Machine Learning from Scratch

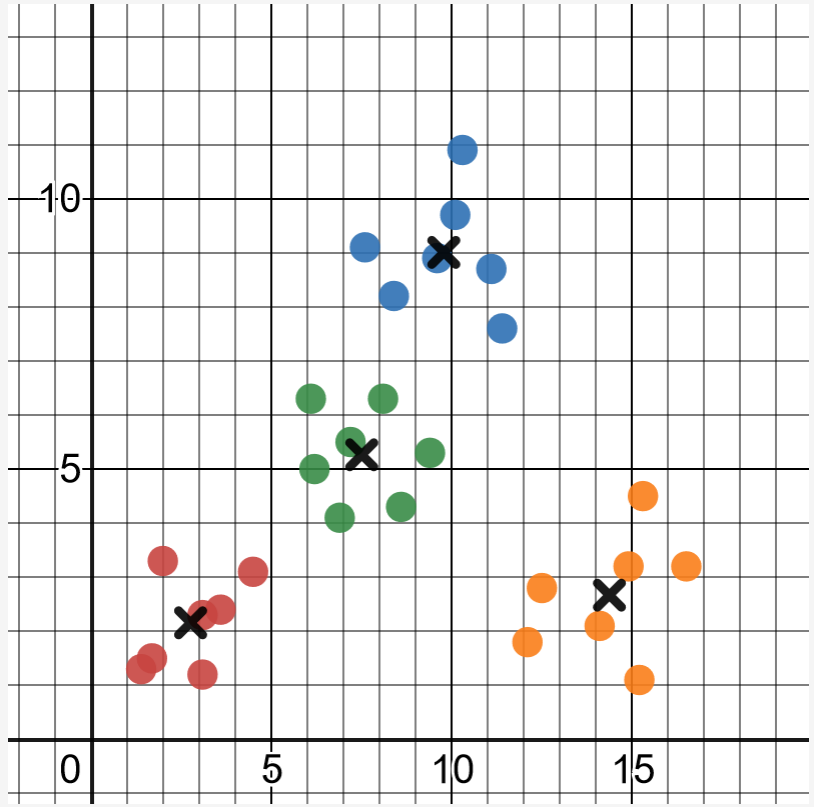
Hacking ML without Libraries

Section I
Overview

Agenda

Here is what we will do for the next 3 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. Classification with **Neural Networks**



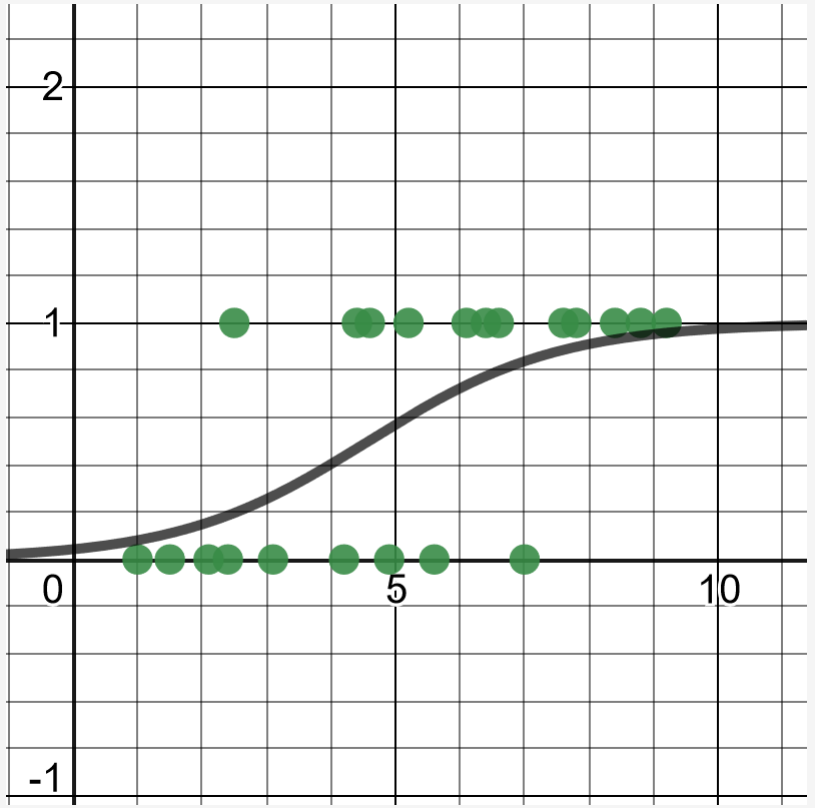
What to Expect

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.

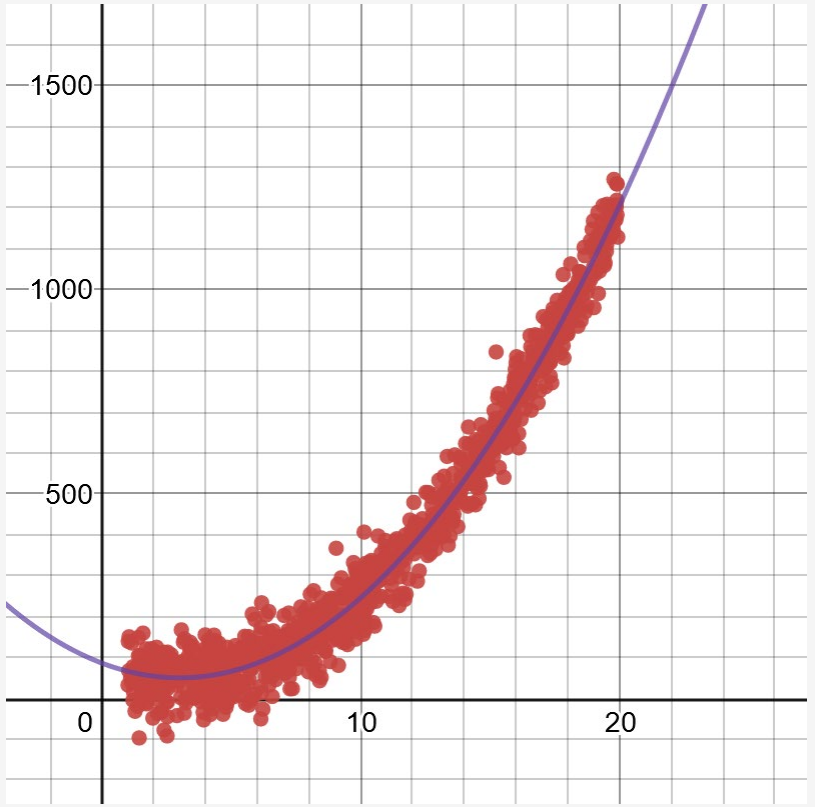


What Not to Expect

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.

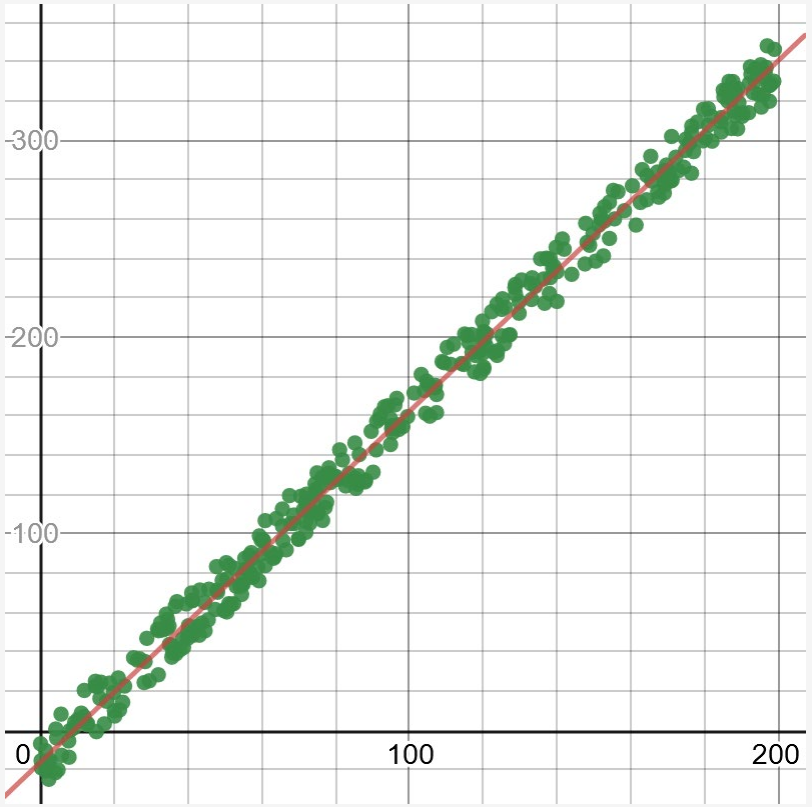
We will not be validating and analyzing our machine learning models formally, and instead focus on building the ML algorithms.



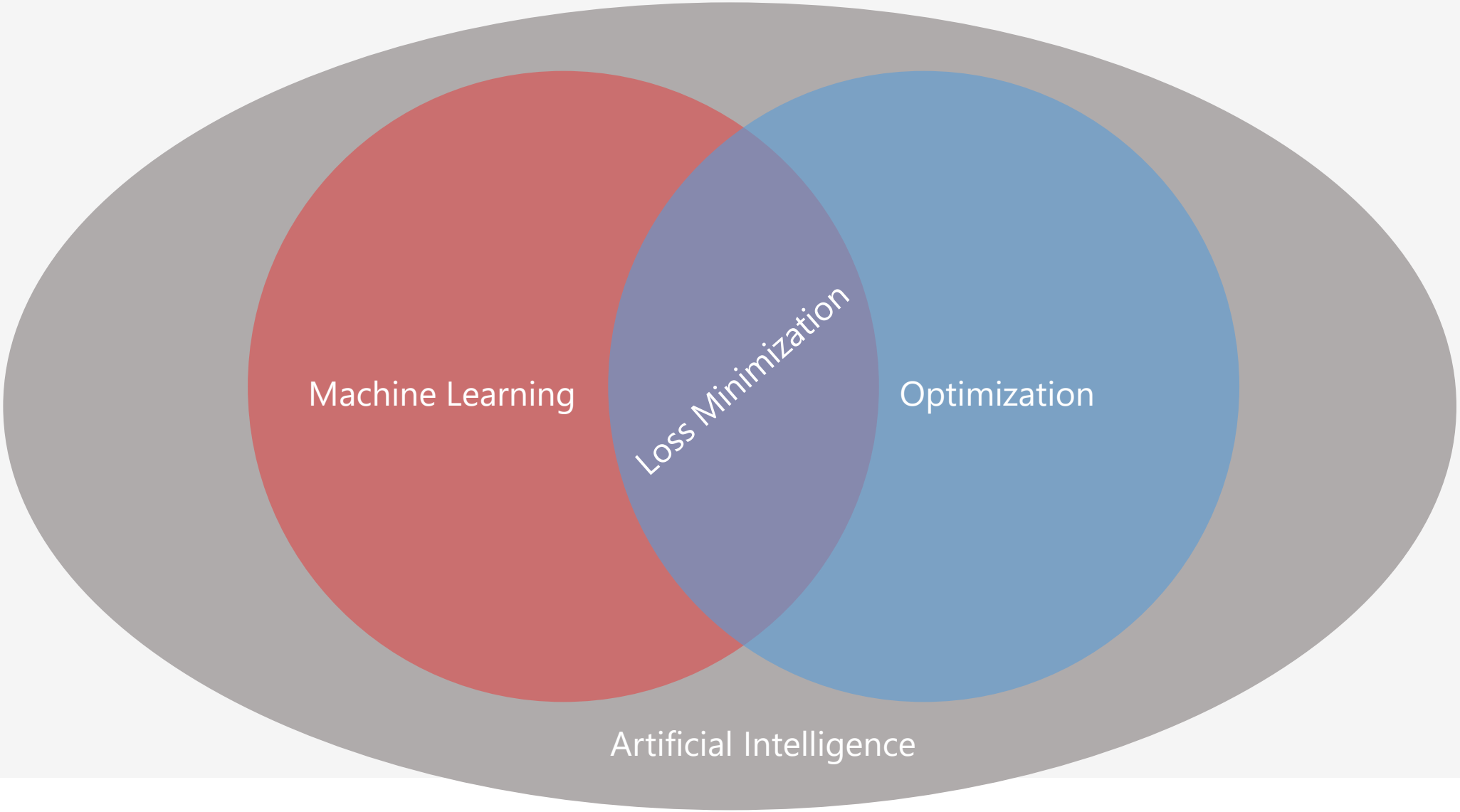
What Not to Expect

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

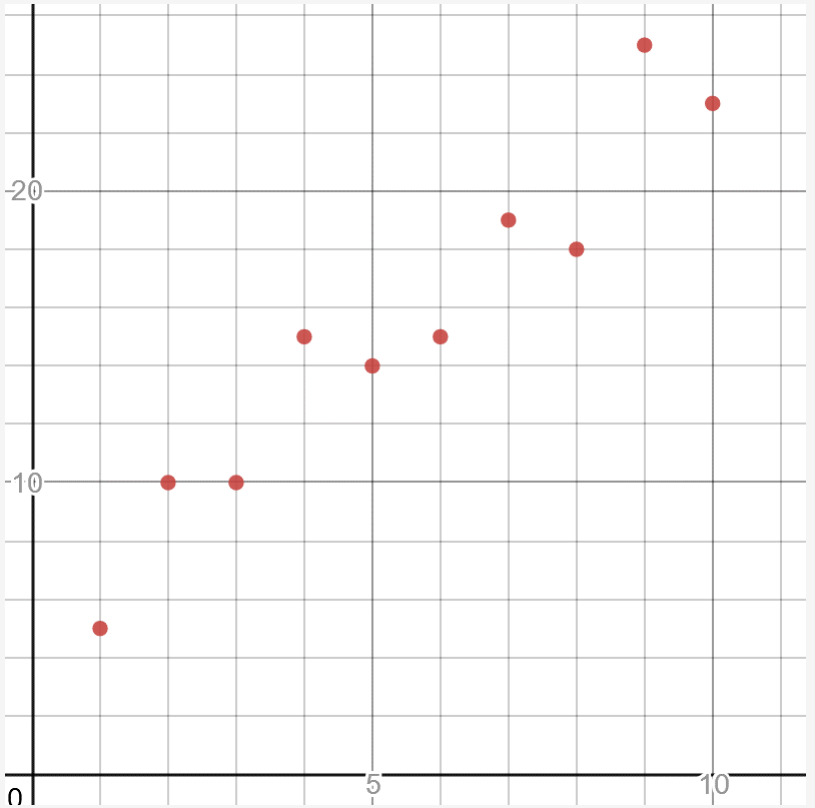
Simple Linear Regression

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

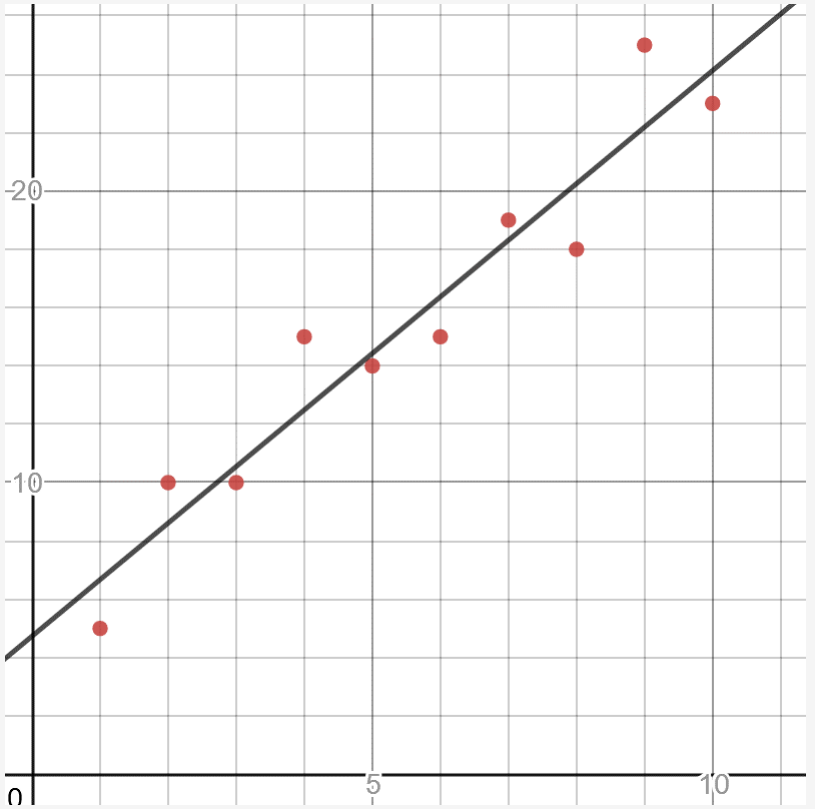
For a simple linear regression, we want to find a function $\boldsymbol{y} = \boldsymbol{mx} + \boldsymbol{b}$ that best fits to these points.

$$y = mx + b$$

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

So the missing information is "what $\boldsymbol{m}$ and $\boldsymbol{b}$ values will create the best fit line"?

But before we solve for the $\boldsymbol{m}$ and $\boldsymbol{b}$ values, let's first ask "what defines a best fit anyway?"



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

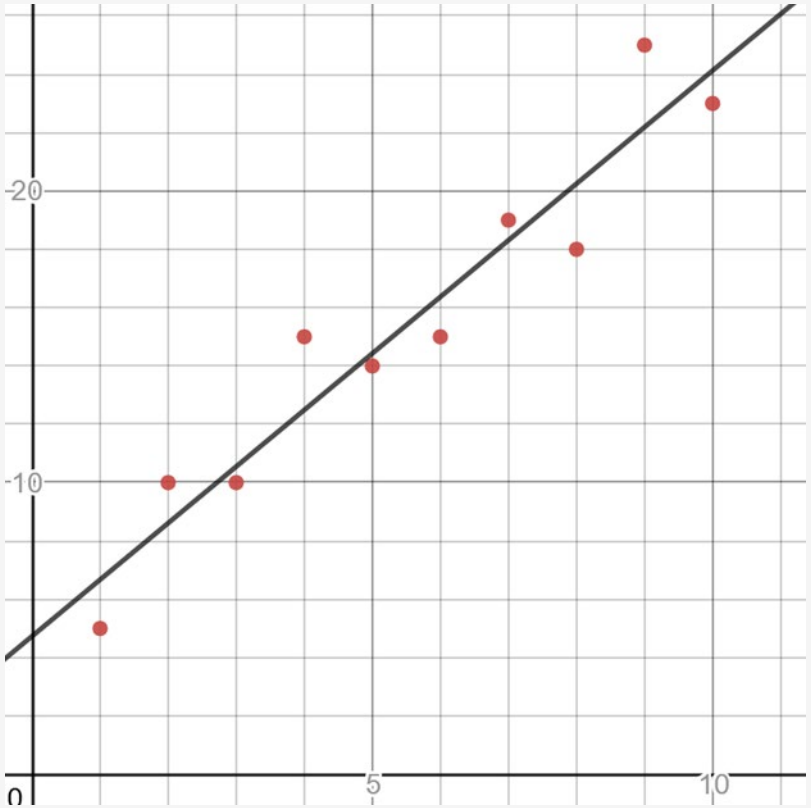
Simple Linear Regression

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.



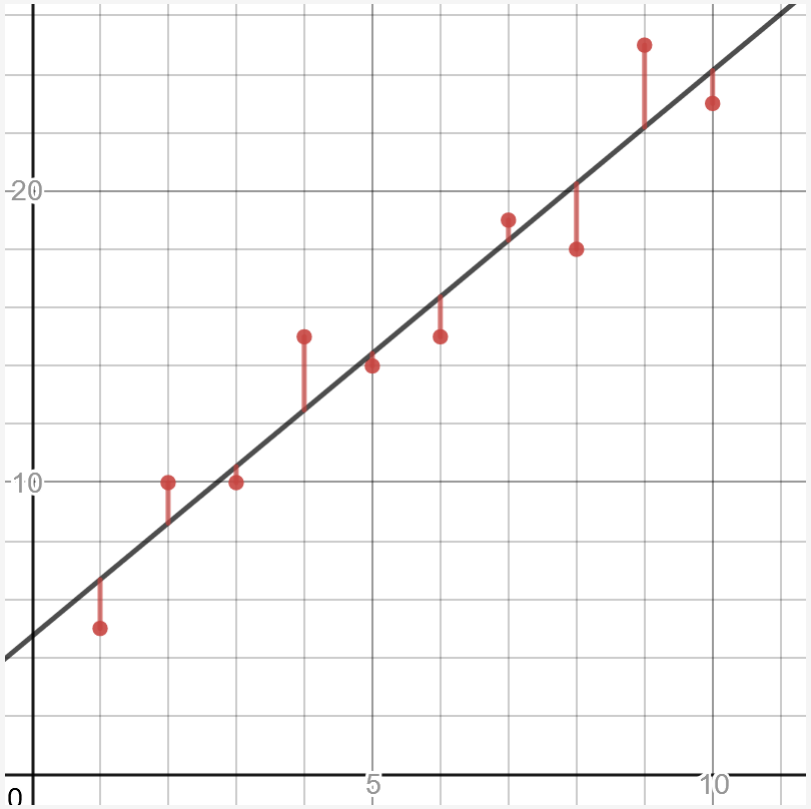
https://www.desmos.com/calculator/fmhotfn3qm

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?



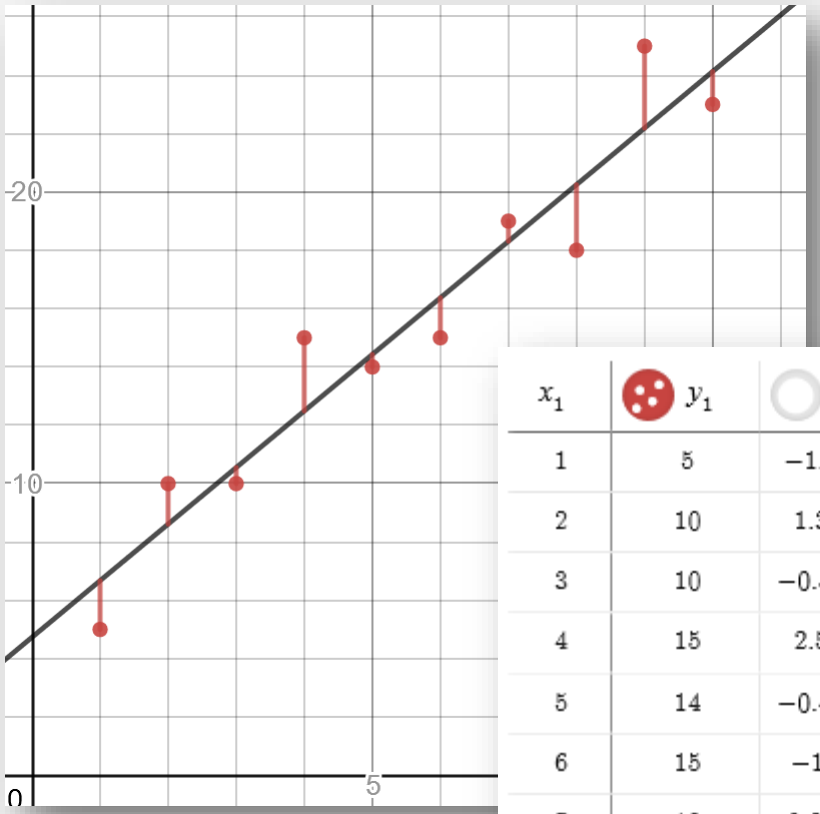
https://www.desmos.com/calculator/fmhotfn3qm

Sum of Least Squares

When we solve for $\boldsymbol{m}$ and $\boldsymbol{b}$ 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?



x_1	y_1	e_1
1	5	−1.6727273
2	10	1.3878788
3	10	−0.55151515
4	15	2.5090909
5	14	−0.43030303
6	15	−1.369697
7	19	0.69090909
8	18	−2.2484848
9	25	2.8121212
10	23	−1.1272727

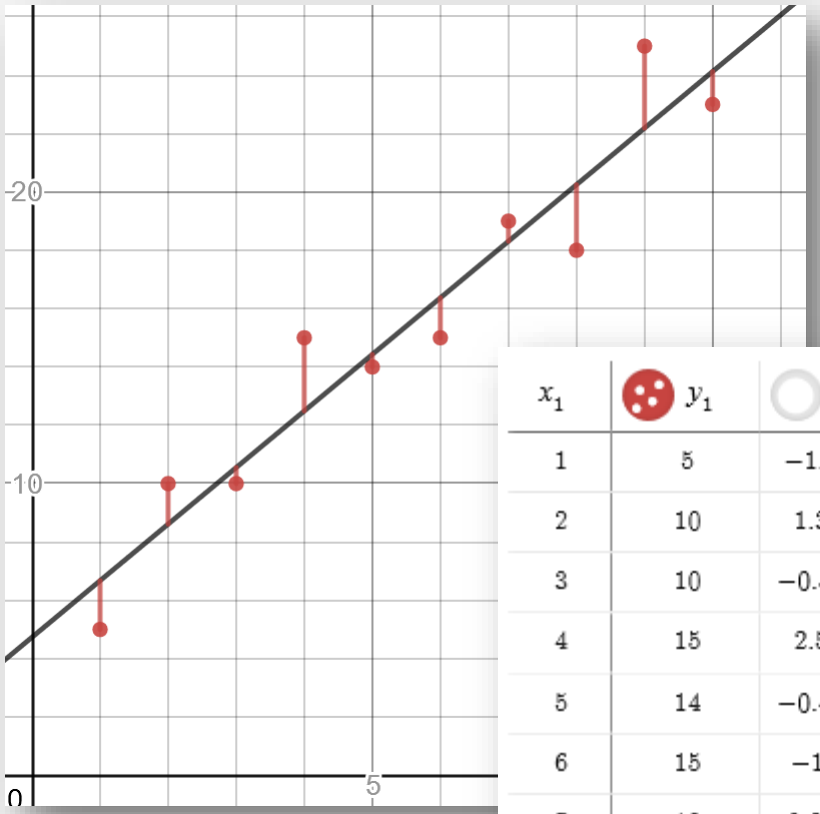
Sum of Least Squares

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 $\boldsymbol{m}$ and $\boldsymbol{b}$ values that will find the sum of least squares.

