of *Essential Math for Data Science*.

O'REILLY®

## Essential Math for Data Science

Take Control of Your Data with Fundamental Linear Algebra, Probability, and Statistics



Thomas Nield

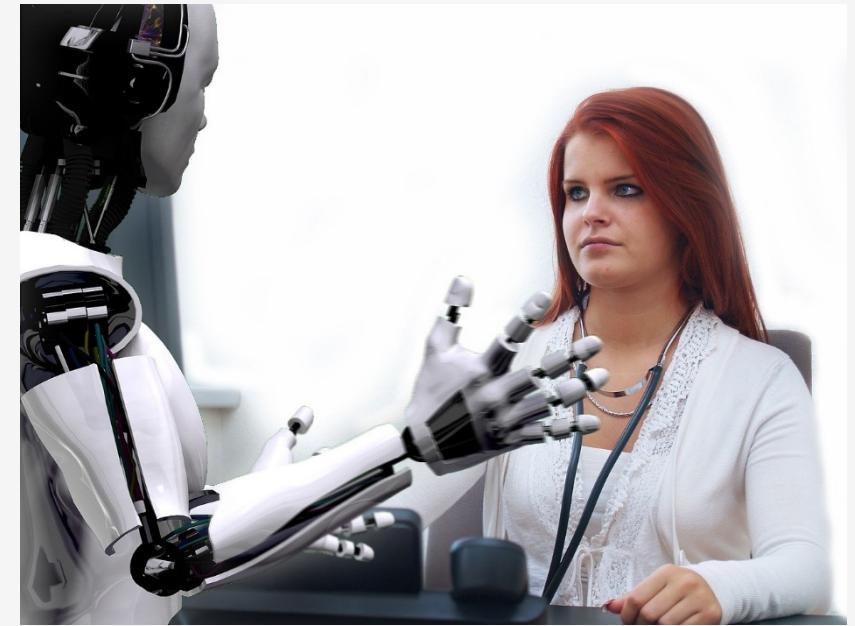
# Neural Networks - Theoretical versus Practical Applications

---

Neural networks have usefulness in certain domains, but it is important to discern theoretical versus practical capabilities.

- **Theoretical capabilities** range from interesting research ideas to sensational science fiction, like artificial general intelligence.
- **Practical capabilities** are often narrow tasks like image/audio/video/language recognition and generation.
- Discerning the above will help prevent confusion and not chase red herrings caused by hype.

Neural networks are *loosely* inspired by biological brains but are by no means a replication of them, so keep that in mind when managing expectations.

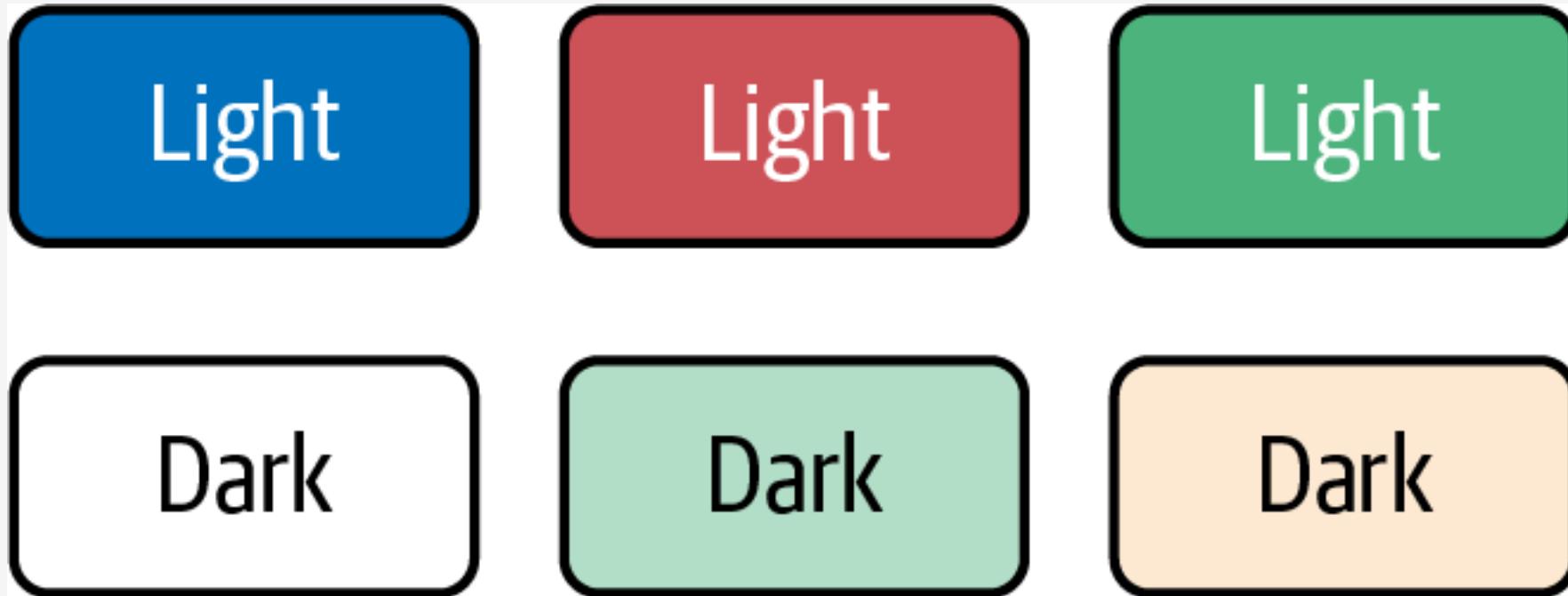


A typical stock photo for AI articles and media, which often contain anthropomorphized androids and other science fiction imagery.

# Objective

---

We want to predict a light (1) or dark (0) font for different background colors.

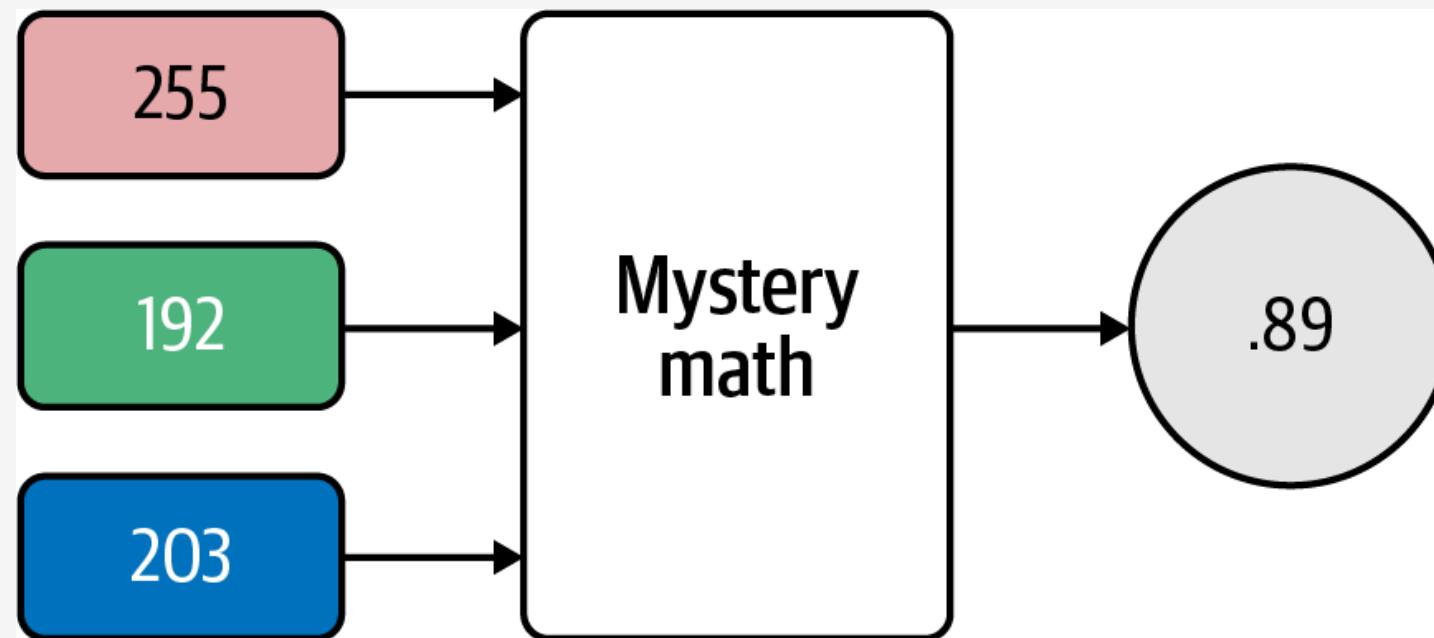


The best way to solve this would probably be a logistic regression or even a [plain formula](#), but we will use a neural network as a simple exercise.

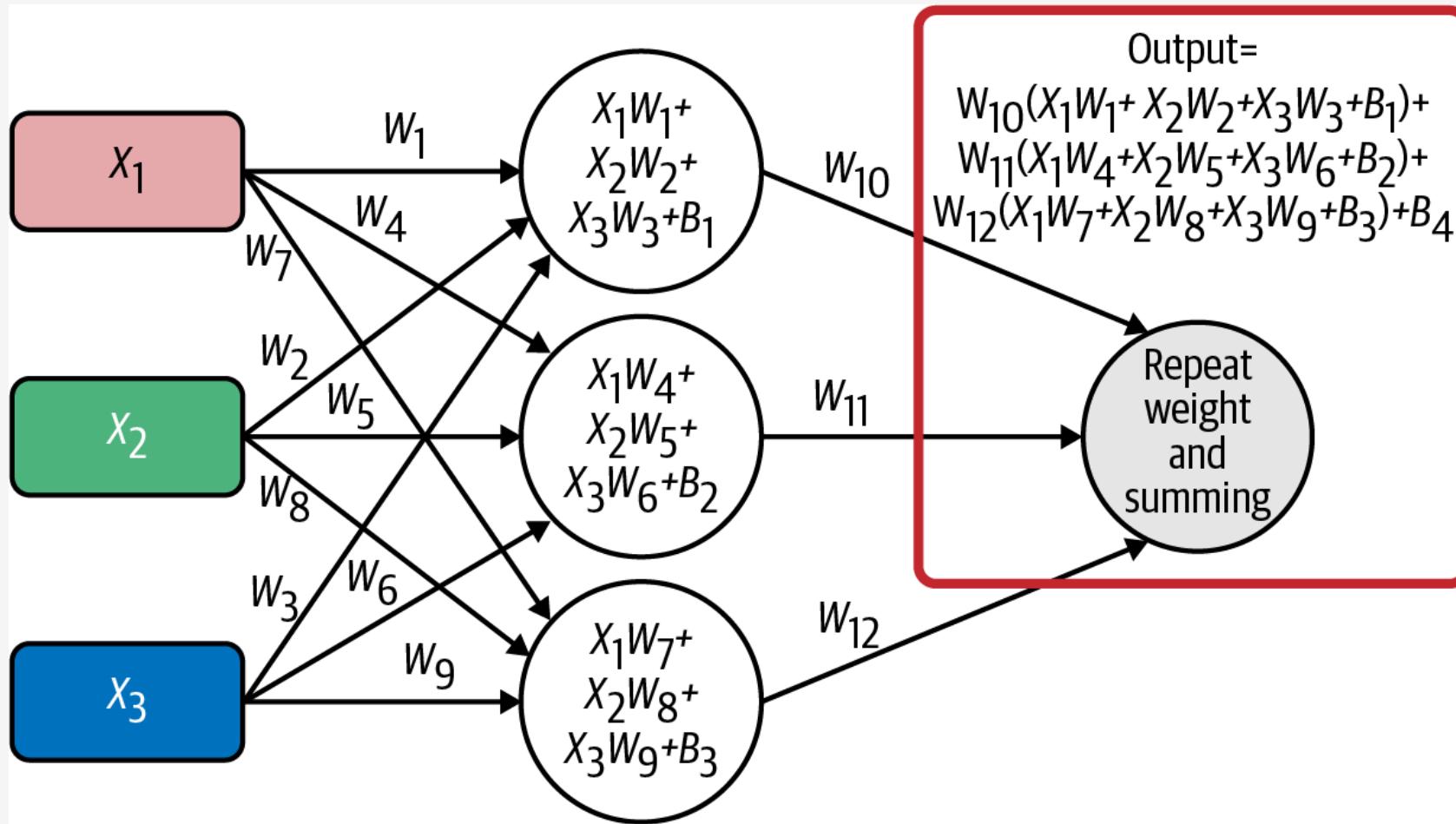
# A Simple Neural Network

---

Let's represent background color as 3 numeric RGB inputs, and predict whether a DARK/LIGHT font should be used.

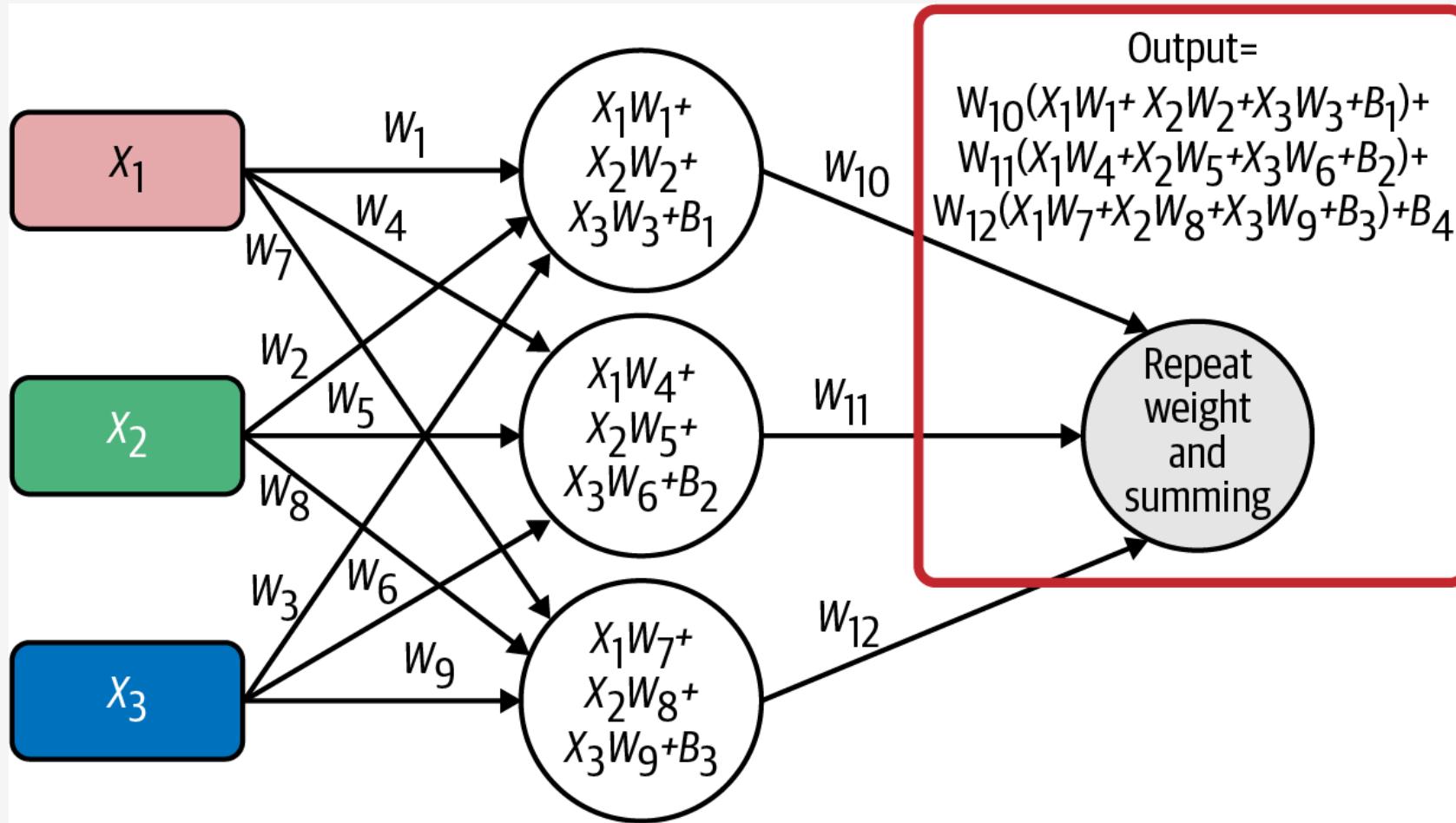


# A Simple Neural Network



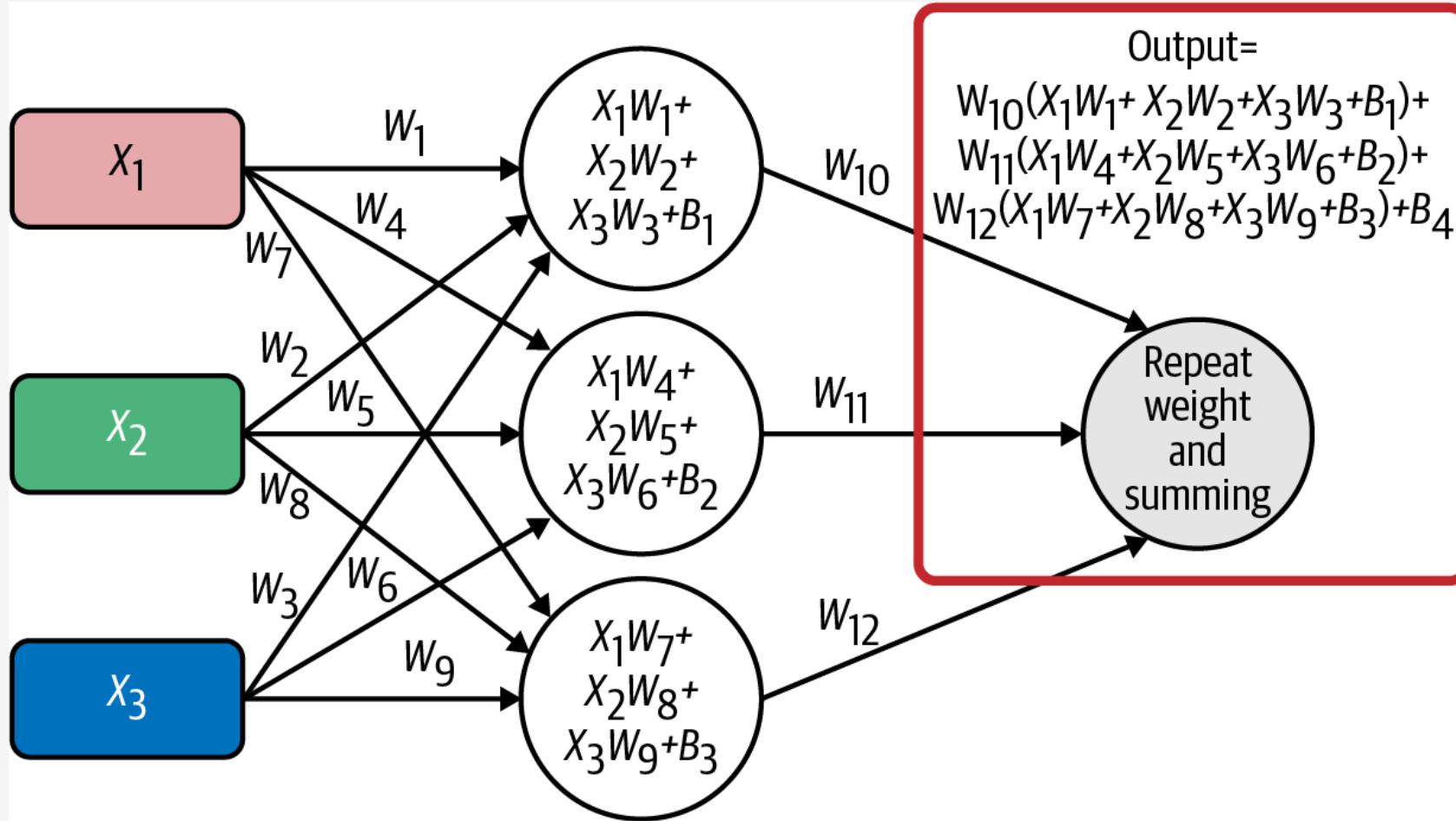
# A Simple Neural Network

This is the “mystery math”