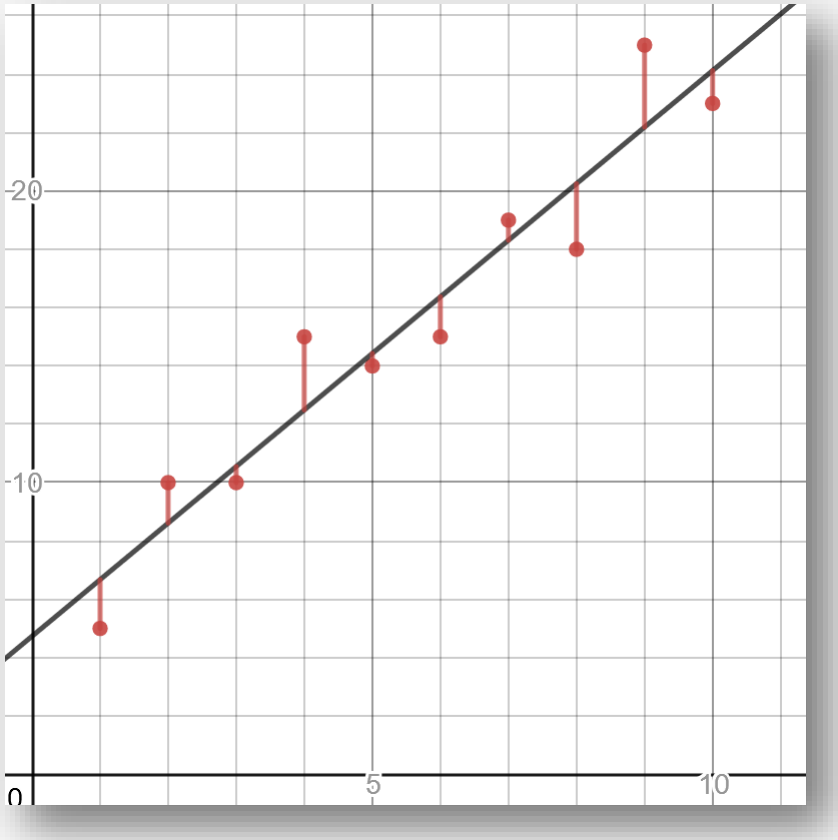
https://www.desmos.com/calculator/fvrnuhw0hy



x_1	y_1	e_1
1	5	−1.6727273
2	10	1.3878788
3	10	−0.55151515
4	15	2.5090909
5	14	−0.43030303
6	15	−1.369697
7	19	0.69090909
8	18	−2.2484848
9	25	2.8121212
10	23	−1.1272727

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



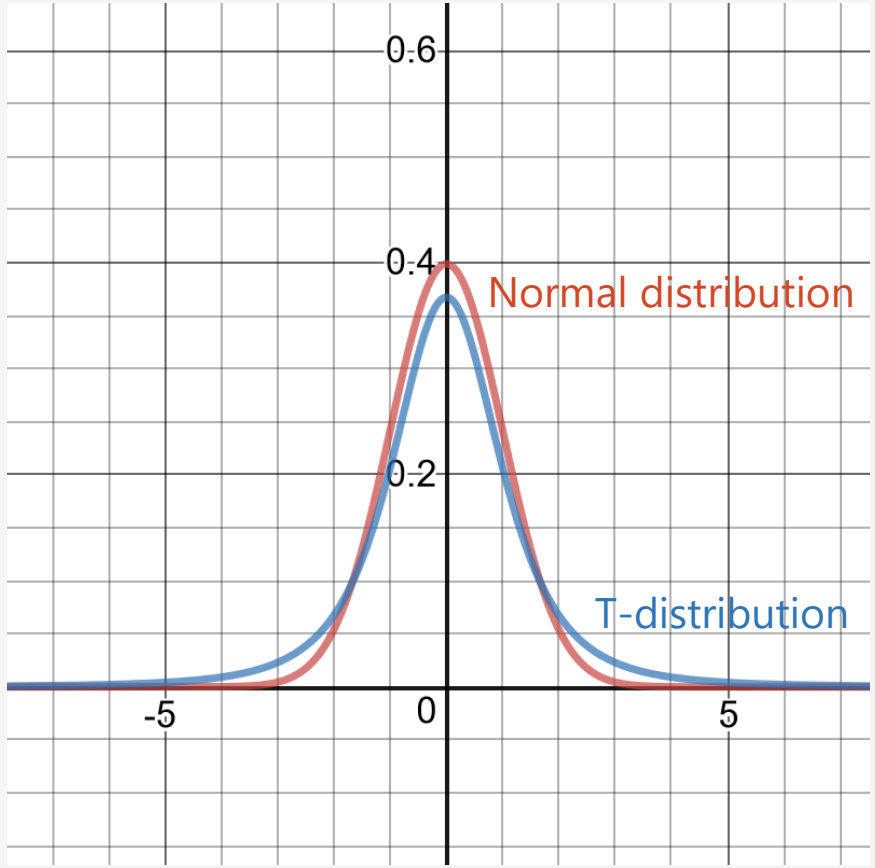
Solving for a Fit

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

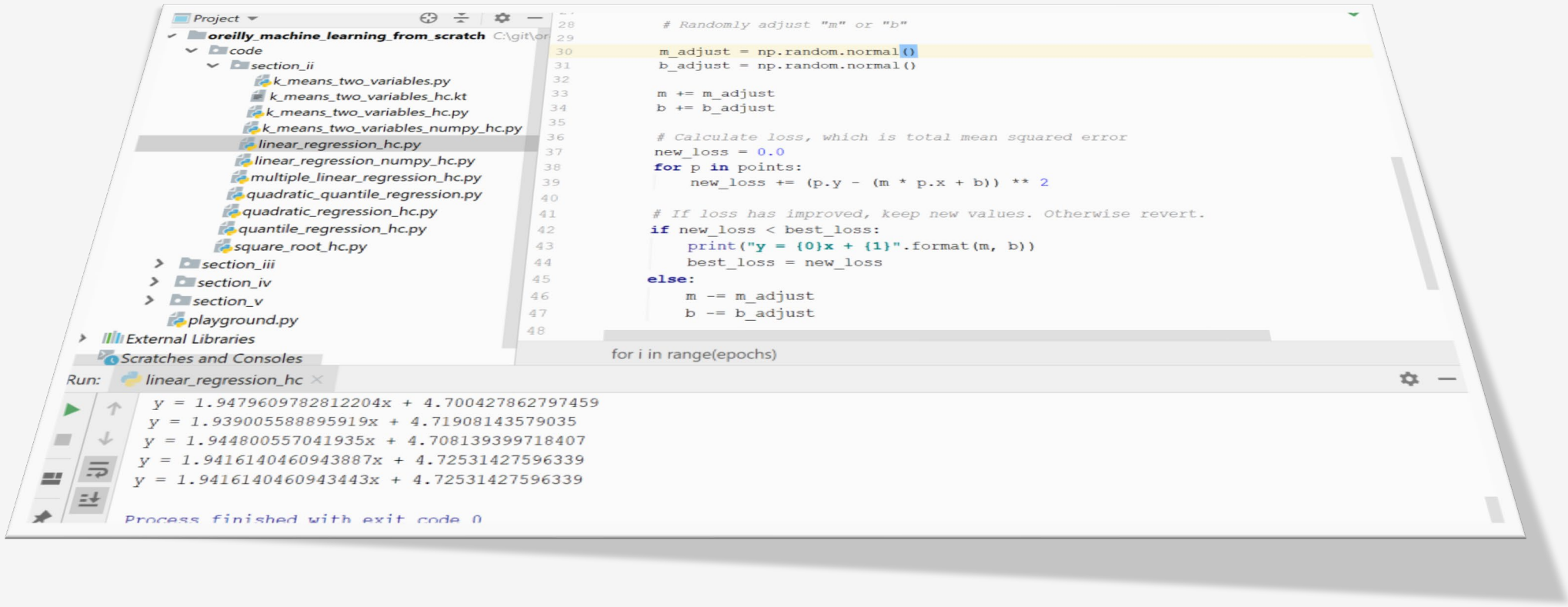
1. Start with a random or initial solution, even if it is poor quality.

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



Multiple Linear Regression

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

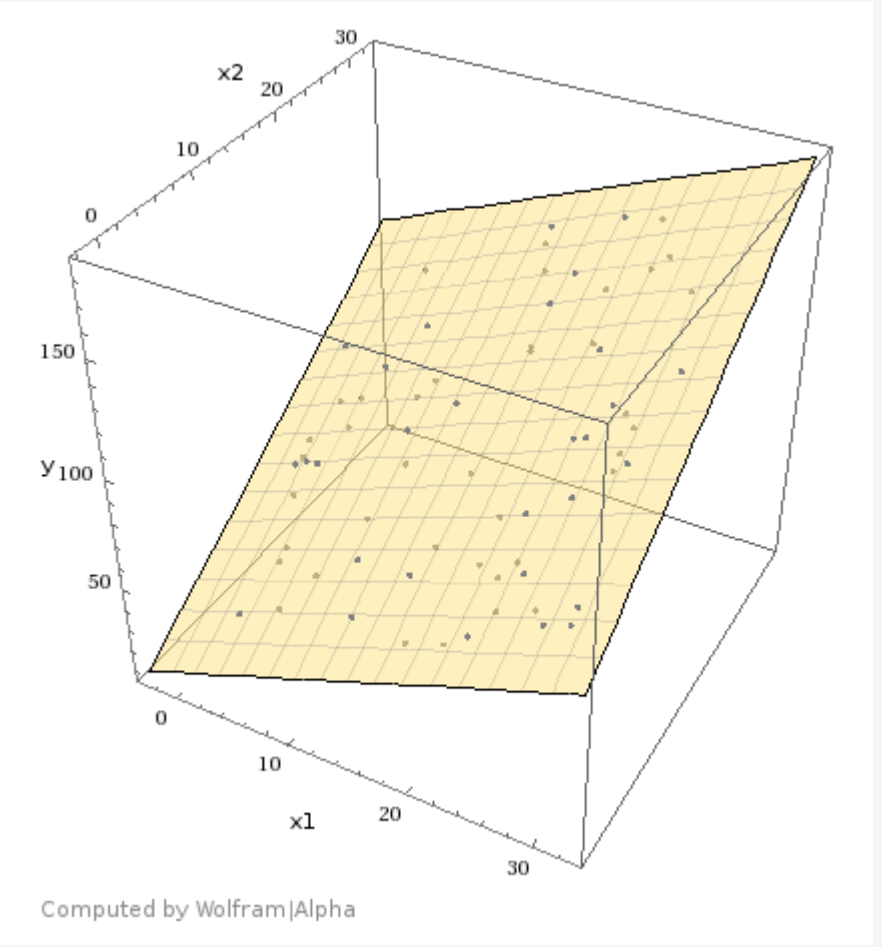
We can also solve for multiple independent variables, like x_1, x_2, x_3, and so on…

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

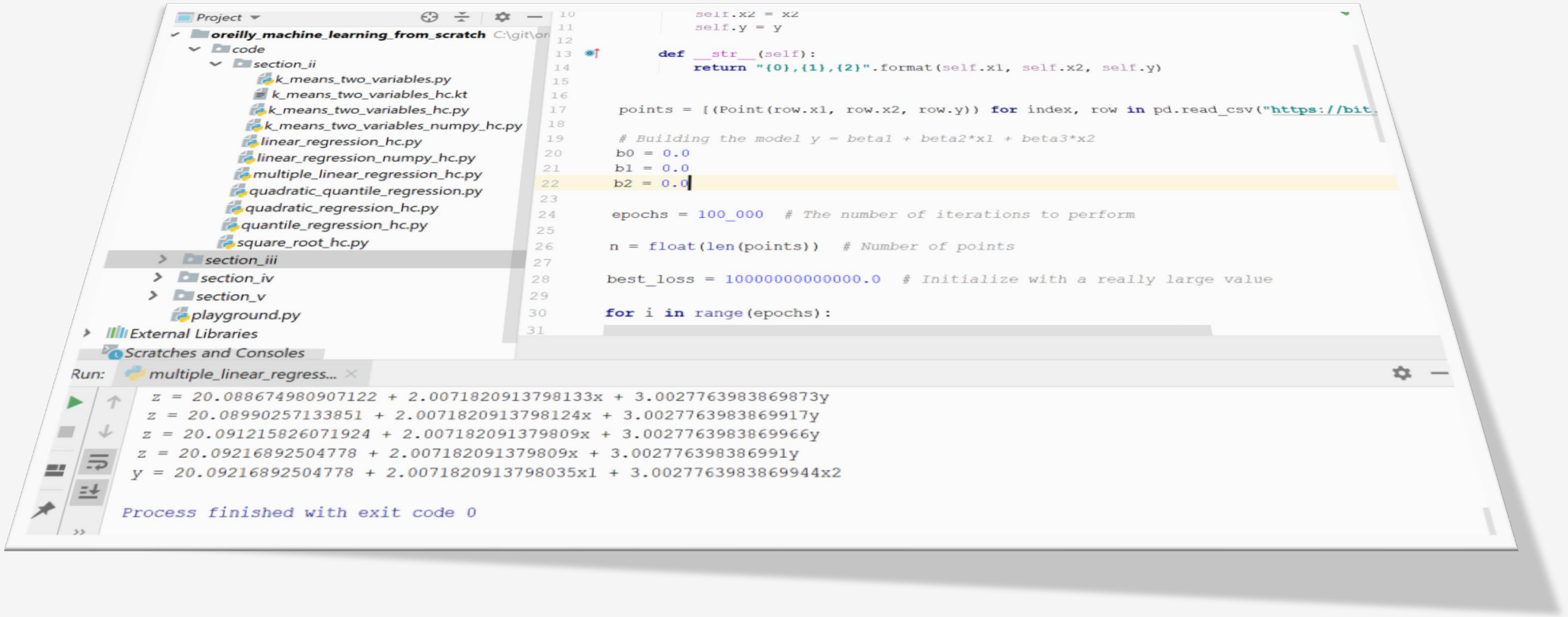
$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Instead of $\boldsymbol{m}$ and $\boldsymbol{b}$ constants, we now need to solve for each β parameter, where β_0 is the y-intercept and β_1 and β_2 are slopes for the respective x variables.

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



Hands-On: Multiple Linear Regression

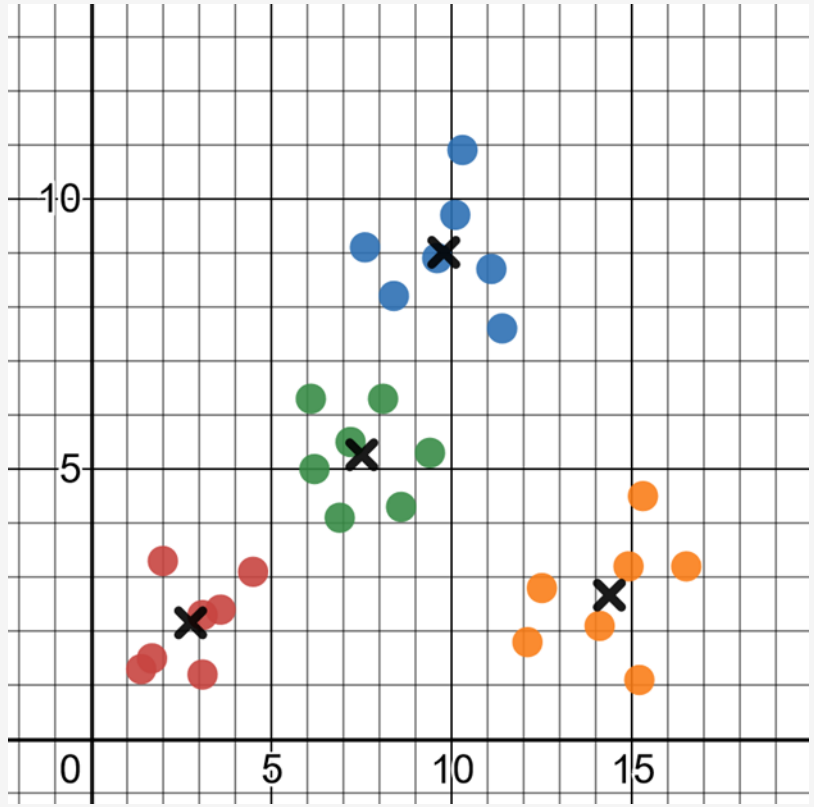


K-Means Clustering

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

K-means 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.



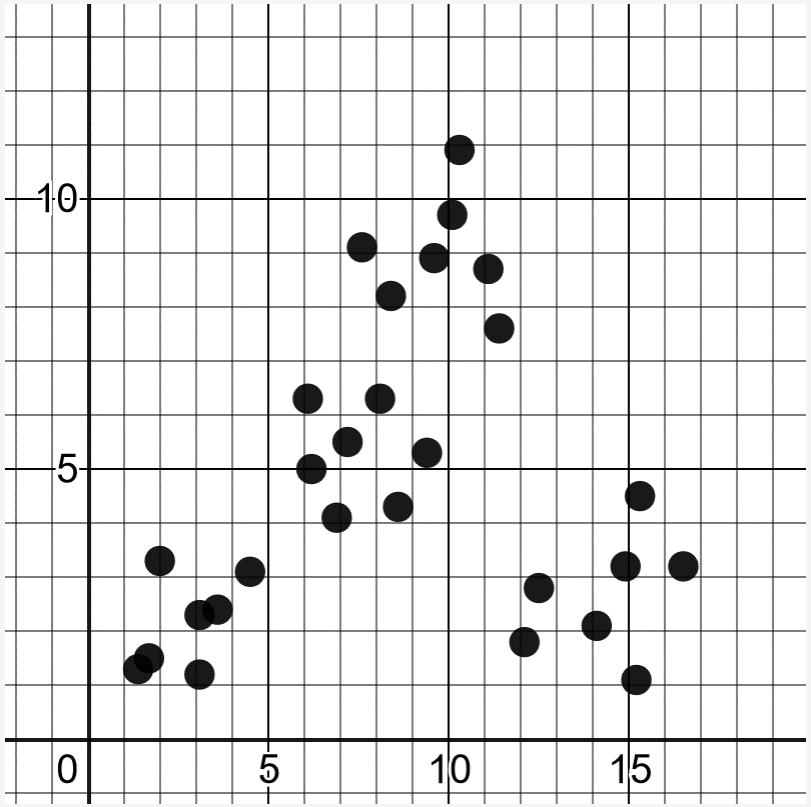
K-Means Clustering

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.

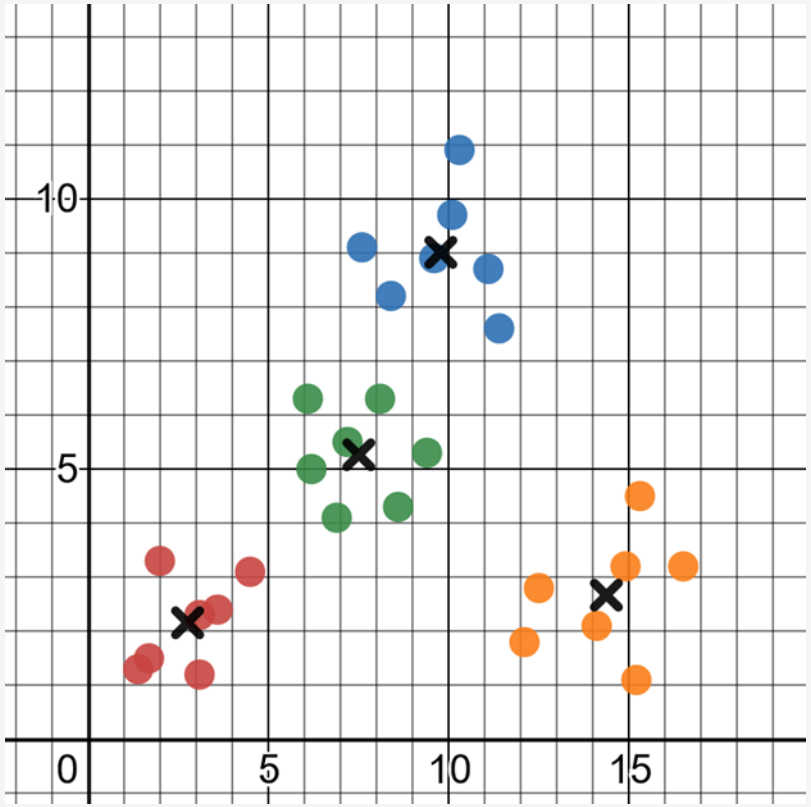


K-Means Clustering

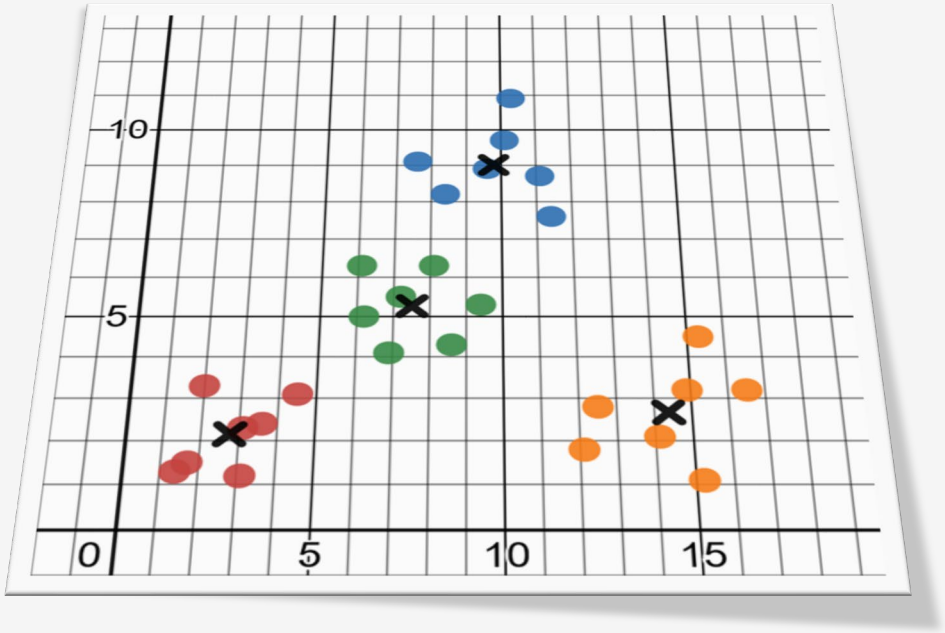
We will start each centroid at (0,0).

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.



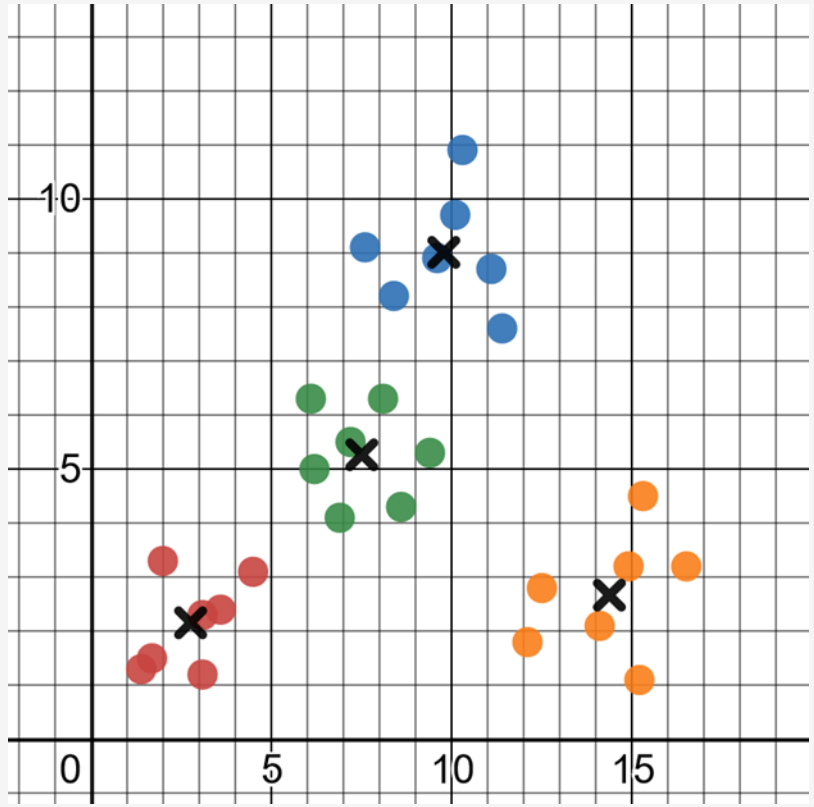
Hands-On: K-Means Clustering



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.



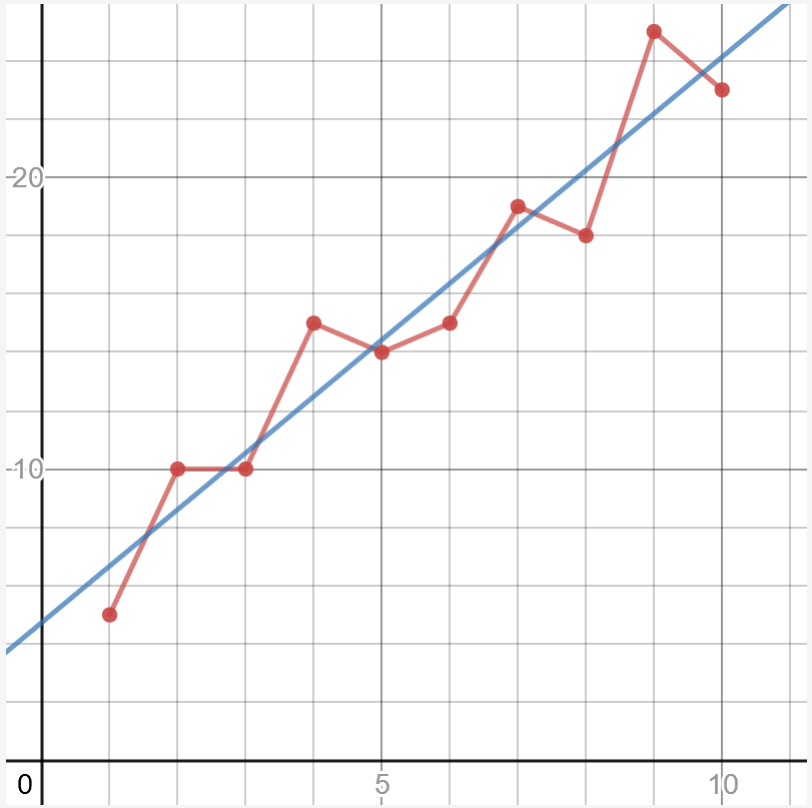
A Note About Overfitting

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.

A Note About Normalization

Normalization is another topic we do not have time to explore deeply in this 4-hour class but should at least get mentioned.

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.

Quiz Time!

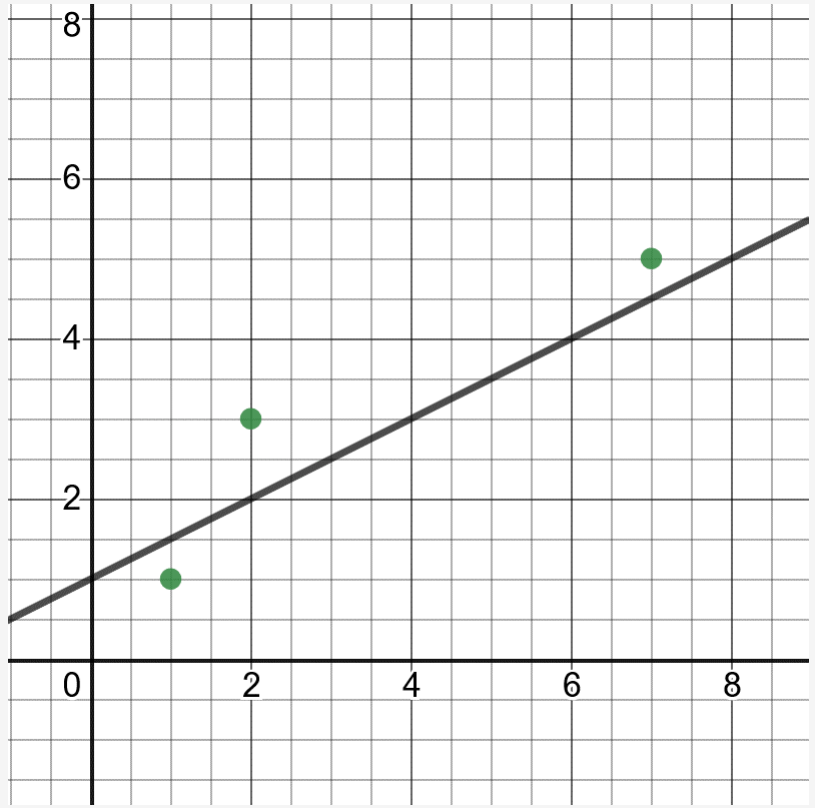
You have three points (1,1), (2,3), and (7,5). You are testing a fit for a line $y = \frac{1}{2}x + 1$. What is the loss calculated by sum of squares?

A) 1.5

B) 1.75

C) 1.0

D) -1.5



Quiz Time!

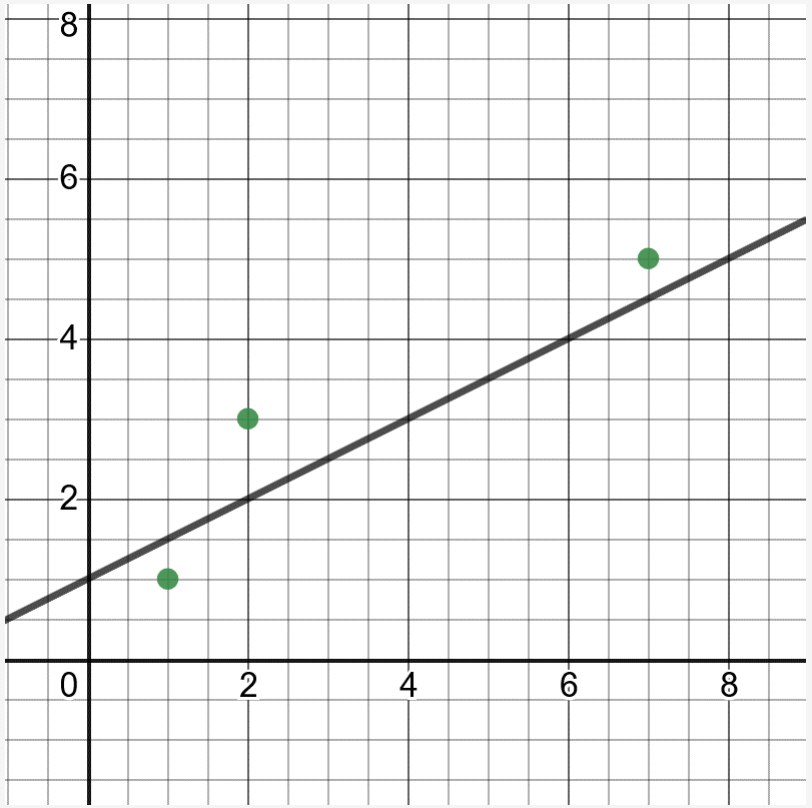
You have three points (1,1), (2,3), and (7,5). You are testing a fit for a line $y = \frac{1}{2}x + 1$. What is the loss calculated by sum of squares?

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

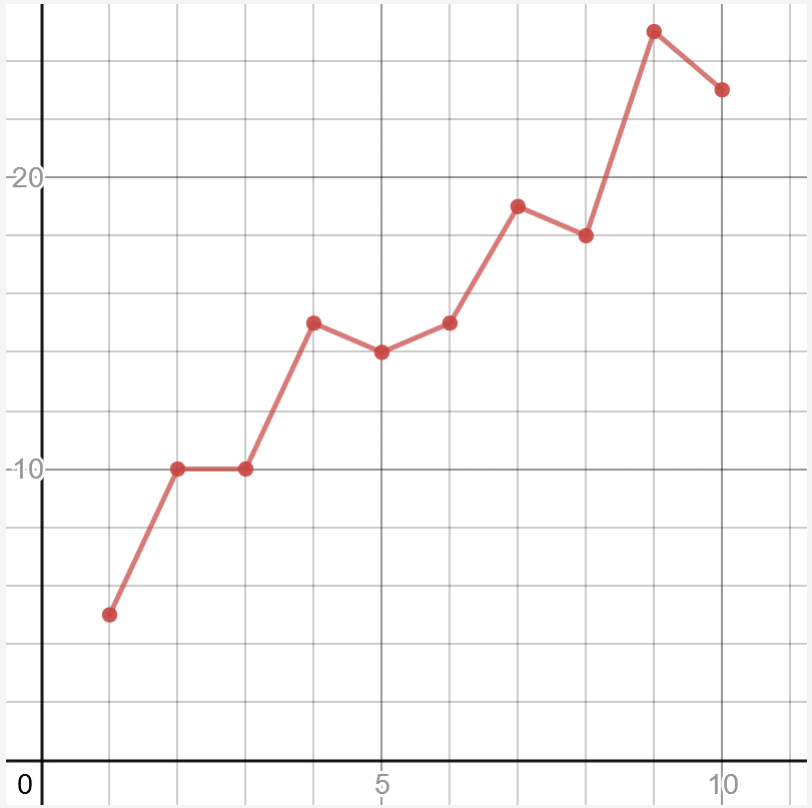


Quiz Time!

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!

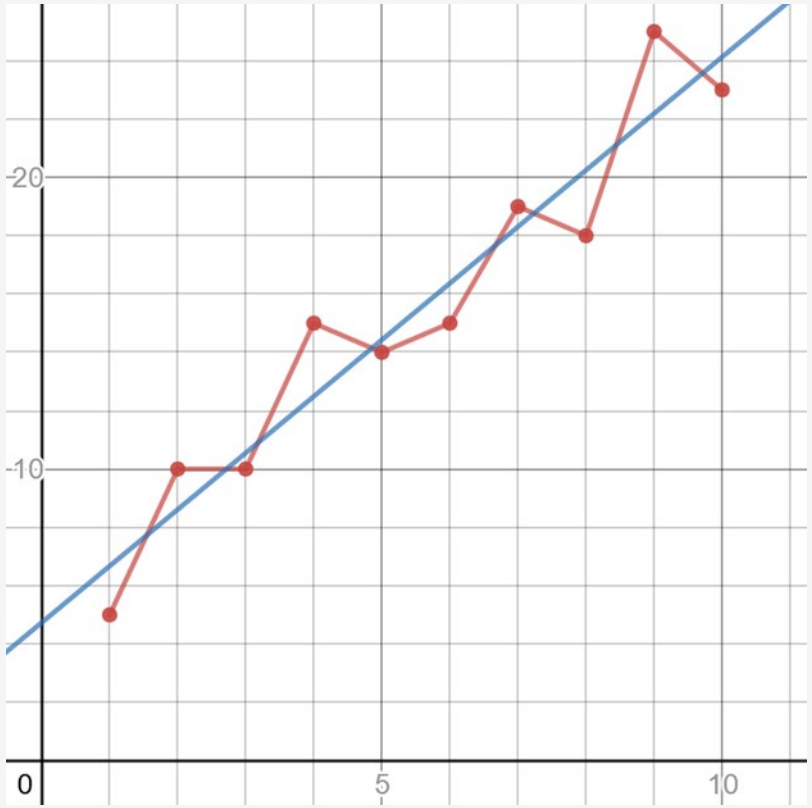
It is good to fit a regression as close to the points as possible, even if you just connect the points as shown to the right.

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*?

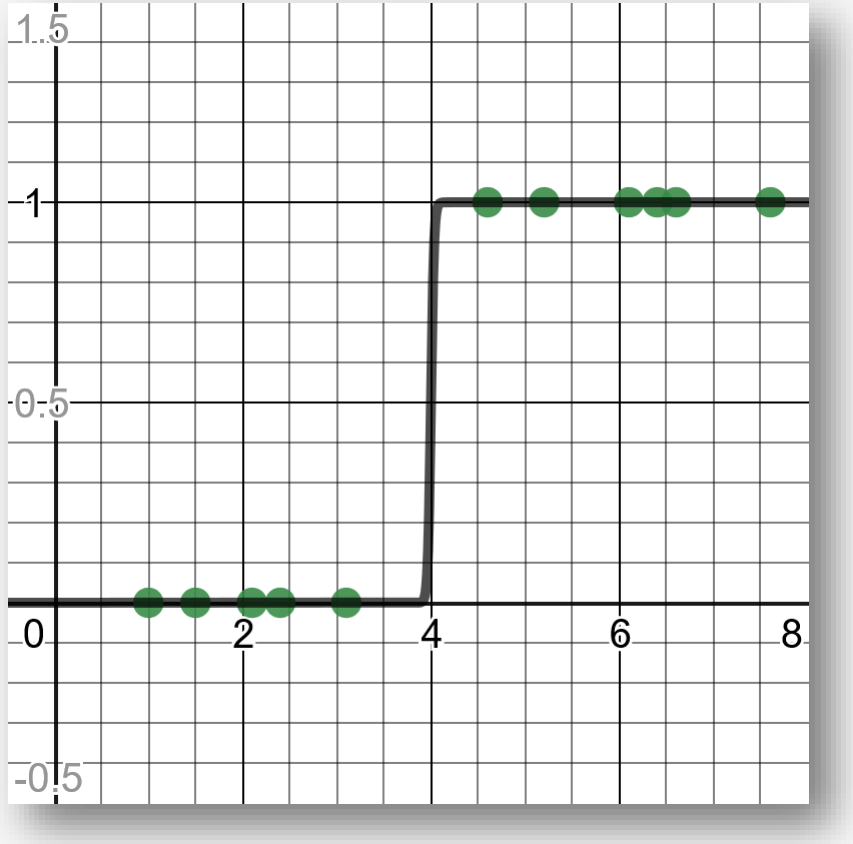
There are several machine learning algorithms that work well for classification, and we are going to learn first about Logistic Regression.

Hot dog: 0.92

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…

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…

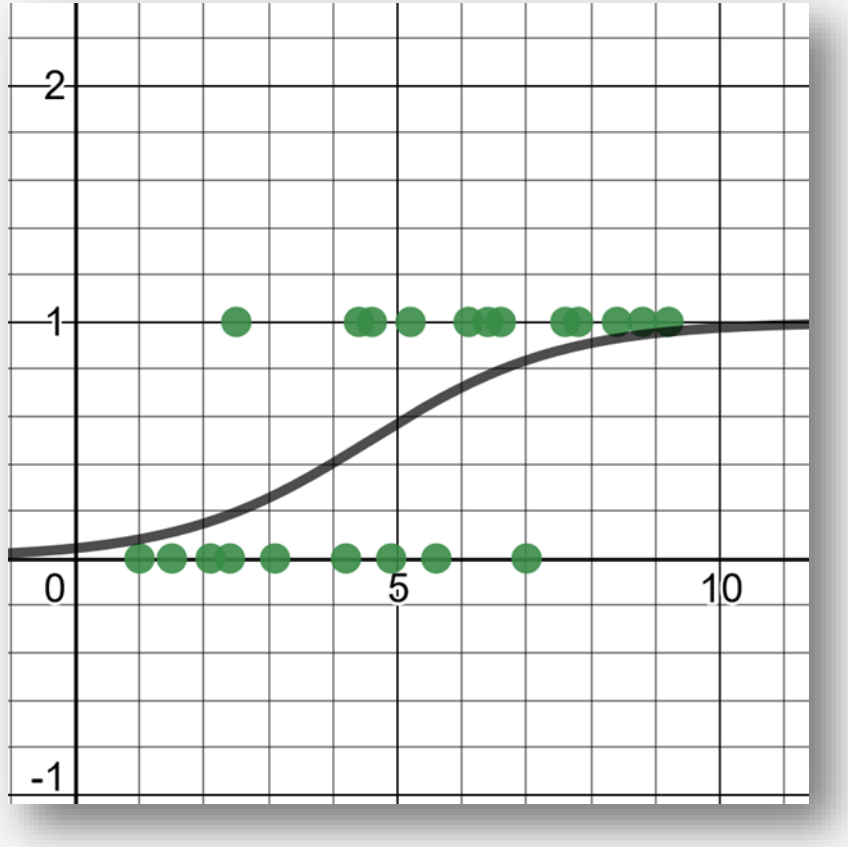
https://www.desmos.com/calculator/prs2p0sofc

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").

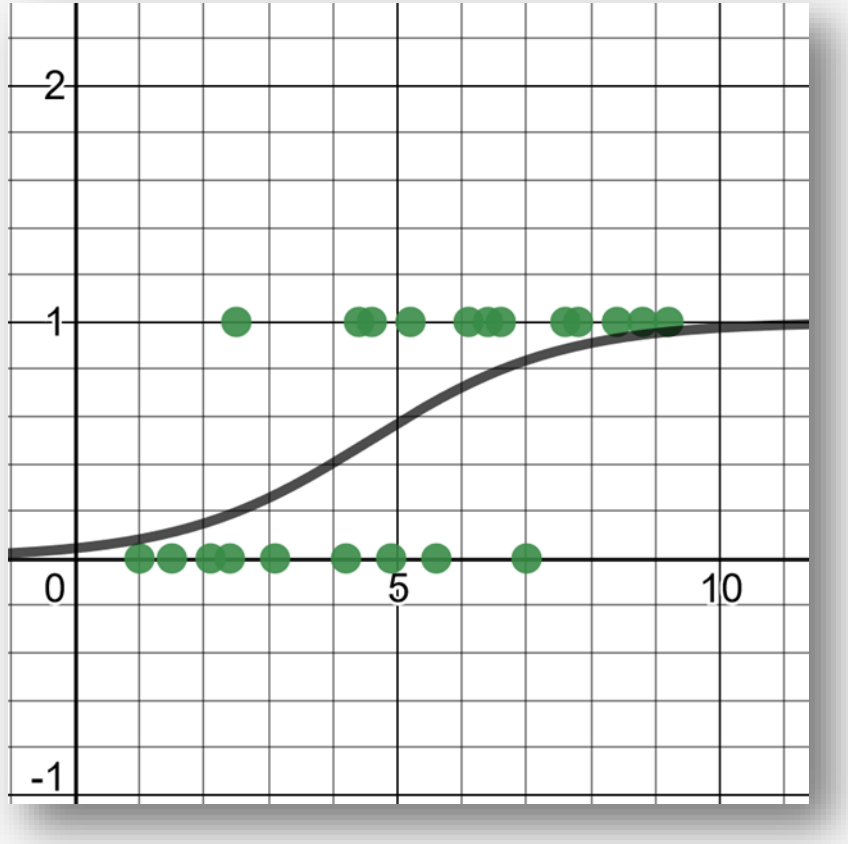


https://www.desmos.com/calculator/bsqtqfians

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.



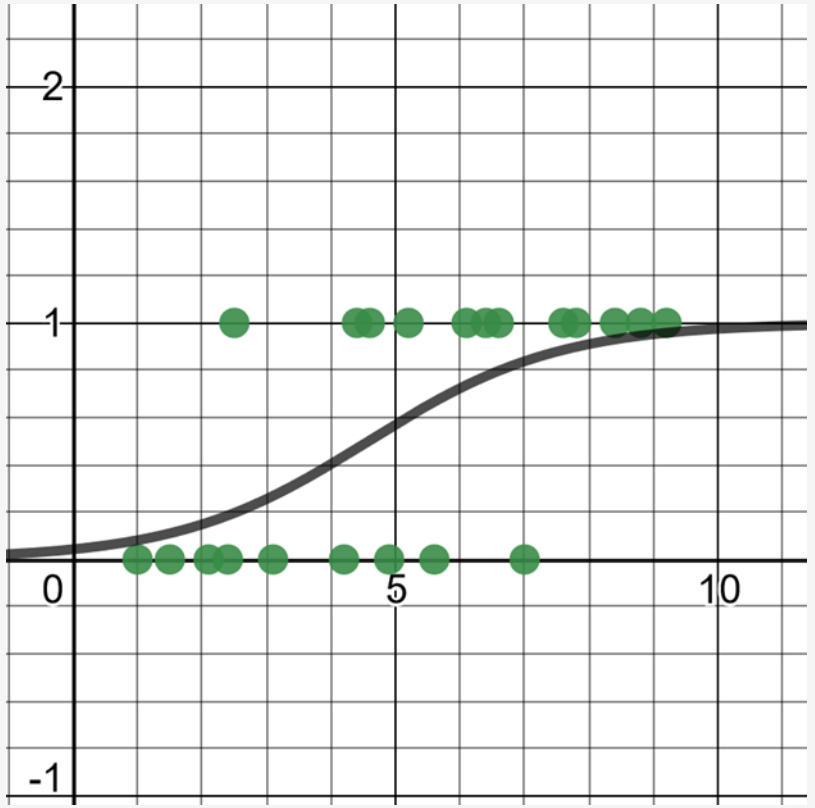
https://www.desmos.com/calculator/bsqtqfians

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.

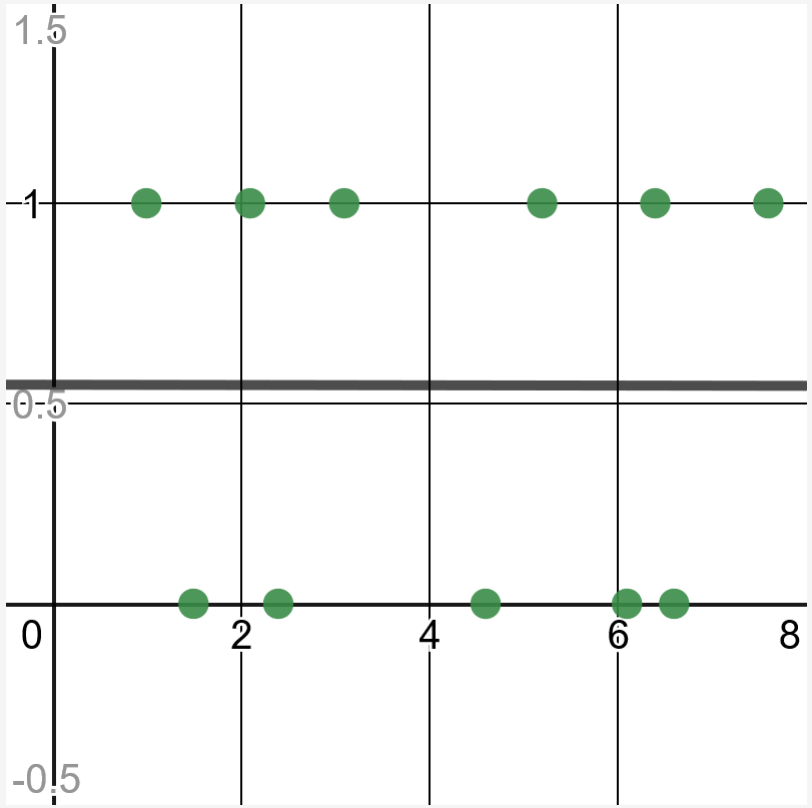
- An S-shaped curve (a **logistic function**) is fit to the points and then used to predict probability.
- If a predicted value (the y-axis) is less than .5 it is typically categorized as false (0), and if the predicted value is greater than/equal to .5 it is typically categorized as true (1).



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

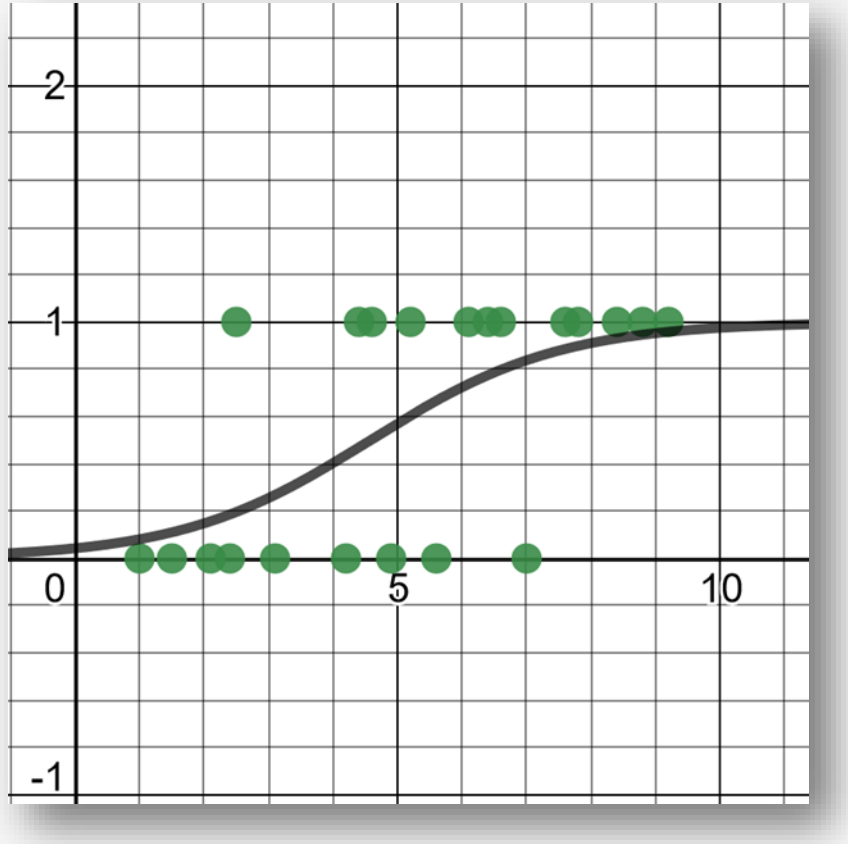
Just Show Me the Math!

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
```



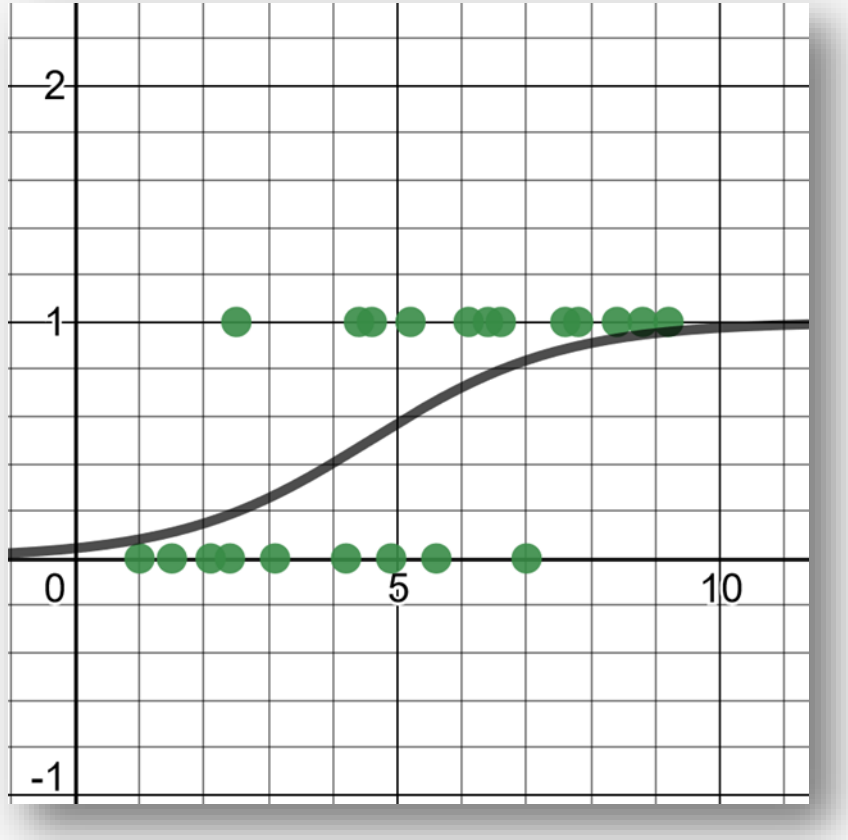
https://www.desmos.com/calculator/blfcwrlnuw

Just Show Me the Math!

You may see 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!

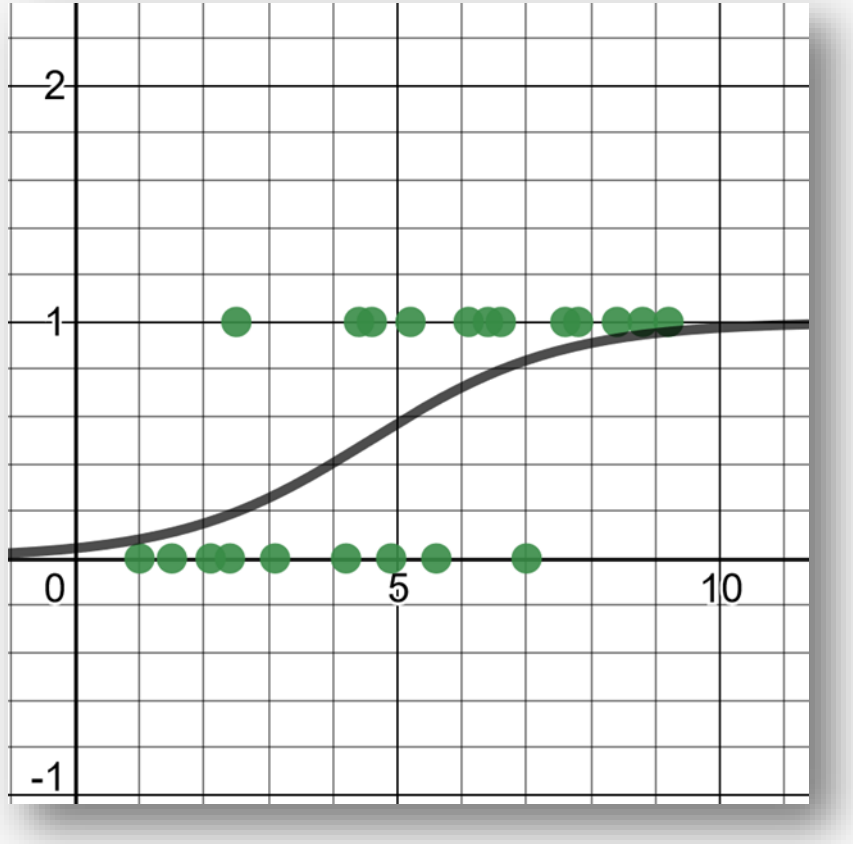
$$y = \frac{1.0}{1.0 + e^{-(\beta_0 + \beta_1 x)}}$$

```
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

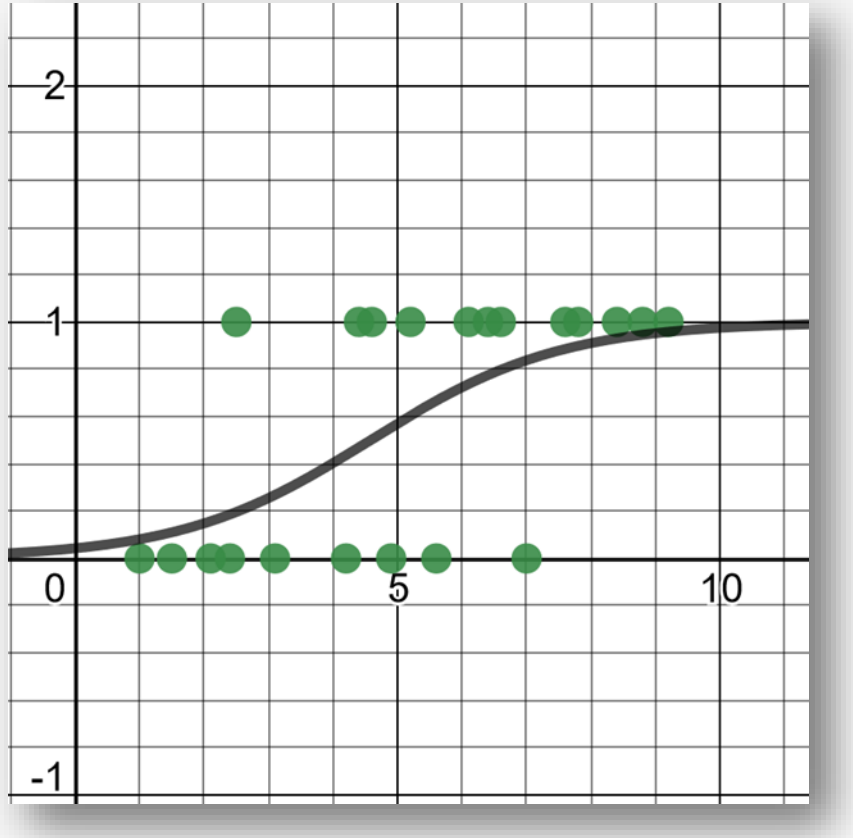
Just Show Me the Math!

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

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

We need to solve for β_0 and β_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 β_0 and β_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 β_0 and β_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 $\boldsymbol{y}$ for each $\boldsymbol{x}$ value, and multiply them together.

2. Get the false (0) data, calculate the probability $(\boldsymbol{1.0 - y})$ for each $\boldsymbol{x}$ 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 coursed.

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 $\boldsymbol{y}$ for each $\boldsymbol{x}$ value, pass it to a $\boldsymbol{log}()$ function, then sum the values.

2. Get the false (0) data, calculate the probability $(\boldsymbol{1.0 - y})$ for each $\boldsymbol{x}$ value, pass it to a $\boldsymbol{log}()$ 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.

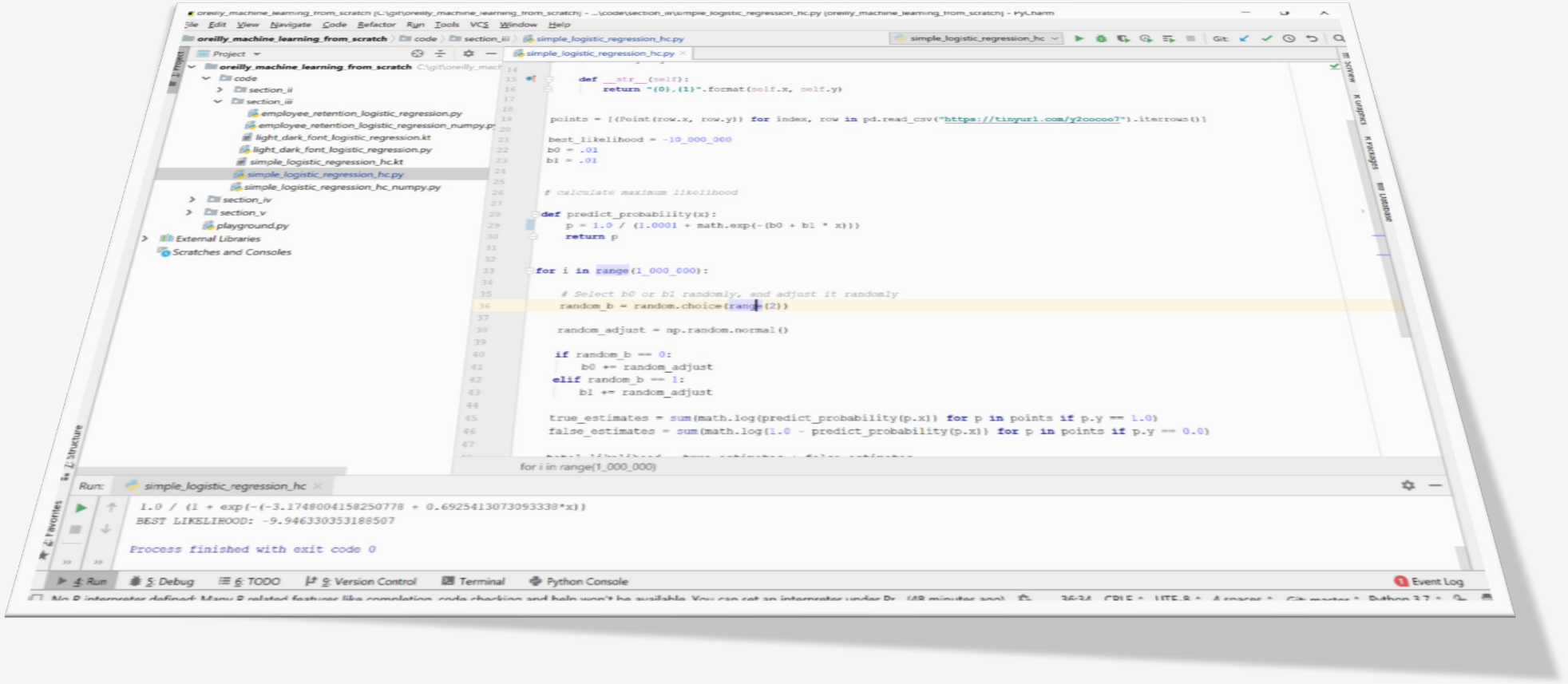
Maximum Likelihood

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

To calculate the maximum likelihood…

1. Randomly adjust the β_0 and β_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 β_0 and β_1 , otherwise revert.
4. Do this for as many iterations as necessary, until the likelihood stops improving.

Hands-On: Logistic Regression



Multivariable Logistic Regression

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

We then solve for those β_x variables the same way as before.

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

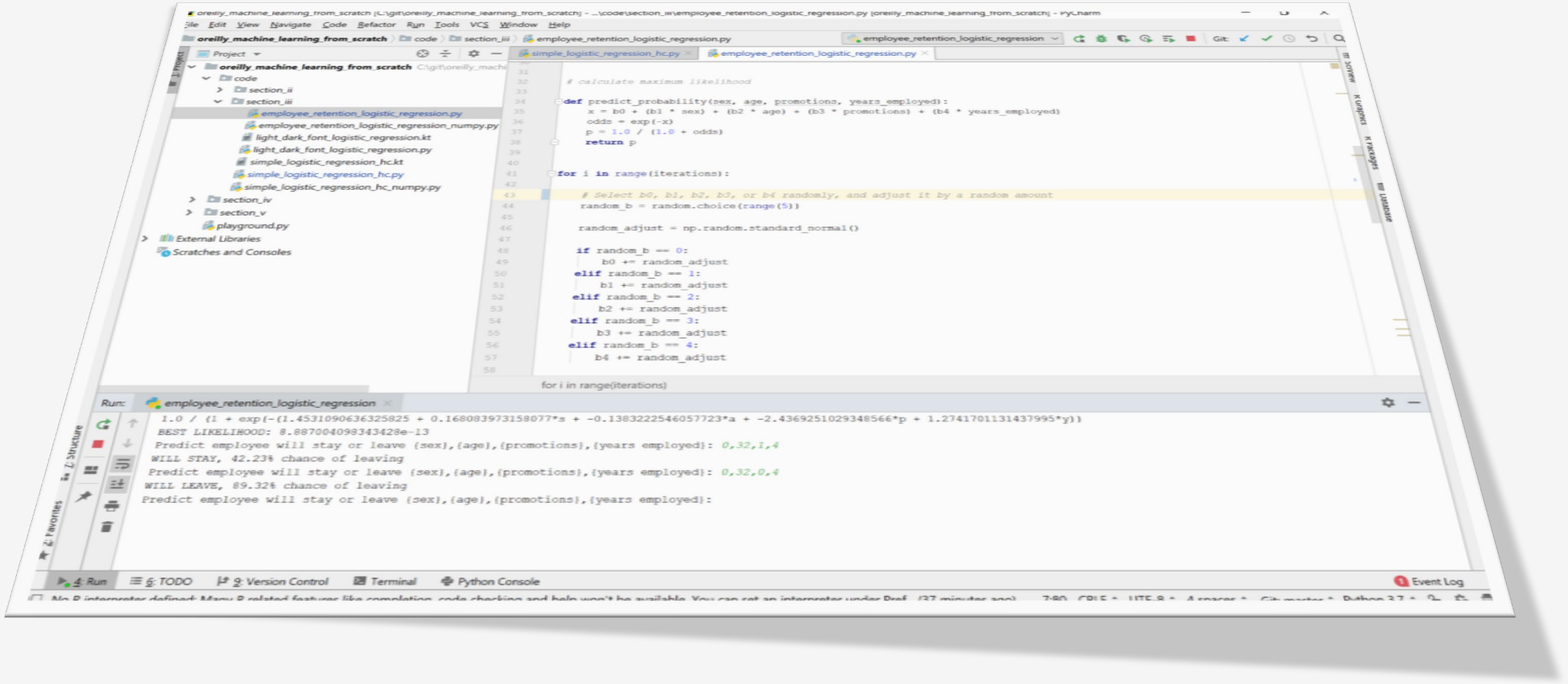
Multivariable Logistic Regression

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



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!