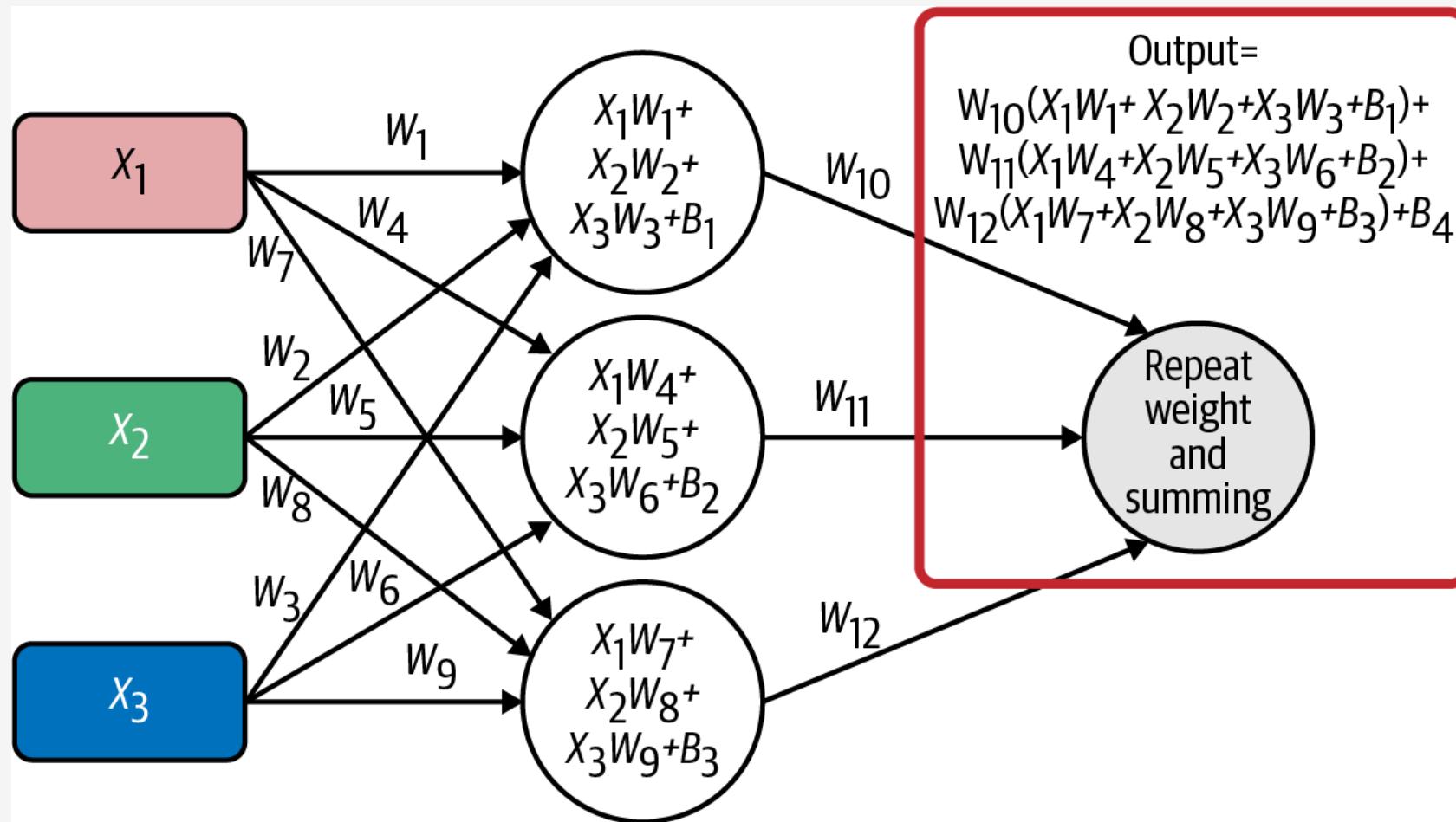
# A Simple Neural Network

Each weight  $w_x$  value is between -1.0 and 1.0



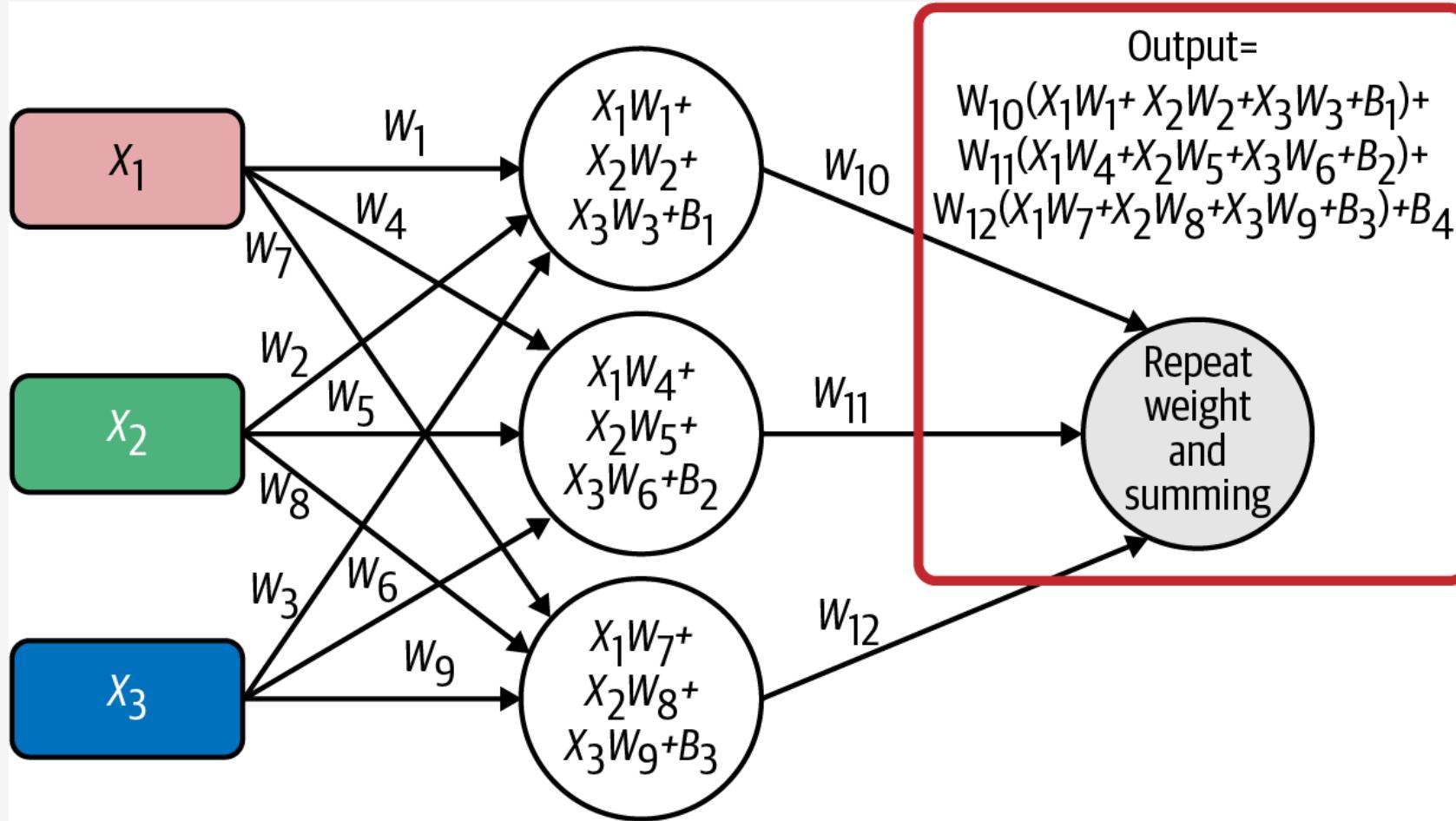
# A Simple Neural Network

Note we should also add a  $b_x$  bias value (between 0 and 1) to each node.



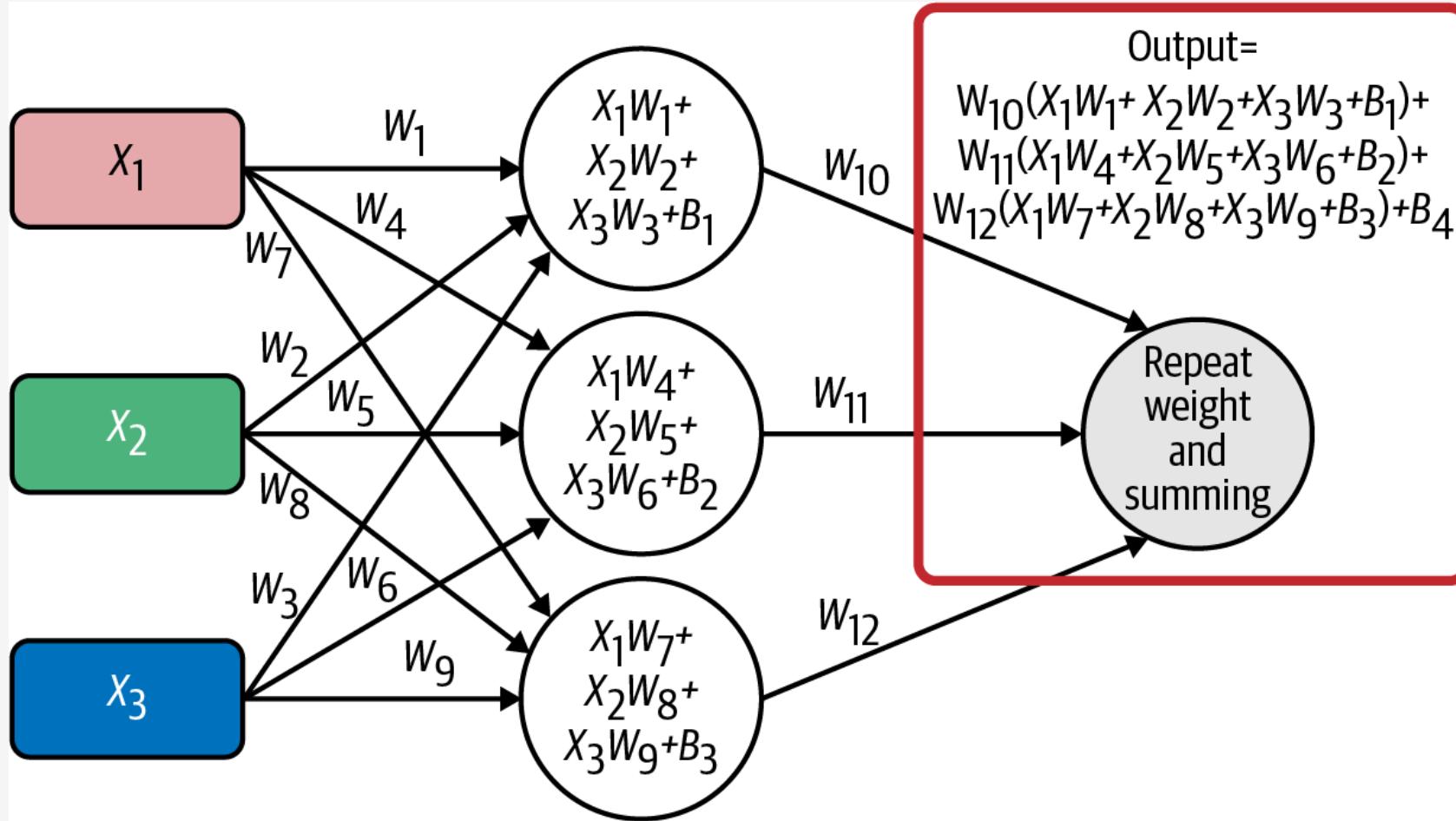
# A Simple Neural Network

What are the *optimal* weight (and bias) values to get the desired output?



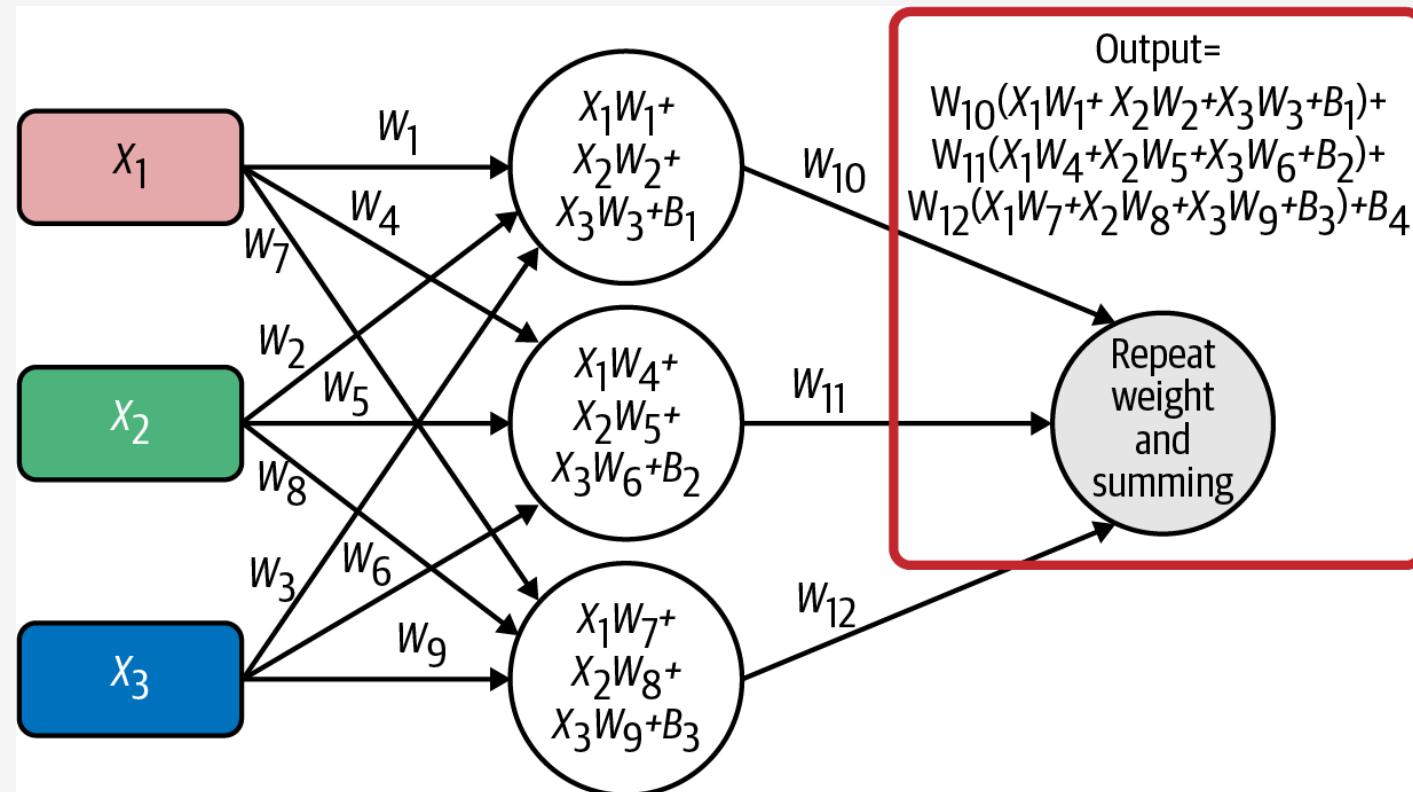
# A Simple Neural Network

**Answer:** Like most machine learning, this is an optimization problem!



# A Simple Neural Network

We need to solve for the weight (and bias) values that gets our training colors as close to their desired outputs as possible using hill climbing, simulated annealing, gradient descent, or other optimization methodologies.

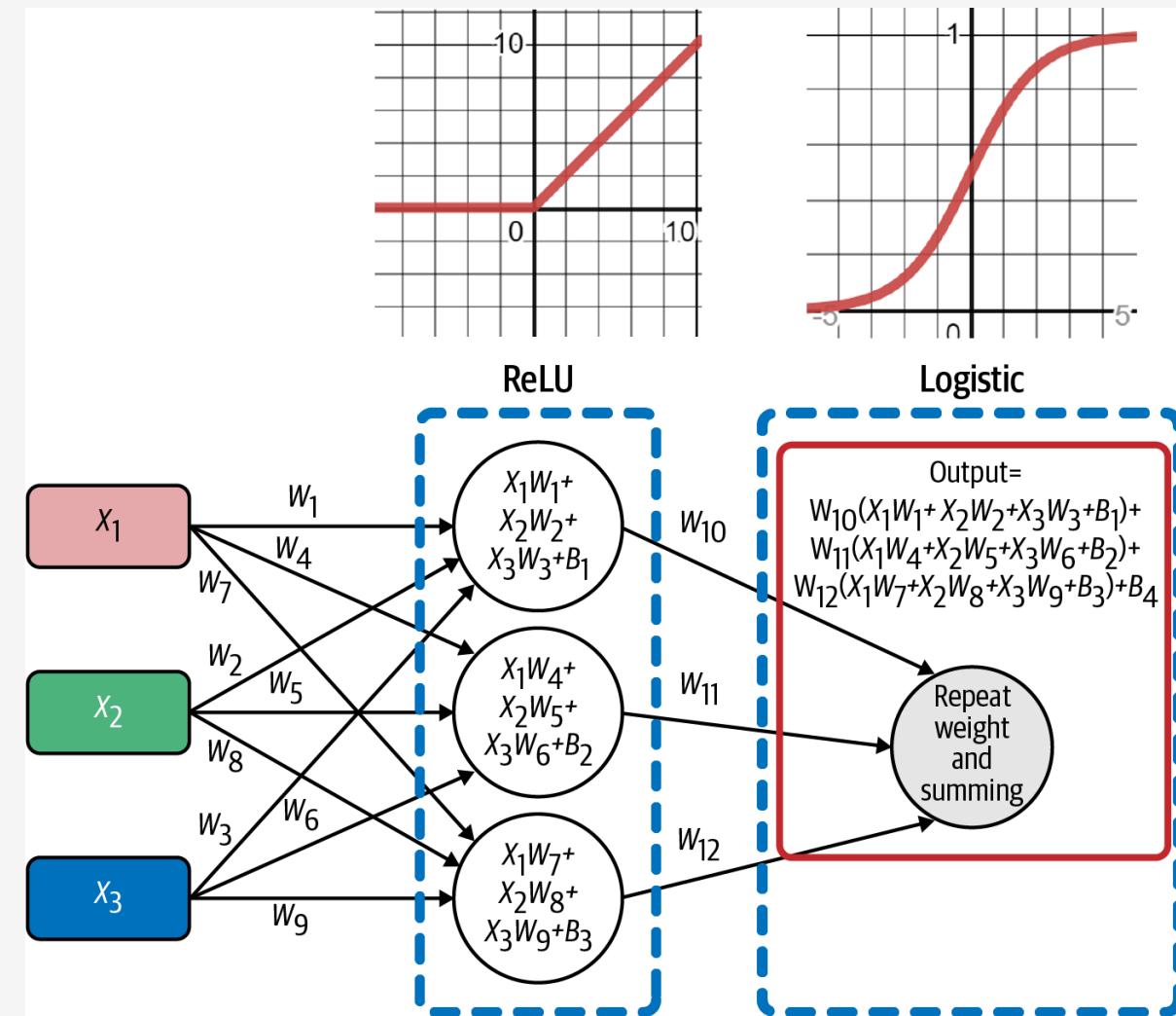


# Activation Functions

You should also use **activation functions** on each layer.

These are nonlinear functions that smooth, scale, or compress the resulting sum values.

Activation functions make the middle layers productive, separating features from each preceding layer.



# Activation Functions

---

```
import numpy as np
from scipy import special
import math

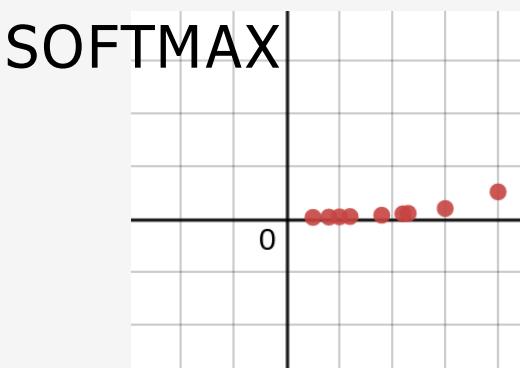
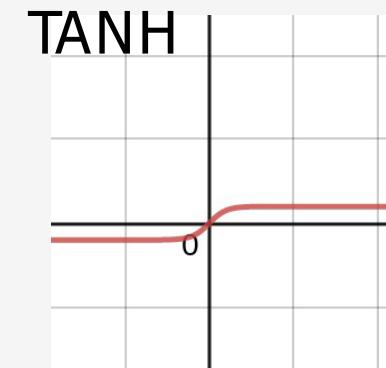
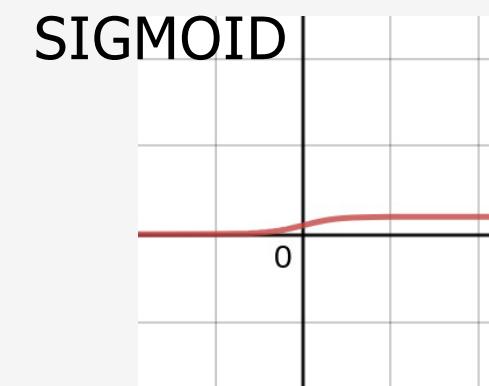
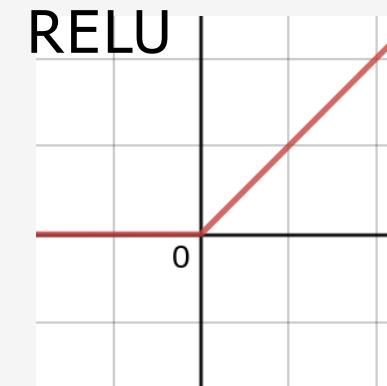
# Activation functions
def tanh(x):
    return np.tanh(x)

def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def relu(x):
    return np.maximum(x, 0)

def softmax(x):
    return special.softmax(x, axis=0)
```

Four common neural network activation functions implemented in Python.



<https://www.desmos.com/calculator/jwjn5rwfy6>

# Building Neural Networks with NumPy

---

We avoided linear algebra up to this point because it can be distracting to understanding machine learning, but it does make neural networks easier to build.

On a practical level, **linear algebra** is about expressing numeric transformations with matrices.

Hopefully a **matrix** is a familiar concept where you express data as grids of numbers that can be multiplied, added, and transformed in various ways.

You can express a neural network with numeric training data and weights in matrix forms.

**Basic Math**

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

**Dangerous Artificial Intelligence**

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} * \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} * \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} * \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

# Building a Neural Network

---

- 1) First use NumPy to express your RGB input colors as a matrix. You will need to transpose the records, so each record is a column.
- 2) Since 255 is the maximum value for a color field, divide each value in the matrix by 255 to scale it between 0 and 1.

$$\begin{array}{ccccccccc} \text{Input RGB Colors} & & & & \text{Scaled Input RGB Colors} & & & & \\ \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & 100 & \dots \\ 0 & 128 & 139 & 205 & 238 & 255 & 0 & \dots \end{matrix} & \times & \frac{1}{255} = & & \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & .392 & \dots \\ 0 & .501 & .545 & .803 & .933 & 1 & 0 & \dots \end{matrix} \end{array}$$

```
training_data = pd.read_csv("https://tinyurl.com/y2qmhf8r")

# Extract the input columns, scale down by 255
training_inputs = training_data.iloc[:, 0:3].values.transpose() / 255
```

# Building a Neural Network

---

- 3) Next create a  $3 \times 3$  matrix of random values between 0 and 1, which will represent the weight values between the input and middle layer. Later we will randomly adjust these values with hill-climbing to get a desired output.
- 4) During each iteration while hill climbing, we will “multiply” these matrices using the **dot product** function, which will multiply each row of the first matrix to the column of the second matrix, then sum respectively (watch [3Blue1Brown on YouTube to learn more](#) ).

Input→Middle Layer Weights	$\times$	Scaled Input RGB Colors	=	First Output
.242 .481 .473		0	=	0 .237 .257 .380 .441 .473 .188 ...
.345 .054 .783		0	=	0 .393 .426 .629 .730 .730 .021 ...
.754 .563 .673		0	=	0 .337 .366 .541 .628 .628 .220 ...
		.501 .545 .803 .933 1 0 ...		

```
input_weights = np.random.rand(3, 3)  
first_output = input_weights.dot(training_inputs)
```

# Building a Neural Network

---

- 5) Declare the biases as a matrix with random values ranging from 0 to 1, and these will be added to the dot product output.
- 6) Apply the first layer's activation function on our first layer output, which in this case we will use Relu which turns negative values into 0.

$$relu \left( \begin{bmatrix} 0 & .237 & .257 & .380 & .441 & .473 & .188 & \dots \\ 0 & .393 & .426 & .629 & .730 & .730 & .021 & \dots \\ 0 & .337 & .366 & .541 & .628 & .628 & .220 & \dots \end{bmatrix} + \begin{bmatrix} .23 \\ .61 \\ .33 \end{bmatrix} \right) = \begin{bmatrix} 0 & .237 & .257 & .380 & .441 & .473 & .188 & \dots \\ 0 & .393 & .426 & .629 & .730 & .730 & .021 & \dots \\ 0 & .337 & .366 & .541 & .628 & .628 & .220 & \dots \end{bmatrix}$$

```
input_weights = np.random.rand(3, 3)
input_bias = np.random.rand(3, 1)

def relu(x):
    return np.maximum(x, 0)

first_output = relu(input_weights.dot(training_inputs) + input_bias)
```

# Alright, I'm going to save time

---

7) Okay, I'm lazy and I want to get to the end result.

Here is what we are getting at. This is how you build the entire neural network with NumPy:

```
# Extract the input columns, scale down by 255
training_inputs = training_data.iloc[:, 0:3].values.transpose() / 255

# Build neural network with weights and biases
middle_weights = np.random.rand(3, 3)
output_weights = np.random.rand(2, 3)

middle_bias = np.random.rand(3, 1)
output_bias = np.random.rand(2, 1)

# Activation functions

def relu(x):
    return np.maximum(x, 0)

def softmax(x):
    return special.softmax(x, axis=0)

training_outputs = softmax(output_bias + output_weights.dot(relu(middle_bias + middle_weights.dot(training_inputs))))
```

The neural network!

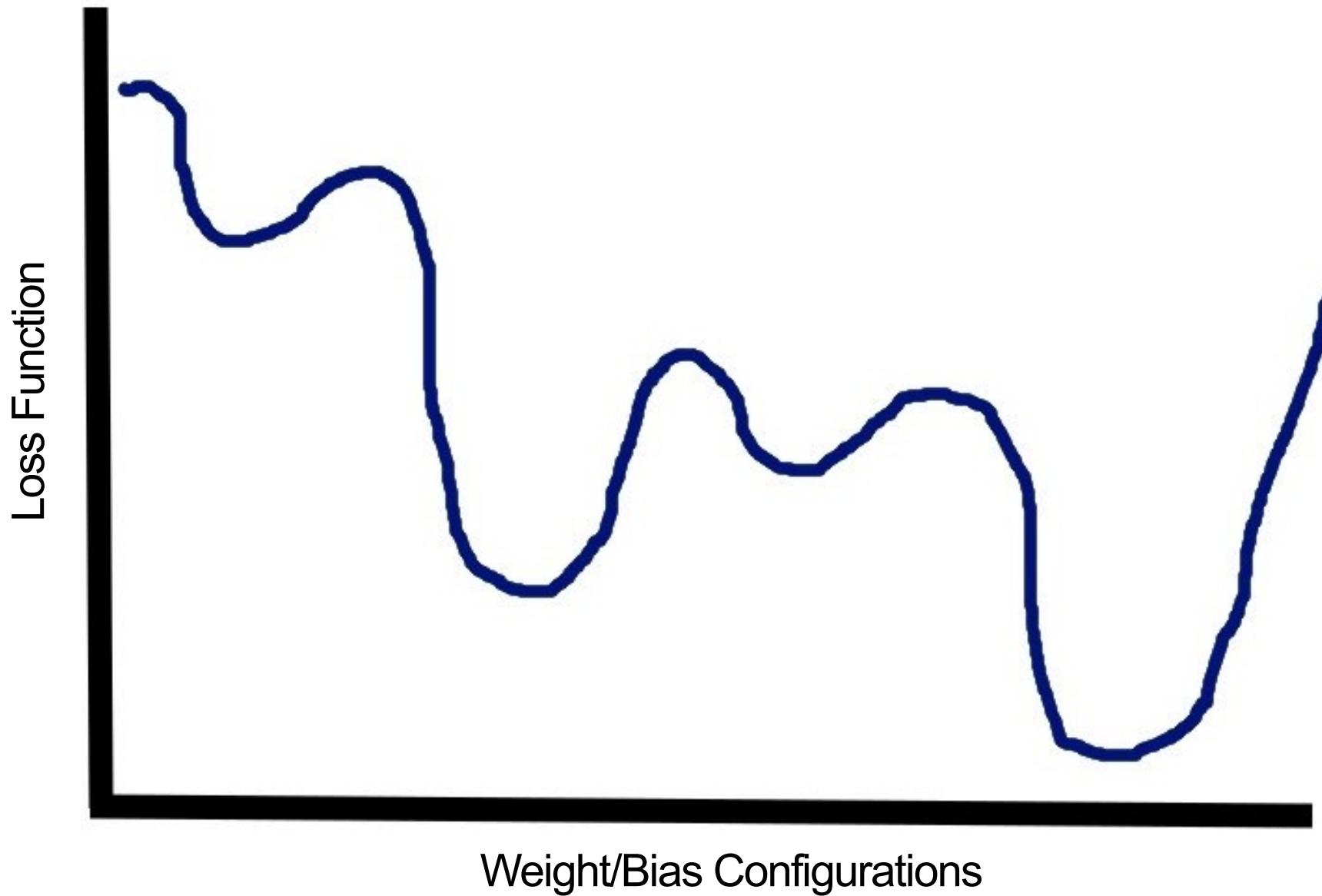


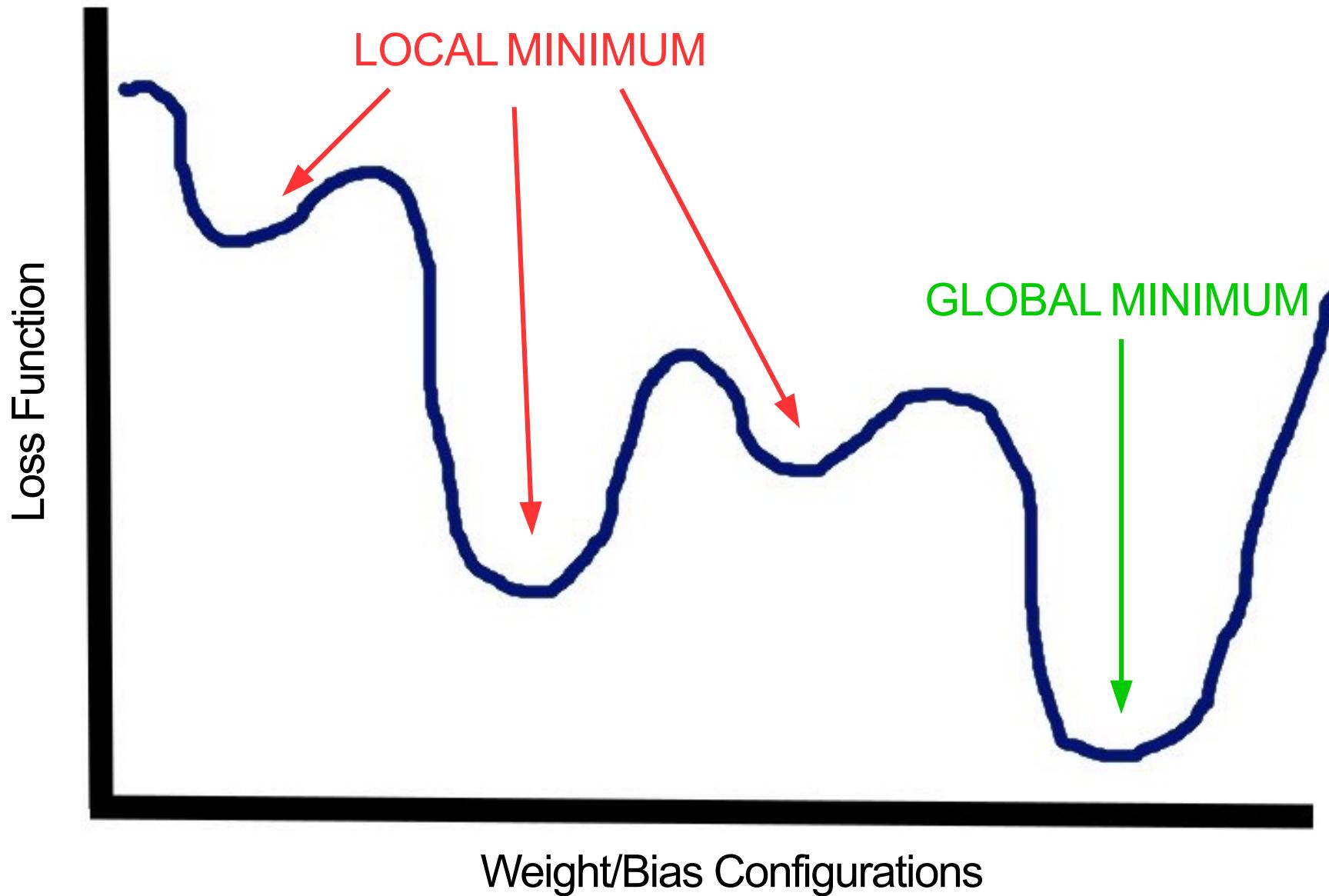
# Alright, I'm going to save time

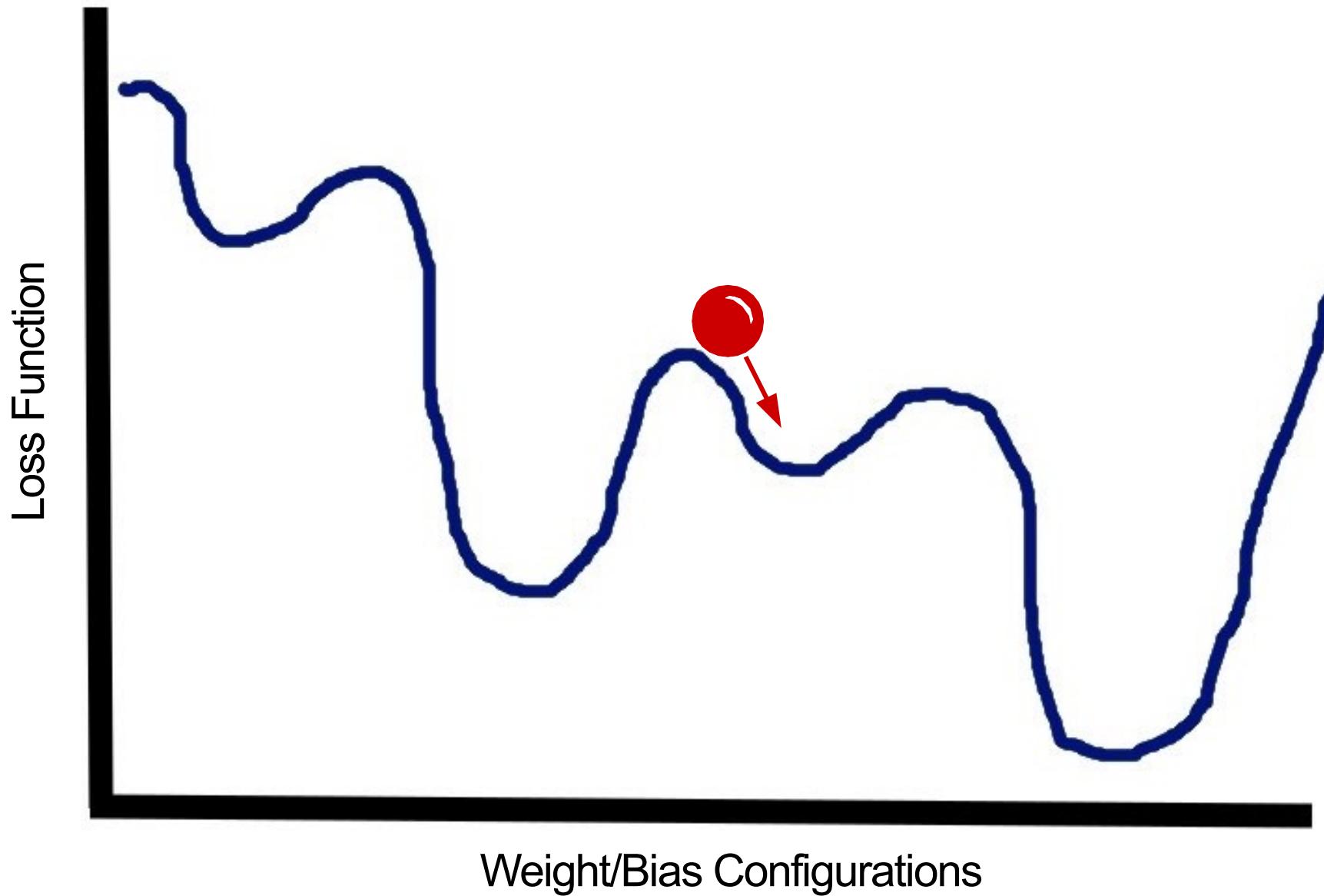
---

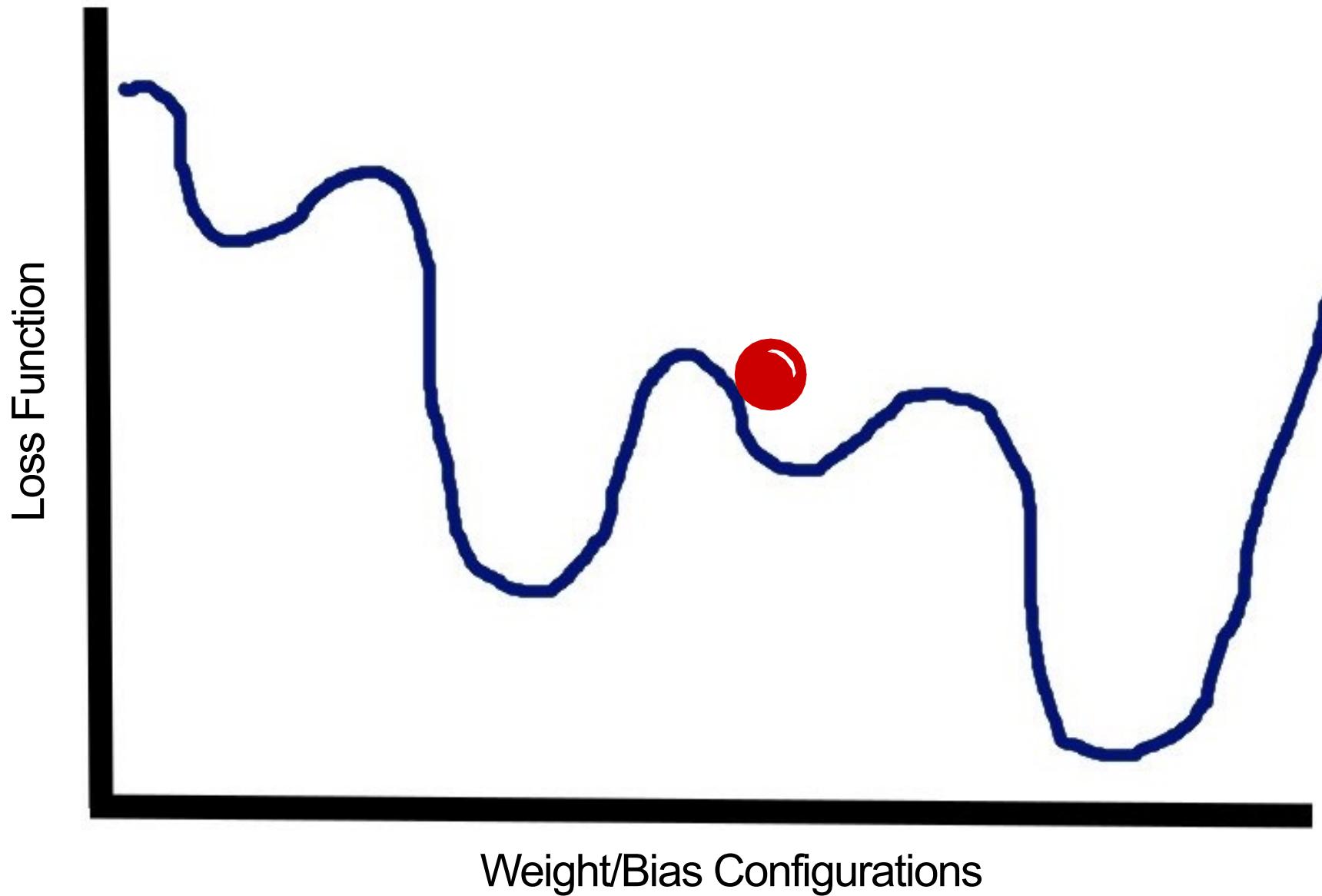
```
training_outputs = softmax(output_bias + output_weights.dot(relu(middle_bias + middle_weights.dot(training_inputs))))
```

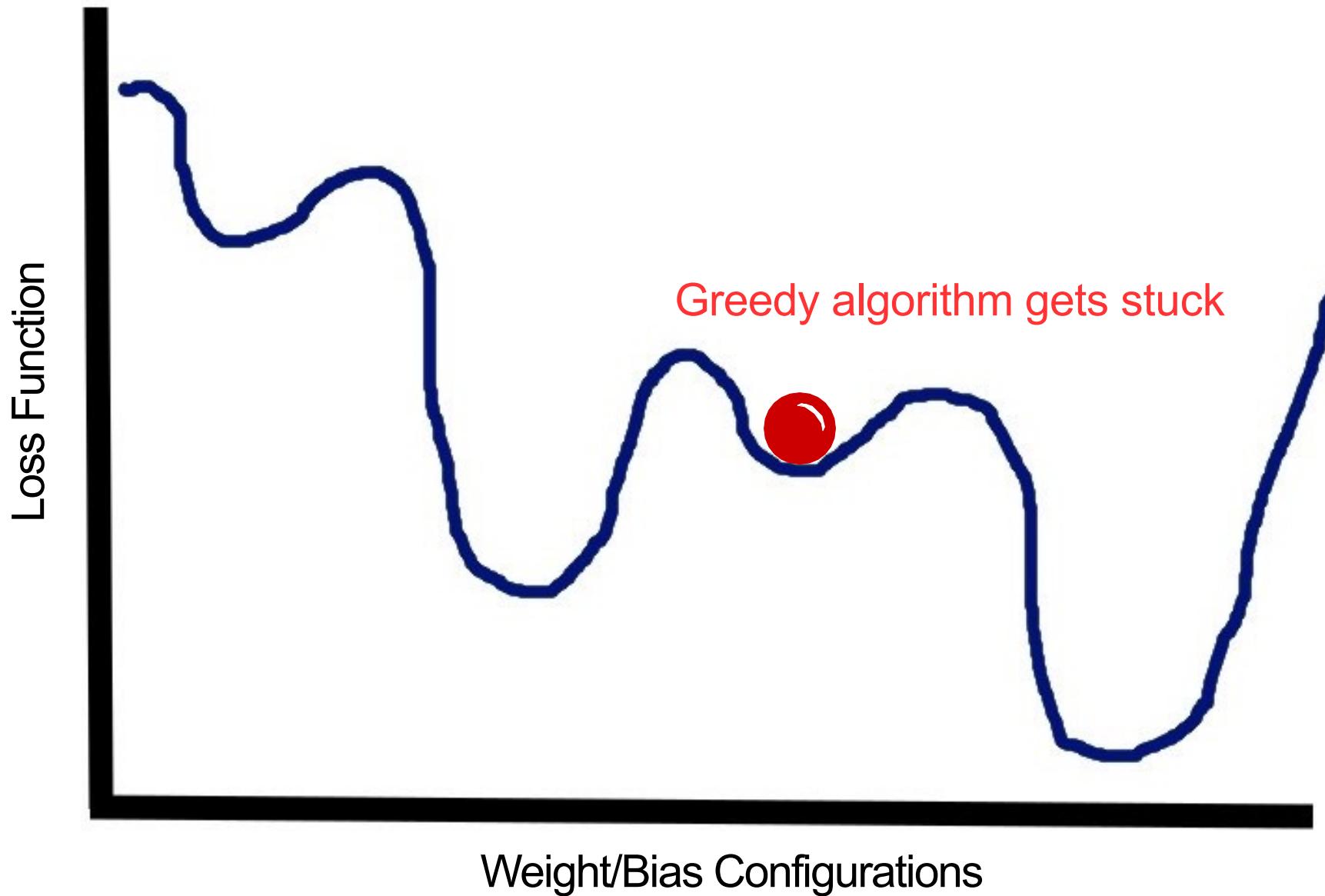
- That beautiful, magical one line executes your input values, multiplies all the weights and adds all the biases for all three layers, applies each activation function, and outputs the recommendation matrix.
- For millions of iterations, we will loop a hill-climbing algorithm randomly adjusting the weights to minimize loss.
- If we are lucky, we will get a good weight configuration that predicts the right outputs for new colors.