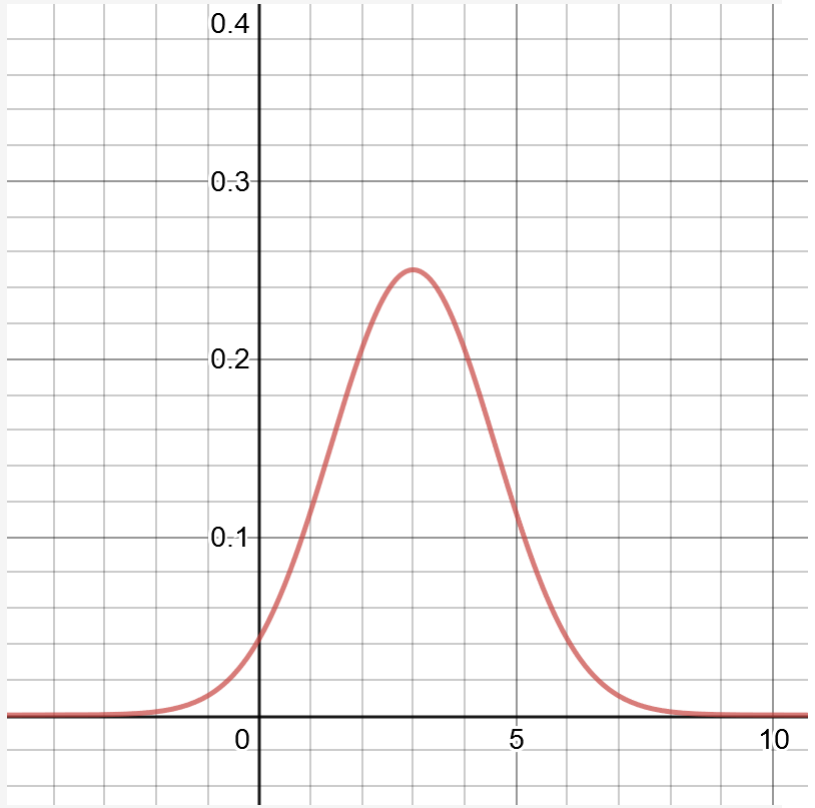
To use logistic regression for more than two categories:

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.

Kind Of Similar – Fitting a Probability Distribution

```python
import random, math

observations = [1.0, 1.0, 1.0, 2.0, 2.0, 3.0, 4.0, 4.0, 5.0, 5.0, 5.0]


def normal_pdf(x: float, mean: float, std_dev: float) -> float:
    return (1.0 / (2.0 * math.pi * std_dev ** 2) ** 0.5) * math.exp(-1.0 * ((x - mean) ** 2 / (2.0 * std_dev ** 2)))


best_likelihood = 0.0
std_dev = 1.0
mean = 1.0

for i in range(100_000):
    adj = random.normalvariate(0, 1)
    selected_variable = random.randint(0, 1)

    if selected_variable == 0:
        mean += adj
    elif selected_variable == 1:
        std_dev += adj

    likelihood = math.exp(sum([math.log(.000000001 + normal_pdf(x, mean, std_dev)) for x in observations]))

    if likelihood > best_likelihood:
        best_likelihood = likelihood
    elif selected_variable == 0:
        mean -= adj
    elif selected_variable == 1:
        std_dev -= adj

print("mean={0}, std_dev={1}".format(mean, std_dev))
```



Quiz Time!

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

1) True

2) False

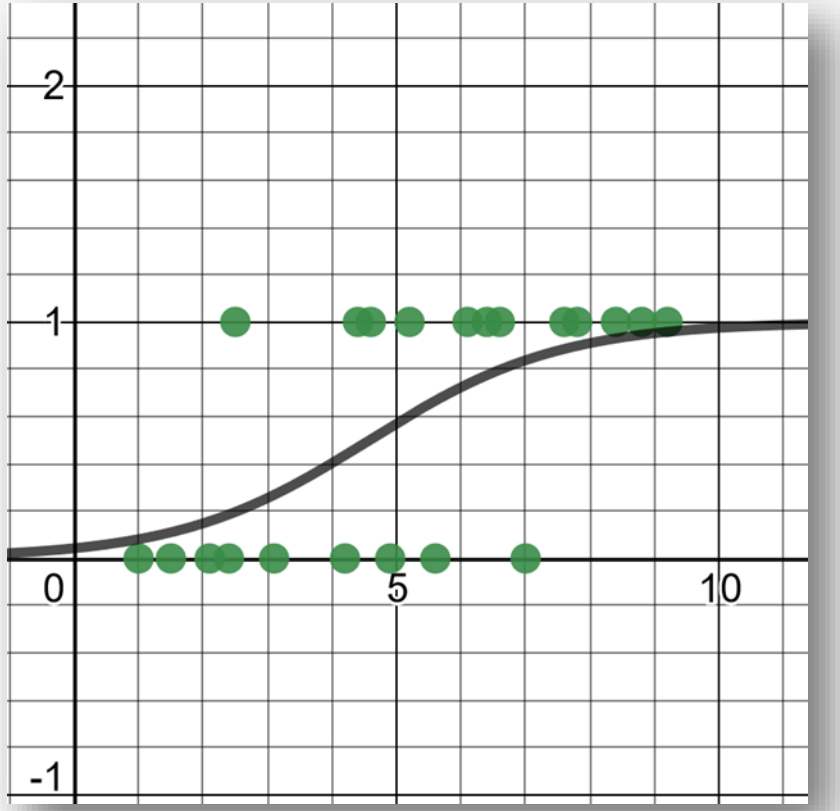
Quiz Time!

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

B) False

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.

Section IV
Naïve Bayes

What is Naïve Bayes?

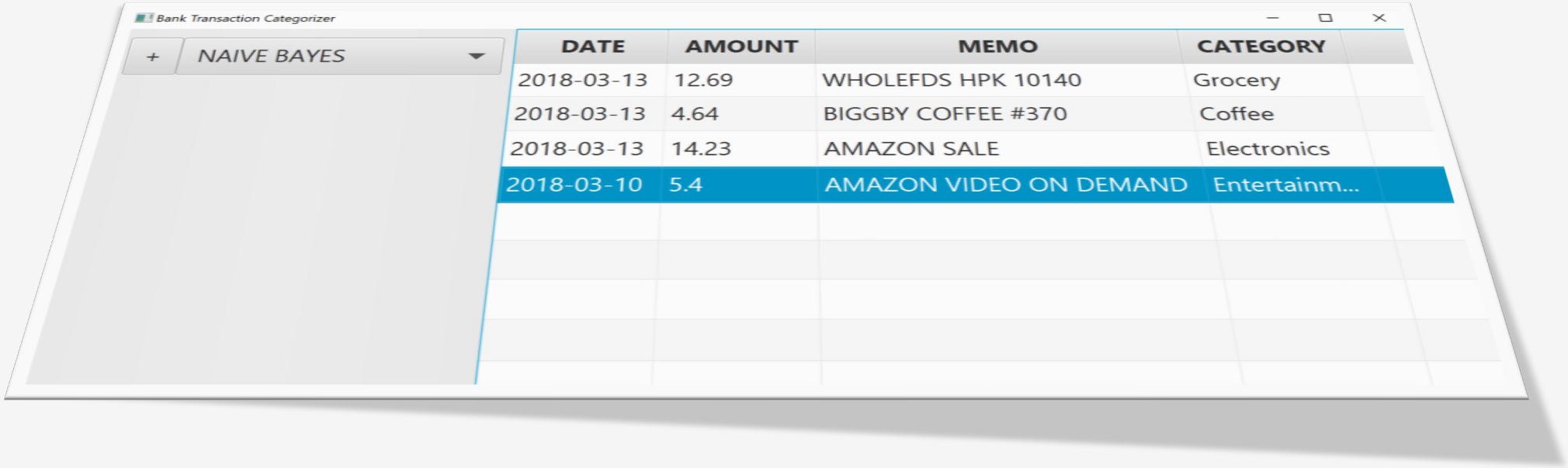
Naïve Bayes is a machine learning application of Bayes Theorem that merges probabilities of multiple features to predict a given category.

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

- Naïve Bayes is effective because it learns quickly, even with little data.
- Naive 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:

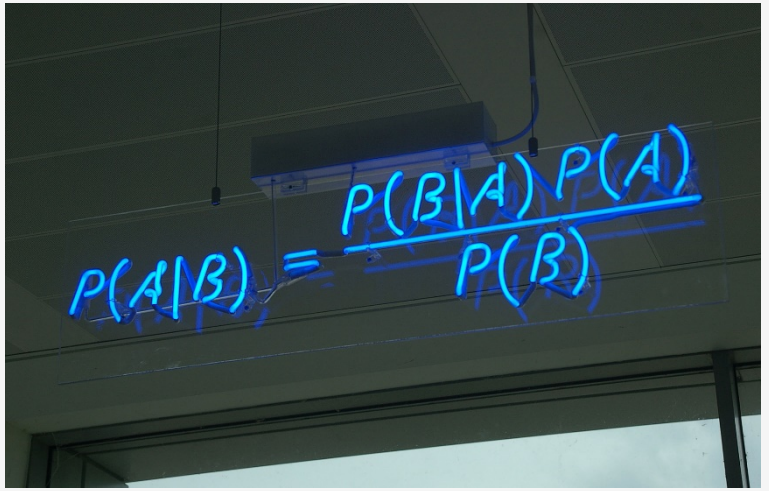
1) For a given category (like ***spam, grocery,*** etc), combine the probabilities of each feature ***occurring*** and ***not occurring*** by multiplying them.

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

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

2) Combine the probabilities above with this division operation:

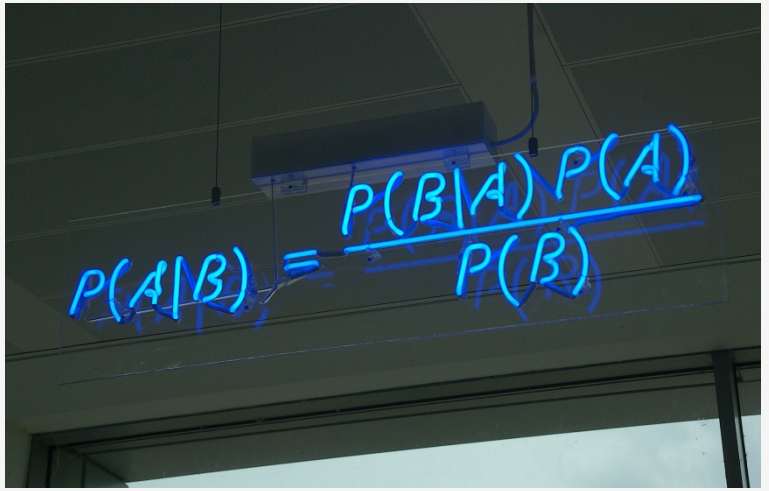
$$Combined\ Probability = \frac{(Occur\ Product)}{(Occur\ Product) + (Not\ Occur\ Product)}$$



How to Build a Naïve Bayes Text Classifier

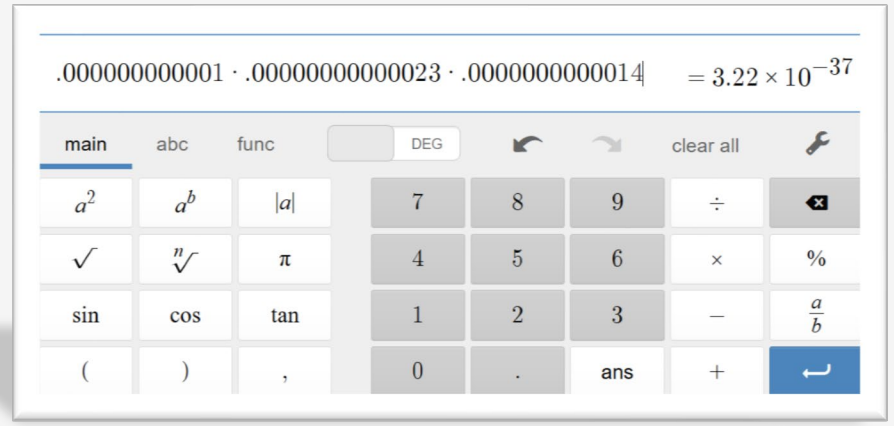
3) Calculate the combined probability for every category, and the one that is the highest is the category you predict.

Easy right? But there is one complication…



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

$$P_{fx} = Probability\ of\ feature\ \text{x}$$

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

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

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

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.

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

Final Naïve Bayes Formulation

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

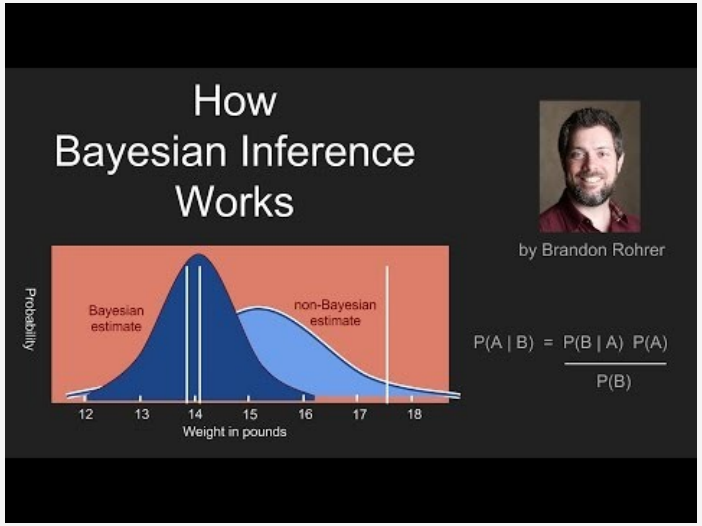
$$Occur\ Product = exp(log(P_{f1}) + log(P_{f2}) + log(P_{f3}) + log(P_{fn}))$$

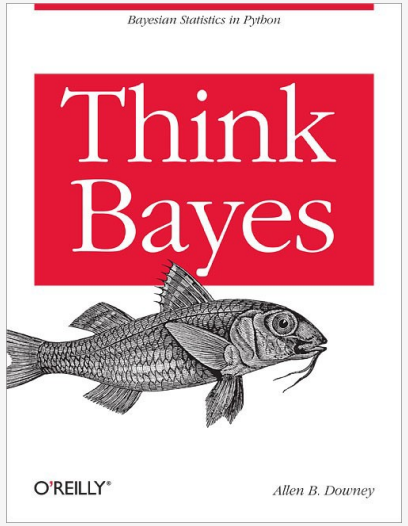
$$Not\ Occur\ Product = \exp(log(1 - P_{f1}) + log(1 - P_{f2}) + log(1 - P_{f3}) + \ldots + log(1 - P_{fn}))$$

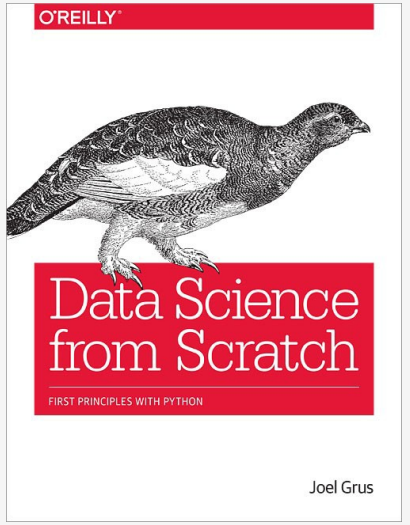
$$Combined\ Probability = \frac{(Occur\ Product)}{(Occur\ Product) + (Not\ Occur\ Product)}$$

Learn More about Bayesian Modeling

Brandon Rohrer - YouTube







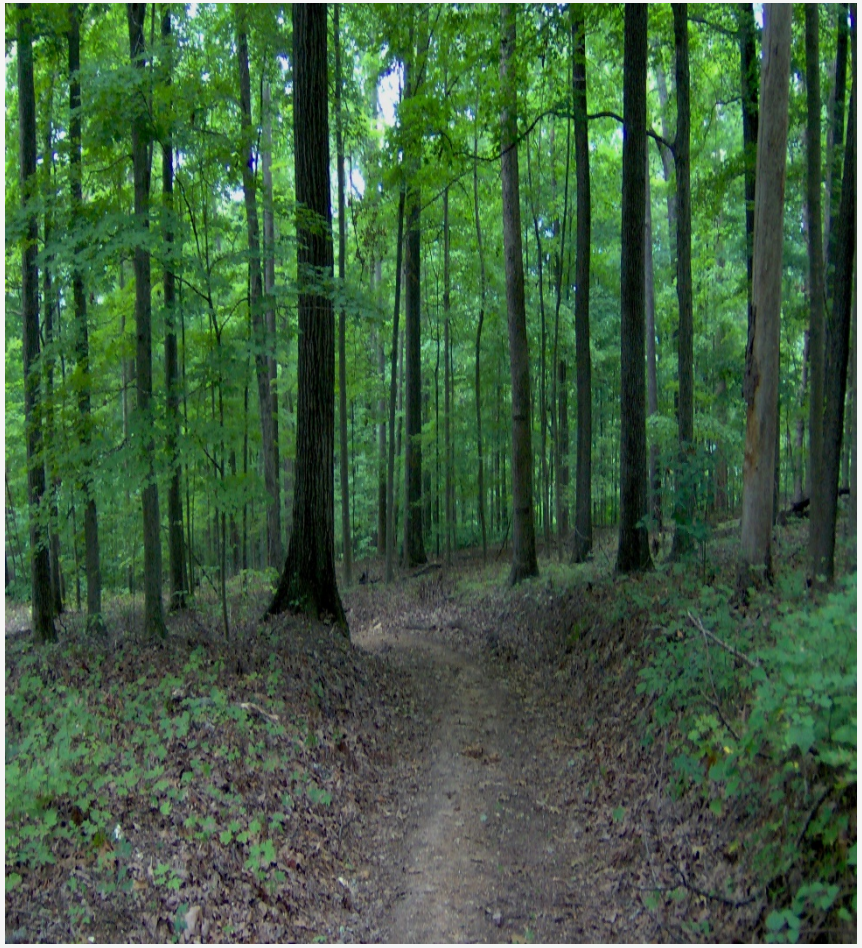
Section V
Decision Trees and Random Forests

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
0/16	14/20

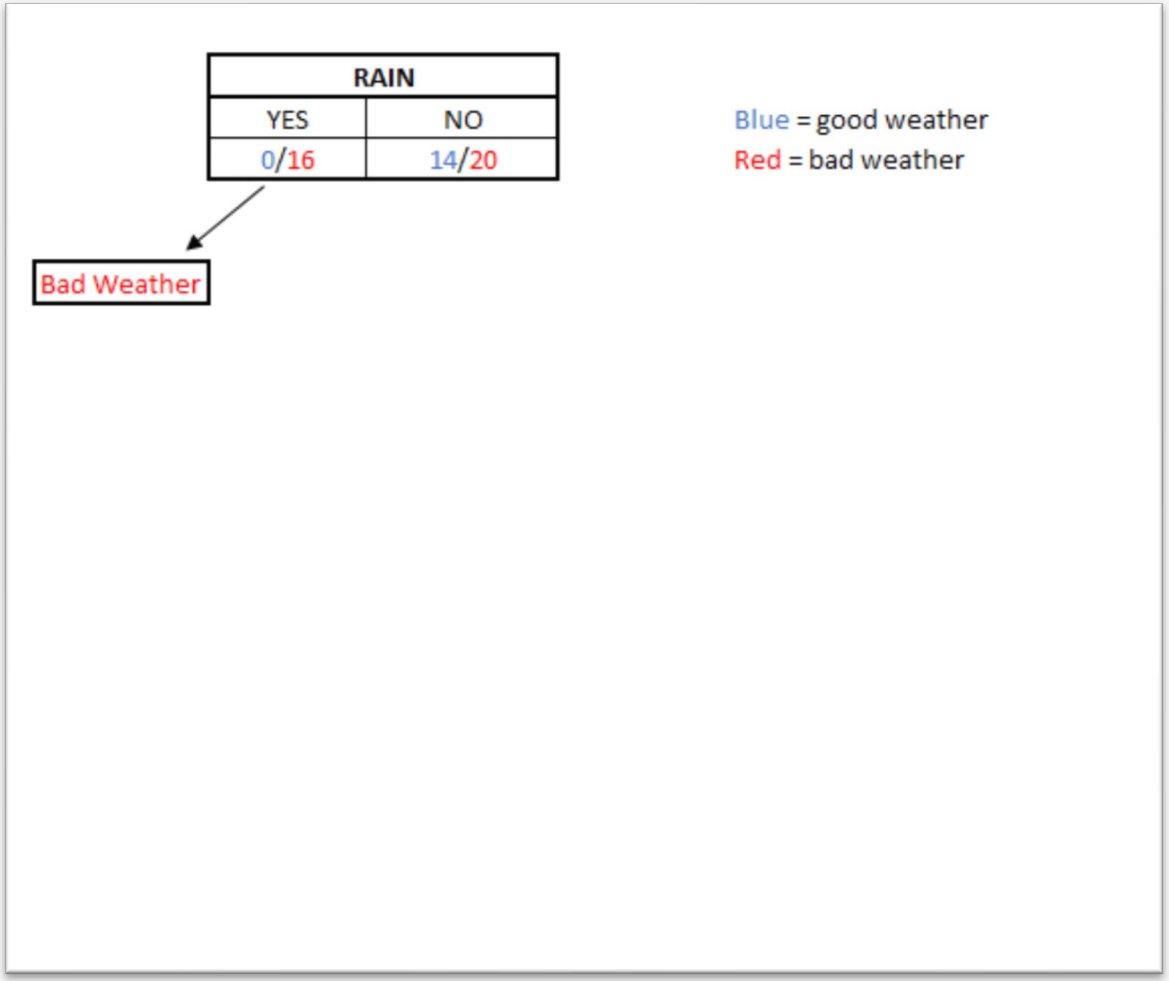
Blue = good weather

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.

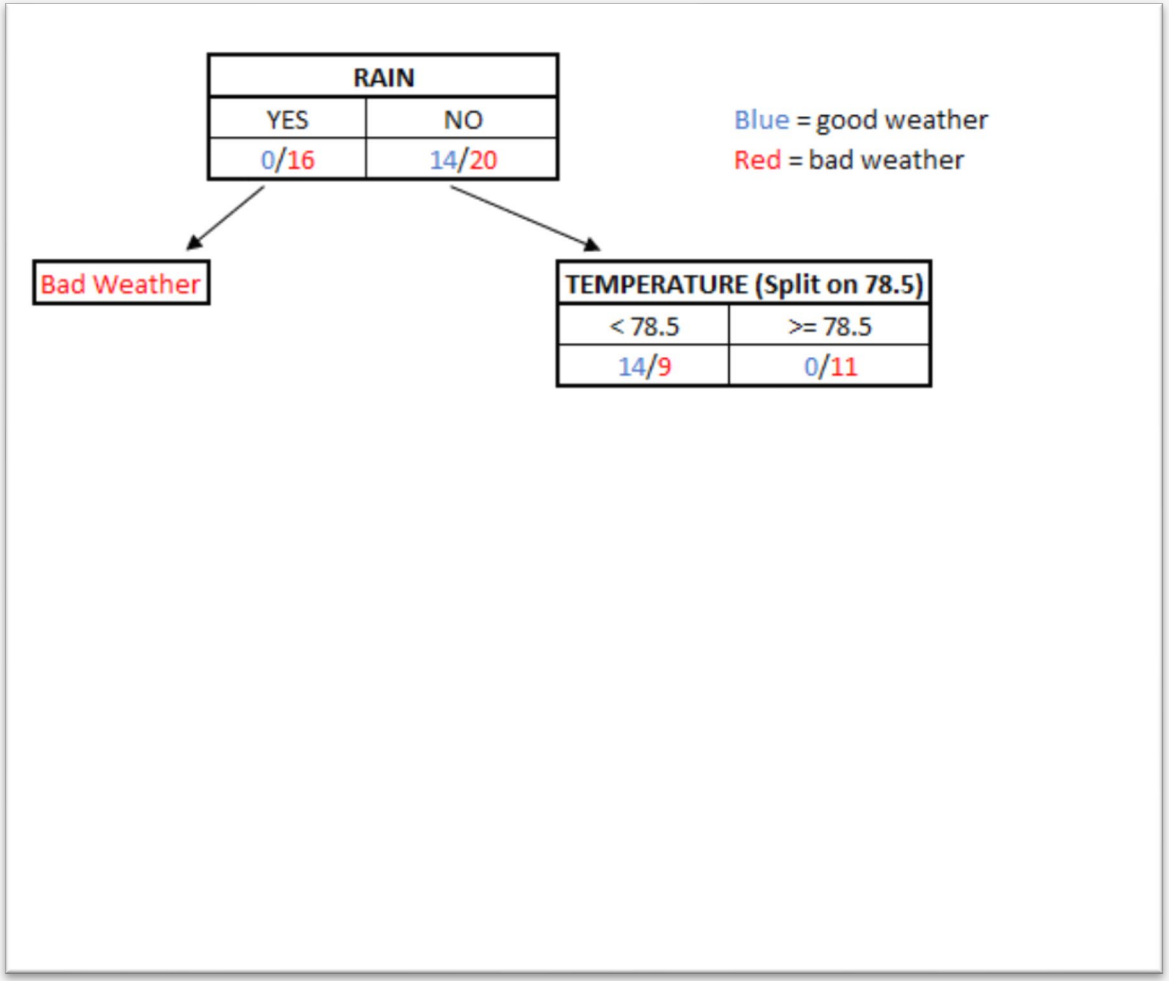
However, records that did not have ***RAIN*** are mixed, where some are good weather, and others are bad weather. These will need to be split again.



Decision Tree Intuition

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

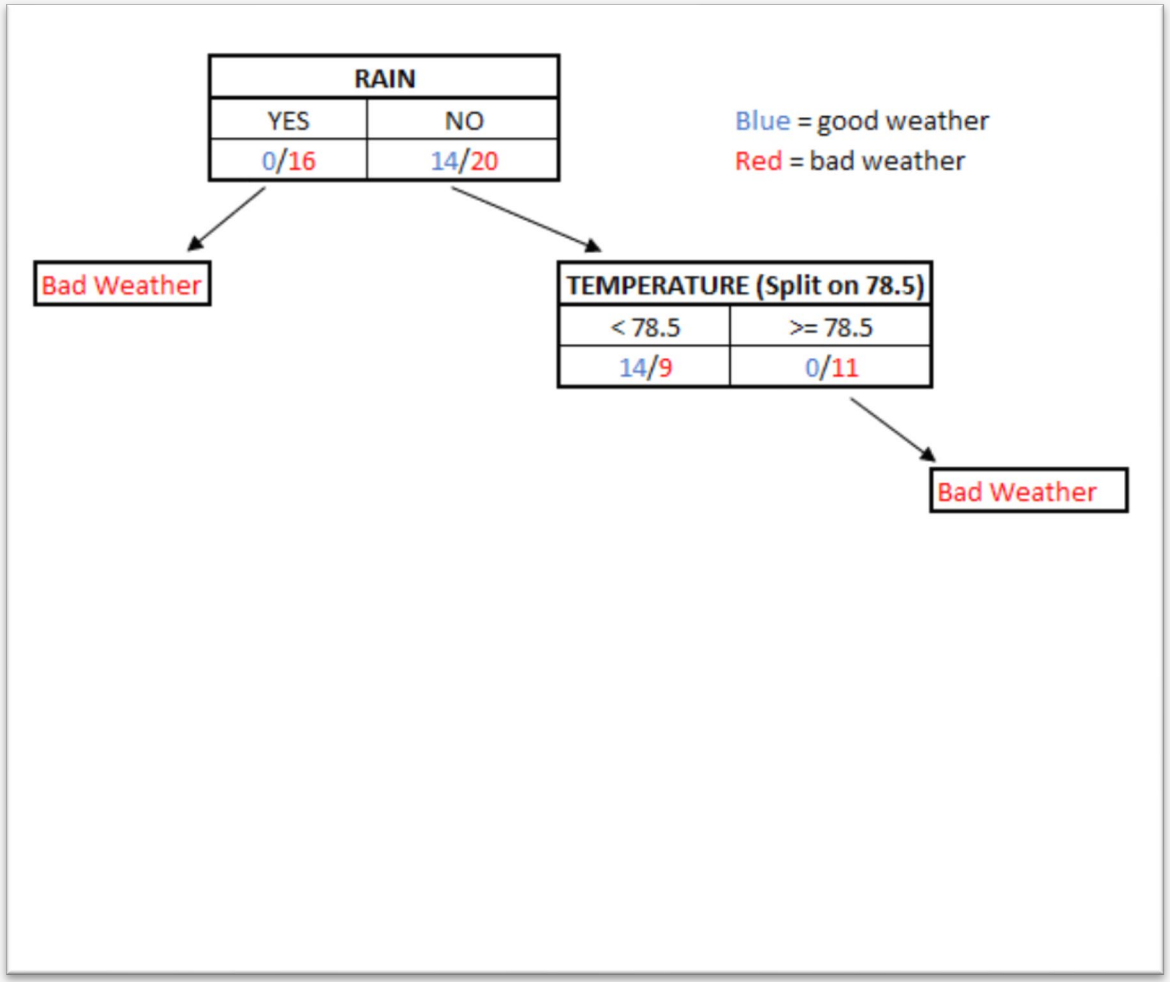
Since ***TEMPERATURE*** is continuous and not a simple *yes/no* binary, we split on a value 78.5 that optimizes the split (more on this later).