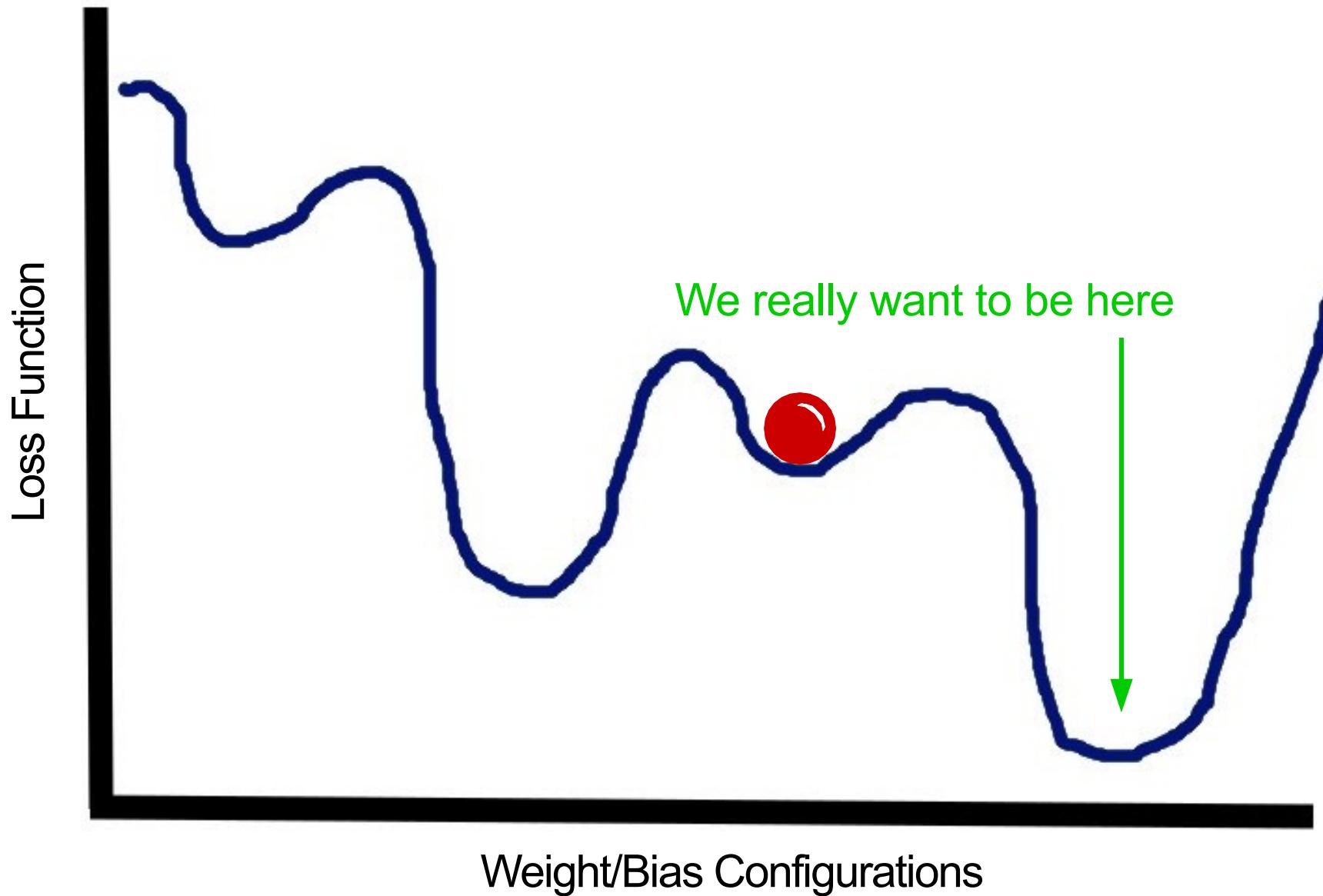


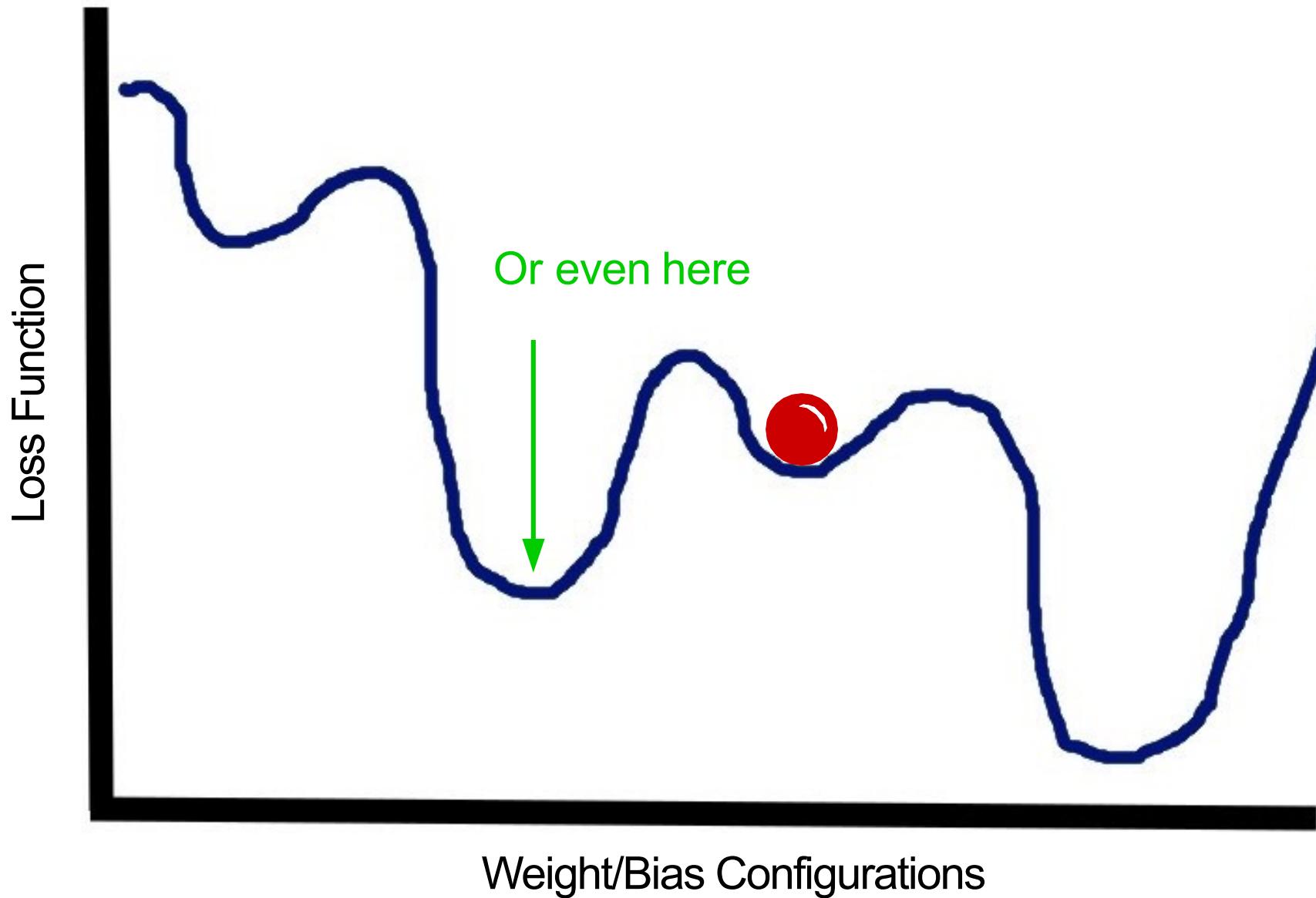


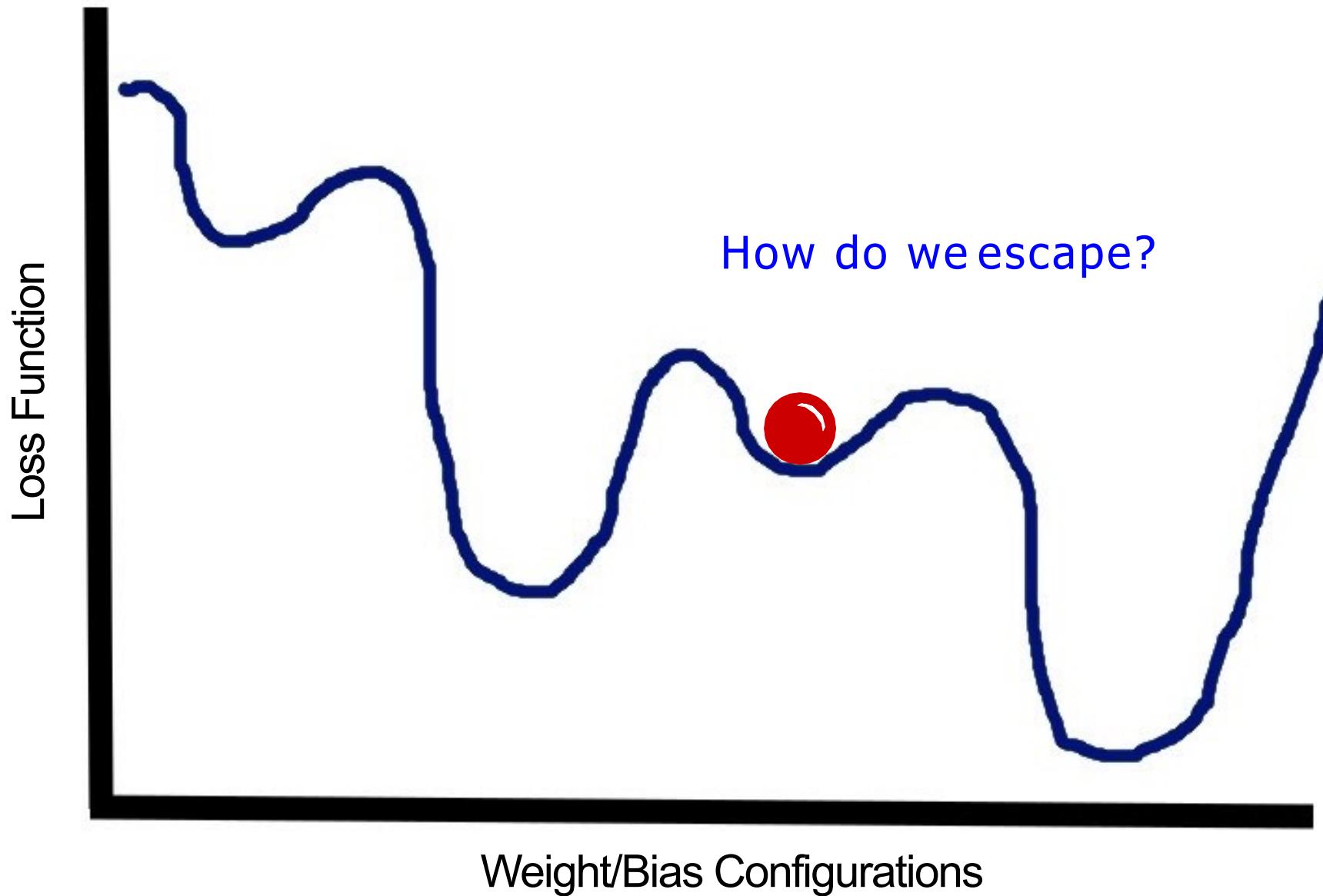


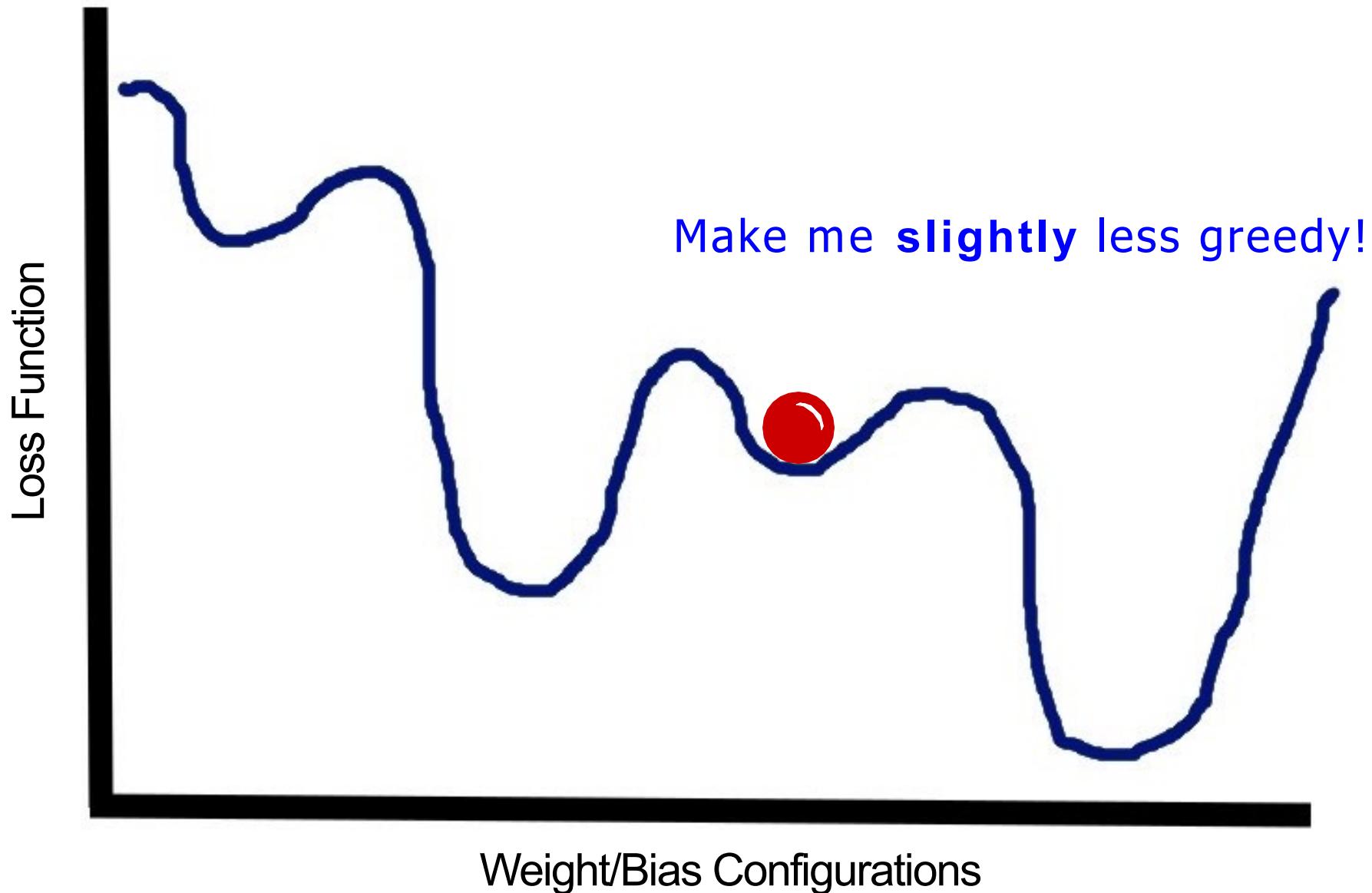


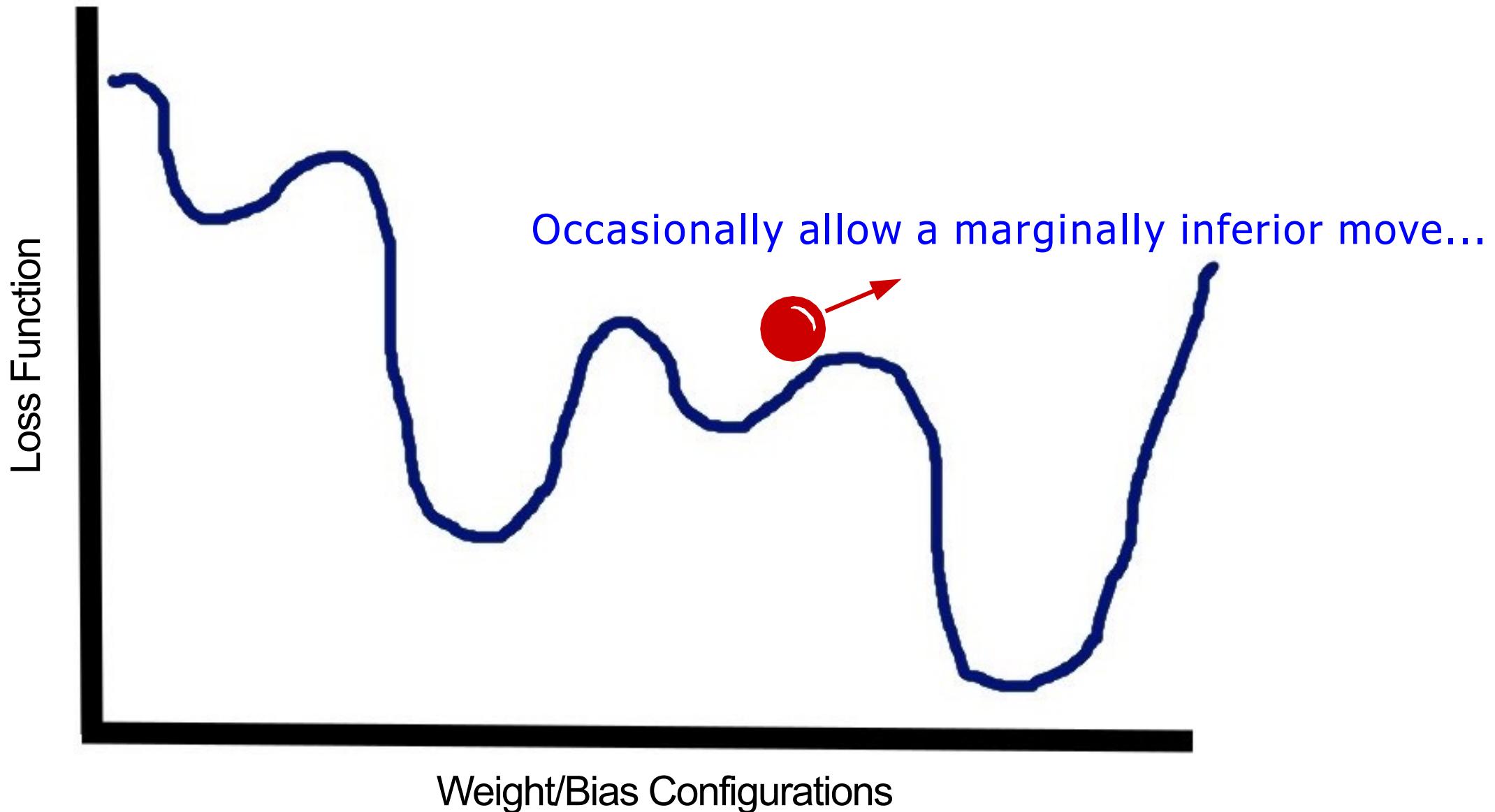


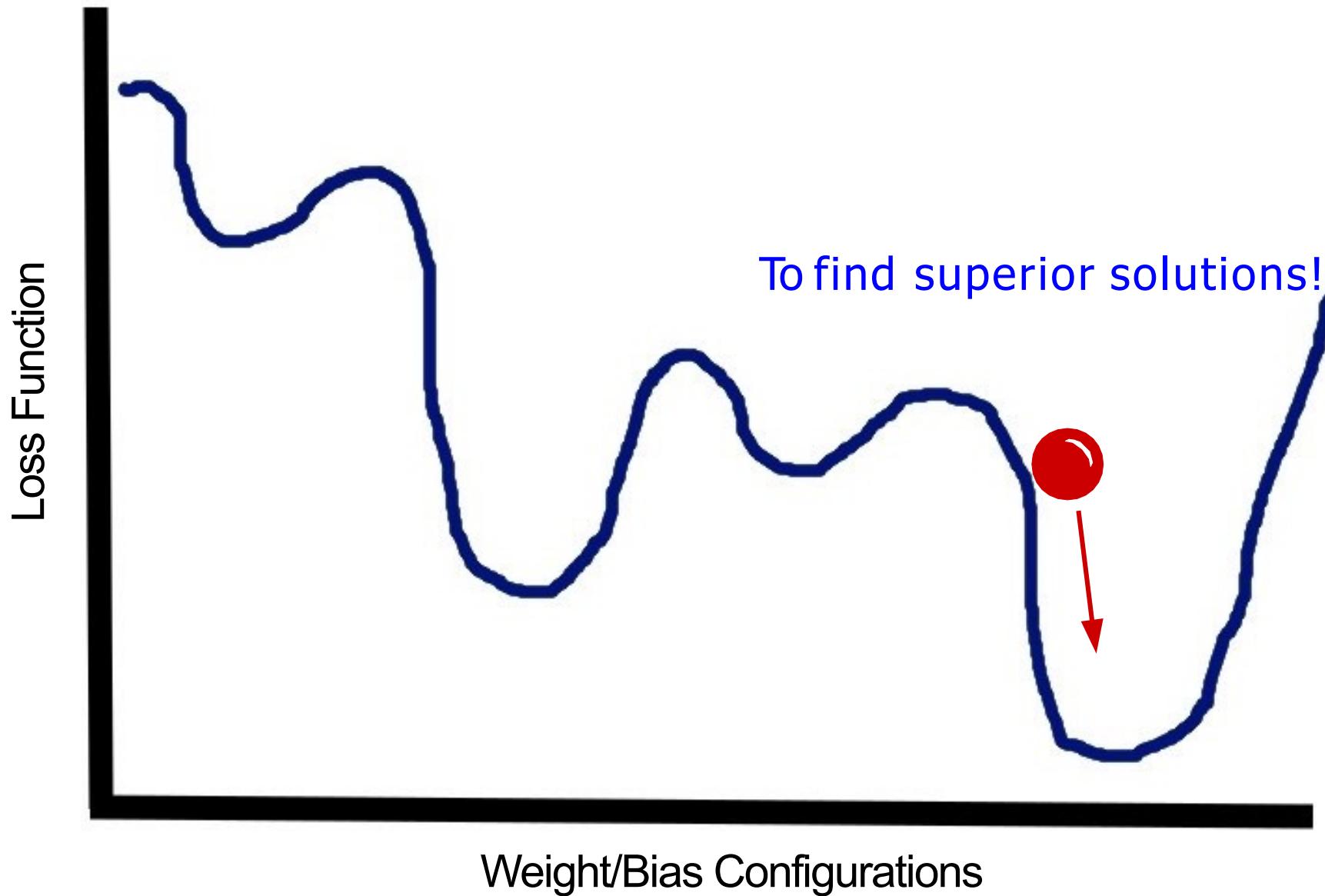


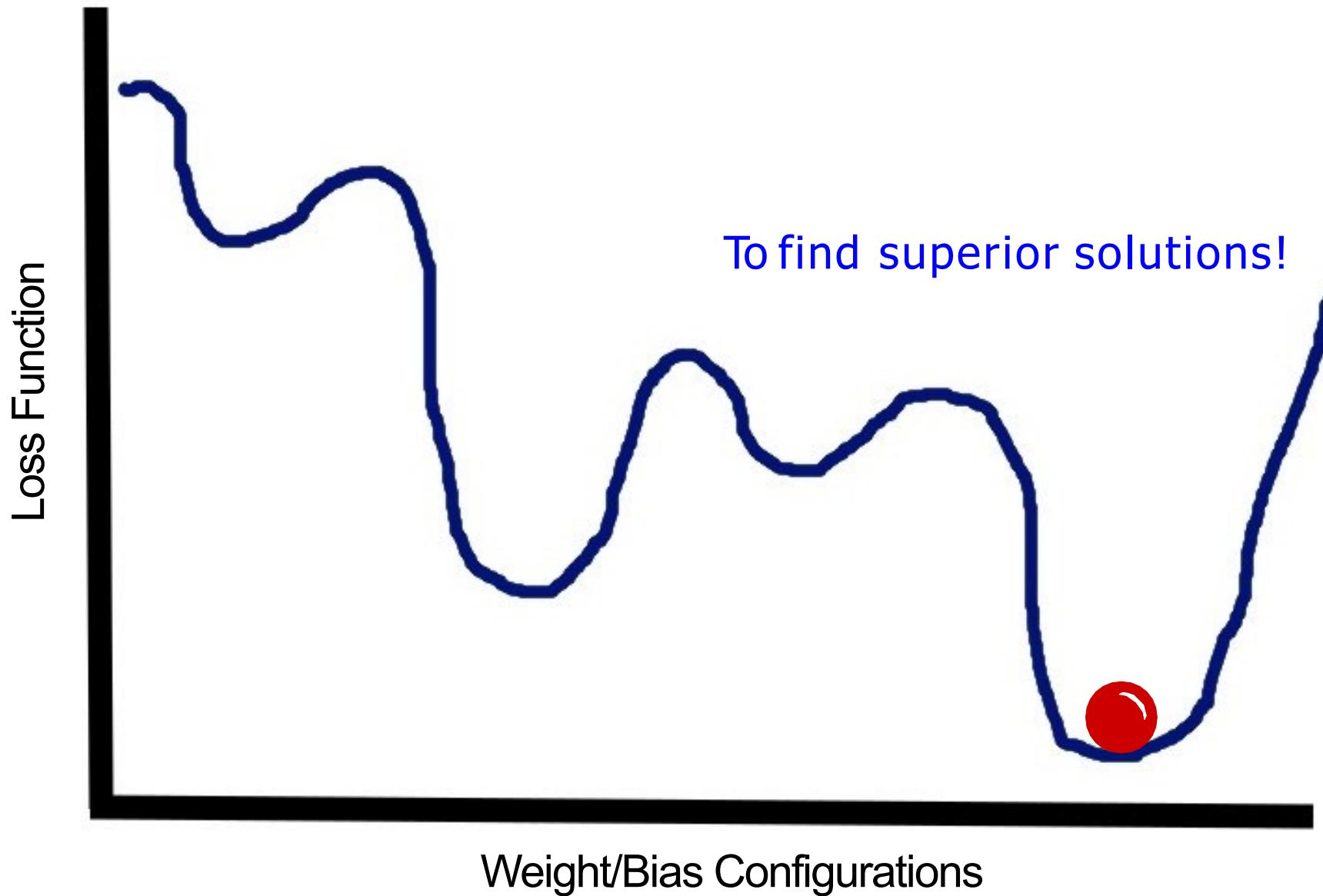


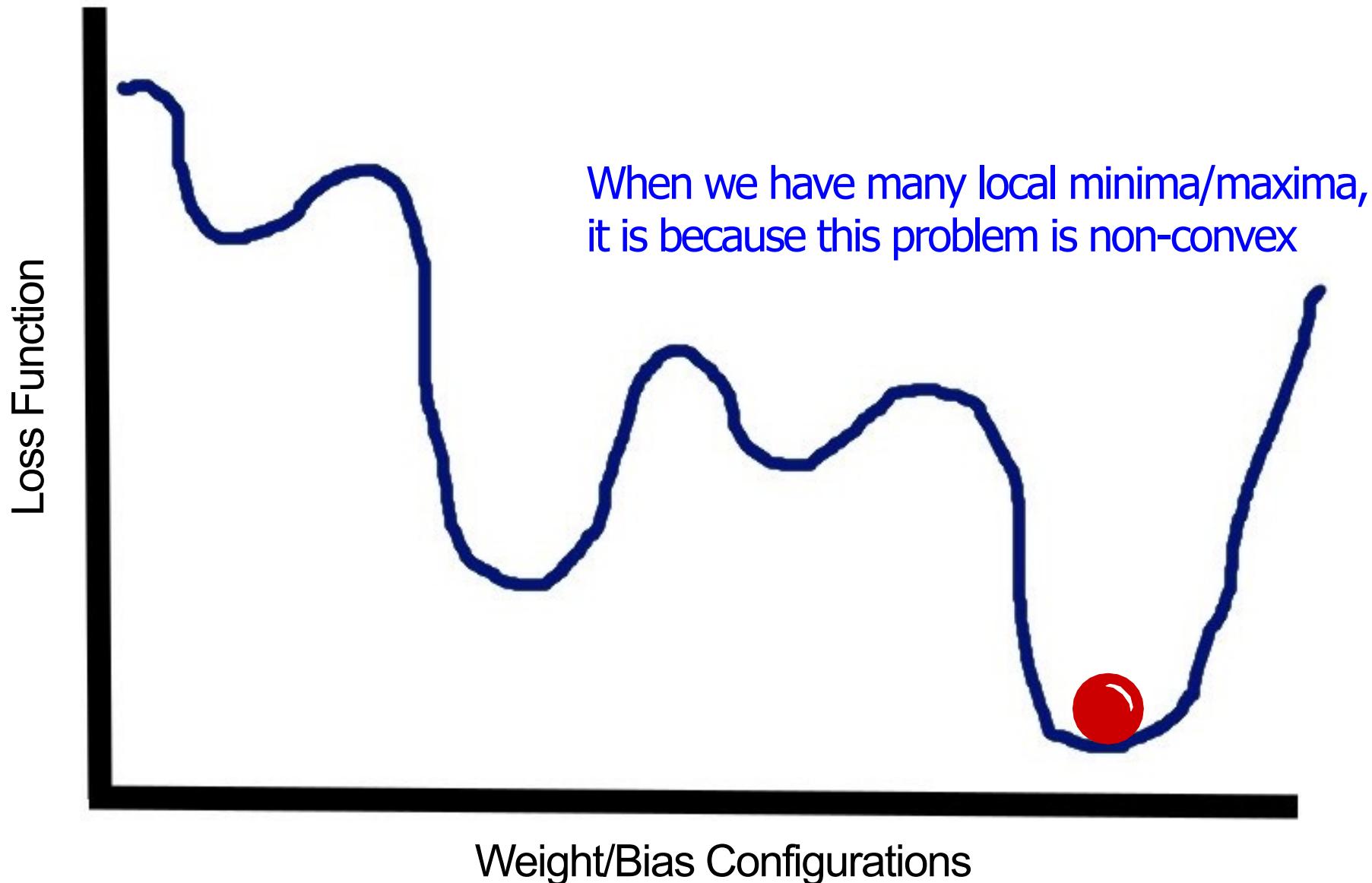






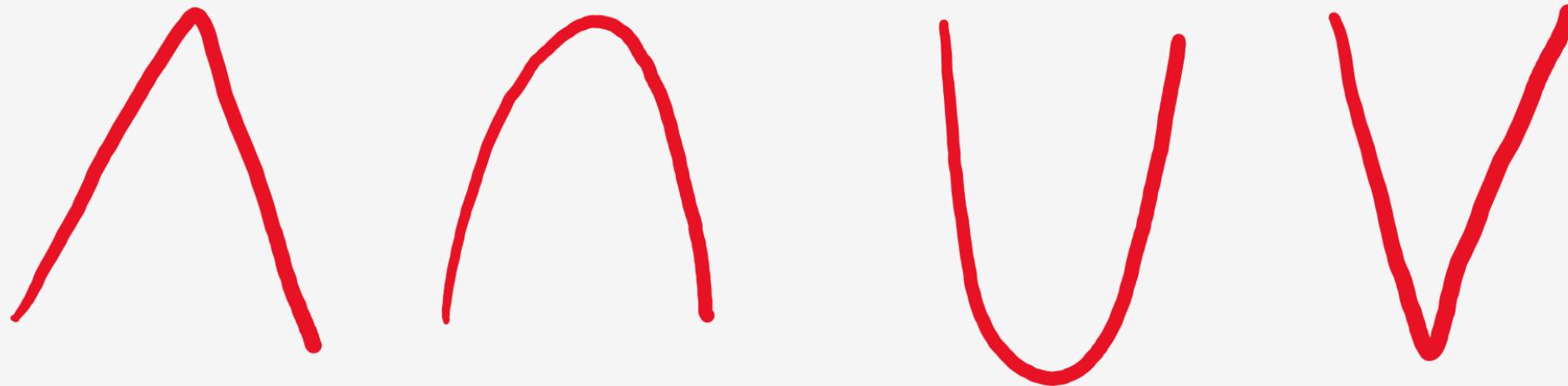






## Convex versus Non-Convex

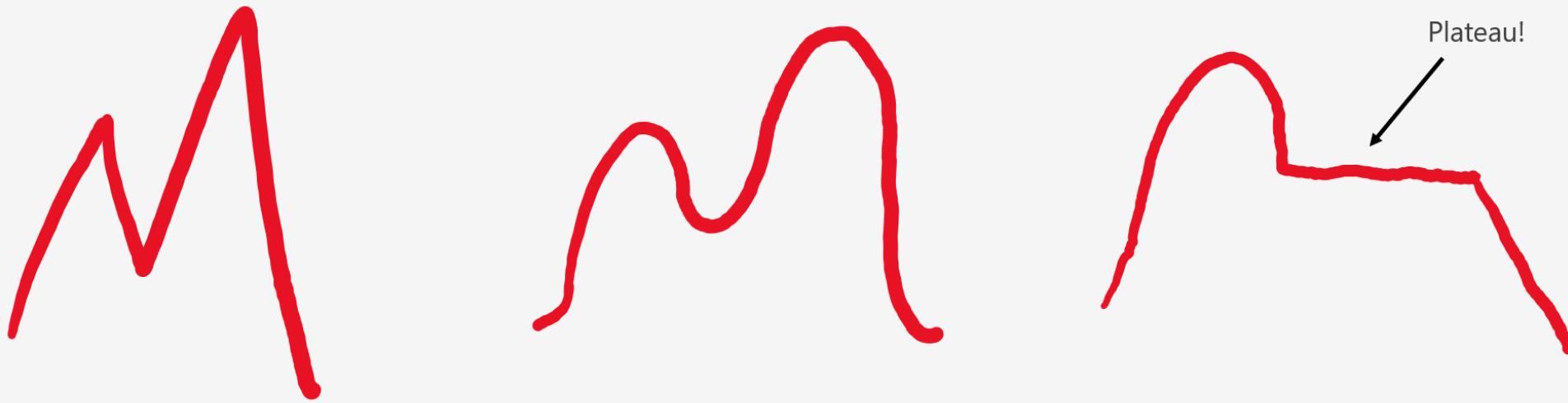
---



Convex

## Convex versus Non-Convex

---



Non-Convex

## So How Do We Optimize the Weights and Biases?

---

Hill climbing is going to perform poorly with neural networks because it will fall into the first local minimum it finds, but so will **gradient descent** which uses Calculus-based slopes to find the direction of the minimum.

We would typically use **stochastic gradient descent** to increase computational speed and avoid getting stuck in local minimums, and we would use **backpropagation** to calculate derivatives for the weights and biases.

However this is beyond the scope of this class, and I cover how to build a neural network (including backpropagation) in Chapter 7 of my book *Essential Math for Data Science*.

O'REILLY®

### Essential Math for Data Science

Take Control of Your Data with Fundamental Linear Algebra, Probability, and Statistics



Thomas Nield

# A Neural Network from Scratch

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

all_data = pd.read_csv("https://tinyurl.com/y2qmhfsr")

# Learning rate controls how slowly we approach a solution
# Make it too small, it will take too long to run.
# Make it too big, it will likely overshoot and miss the solution
L = 0.05

# Extract the input columns, scale down by 255
all_inputs = (all_data.iloc[:, 0:3].values / 255.0)
all_outputs = all_data.iloc[:, -1].values

# Split train and test data sets
X_train, X_test, Y_train, Y_test = train_test_split(all_inputs,
    test_size=1 / 3)
n = X_train.shape[0]

# Build neural network with weights and biases
# with random initialization
w_hidden = np.random.rand(3, 3)
w_output = np.random.rand(1, 3)

b_hidden = np.random.rand(3, 1)
b_output = np.random.rand(1, 1)
```

```
# Activation functions
relu = lambda x: np.maximum(x, 0)
logistic = lambda x: 1 / (1 + np.exp(-x))

# Runs inputs through the neural network to get predicted output
def forward_prop(X):
    Z1 = w_hidden @ X + b_hidden
    A1 = relu(Z1)
    Z2 = w_output @ A1 + b_output
    A2 = logistic(Z2)
    return Z1, A1, Z2, A2

# Derivatives of Activation functions
d_relu = lambda x: x > 0
d_logistic = lambda x: np.exp(-x) / (1 + np.exp(-x)) ** 2

# returns slopes for weights and biases
# using chain rule
def backward_prop(Z1, A1, Z2, A2, X, Y):
    dC_dA2 = 2 * A2 - 2 * Y
    dA2_dZ2 = d_logistic(Z2)
    dZ2_dA1 = w_output
    dZ2_dW2 = A1
    dZ2_dB2 = 1
    dA1_dZ1 = d_relu(Z1)
    dZ1_dW1 = X
    dZ1_dB1 = 1

    dC_dW2 = dC_dA2 @ dA2_dZ2 @ dZ2_dW2.T
```

# A Neural Network from Scratch

```
dc_dB2 = dc_dA2 @ dA2_dz2 * dz2_db2

dc_dA1 = dc_dA2 @ dA2_dz2 @ dz2_da1

dc_dW1 = dc_dA1 @ da1_dz1 @ dz1_dW1.T

dc_db1 = dc_da1 @ da1_dz1 * dz1_db1

return dc_dW1, dc_db1, dc_dW2, dc_db2

# Execute gradient descent
for i in range(100_000):
    # randomly select one of the training data
    idx = np.random.choice(n, 1, replace=False)
    X_sample = X_train[idx].transpose()
    Y_sample = Y_train[idx]

    # run randomly selected training data through neural network
    z1, A1, z2, A2 = forward_prop(X_sample)

    # distribute error through backpropagation
    # and return slopes for weights and biases
    dW1, dB1, dW2, dB2 = backward_prop(z1, A1, z2, A2, X_sample)

    # update weights and biases
    w_hidden -= L * dW1
    b_hidden -= L * dB1
    w_output -= L * dW2
    b_output -= L * dB2