Decision Tree Intuition

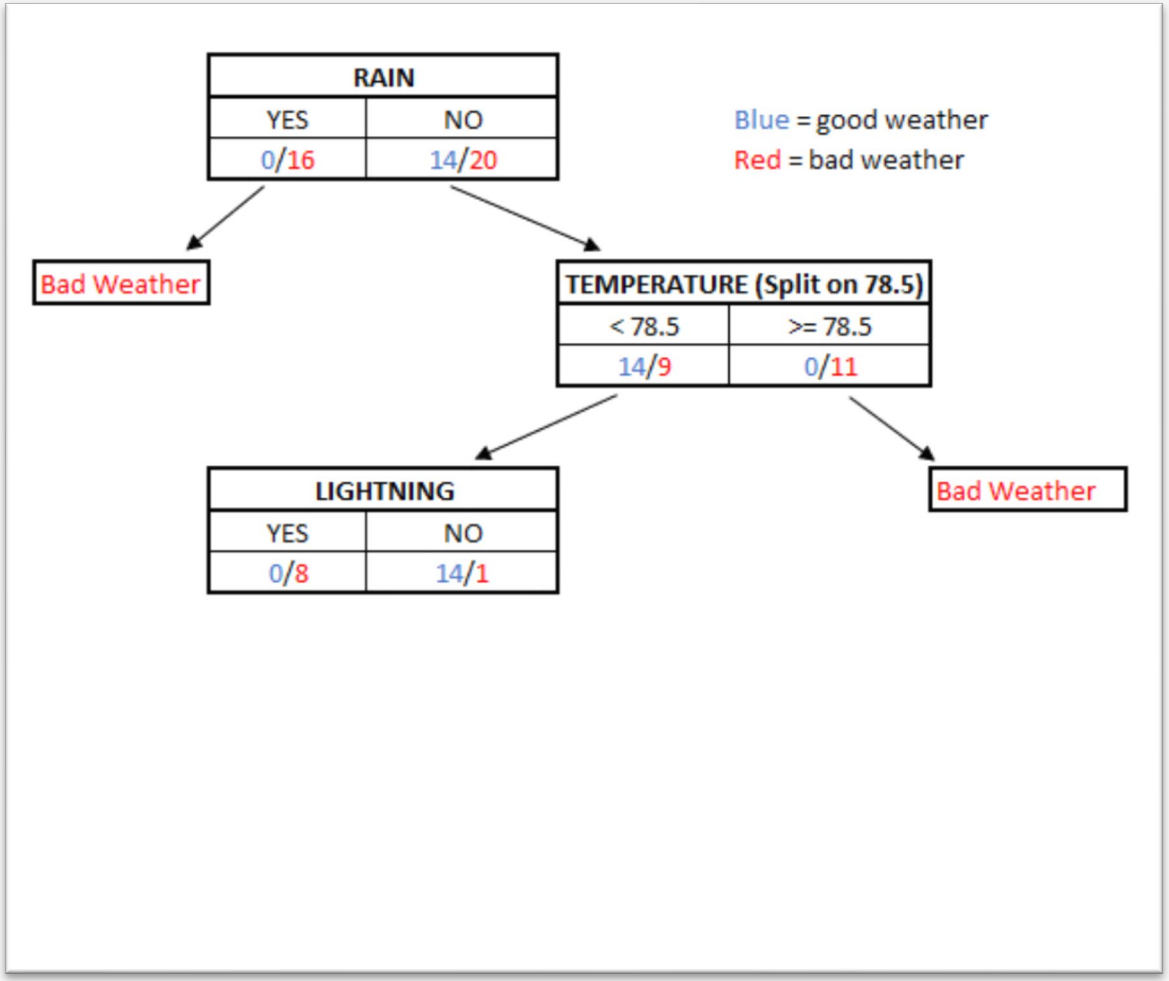
Notice how all records with ***TEMPERATURE*** greater than/equal to 78.5 are bad weather, so we will predict any records with a ***TEMPERATURE*** at least 78.5 as bad weather.

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

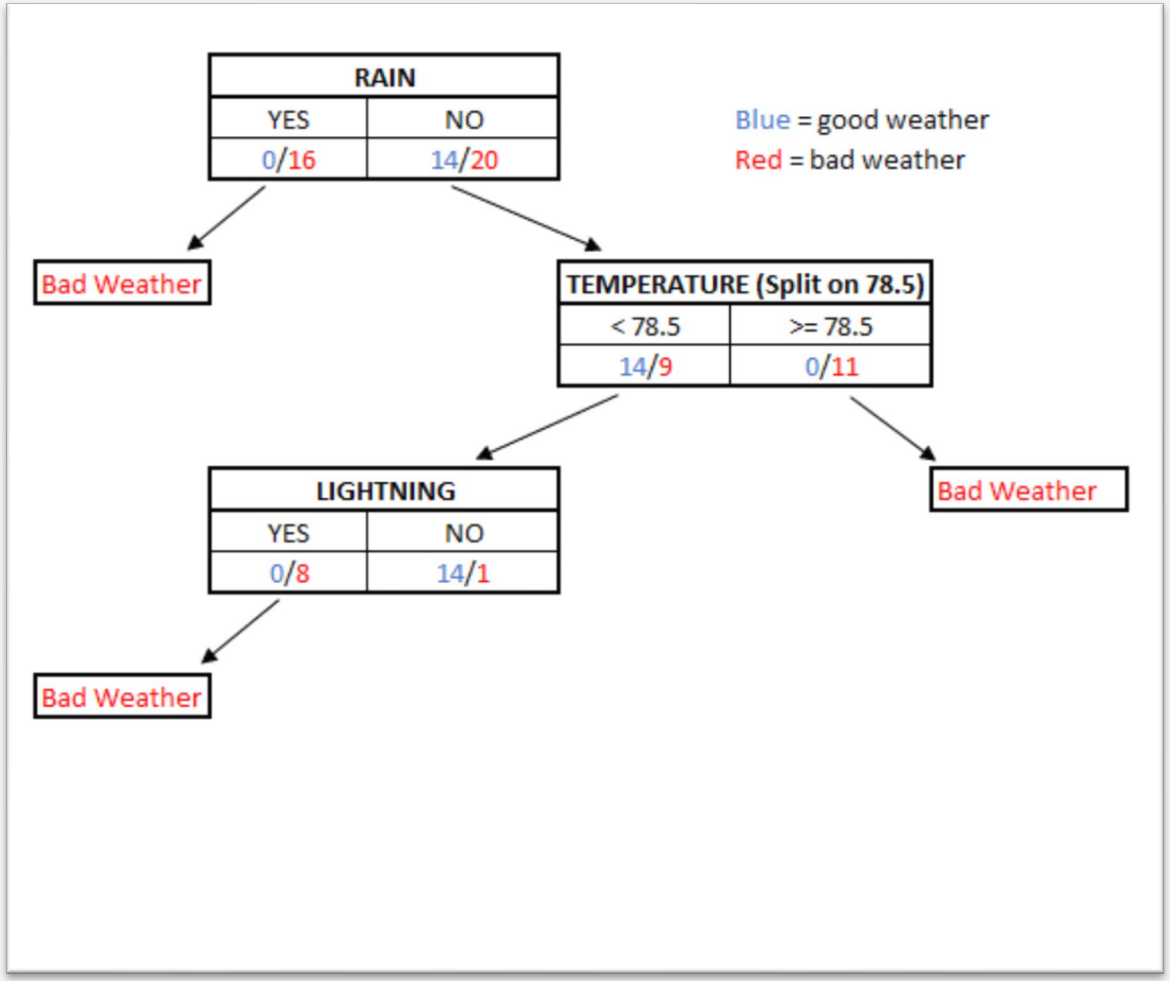
We take those records with ***TEMPERATURE*** less than 78.5 and now split them on whether ***LIGHTNING*** was present.



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.

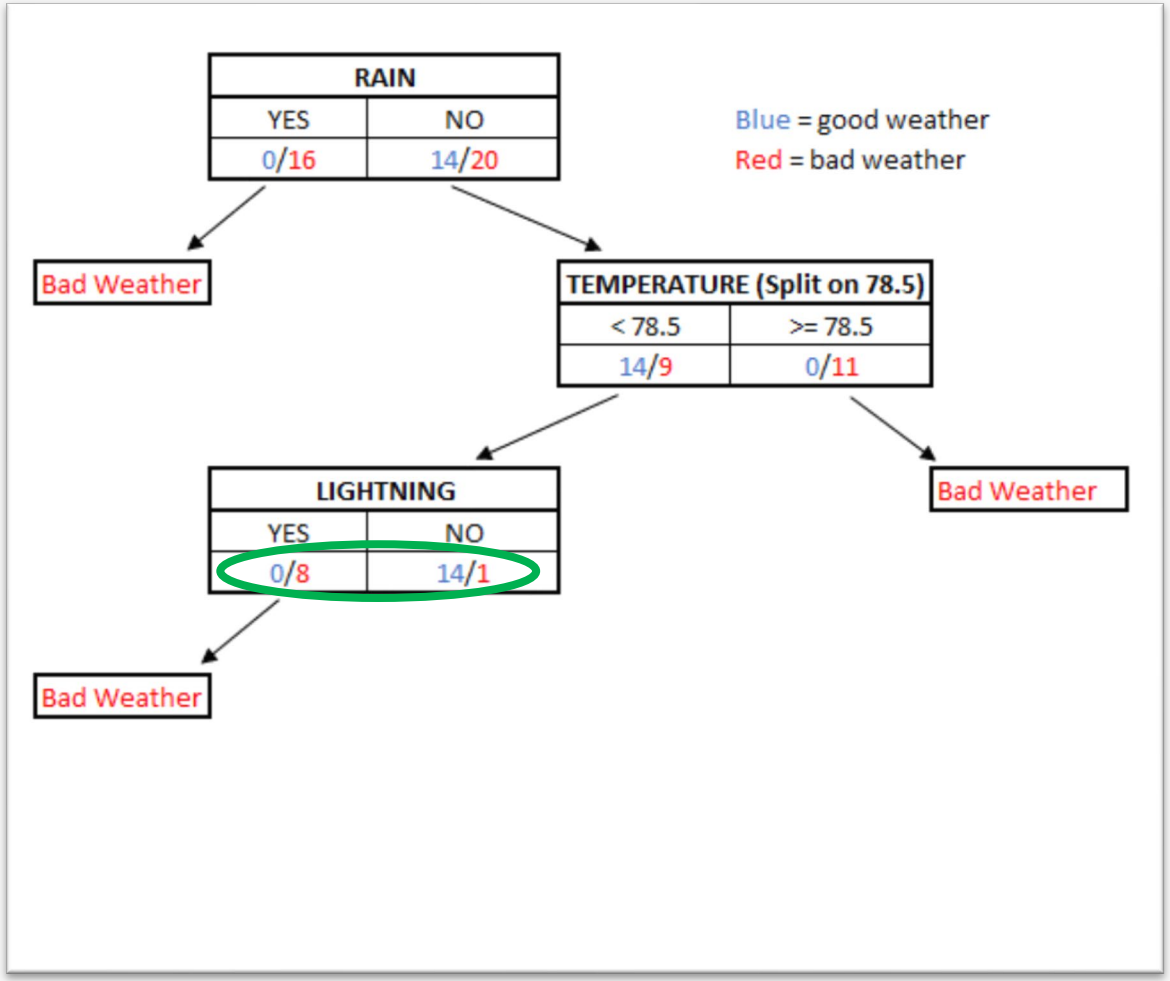


Decision Tree Intuition

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

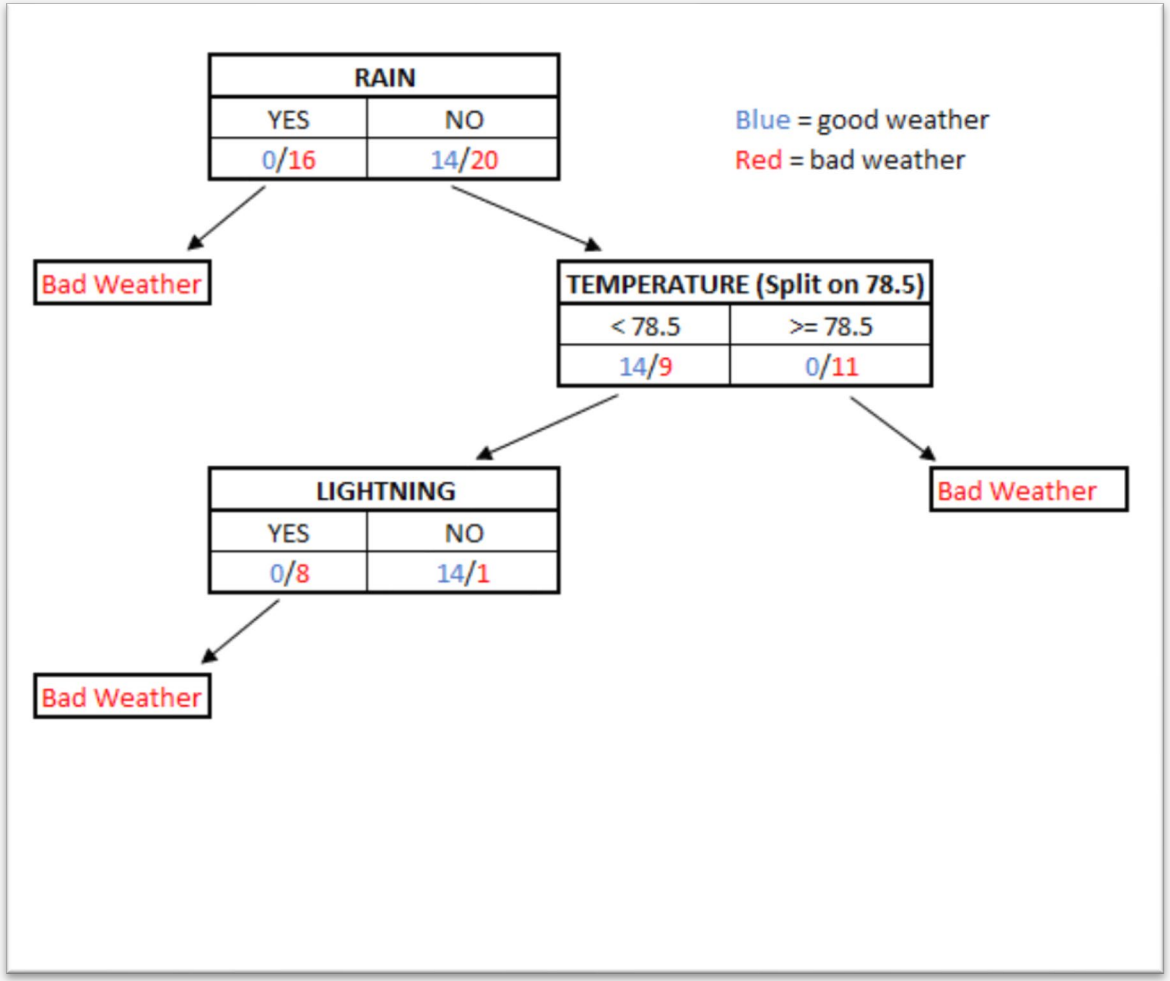
Notice how all records with ***LIGHTNING*** are labelled as bad weather, so we can predict any new record with ***LIGHTNING*** as bad weather.

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



Decision Tree Intuition

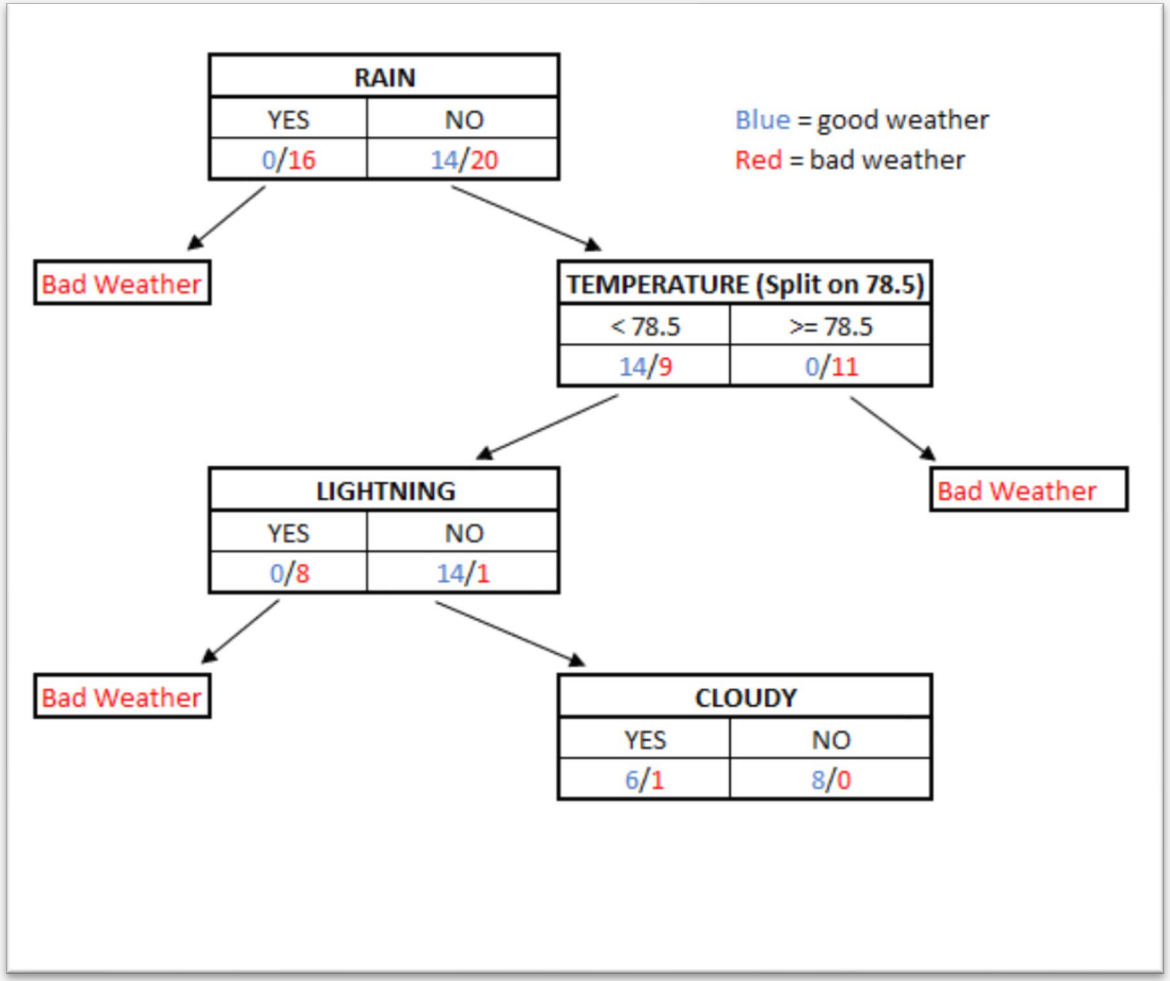
LIGHTNING almost gave us a perfect split, where lightning being present would predict bad weather and no lightning *almost* always meant good weather.



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.

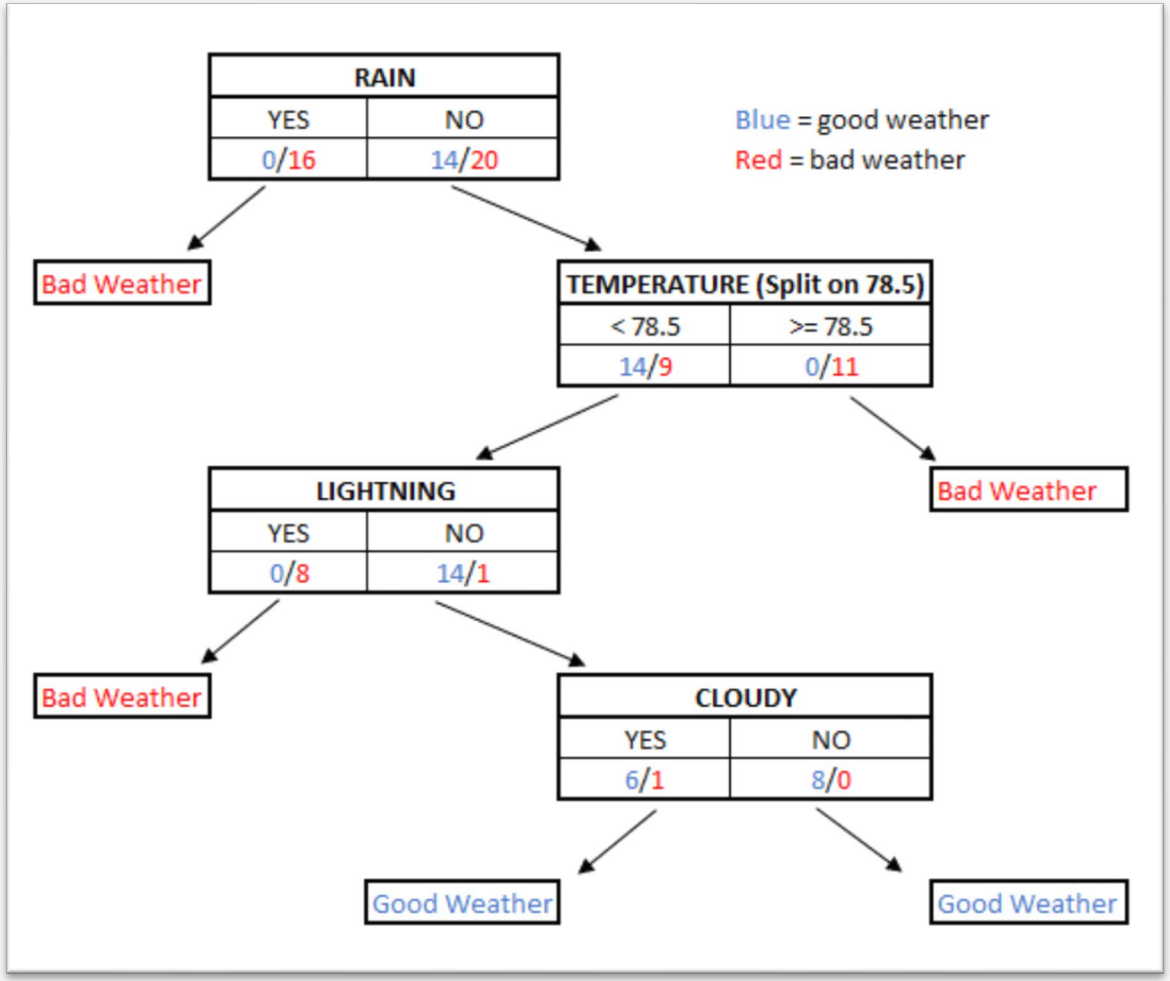


Decision Tree Intuition

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…

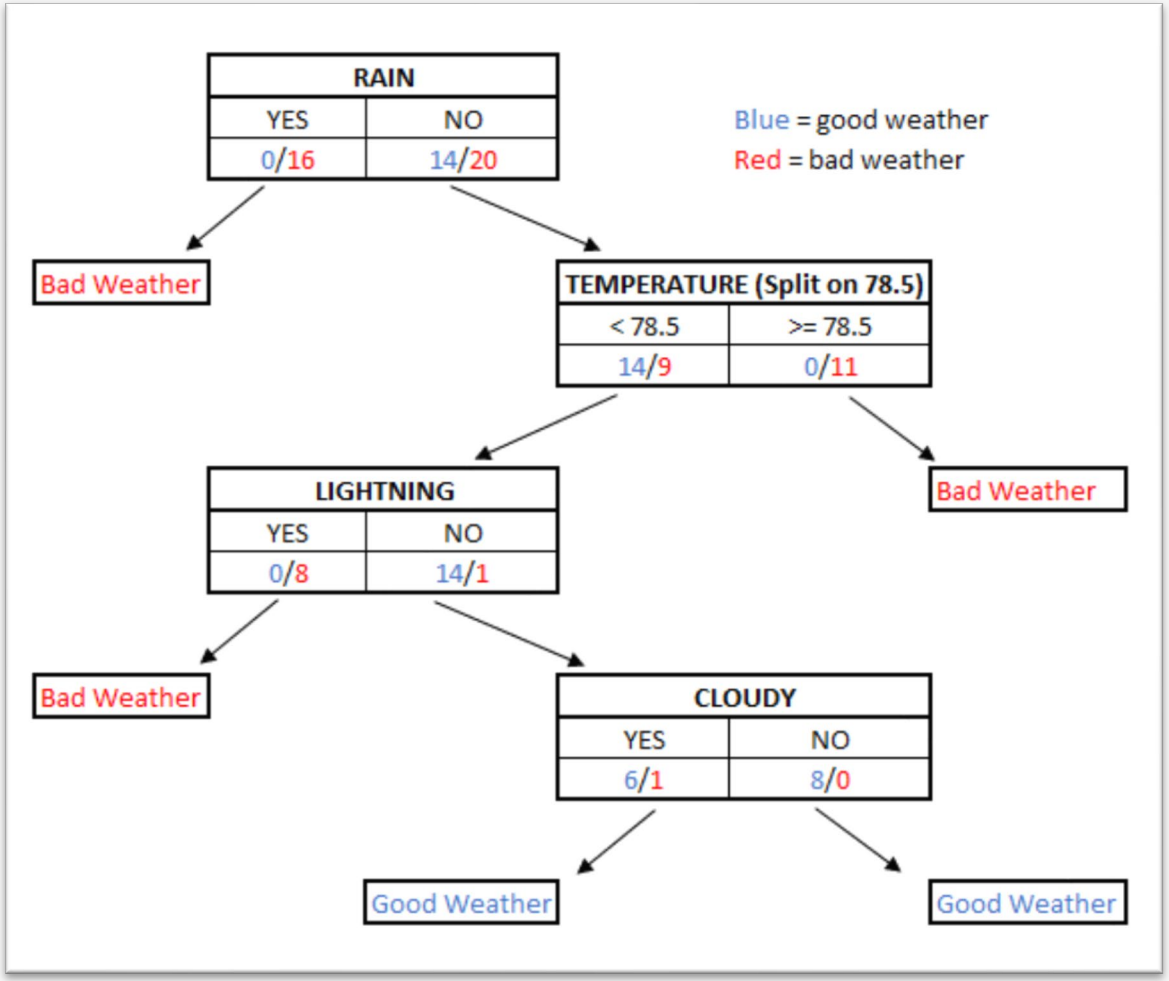


Decision Trees – Getting to the "Decision" Part

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*):

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

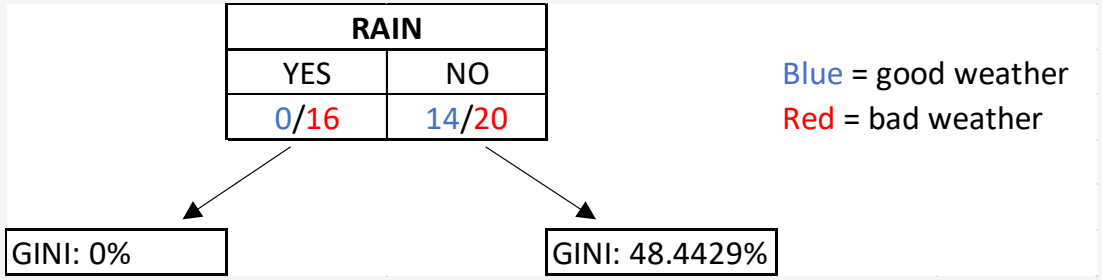
So the Gini impurity of dogs versus cats in the kennel:

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

Gini Impurity

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.

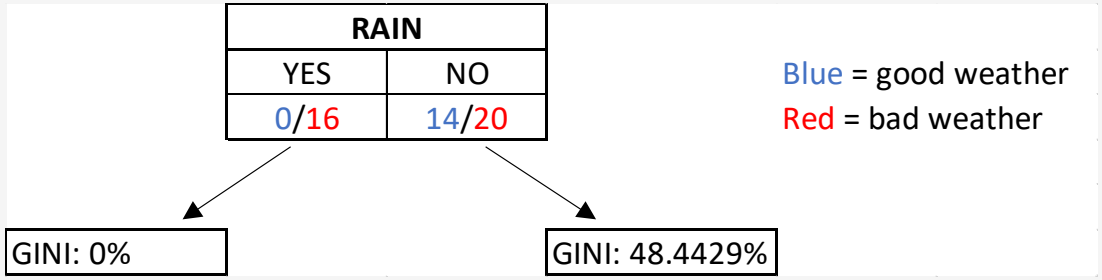


$$1 - \left(\frac{0}{0+16}\right)^2 - \left(\frac{16}{0+16}\right)^2 = 0$$

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.



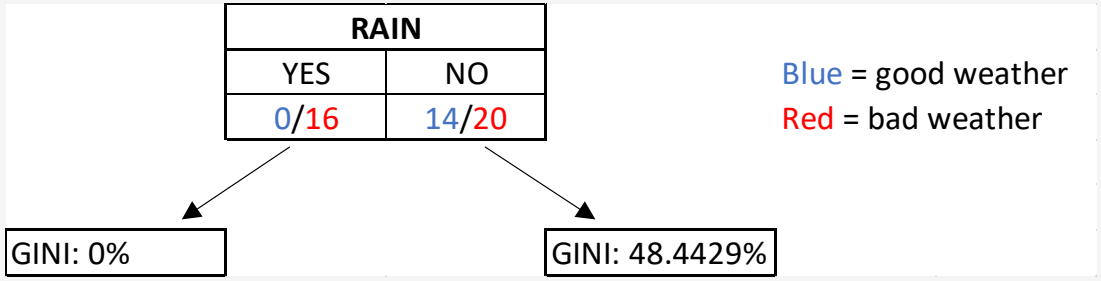
$$1 - \left(\frac{0}{0+16}\right)^2 - \left(\frac{16}{0+16}\right)^2 = 0$$

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!*

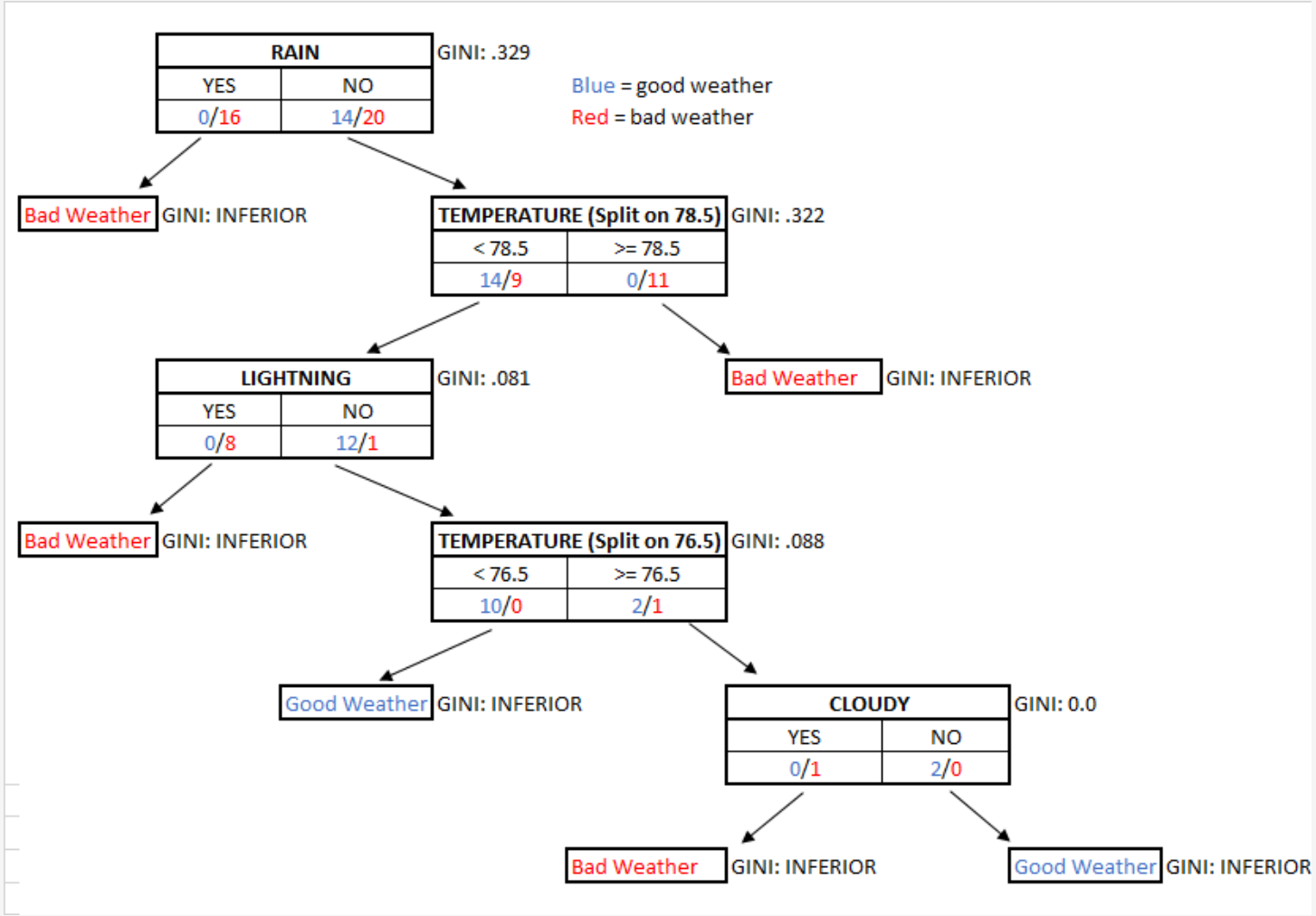
At every step we choose the property that provides the least weighted average Gini impurity.

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.

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

TEMERATURE
74
76
77
80
81
83
84

TEMPERATURE (Split on 78.5)	
< 78.5	>= 78.5
14/9	0/11

How to Split Continuous Variables?

Continuous variables extend the GINI concept further.

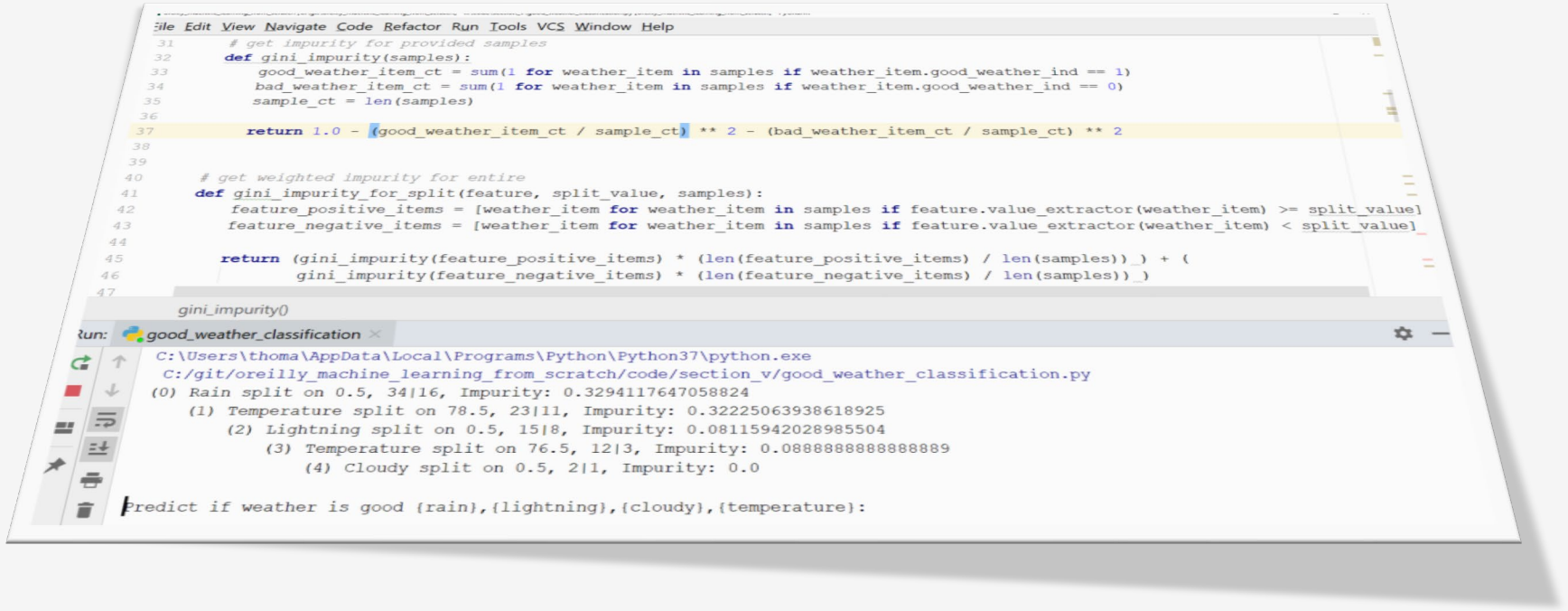
When you evaluate a continuous variable for its GINI, first sort the data for that variable then produce the rolling 2-value averages, each of which is a candidate for the split value.

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.

TEMERATURE	ROLLING 2-VALUE AVG
74	
76	75
77	76.5
80	**78.5**
81	80.5
83	82
84	83.5

TEMPERATURE (Split on 78.5)	
< 78.5	>= 78.5
14/9	0/11

Hands On: Decision Trees



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.

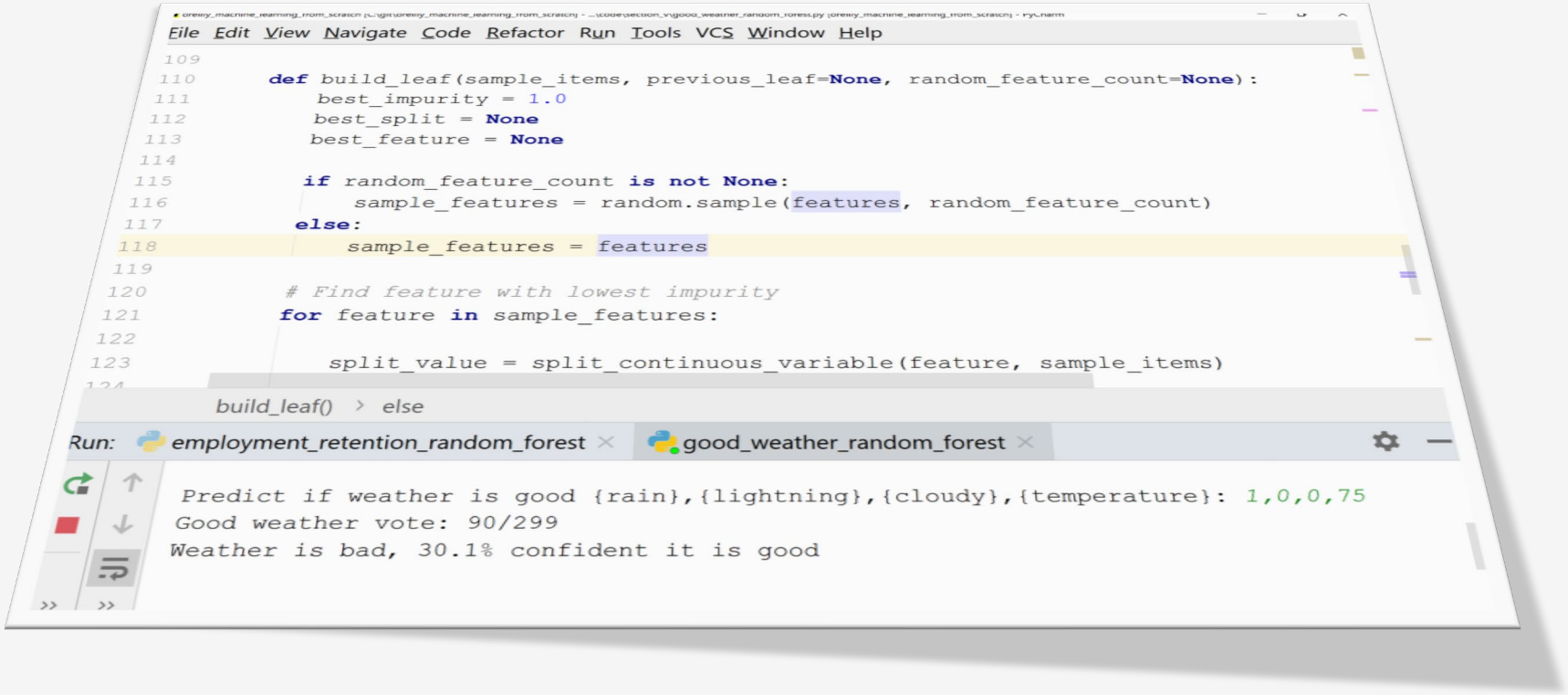
Fine-Tuning Random Forests

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



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.

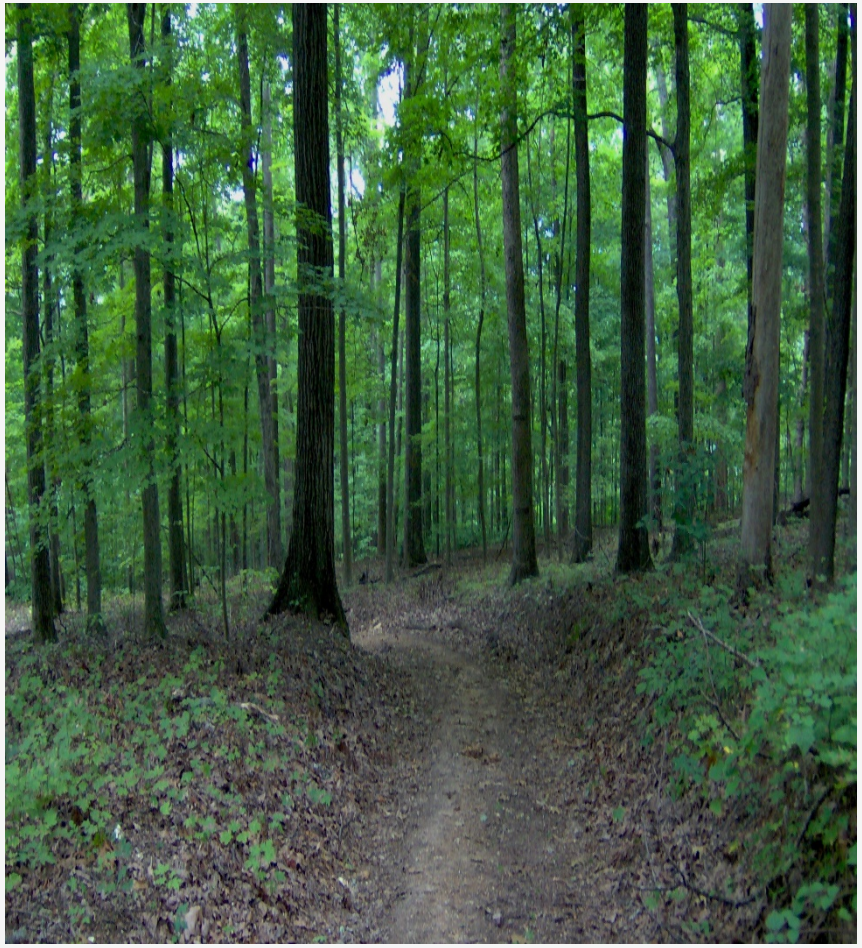
If I were to take one machine learning algorithm with me, it would be decision trees and all of its variants.

Quiz Time!

Decision trees are effective because they do not overfit.

A) True

B) False



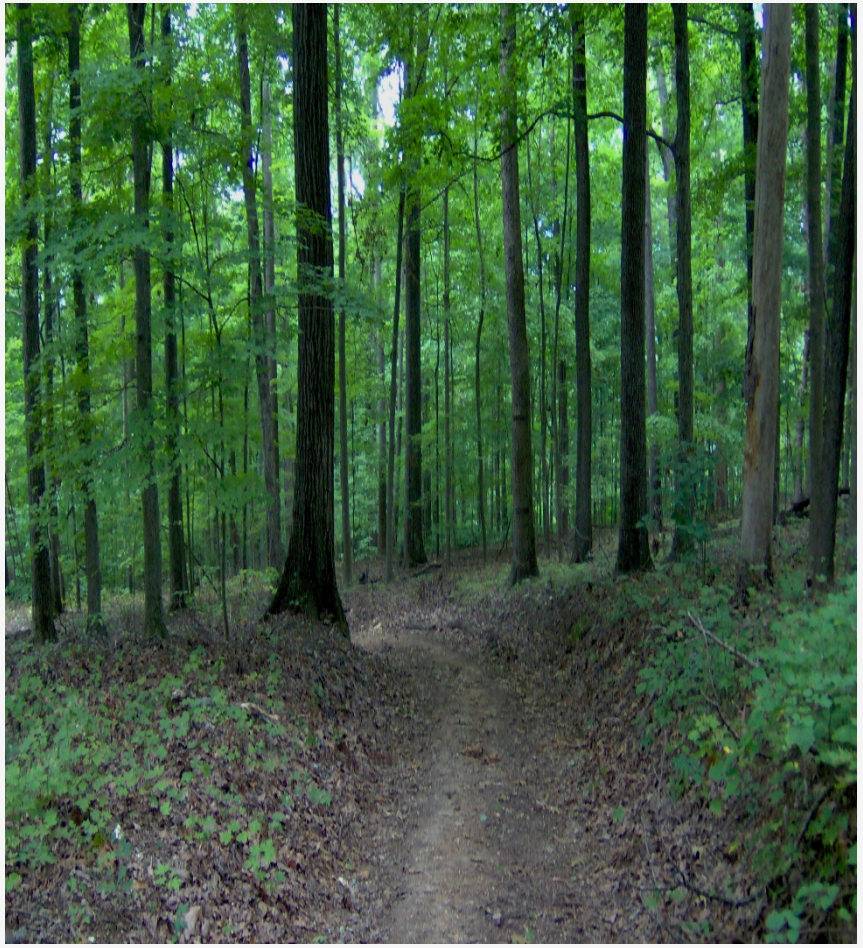
Quiz Time!

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?

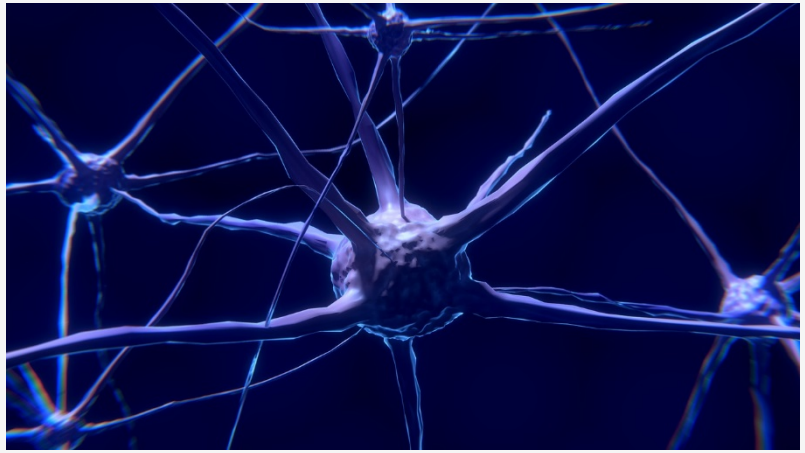
A) .50 B) .2537 C) .4506 D) .45

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

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.

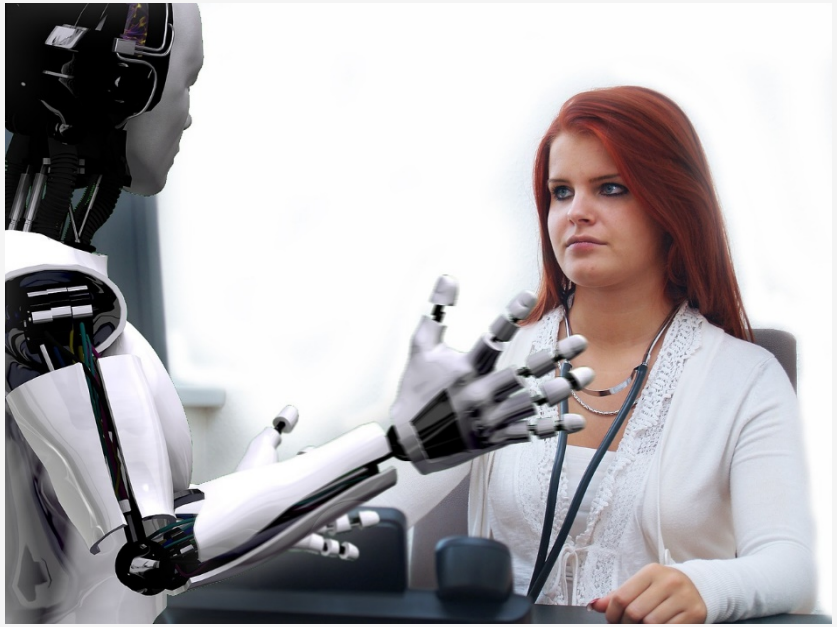
Neural networks are utilized to employ **deep learning**, which utilizes middle layers hence the name “deep”.

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.

The Problem

Suppose we wanted to take a background color (in RGB values) and predict a light/dark font for it.

Hello Hello

If you search around Stack Overflow, there is a nice formula to do this:

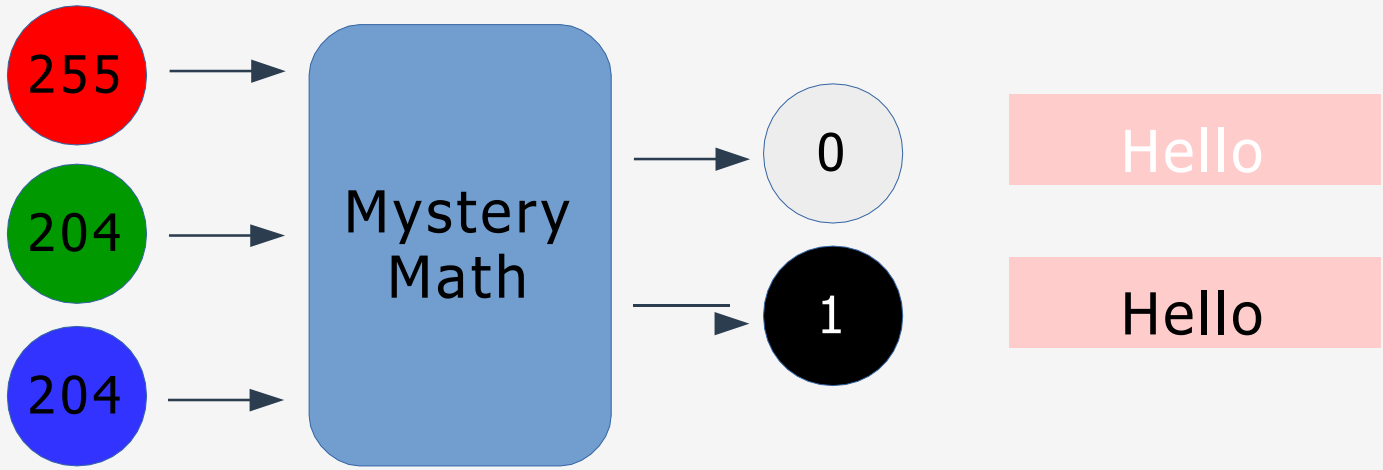
$$L = (.299R + .587G + .114B)/255$$

If you are unsure of the underlying model, you could solve this with logistic regression or decisions trees.

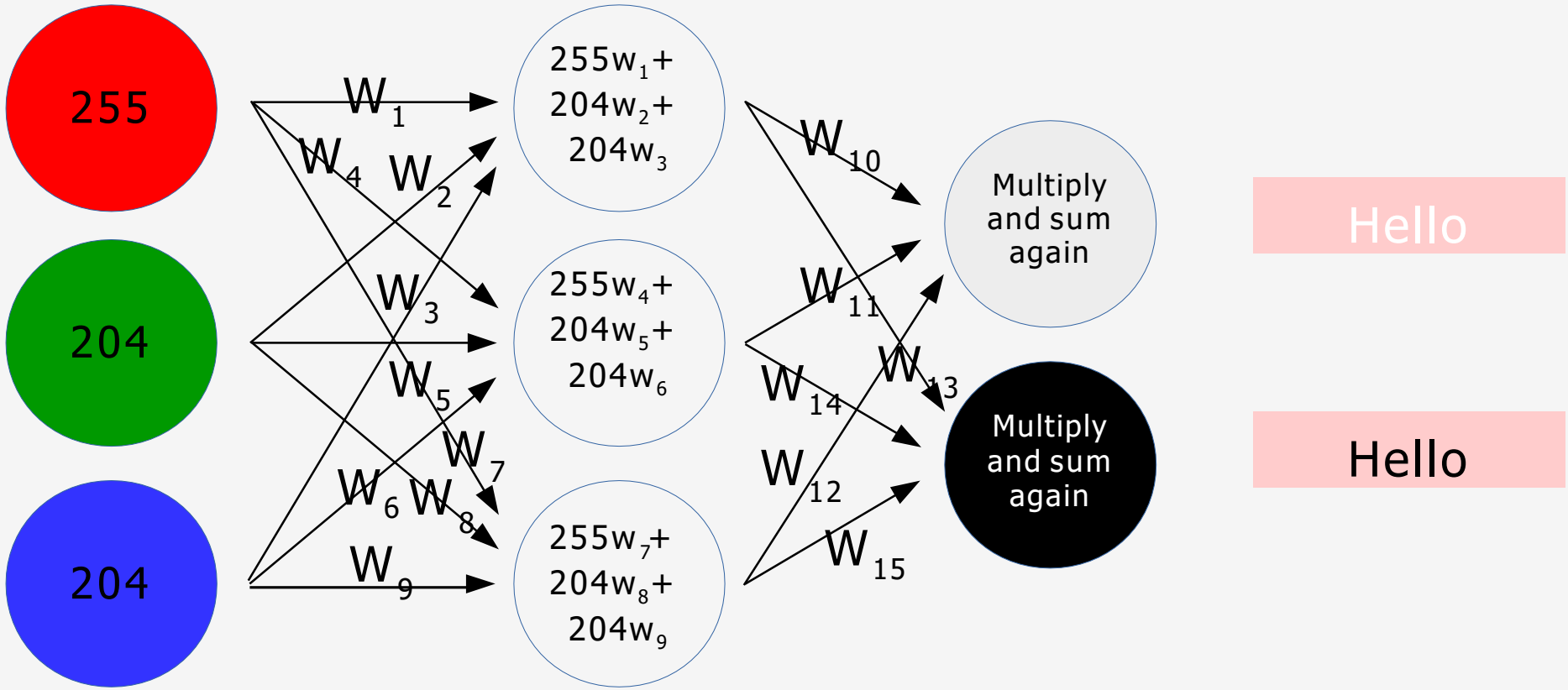
But let's use a neural network.

A Simple Neural Network

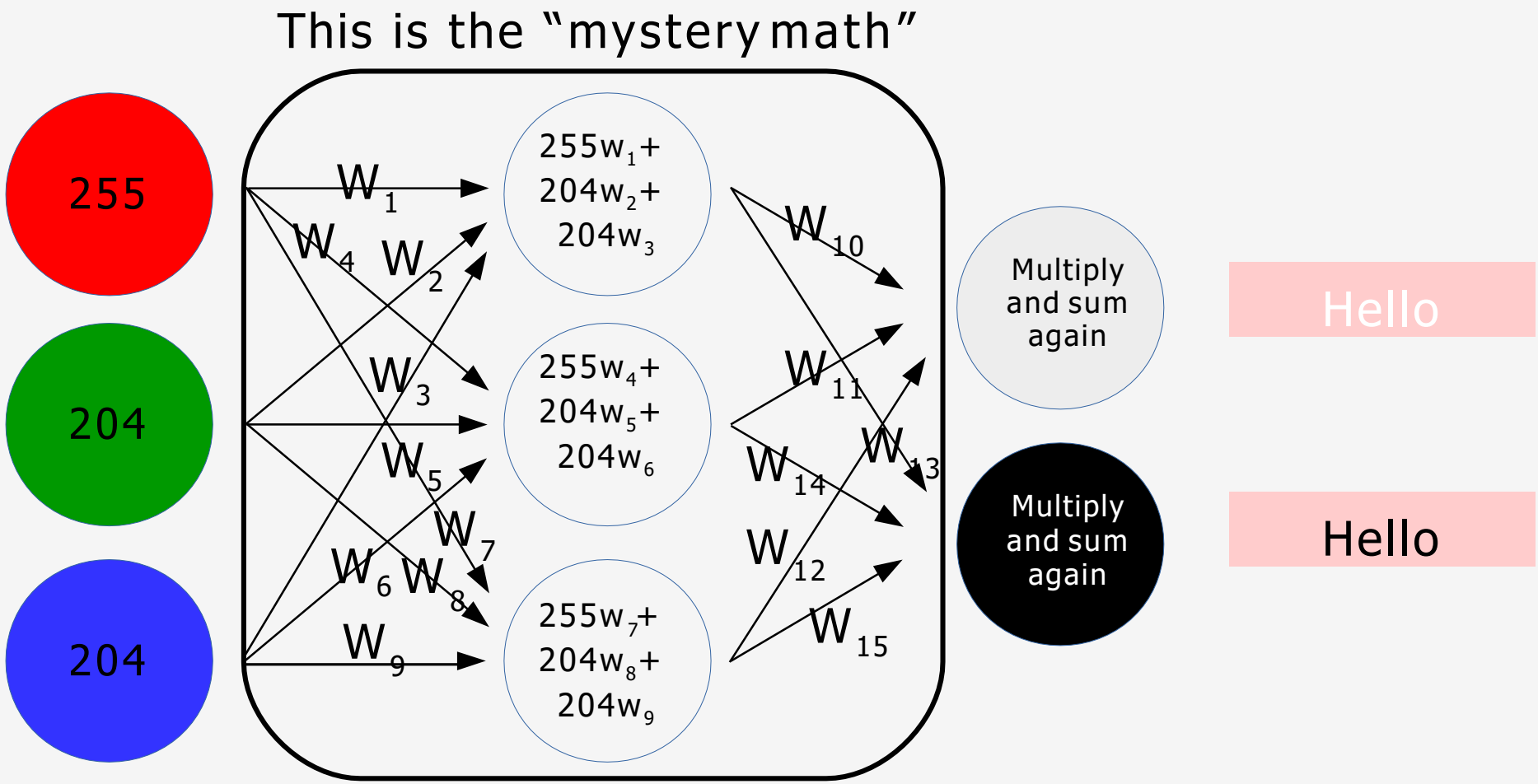
Let's represent background color as 3 numeric RGB inputs, and predict whether a DARK/LIGHT font should be used.