# Calculate accuracy
test_predictions = forward_prop(X_test.transpose())[3] # grab
test_comparisons = np.equal((test_predictions >= .5).flatten())
accuracy = sum(test_comparisons.astype(int) / X_test.shape[0])
print("ACCURACY: ", accuracy)
```

# Other Design Decisions

---

**While optimizing weights is a core part of neural networks, there are other components to consider too:**

How many middle *layers* are needed?

How many *nodes* are needed in each layer?

What *activation functions* should be applied to each layer?

**Middle Layers** - Should the layers be recurrent, recursive, convolutional, etc?

**Loss function** – How should we measure error?

**Learning Rate** - How aggressively should the optimization move towards the local minimum?

# Problems with Deep Learning

---

Machine learning algorithms, especially deep learning, are not perfect and easily error-prone especially with outliers. **Correlation is not Causation!**

To the right are examples of a well-trained neural network unable to recognize images correctly due to objects in abnormal positions

SOURCE: <https://arxiv.org/abs/1811.11553>



# Section VII

# Going Forward

To Learn More...

---

O'REILLY®

# Essential Math for Data Science

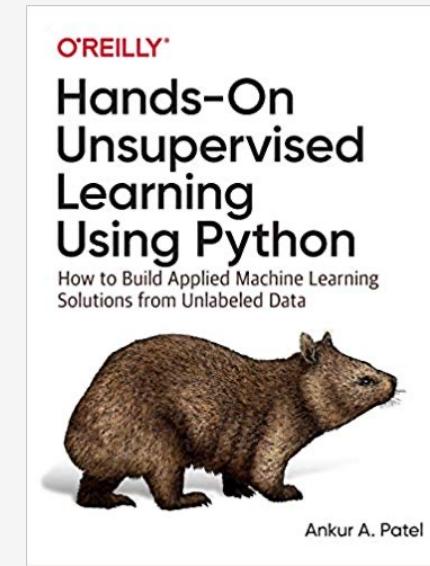
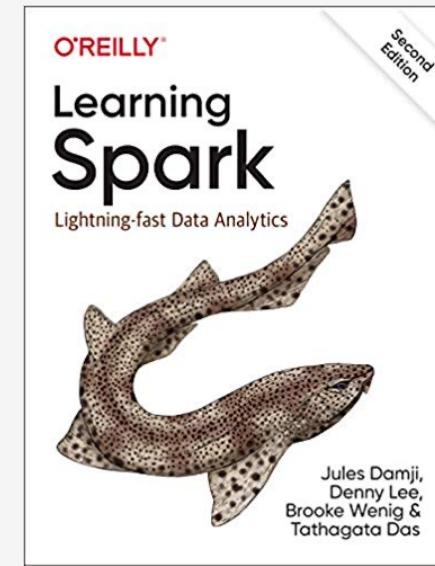
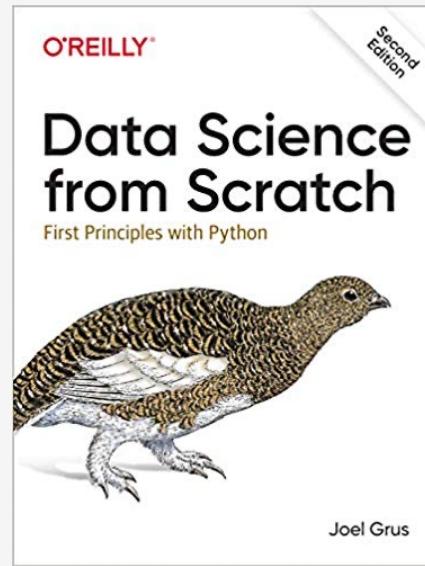
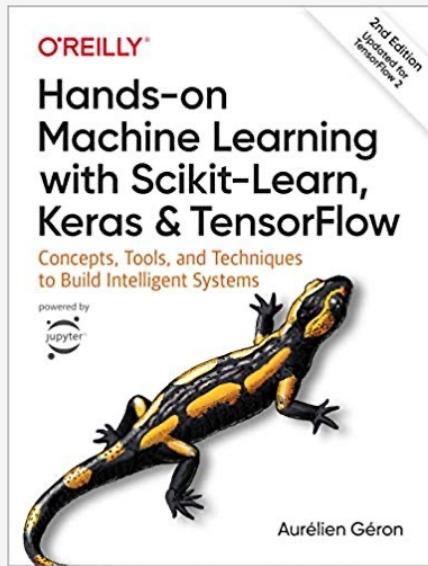
Take Control of Your Data with Fundamental  
Linear Algebra, Probability, and Statistics



Thomas Nield

# To Learn More...

---



Homework Assignments  
(Answers Are In “code” Folder)

# Homework #1

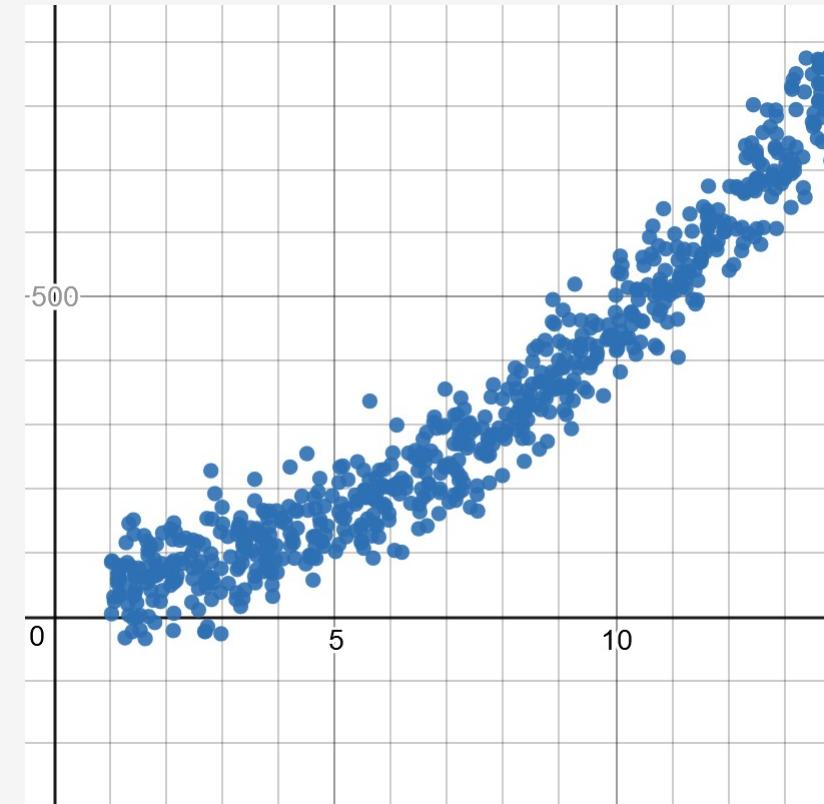
---

You notice some data (accessible at <https://bit.ly/2UBhrMG>) has a familiar shape on a scatterplot, and you believe that the best regression will fit to a function

$$y = ax^2 + b$$

where  $a$  and  $b$  are some constants. Find the  $a$  and  $b$  constants for the best fit regression based on sum of squares. I recommend using this [Desmos Graph](#) to graph your points and function to eyeball whether a good fit occurred.

**HINT:** Think of how you solved a linear regression using hill-climbing, and how it could apply here. Solution is in the folder `code/homework_answers`



<https://www.desmos.com/calculator/canygmx67n>

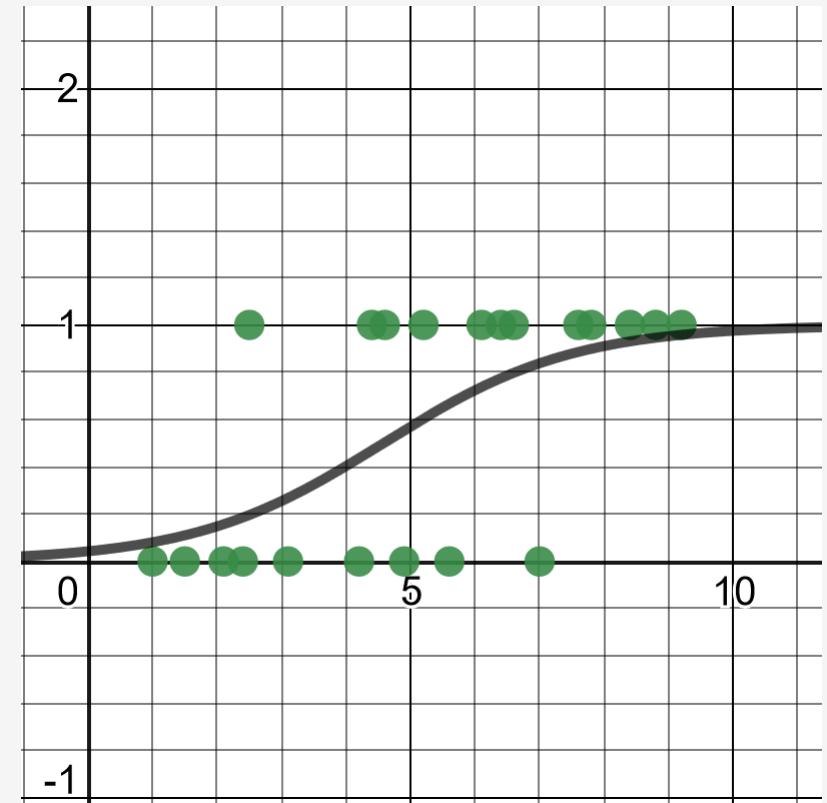
## Homework #2

---

Take the RGB background color data (accessible at <https://tinyurl.com/y2qmhsr>) that recommends a light (1) or dark (0) font. Create a logistic regression model that trains with that data and predicts a light/dark font for new background colors. Test to make sure a black background (0,0,0) will predict a LIGHT font and a white background (255,255,255) will predict a DARK font.

**HINT:** Remember the first three columns are the numeric inputs (red, green, blue) and the fourth column is the output ("1" for light, "0" for dark).

Solution is in the folder *code/homework\_answers*



# Homework #3

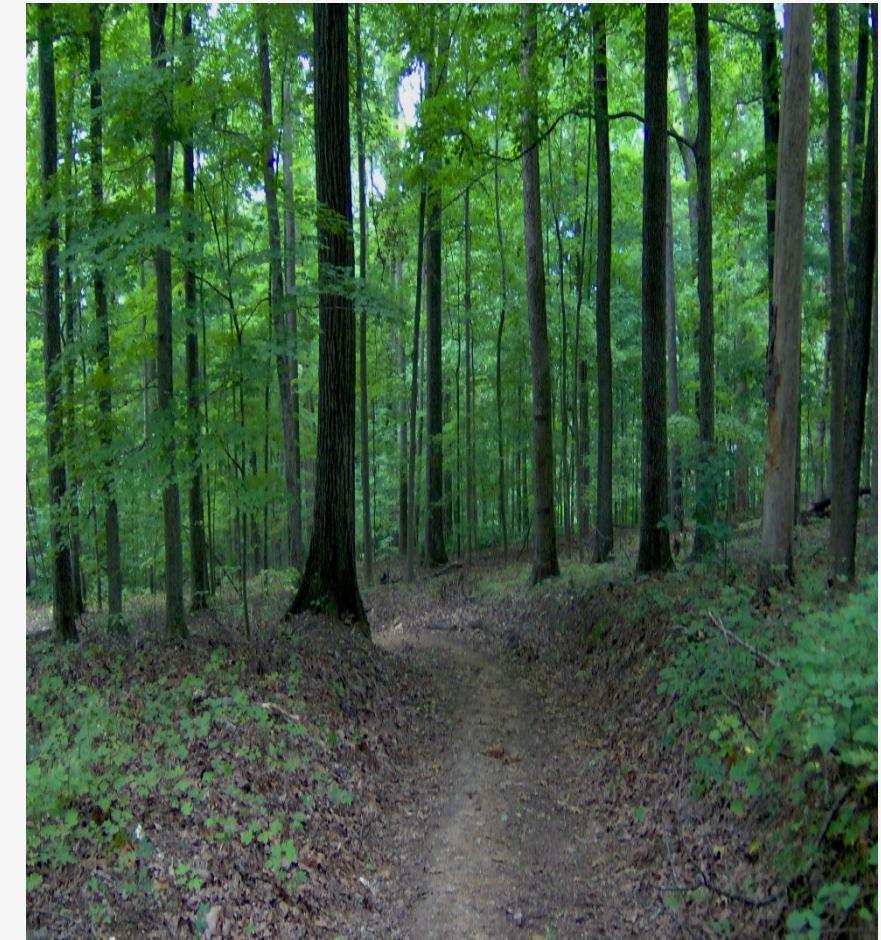
---

Take the employee retention data (accessible at <https://tinyurl.com/y6r7qjrp>) and make a decision tree to predict whether the employee will quit (1) or not quit (0).

As a quick and dirty test, check to see if employees who are not promoted for over two years generally quit by putting some in as inputs.

Solution is in the folder *code/homework\_answers*

**BONUS:** Can you convert the model into a random forest with 600 decision trees?



# Appendix A: Classification Validation

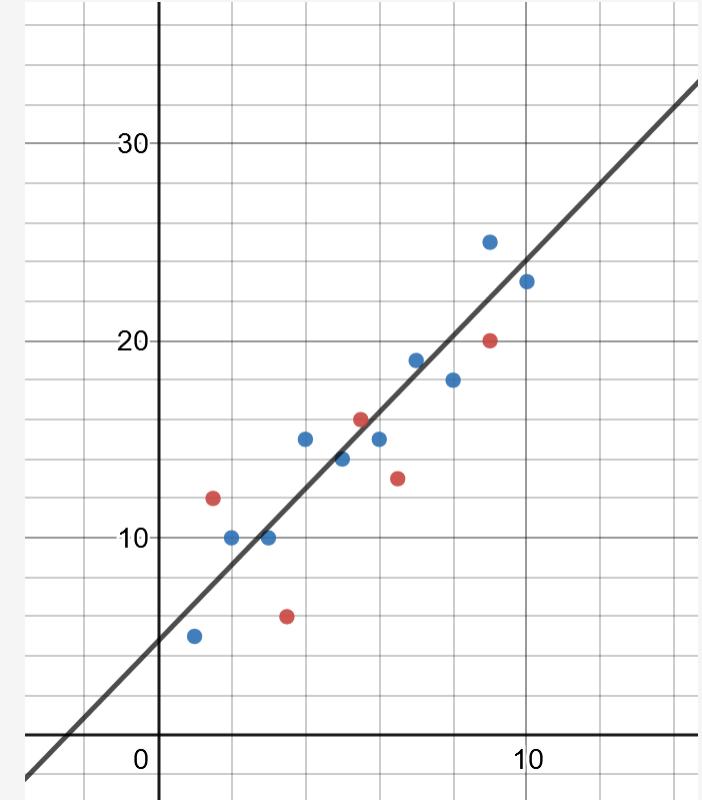
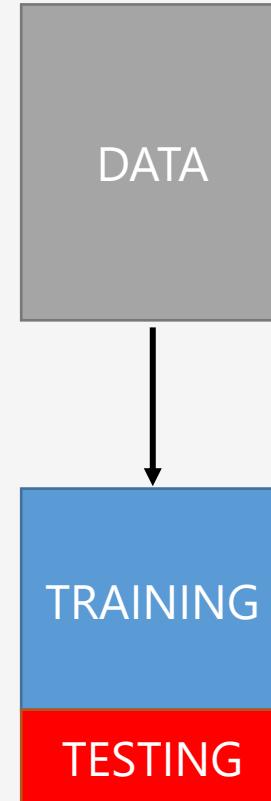
# Training and Testing Data

---

A common practice to proactively prevent overfitting in machine learning is to separate training data and testing data.

- **Training data** is data used to fit a model and is typically 2/3 of the data.
- **Test data** is used to test the model and is the remaining 1/3 of the data.

By omitting the testing data from training, we see how well the model works on data it has not seen before and change our parameters accordingly.



# Cross-Validation

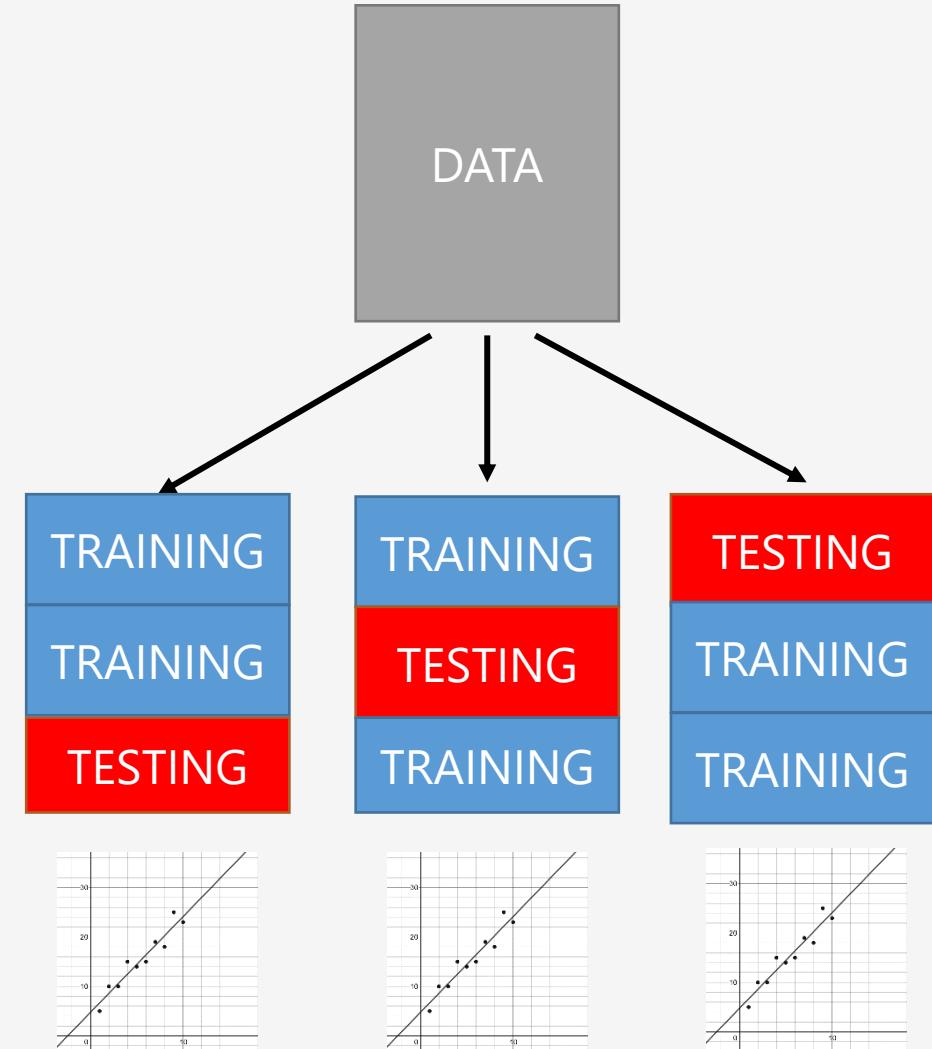
---

We can take this concept of training/testing data a step further, and test different combinations of training and testing data.

This is known as **cross-validation**, the gold standard of validation techniques.

To the right we have **3-fold cross validation** which breaks the data into thirds and uses one of the pieces for testing.

We can then evaluate how well each of these perform, being able to compare different parameters and models (e.g. linear regression vs decision trees) and see which setup produces the best performance.



# Cross-Validation

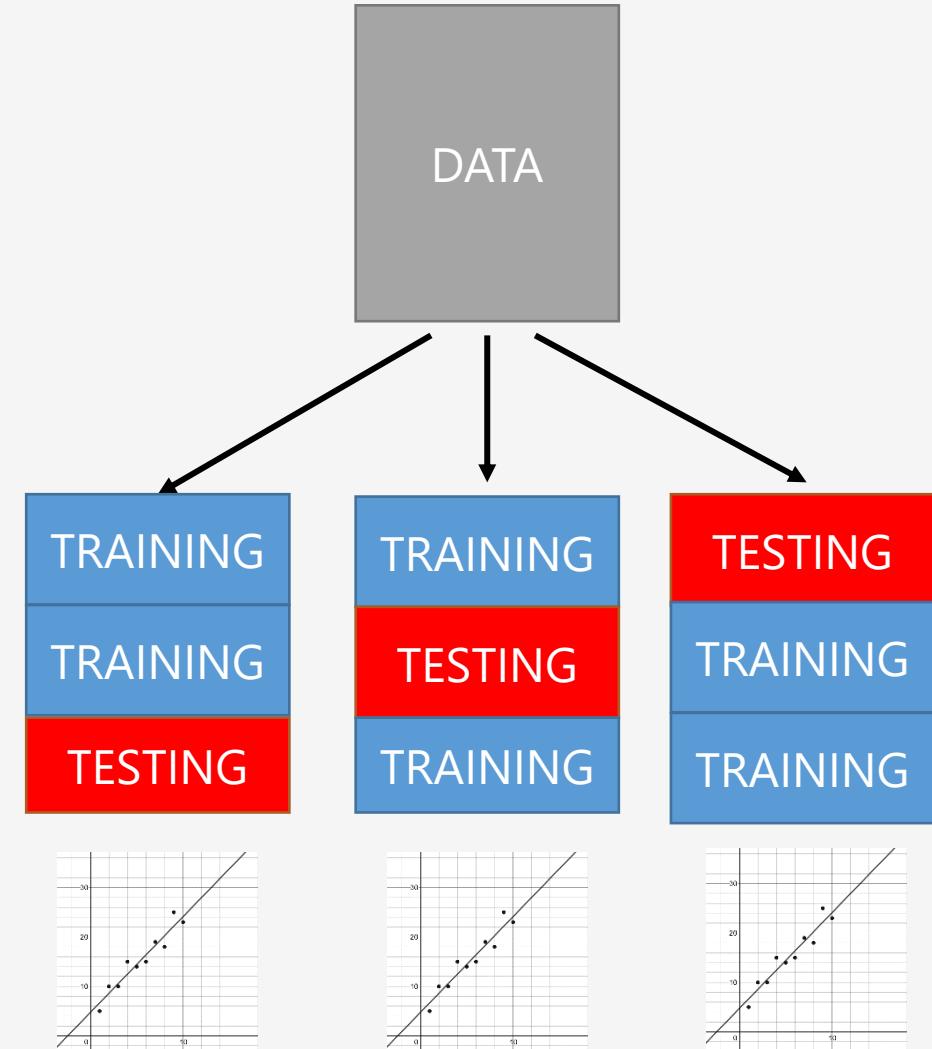
---

We can take this concept of training/testing data a step further, and test different combinations of training and testing data.

This is known as **cross-validation**, the gold standard of validation techniques.

To the right we have **3-fold cross validation** which breaks the data into thirds and uses one of the pieces for testing.

We can then evaluate how well each of these perform, being able to compare different parameters and models (e.g. linear regression vs decision trees) and see which setup produces the best performance.



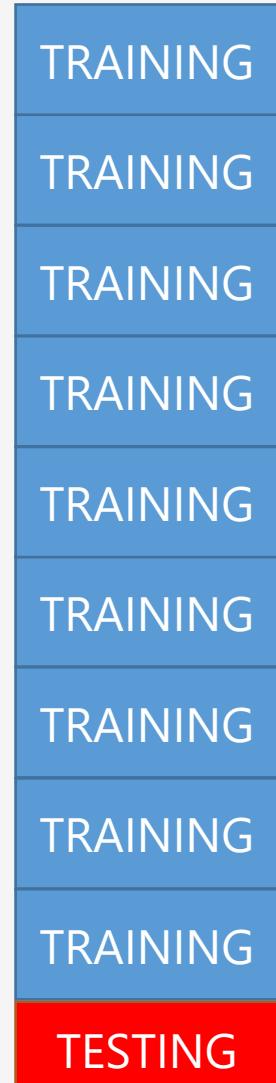
# Cross-Validation

---

Note that **k-fold cross validation** allows us to slice our data into any number and not just 3 (typically 3, 5, or 10).

For example we can do 10-fold cross validation and validate 10 different combinations of training/test data.

The most extreme form of folding is **leave-one-out cross validation**, which omits one data record for testing and uses the remaining records for training, and this is done repeatedly.



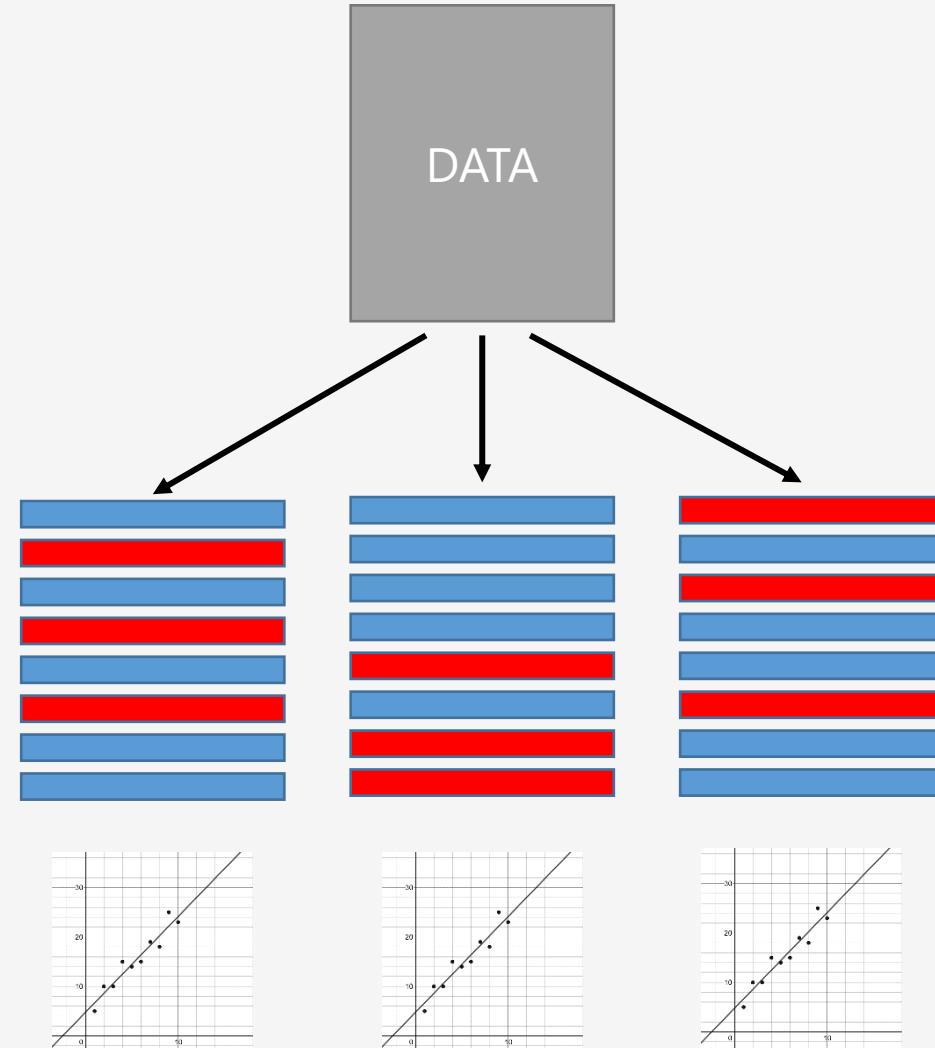
# Random Fold Validation

---

As you may be noticing, machine learning often tries to overcome data variance with randomness.

A variant with fold validation is **repeated random fold validation**, where we randomly shuffle the data and create random train/test folds as many times as desired.

This is helpful when we need to mitigate variance in the model.



# Which Validation to Use?

---

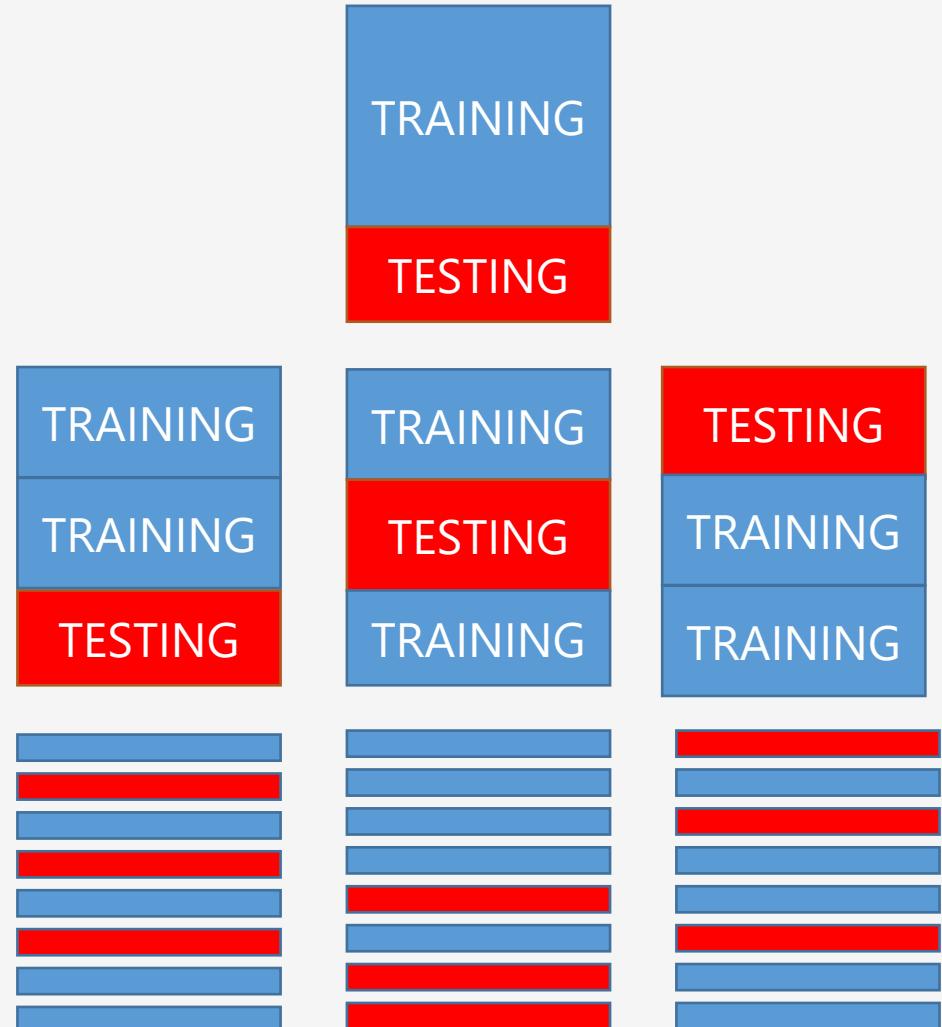
**Generally, you will want to prefer k-fold validation as it is the gold standard.**

A single train/test split might be warranted if performance of machine learning algorithm is slow and has enough data with lower bias.

Use the random fold split to mitigate variance in the model while balancing training speed and dataset sizes.

**We will talk about class imbalance later, but if you do not have an equal number of samples for each class, you might want to consider using stratification in your k-fold validation.**

**Stratification** means an equal proportion of data for each class is sampled for training and testing data (even it is sampled repeatedly), so that no class is neglected.



# A Stratified Validation in Scikit-Learn

---