A Simple Neural Network

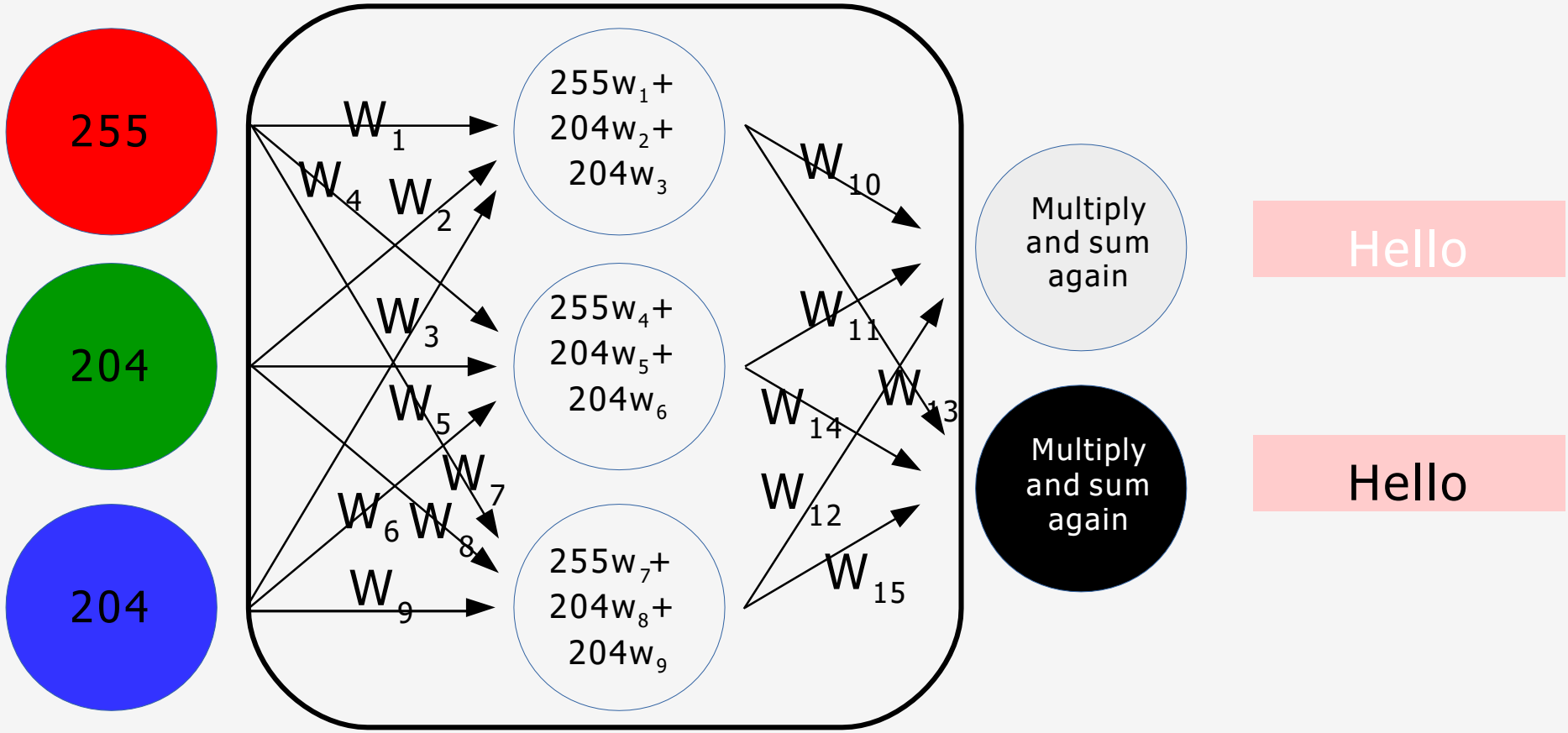


A Simple Neural Network



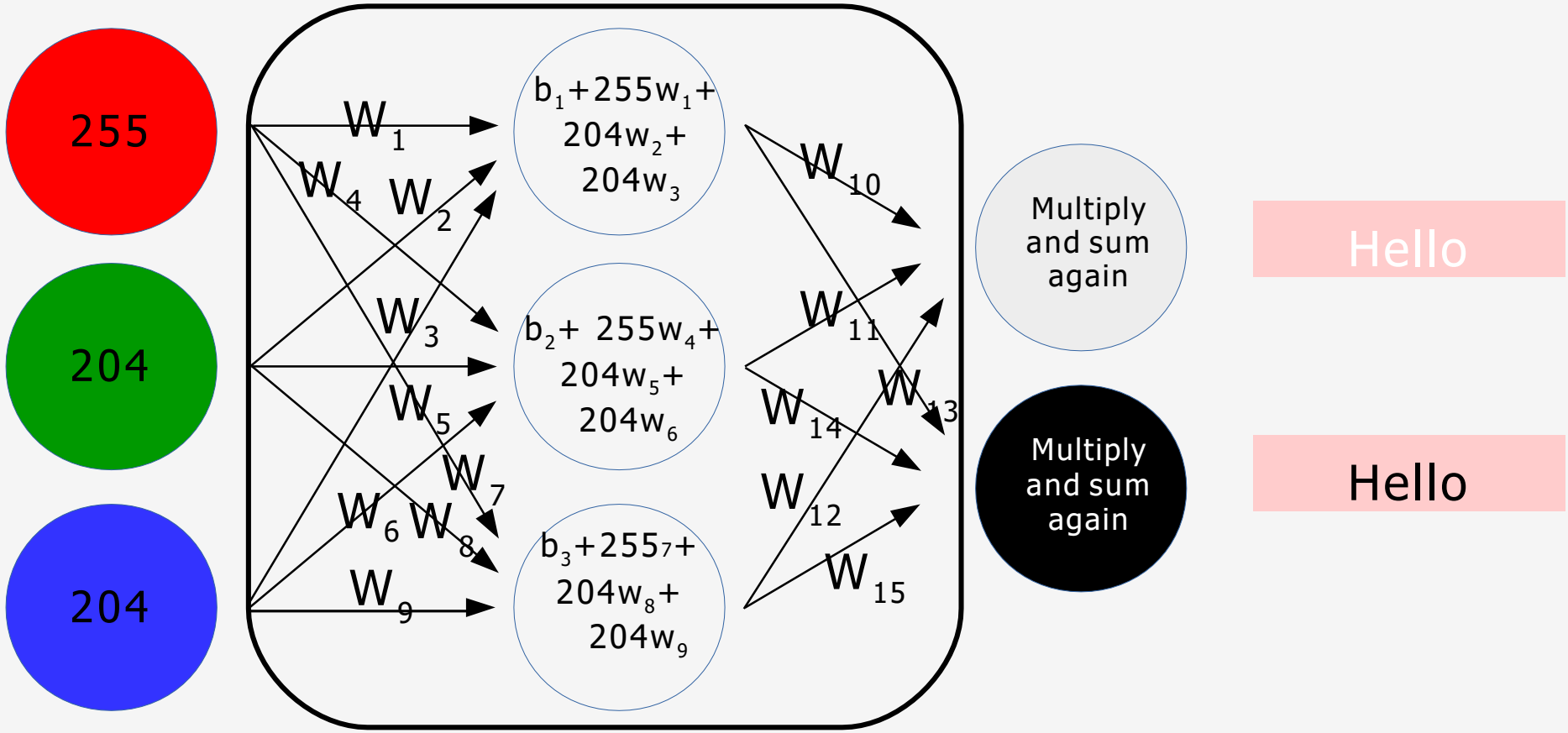
A Simple Neural Network

Each weight w_x value is between -1.0 and 1.0



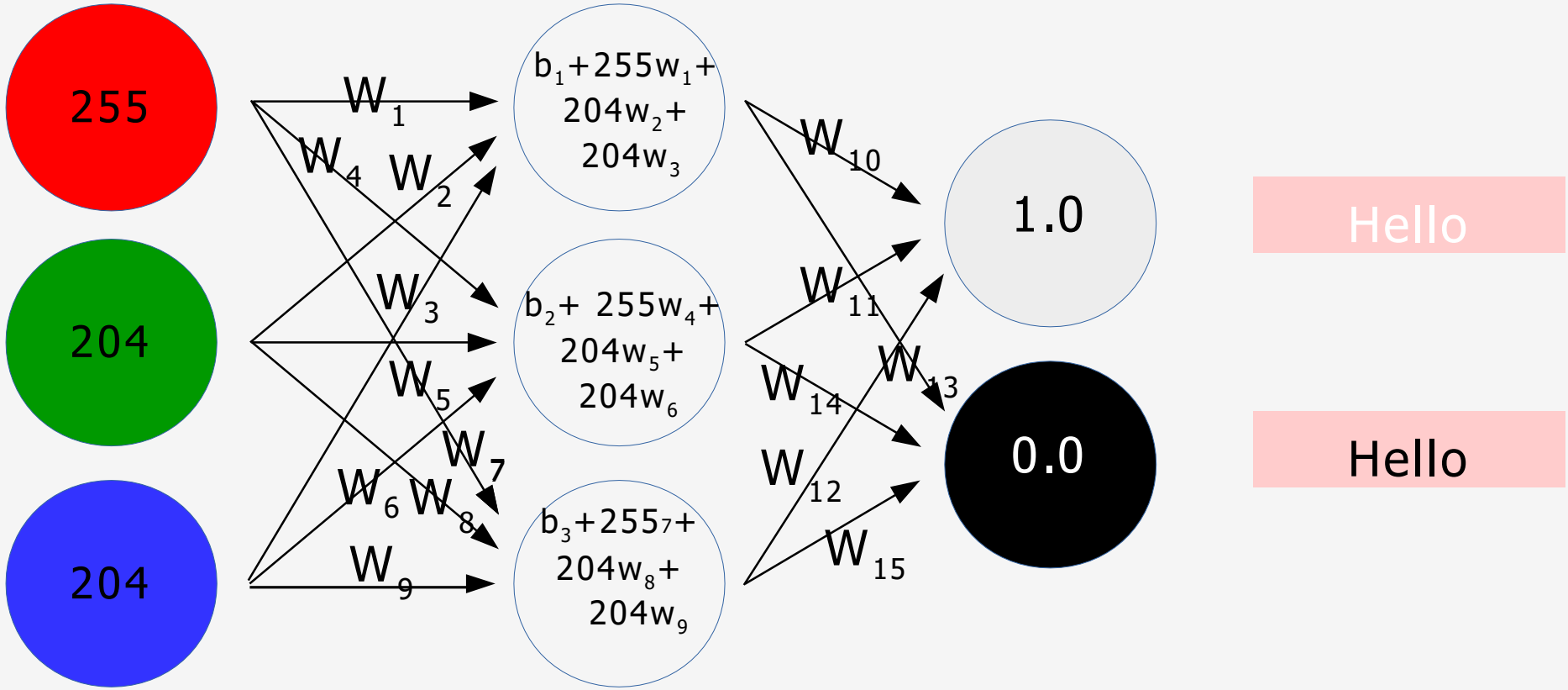
A Simple Neural Network

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



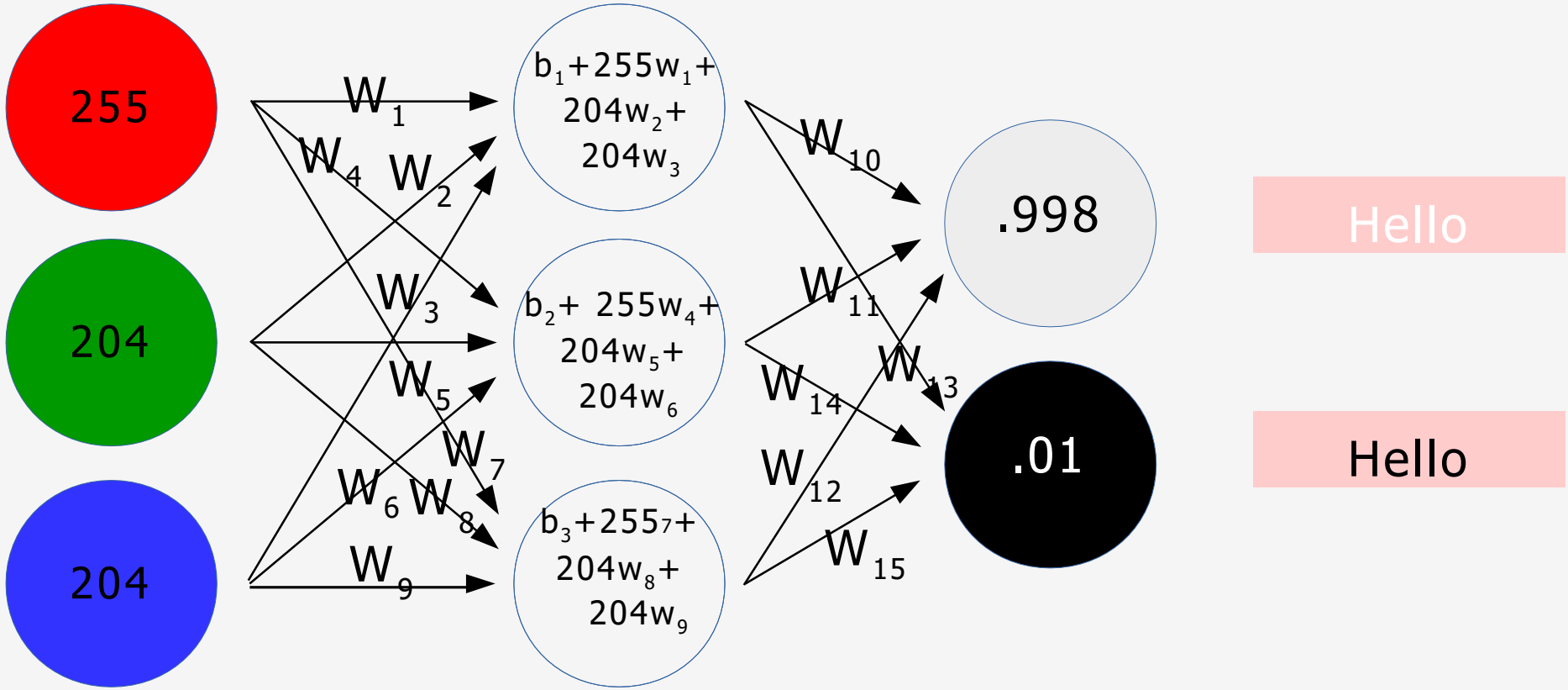
A Simple Neural Network

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



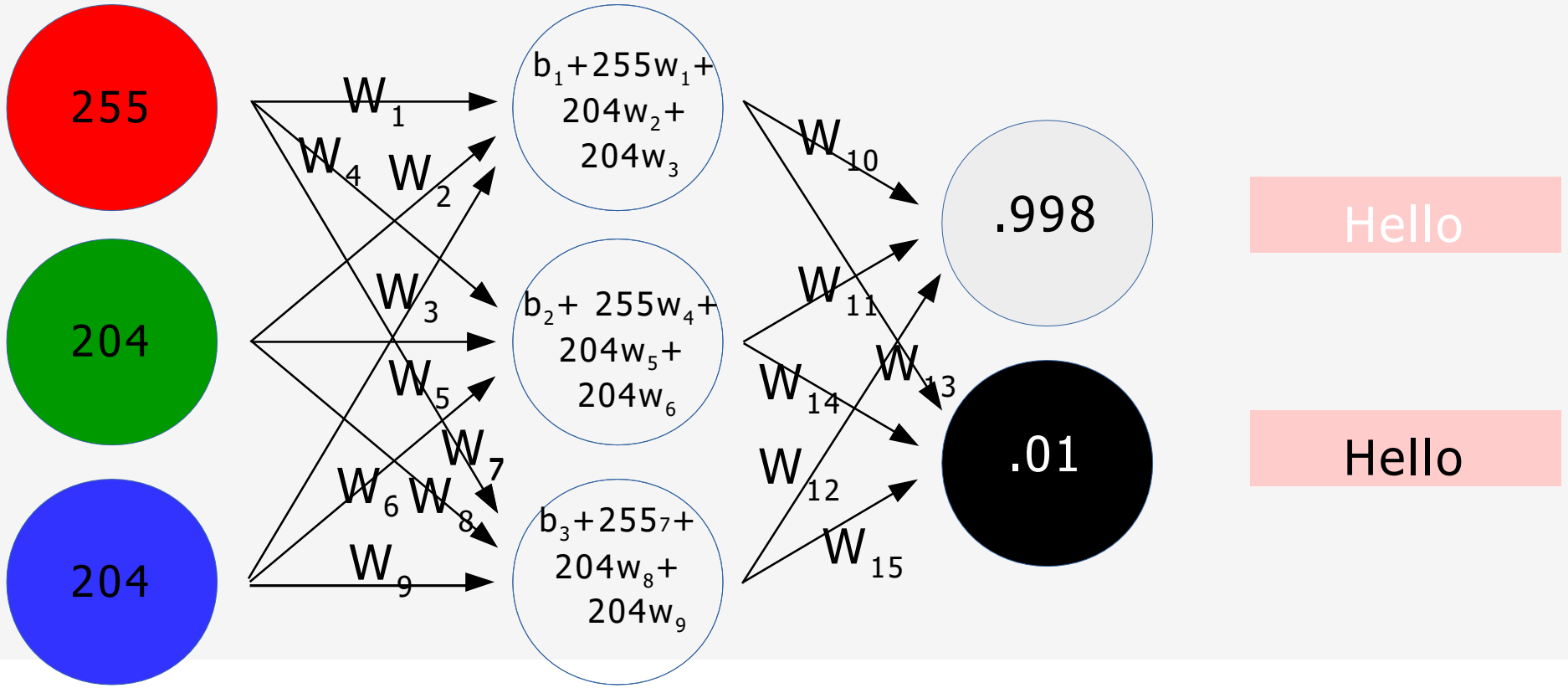
A Simple Neural Network

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.

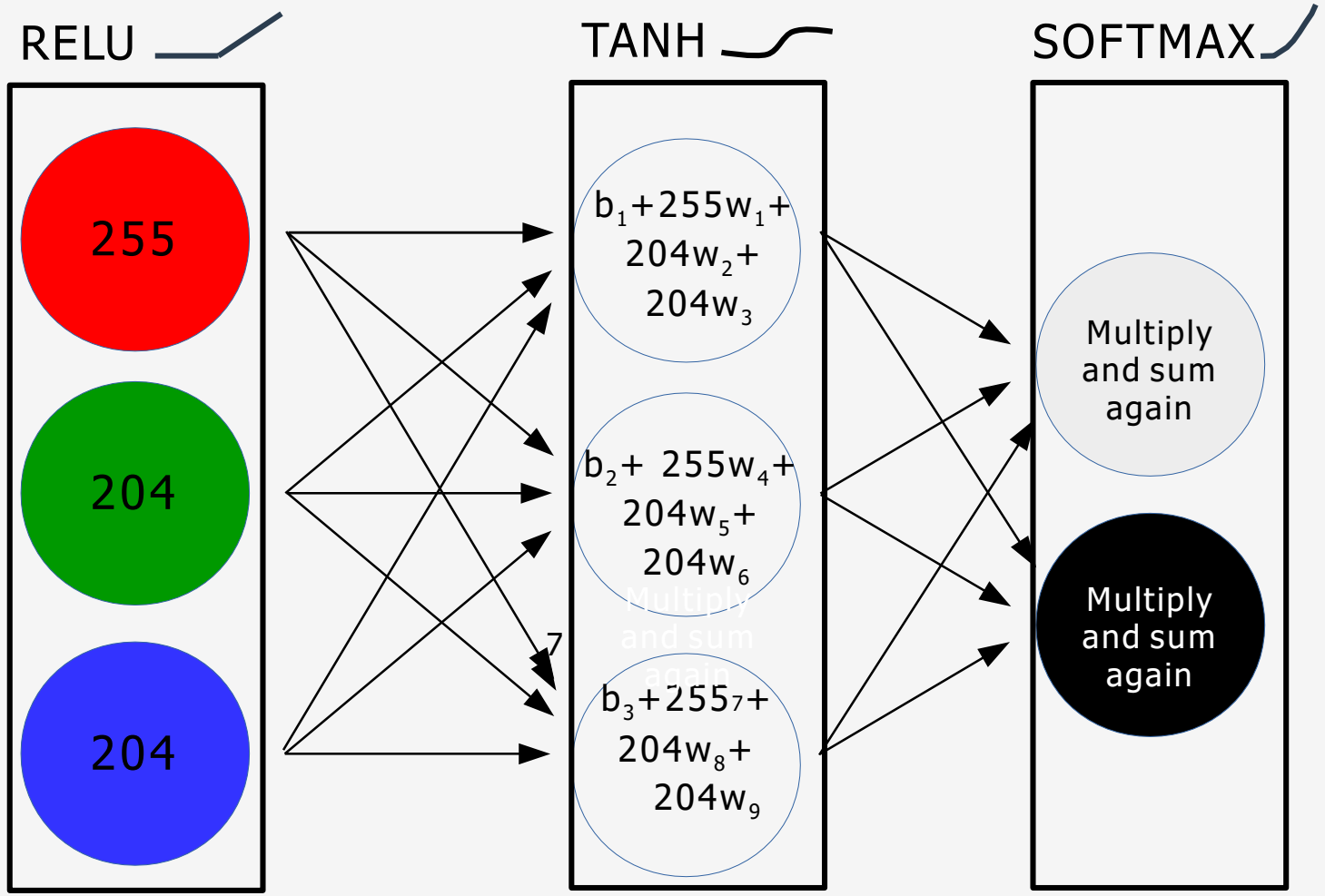


Activation Functions

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.



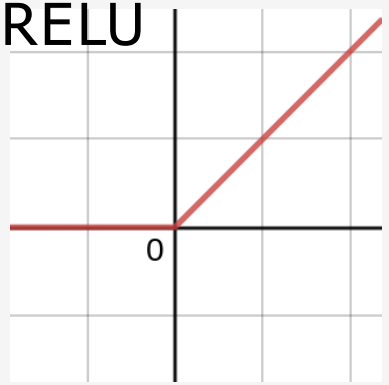
https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Activation Functions

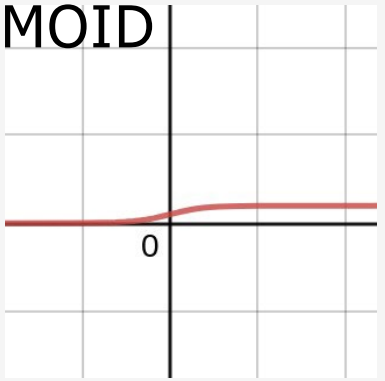
```
import numpy as np
from scipy import special
import math


# 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.

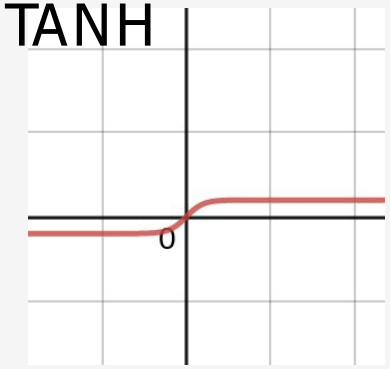
RELU



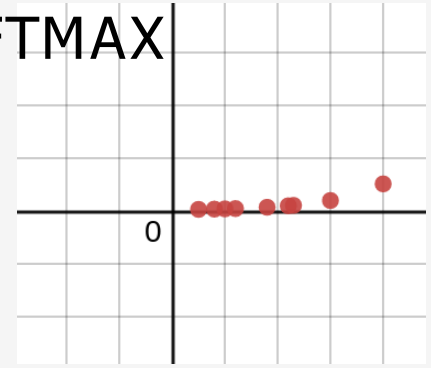
SIGMOID



TANH



SOFTMAX



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

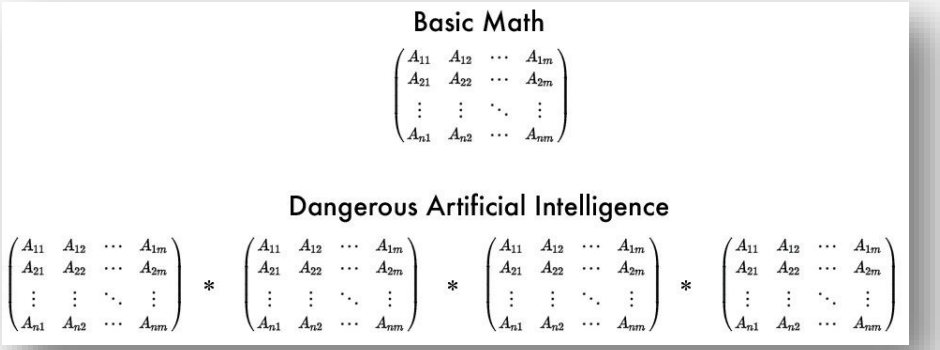
Building Neural Networks with NumPy

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.

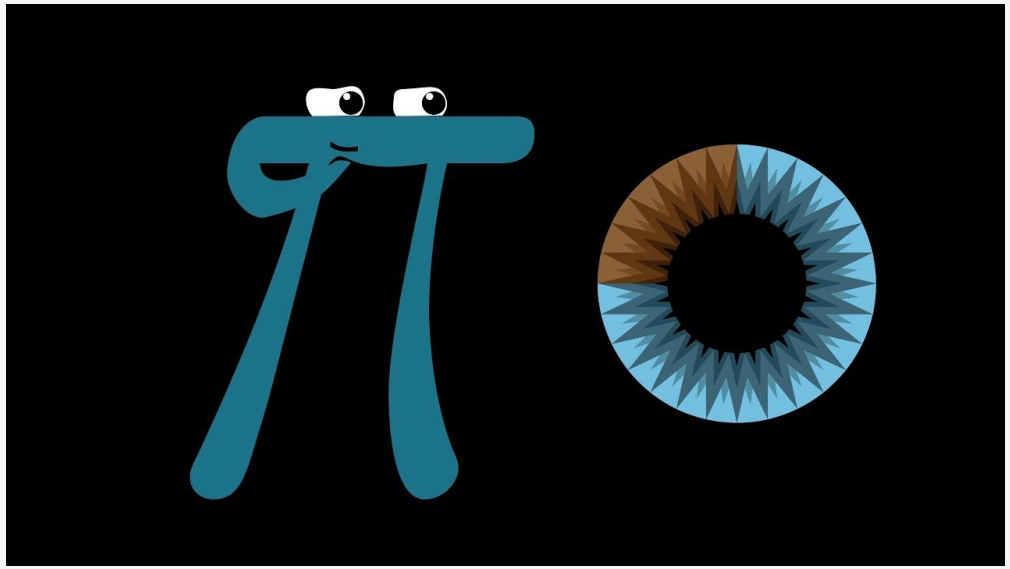
Hopefully a **matrix** is a familiar concept where you express data as grids of numbers that can be multiplied, added, and transformed in various ways.

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



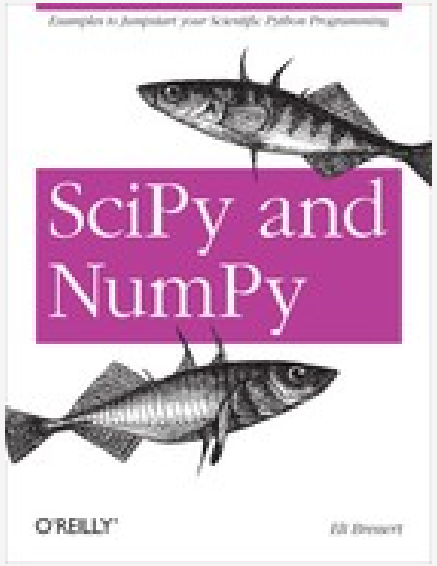
Resources to Learn Linear Algebra

3Blue1Brown - YouTube



https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

O'Reilly – SciPy and NumPy



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.

Input RGB Colors

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & 100 & \dots \\ 0 & 128 & 139 & 205 & 238 & 255 & 0 & \dots \end{matrix} \times \frac{1}{255} =$$

Scaled Input RGB Colors

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 0 & .392 & \dots \\ 0 & .501 & .545 & .803 & .933 & 1 & 0 & \dots \end{matrix}$$

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

# 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 x 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_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)
```