```
import numpy as np
import pandas as pd
# Load data
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

df = pd.read_csv("https://tinyurl.com/y6r7qjrp", delimiter=",")

# Extract input variables (all rows, all columns but last column)
# Note we should do some linear scaling here
X = (df.values[:, :-1] / 255.0) + .01

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Get a count of each group to ensure samples are equitably balanced
print(df.groupby(["class"]).agg({"class" : [np.size]}))

# Separate training and testing data
# Note that I use the 'stratify' parameter to ensure
# each class is proportionally represented in both sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.33, stratify=Y, stratify=Y)

nn = MLPClassifier(solver='sgd', hidden_layer_sizes=(100, ),
                    max_iter=480, learning_rate_init=.1)

nn.fit(X_train, Y_train)

print("Training set score: %f" % nn.score(X_train, Y_train))
print("Test set score: %f" % nn.score(X_test, Y_test))
```

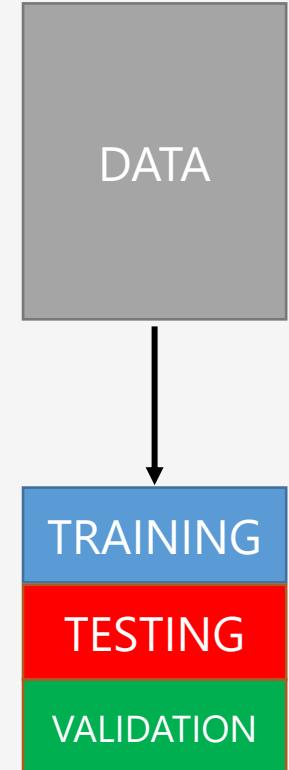
# Validation Data

---

**Validation** is a separate type of testing data used to compare performance of different models.

When you are comparing two or more models (e.g. logistic regression versus decision trees), you may hold back one more chunk of data for validation.

It is different than the testing data, which is used to tune parameters of the individual model, not compare different models altogether.

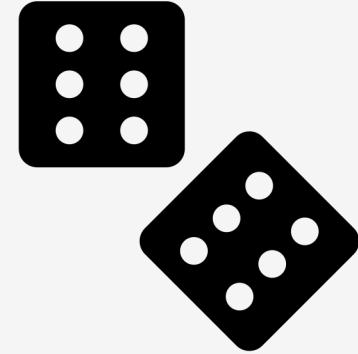


# Random Seeding

---

**Machine learning deals with a lot of random-based operations:**

- Randomly sampled training/testing data
- Random-based optimization and training (e.g. stochastic gradient descent, simulated annealing)
- Randomly generated data from simulations



**Testing can be challenging as the above operations make outputs nondeterministic, so setting a **random seed** is common practice to make them deterministic.**

**This allows tests to be conveniently reproducible, where sequences of randomly generated values are the same.**

**Be careful to not rely on random seeds too much! They can give the illusion of determinism so test many random seeds, not just one, to ensure they converge towards the same outcome.**

## Random Seeding a 3-Fold Split for Reproducibility

---

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold, cross_val_score

# Load the data
df = pd.read_csv("https://tinyurl.com/y6r7qjrp", delimiter=",")

X = df.values[:, :-1]
Y = df.values[:, -1]

# "random_state" is the random seed, which we fix to 7
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LogisticRegression(solver='liblinear')
results = cross_val_score(model, X, Y, cv=kfold)

print("Accuracy Mean: %.3f (stdev=%.3f)" % (results.mean(), results.std()))
```

# Why “Accuracy” is a Bad Measure for Classification

---

Suppose your machine learning model observed people with the name “Michael” quit their job.

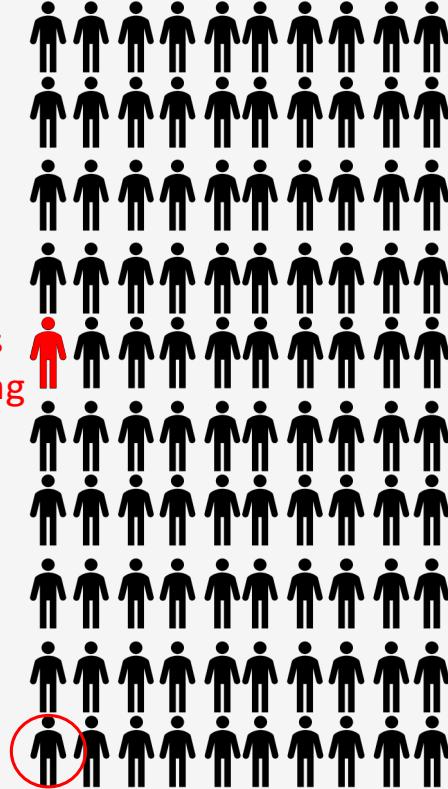
Therefore it simply predicts any employee with the name “Michael” will quit, and everyone else will stay.

If there was one Michael in a company of 100 employees, but a different employee ended up quitting, the machine could still boast 98% accuracy.

**Why stop there? If 99% of employees do not quit, you might as well predict every employee *will not quit* and get 99% accuracy.**

**Machine learning makes shortcuts like this all the time especially when the event of interest is rare (e.g. diseases, security breaches, employee attrition, rare road/highway events).**

This employee quits  
Prediction was wrong  
But I'm still 98%  
accurate!



Prediction  
This employee is named “Michael”  
This employee will quit

# Confusion Matrix

---

As seen in the previous example, **accuracy** (the rate of correct labeling) can be a misleading measure of performance, especially in classification problems with class imbalance.

A better way to evaluate classification models is the **confusion matrix**, which keeps track of **false positives** (Type I Error) and **false negatives** (Type II Error).

We want to evaluate how many employees who were predicted to quit actually did quit (**true positives**).

Conversely, we also want to evaluate how many employees who were predicted to stay actually did stay (**true negatives**).

	<b>Actually Quits (True)</b>	<b>Actually Stays (False)</b>
<b>Predicted will quit (True)</b>	0	1
<b>Predicted will stay (False)</b>	1	98

# Confusion Matrix

---

From the confusion matrix, we can derive all sorts of useful metrics beyond just accuracy.

We can easily see that **precision** (how accurate positive predictions were) and **sensitivity** (rate of identified positives) are 0, meaning this machine learning model fails entirely at positive predictions.

	Actually Quits	Actually Stays		
Predicted will quit	0 (TP)	1 (FN)	<b>Sensitivity</b> $\frac{TP}{TP+FN} = \frac{0}{0+1} = 0$	<b>F1 Score</b> $\frac{2 * Precision * Recall}{Precision + Recall} = \text{Undefined}$
Predicted will stay	1 (FP)	98 (TN)	<b>Specificity</b> $\frac{TN}{TN+FP} = \frac{98}{98+1} = .989$	
	<b>Precision</b> $\frac{TP}{TP+FP} = \frac{0}{0+1} = 0$	<b>Negative Predicted Value</b> $\frac{TN}{TN+FN} = \frac{98}{98+1} = .989$	<b>Accuracy</b> $\frac{TP+TN}{TP+TN+FP+FN} = \frac{98+0}{0+98+1+1} = .98$	

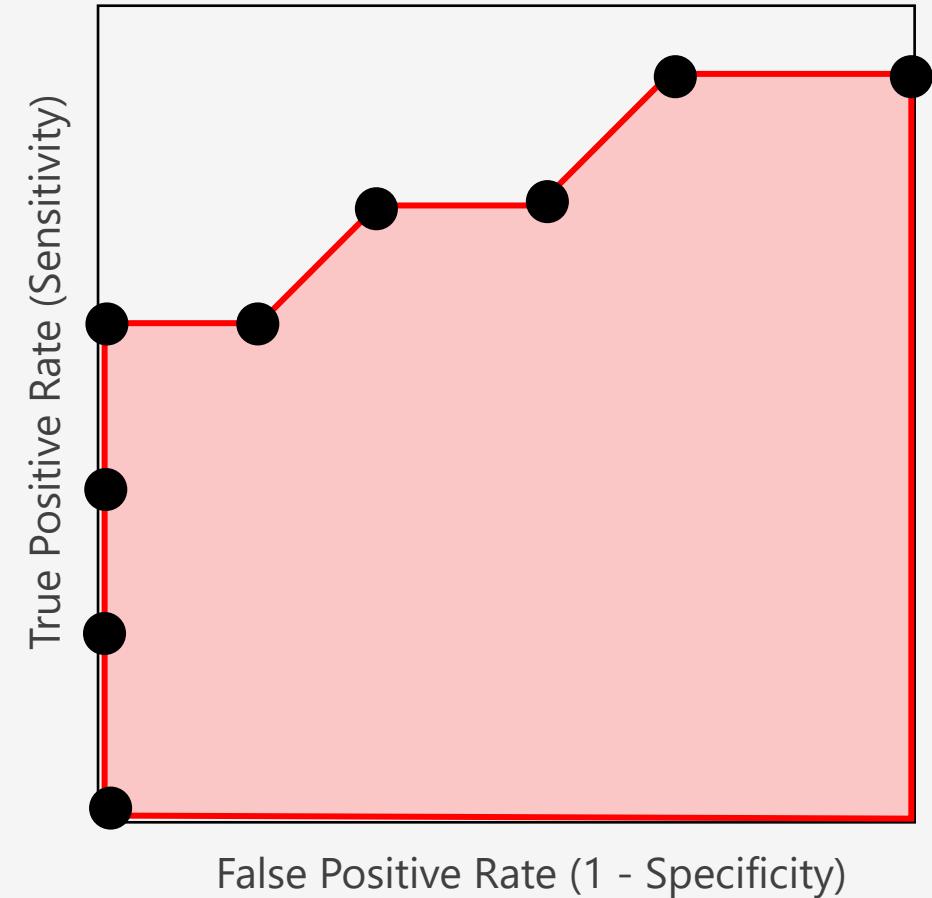
# ROC and AUC

---

When we are evaluating different machine learning configurations, we may end up with dozens, hundreds, or thousands of confusion matrices.

These can be tedious to review, so we can summarize all of them with a **receiver operator characteristic (ROC)** curve as shown to the right.

This allows us to see each testing instance (each represented by a black dot) and find an agreeable balance between true positives and false positives.



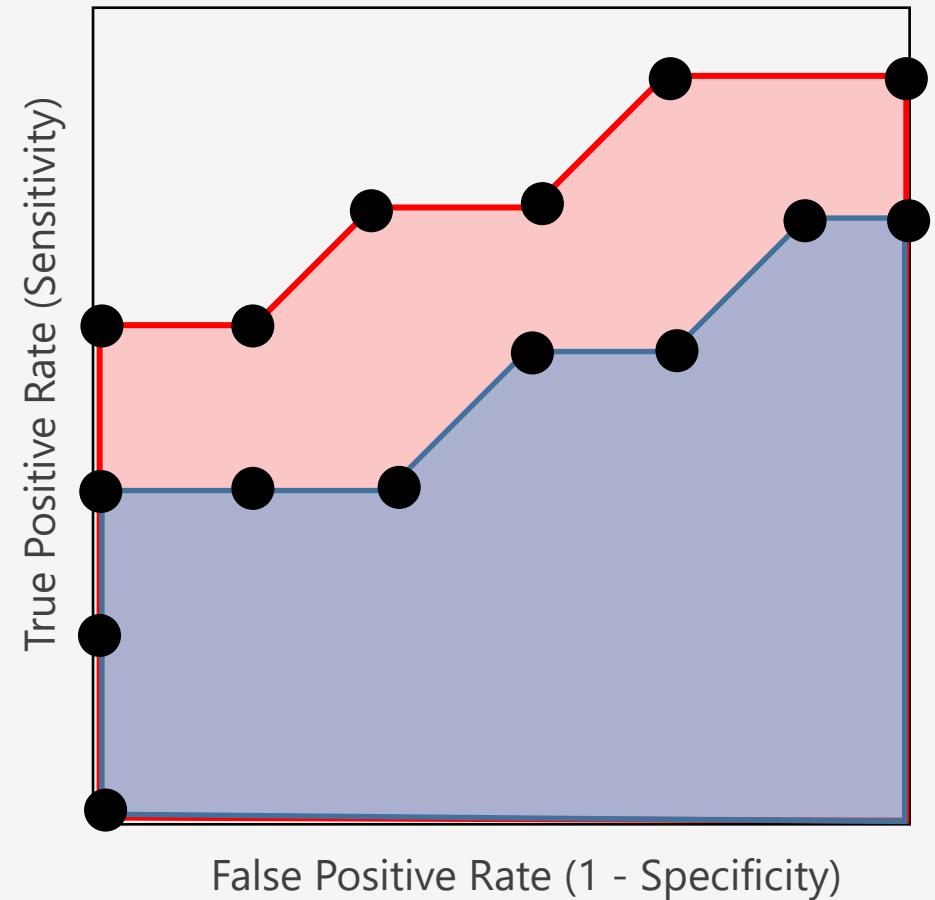
# ROC and AUC

---

We can also compare different machine learning models by creating separate ROC curves for each.

For example, if our red curve represents a logistic regression and the blue curve represents a decision tree, we can see the performance of them side-by-side.

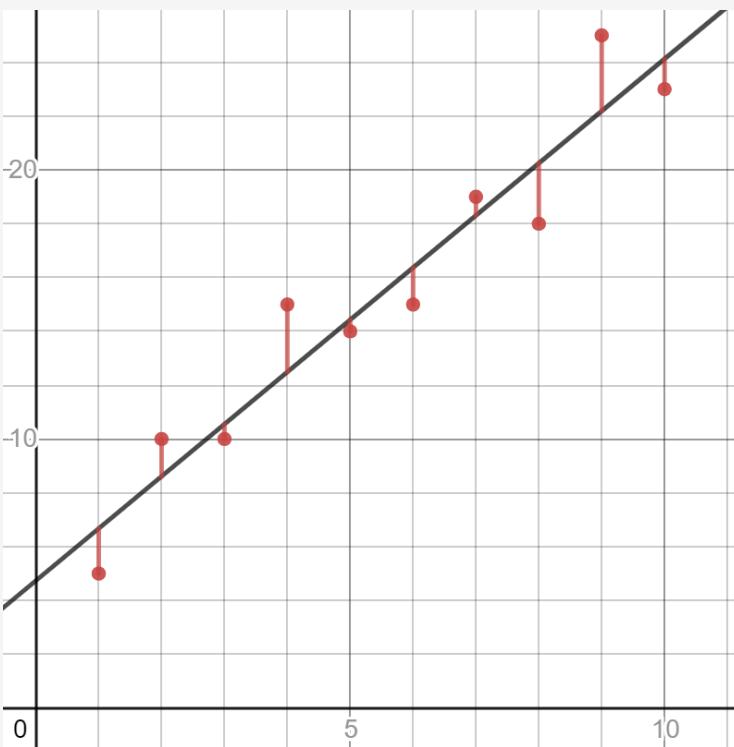
The **area under the curve (AUC)** is a good metric for choosing which model to use. Since the red curve (logistic regression) has a greater area, this suggests it is a superior model.



# Appendix B: Regression Validation

# Mean Absolute Error

**Mean absolute error (MAE)** simply measures the sum of absolute values of residuals (difference between predicted values and actual values).



```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score, validation_curve

df = pd.read_csv('https://bit.ly/3clH97A', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column) \
Y = df.values[:, -1]

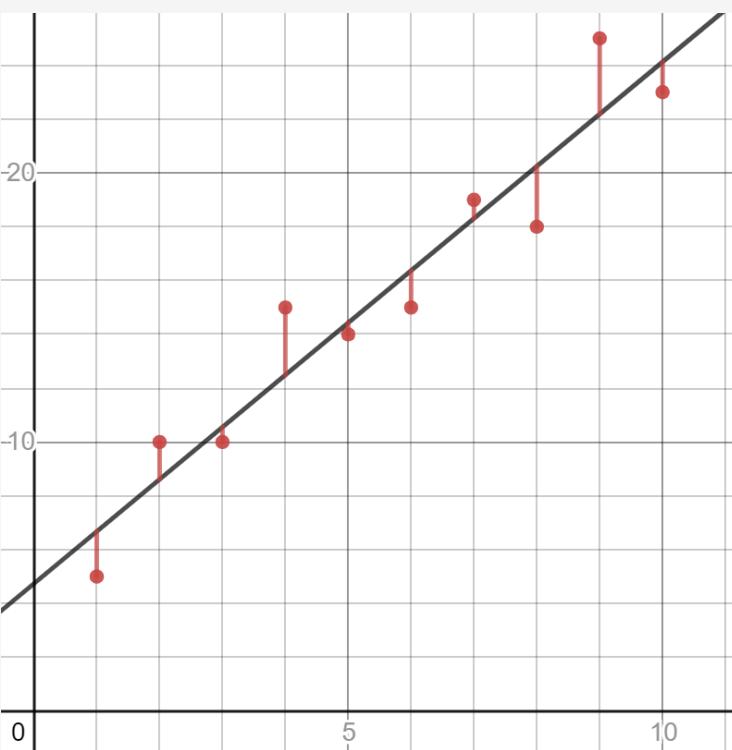
# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()

# Use negative mean absolute error
# It is negative because it is inverted
results = cross_val_score(model, X, Y, cv=kfold, scoring='neg_mean_absolute_error')

print("MAE: mean=% .3f (stdev-% .3f)" % (results.mean(), results.std()))
# prints MAE: mean=-2.446 (stdev-0.791)
```

# Mean Squared Error

**Mean squared error (MSE)** simply measures the average squared values of residuals (difference between predicted values and actual values).



```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score, validation_curve

df = pd.read_csv('https://bit.ly/3nJBdj9', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()

# Use negative mean squared error
# It is negative because it is inverted
results = cross_val_score(model, X, Y, cv=kfold, scoring='neg_mean_squared_error')

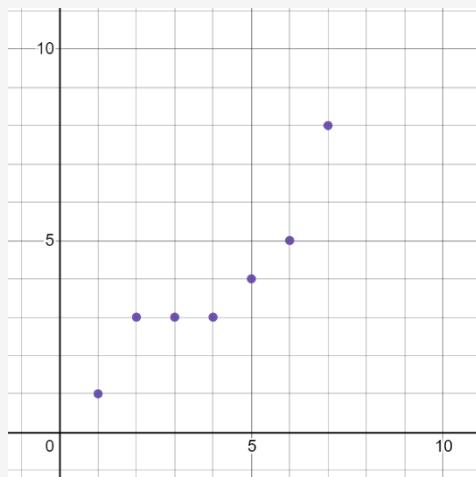
print("MAE: mean=% .3f (stdev-% .3f)" % (results.mean(), results.std()))
# prints MSE: mean=-7.664 (stdev-3.110)
```

# Pearson Correlation

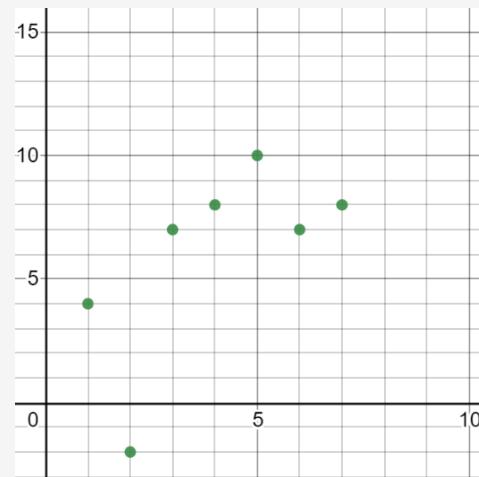
---

Using a **Pearson correlation**, we can measure the strength of a correlation between two variables (from a range of -1.0 to 1.0).

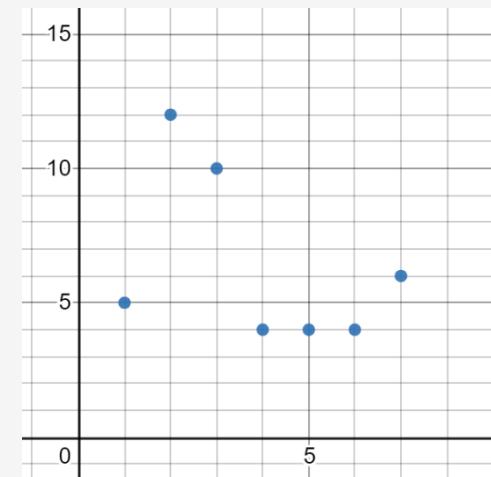
A correlation closer to 1.0 means a strong positive correlation, 0.0 means no correlation, and closer to -1.0 indicates negative correlation.



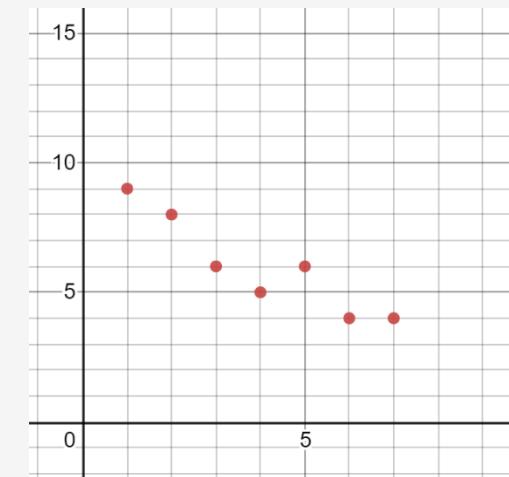
PEARSON CORRELATION: .923133



PEARSON CORRELATION: .643237



PEARSON CORRELATION: -.44984



PEARSON CORRELATION: -.9267

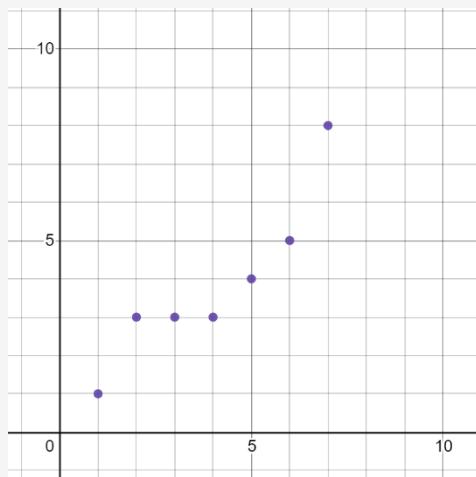
# Pearson Correlation

---

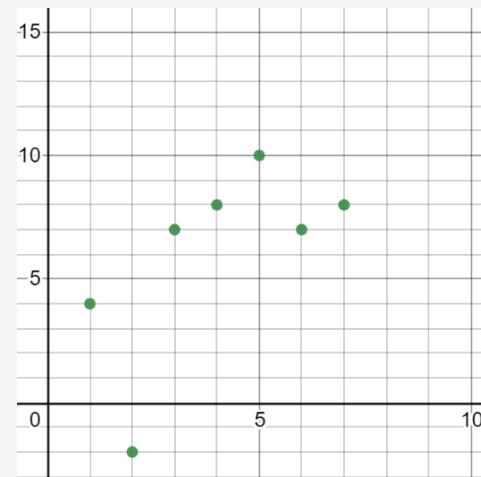
The more data you collect, the more confident you can be in your correlation value.

Having more data decreases the **P-value**, the probability you would observe this correlation randomly by chance.

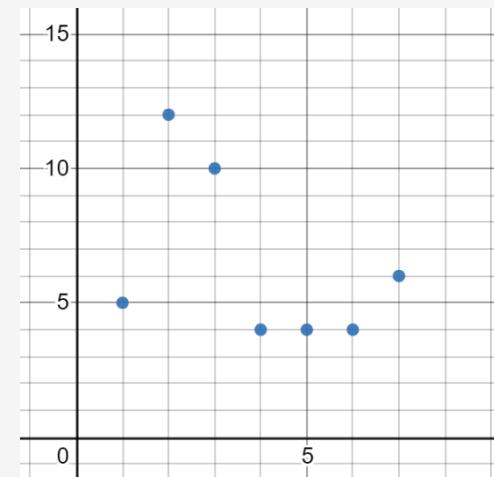
If our correlation is unlikely to have happened at random, that means we are more likely to have discovered something meaningful rather than by chance.



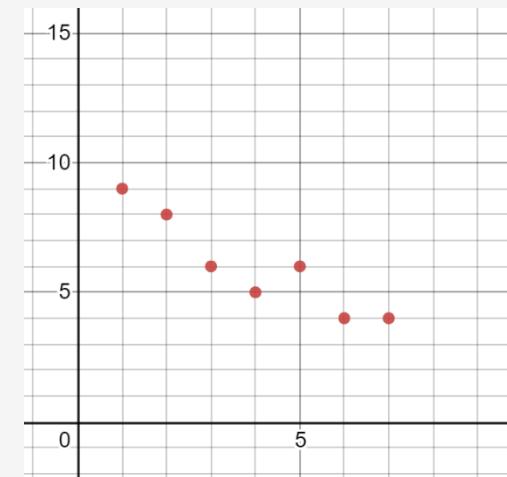
PEARSON CORRELATION: .923133



PEARSON CORRELATION: .643237



PEARSON CORRELATION: -.44984



PEARSON CORRELATION: -.9267

# R<sup>2</sup>

---

Probably the most useful performance metric is the **R-square (R<sup>2</sup>)**, which ratios the residuals against the average y-value to the average of the residuals.

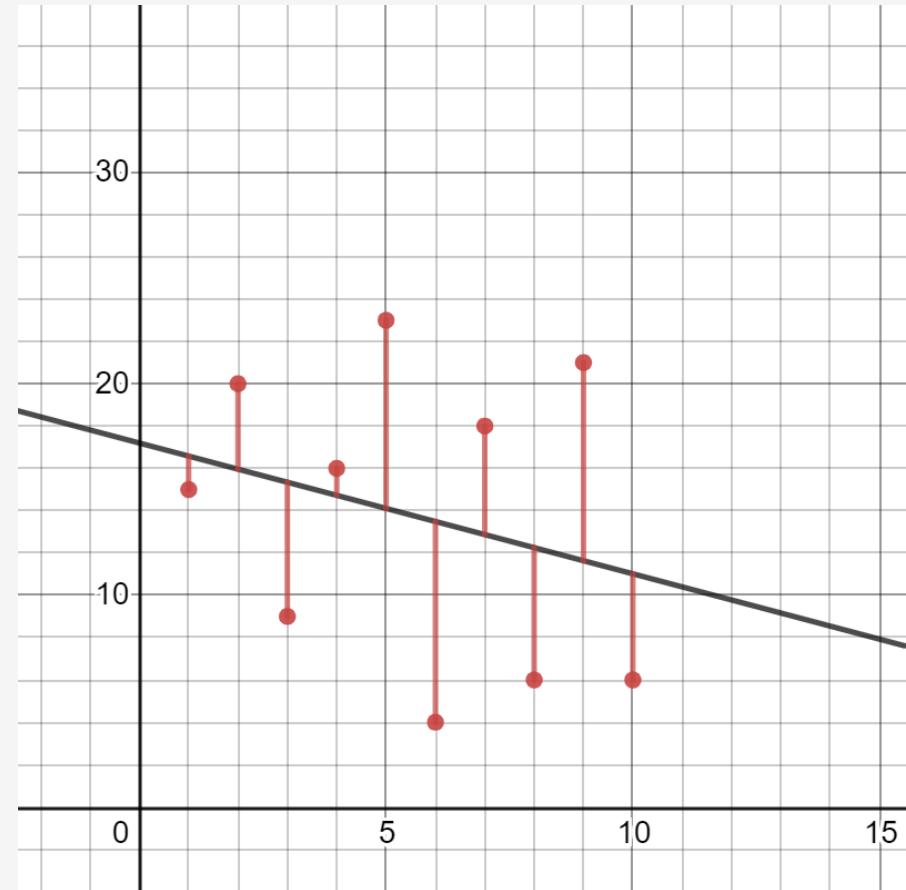
Note R<sup>2</sup> is also called the **coefficient of determination** and is the square of the Pearson correlation.

It's used to evaluate the quality of a model.

Think of R<sup>2</sup> as a measurement (between 0.0 and 1.0) of how well independent variables explain a dependent variable, rather than predicting with a simple average.

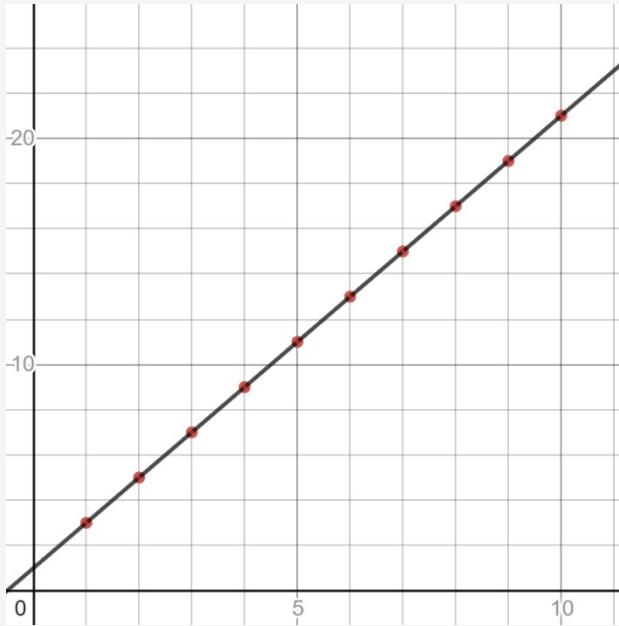
An R<sup>2</sup> of 1.0 indicates a variable perfectly explains a variable while 0.0 indicates there's no explanatory connection at all.

**EXAMPLE:** If **x** was calories consumed and **y** was number of pounds gained, and I got an R<sup>2</sup> of .85 on my test data, that means calories explains weight gain 85% better than just predicting with the average weight **y**.



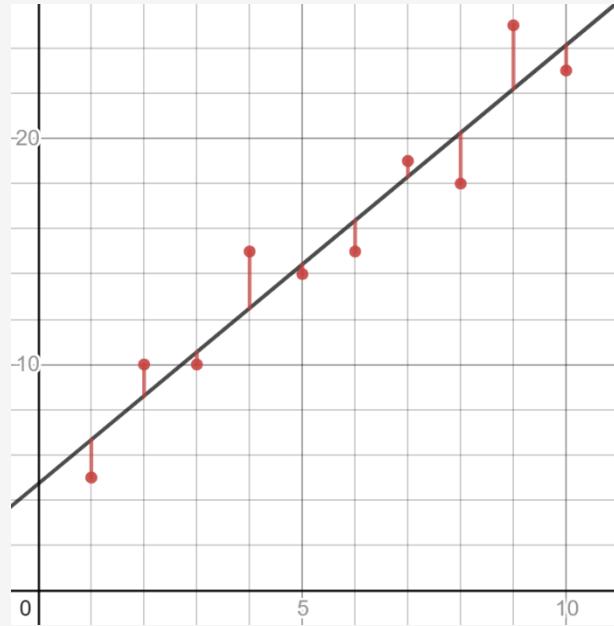
# $R^2$ Visual Examples

---



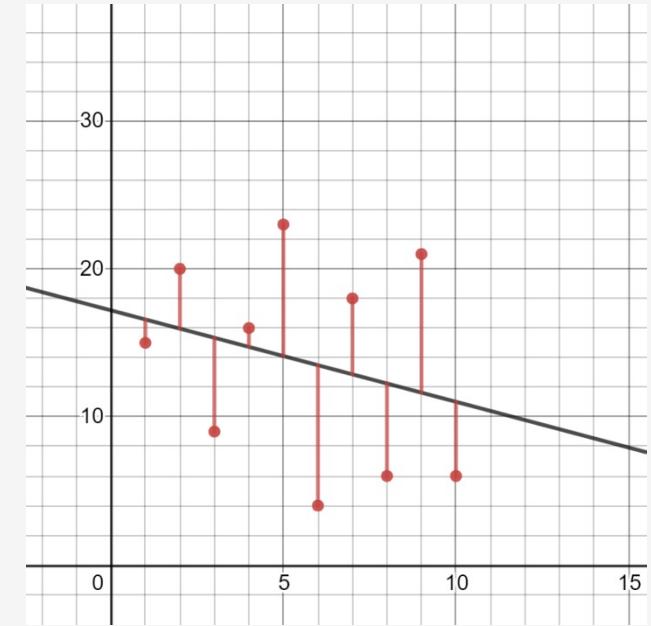
$$R^2 = 1.0$$

(PERFECT!)



$$R^2 = 0.917$$

(GOOD!)



$$R^2 = 0.071$$

(POOR!)

# How to Calculate R<sup>2</sup> by Hand

---

- 1) Separate training and test data, perform linear regression on training data.
- 2) **Calculate sum of squares for regression:** using test data, subtract each predicted y value from each actual y value, square and sum
- 3) **Calculate sum of squares for average:** using test data, subtract the average y value from each actual y value, square and sum
- 4) Divide the sum of squares for regression by the sum of squares for average, subtract that from 1.0.

Now you calculated R<sup>2</sup> from scratch!

```
# Calculating R-square from scratch

slope = 1.73333333
y_intercept = 5.499999999999999

x_test = [9, 3, 6, 7]
y_test = [25, 10, 15, 19]

y_test_avg = sum(y_test) / len(y_test)
# 17.25

y_test_predict = [(slope * x) + y_intercept for x in x_test]
#[21.09999969999992, 10.69999989999991, 15.8999997999999, 17.6333330999999]

sum_sq_regression = sum((y_test[i] - y_test_predict[i])**2 for i in range(0,len(y_test)))
sum_sq_total = sum((y_test[i] - y_test_avg)**2 for i in range(0,len(y_test)))

r_square = 1.0 - sum_sq_regression / sum_sq_total
# 0.8478030805337009

print(r_square)
```

# Scikit-Learn and R<sup>2</sup>

---

## Using 3-Fold

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score,
train_test_split

df = pd.read_csv('https://bit.ly/3m20B31', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.33)

# Perform a simple linear regression
model = LinearRegression()
model.fit(X_train, Y_train)

result = model.score(X_test, Y_test)
print(result) # prints 0.8478030825856914
```

## Using Cross Validation

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score

df = pd.read_csv('https://bit.ly/3nJBdj9', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)\
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()

# Use R-square to evaluate the performance
# An R-square of 0 is no-fit, and 1 is perfect fit
results = cross_val_score(model, X, Y, cv=kfold, scoring='r2')

print("R^2: mean=% .3f (stdev-% .3f)" % (results.mean(), results.std()))
# prints R^2: mean=0.590 (stdev-0.219)
```

# Appendix C

## Rescaling Data

# Normalization

---

**Normalization is scaling and converting the values of each variable, so they are relatively close together.**

- Imagine you had a variable ***age*** whose values typically range in 0-99.
- But you had another variable ***income*** that typically ranges from 30,000 to 1,000,000.
- Because these ranges are so drastically different, fitting a model is not going to be productive until you transform them somehow.



**There are a variety of techniques you can employ from linear scaling to fitting to a standard normal distribution.**

# Rescaling

---

**When you have different fields with varying scales, it can be helpful to **rescale**, or normalize, the data by compressing it between 0.0 and 1.0.**

This can be helpful for optimization algorithms that perform training, making it easier to iterate parameters to fit the data more productively.

**Note you do not always have to rescale, but sometimes it can give a better result especially for data with fields that vary in scale.**

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

df = pd.read_csv('https://bit.ly/33iTfS9', delimiter=',')
# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, 5th column)
Y = df.values[:, 4]

# Rescale all the input variables to be between 0 and 1
scaler = MinMaxScaler(feature_range=(0.0, 1.0))
rescaled_X = scaler.fit_transform(X)

print(rescaled_X)
```

*Doing a simple rescale between 0.0 and 1.0 above in scikit-learn*



[[ 0 25 2 3]	[[ 0. 0. 0.5 0.2 ]]
[ 0 30 2 3]	[ 0. 0.20833333 0.5 0.2 ]
[ 0 26 2 3]	[ 0. 0.04166667 0.5 0.2 ]
[ 0 25 1 2]	[ 0. 0. 0.25 0.1 ]
[ 0 28 1 2]	[ 0. 0.125 0.25 0.1 ]
[ 0 30 2 4]	[ 0. 0.20833333 0.5 0.3 ]
[ 0 49 4 8]	[ 0. 1. 1. 0.7 ]

# Standardization

**Standardization** is transforming numeric variables to fit a standard normal distribution (with a mean of 0 and standard deviation of 1) and express each value in standard deviations.

It can be preferable to rescaling, if the data can assume a normal distribution for each variable.

In other words, standardization is finding the z-scores for each variable.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

df = pd.read_csv('https://bit.ly/3fIJSR', delimiter=',')

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Rescale all the input variables to be between 0 and 1
scaler = StandardScaler().fit(X)
rescaled_X = scaler.fit_transform(X)

print(rescaled_X)
```

**Doing a standardization on all data fields**

```
[[ 6.    148.    72.    ...  33.6   0.627  50.   ]
 [ 1.     85.    66.    ...  26.6   0.351  31.   ]
 [ 8.    183.    64.    ...  23.3   0.672  32.   ]]
```



```
[[ 0.63994726  0.84832379  0.14964075 ...  0.20401277  0.46849198
  1.4259954 ]
 [-0.84488505 -1.12339636 -0.16054575 ... -0.68442195 -0.36506078
 -0.19067191]
 [ 1.23388019  1.94372388 -0.26394125 ... -1.10325546  0.60439732
 -0.10558415]]
```

# Unit Vector Normalization

---

**Not to be confused with standardization, unit vector normalization is rescaling each data record to have a vector length of 1.0.**

This is probably the most complicated transformation, as it involves some understanding of linear algebra.

It works well for sparse data sets and it compresses the data into a smaller space, which is especially helpful for high-dimension datasets.

It is often used for neural networks, k-nearest neighbors, and a few other ML algorithms.

```
import pandas as pd
from sklearn.preprocessing import Normalizer

df = pd.read_csv('https://bit.ly/3fIZJSR', delimiter=',')

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Rescale all the input variables to be normalized
scaler = Normalizer().fit(X)
rescaled_X = scaler.fit_transform(X)

print(rescaled_X)
```

***Doing a normalization on all data fields***

```
[[ 6.    148.    72.    ...   33.6   0.627  50.   ]
 [ 1.     85.    66.    ...   26.6   0.351  31.   ]
 [ 8.    183.    64.    ...   23.3   0.672  32.   ]]
```



```
[[0.03355237 0.82762513 0.40262844 ... 0.18789327 0.00350622 0.27960308]
 [0.008424 0.71604034 0.55598426 ... 0.22407851 0.00295683 0.26114412]
 [0.04039768 0.92409698 0.32318146 ... 0.11765825 0.00339341 0.16159073]]
```