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

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))))
```

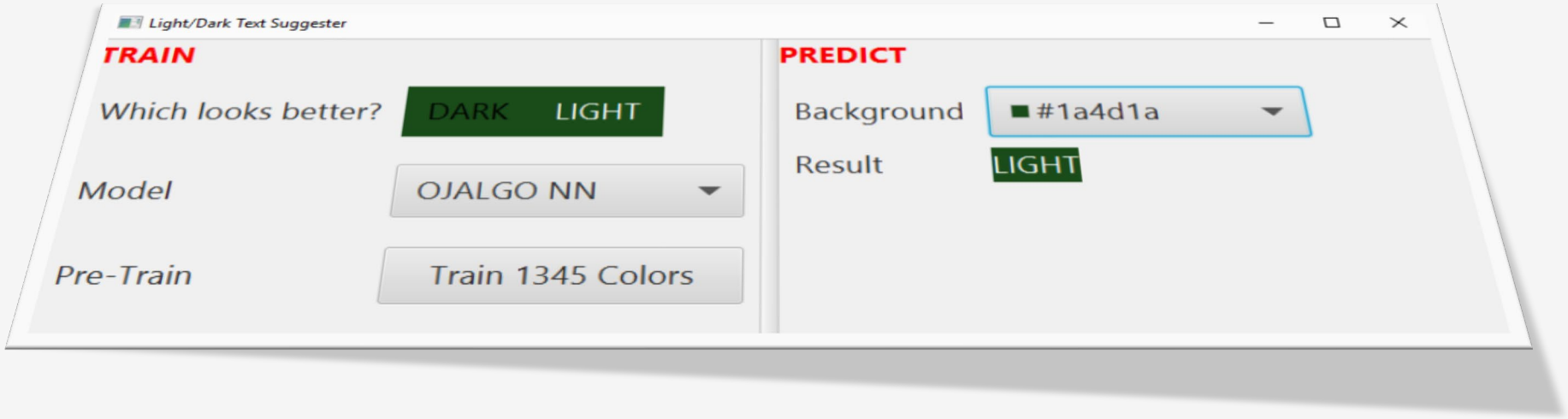
The neural network!

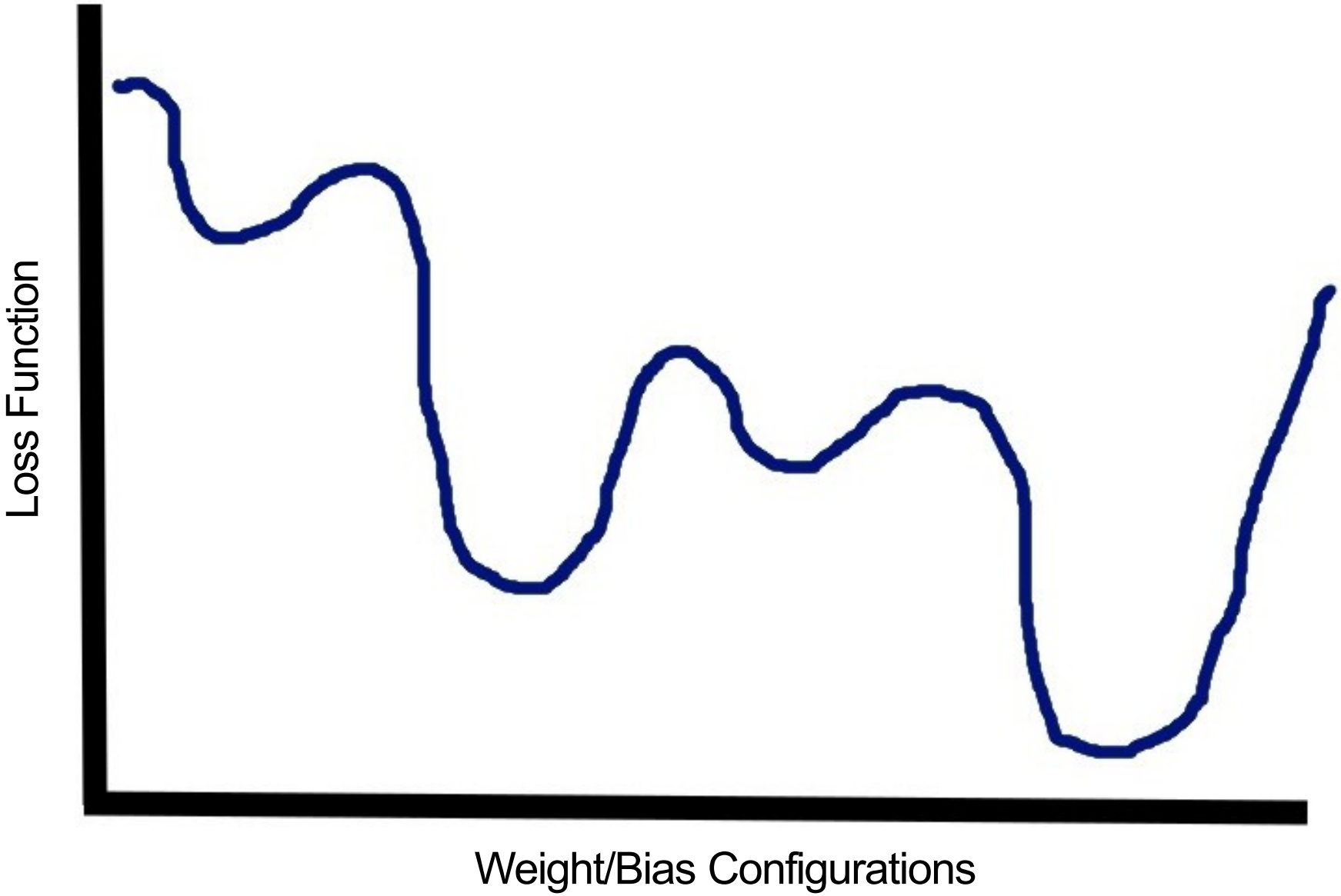
Alright, I’m going to save time

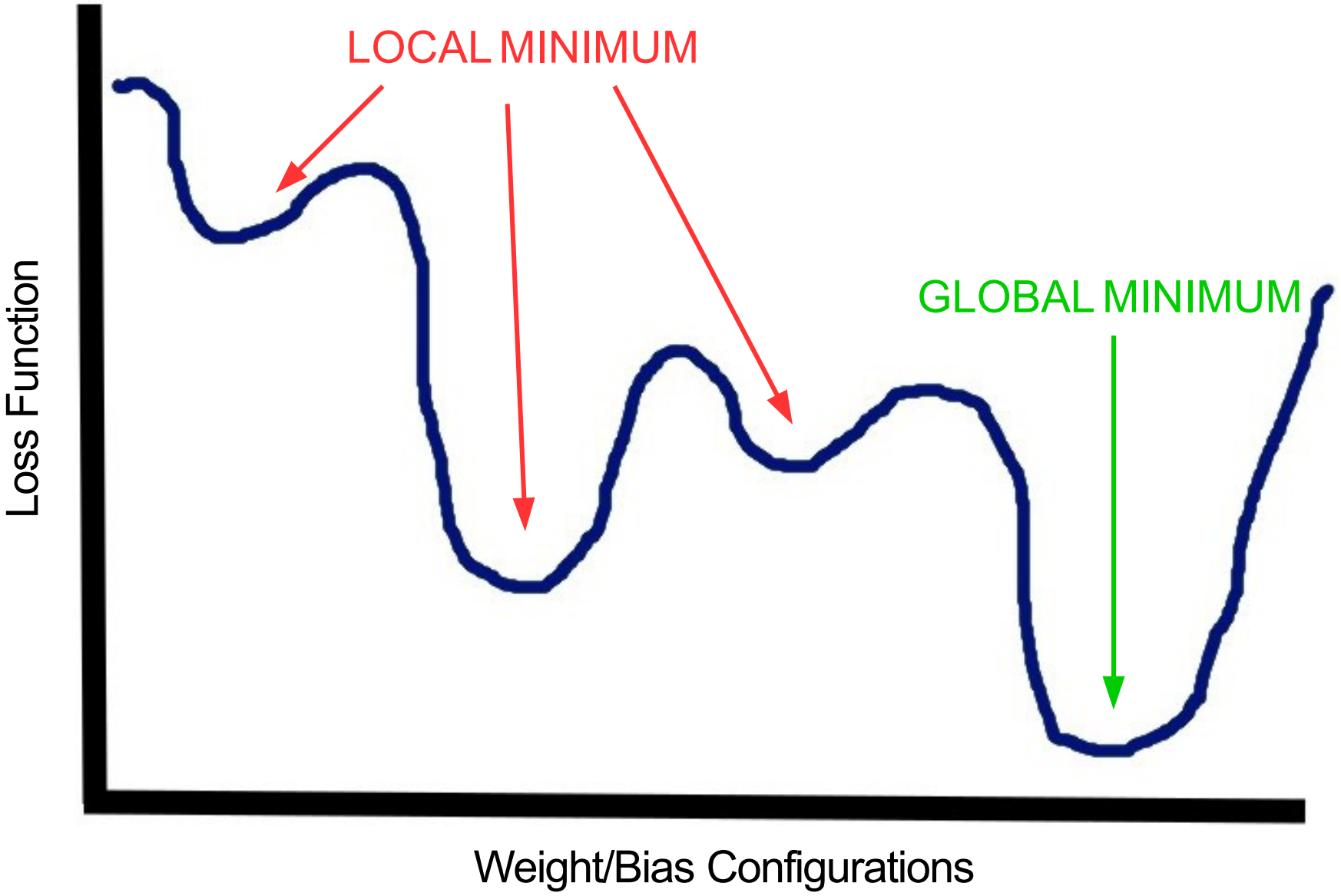
```
training_outputs = softmax(output_bias + output_weights.dot(relu(middle_bias + middle_weights.dot(training_inputs))))
```

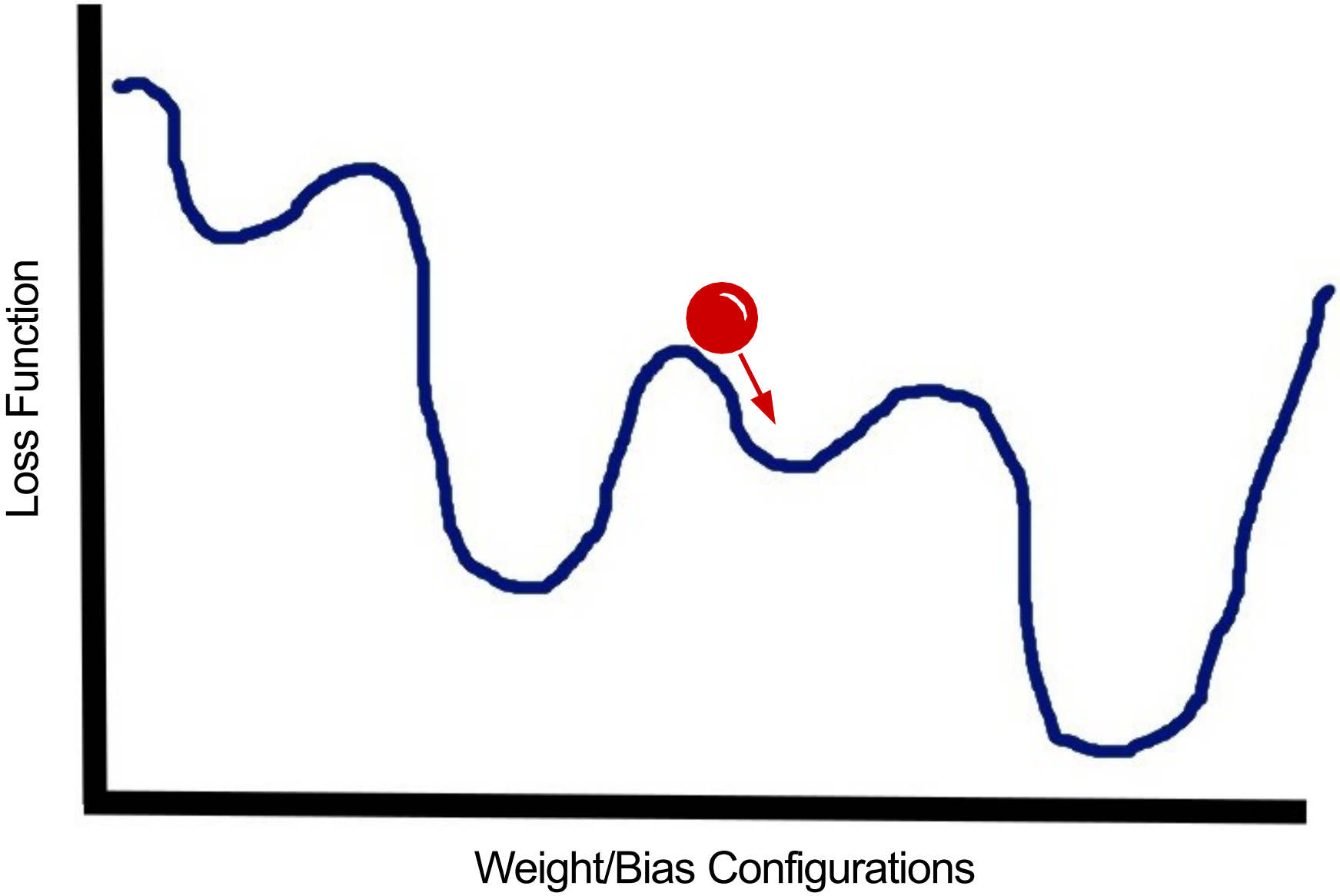
- That beautiful, magical one line executes your input values, multiplies all the weights and adds all the biases for all three layers, applies each activation function, and outputs the recommendation matrix.
- For millions of iterations, we will loop a hill-climbing algorithm randomly adjusting the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

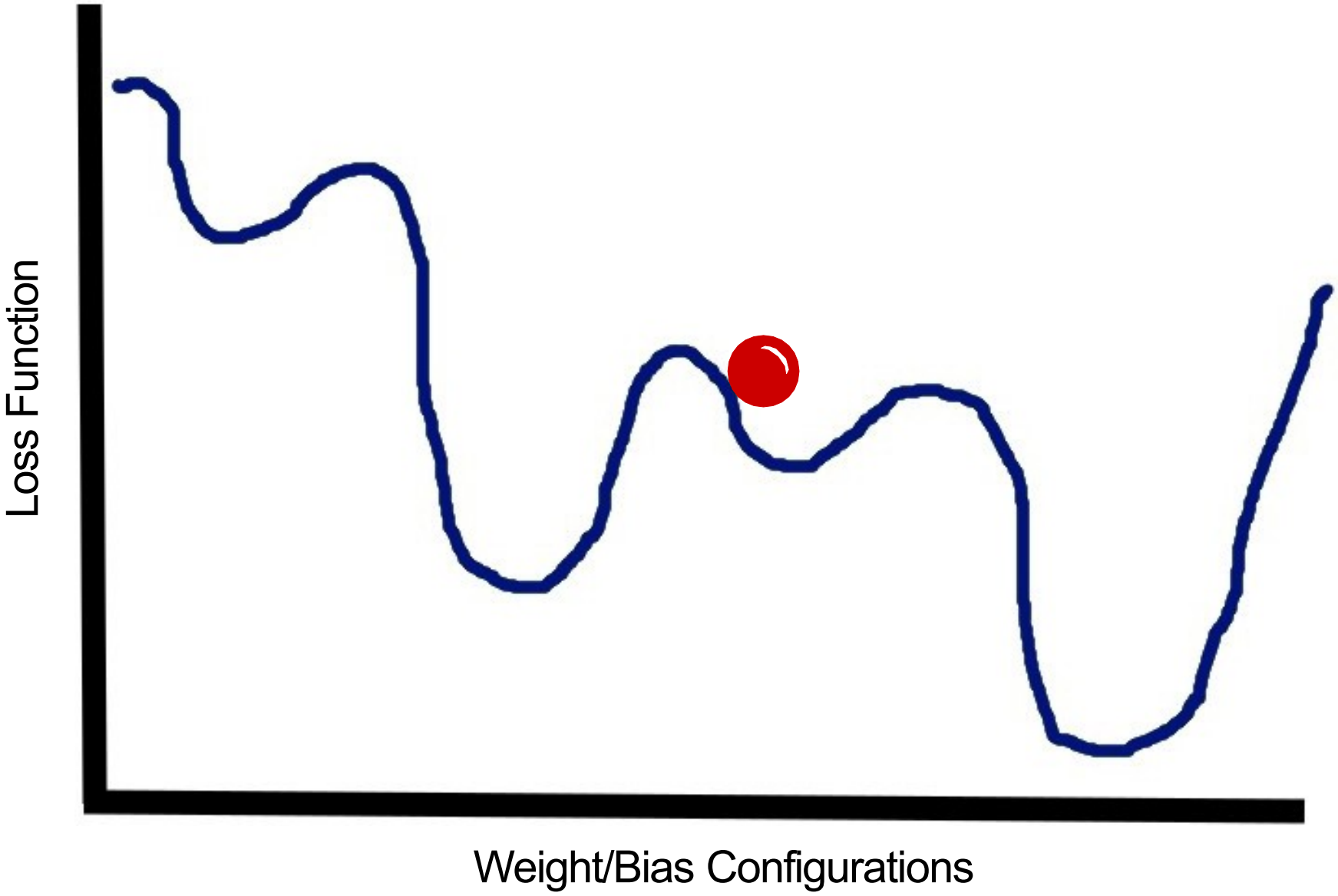
Demo: Training a Neural Network

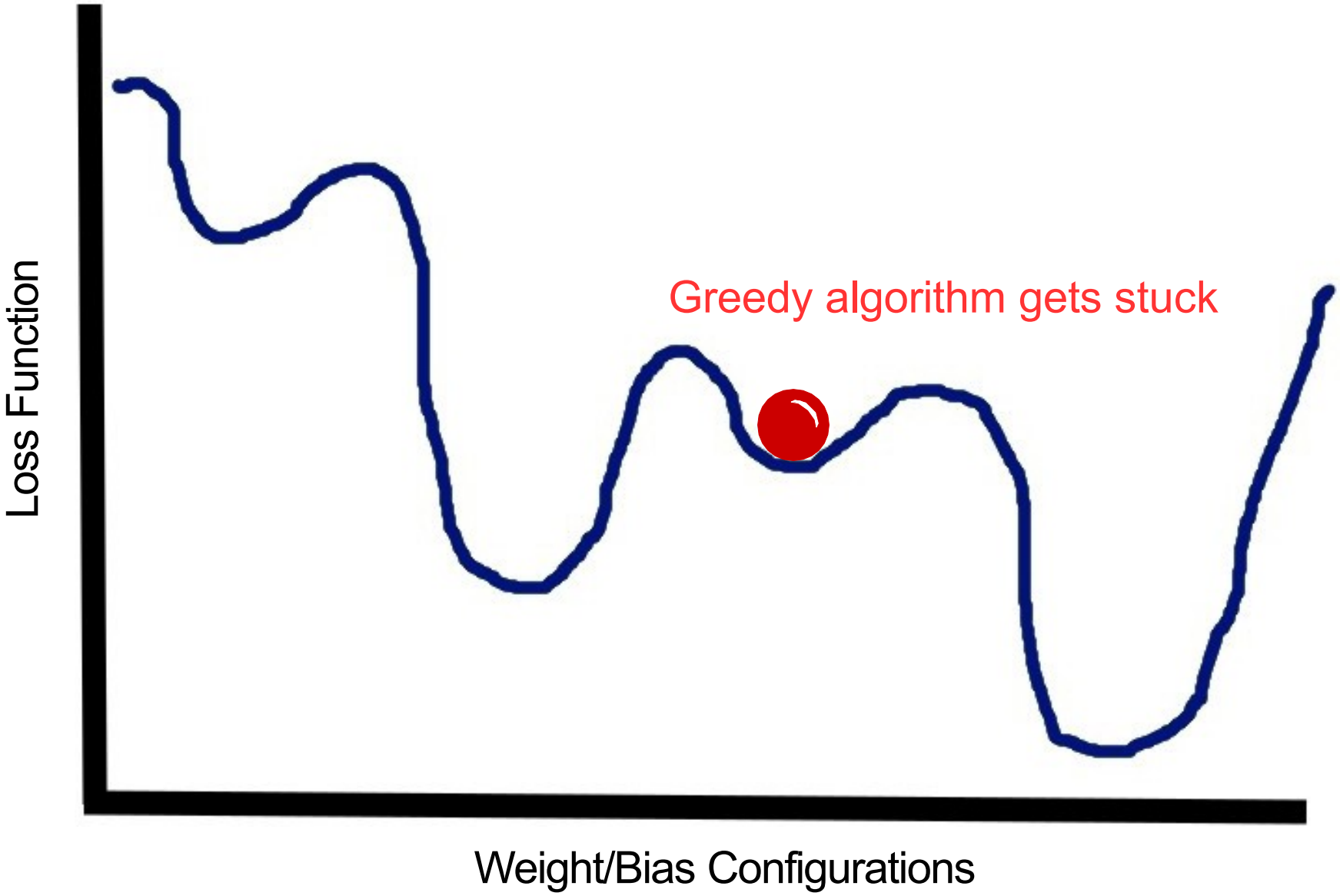


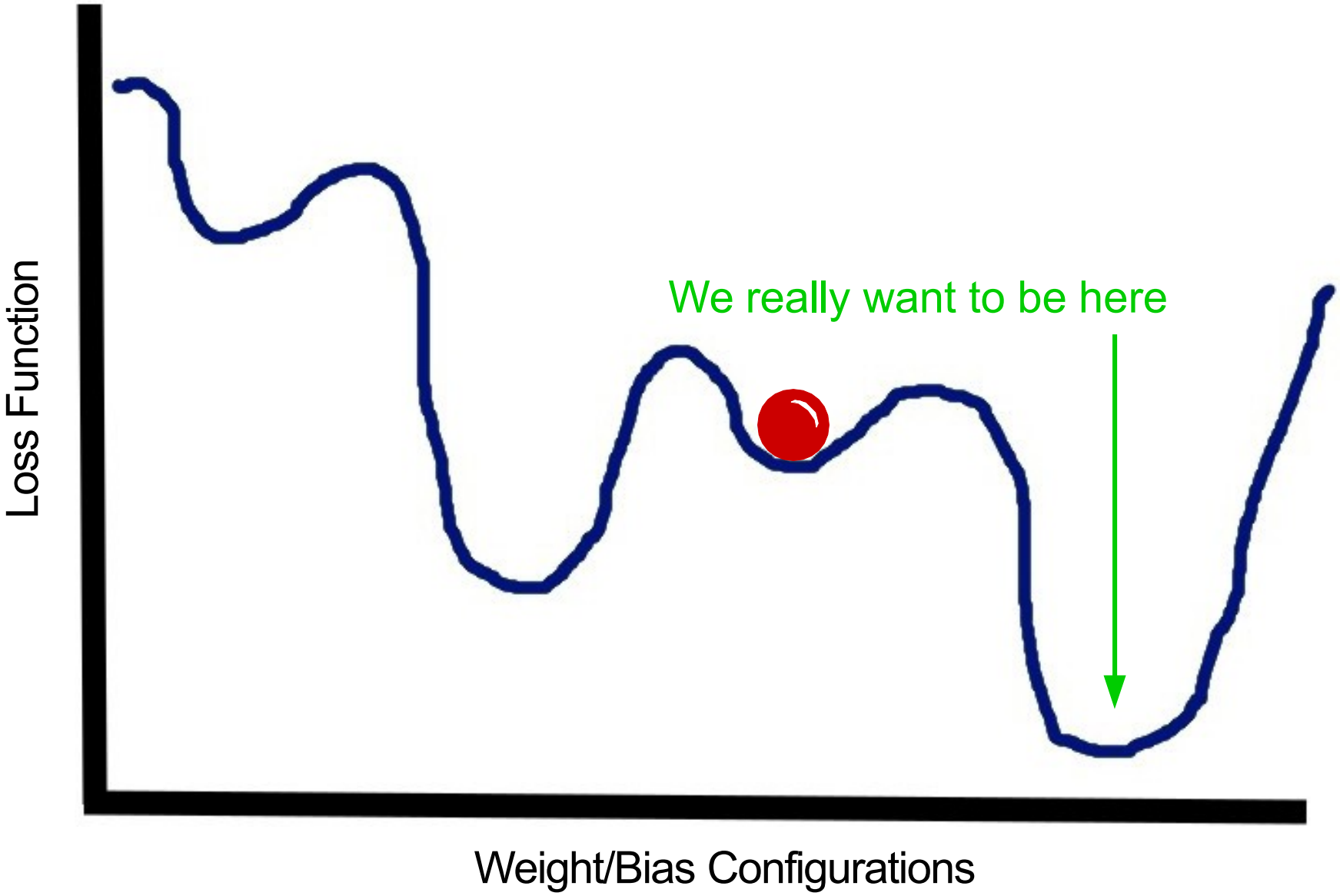

Loss Function
Weight/Bias Configurations

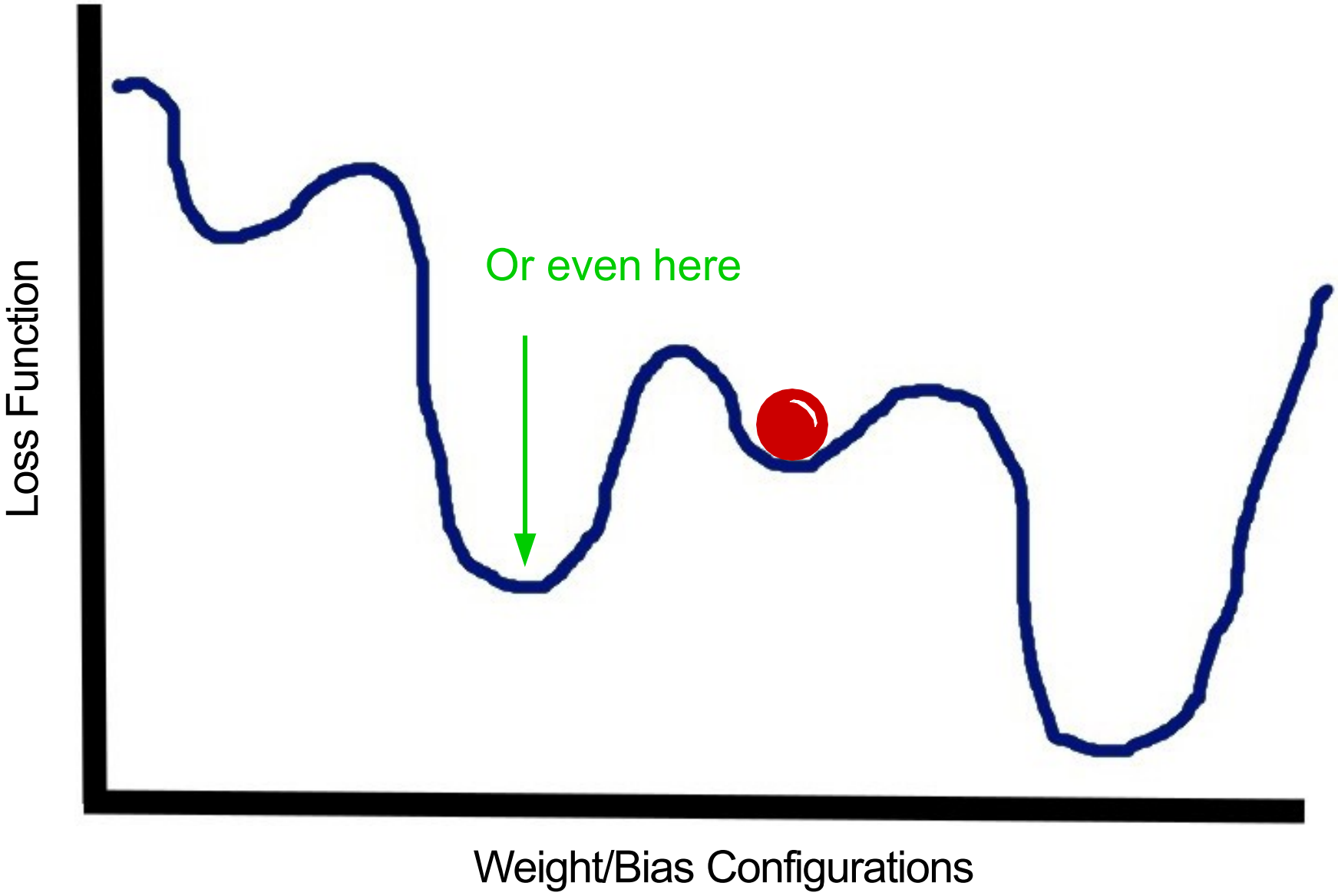

LOCAL MINIMUM
GLOBAL MINIMUM
Loss Function
Weight/Bias Configurations

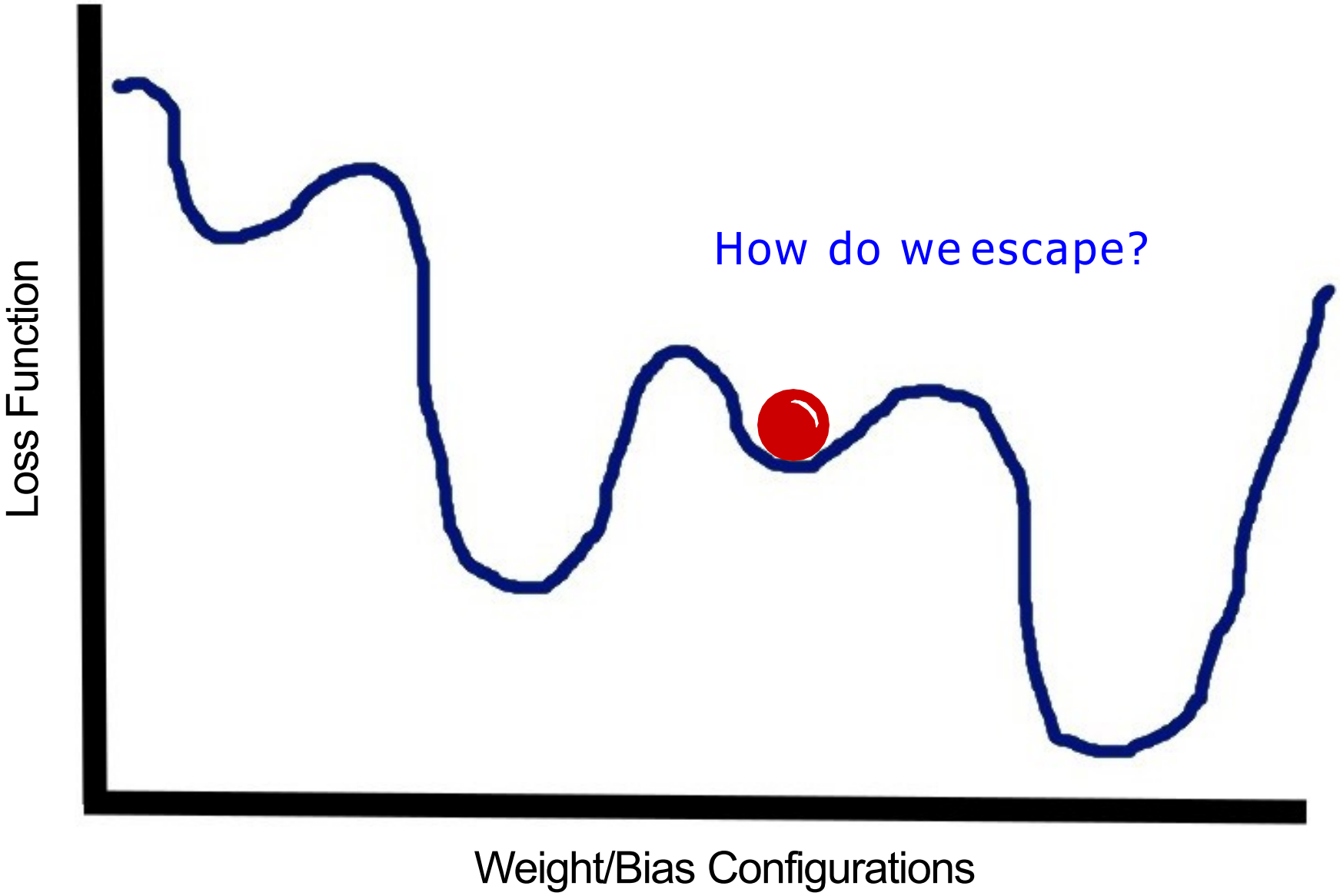

Loss Function
Weight/Bias Configurations

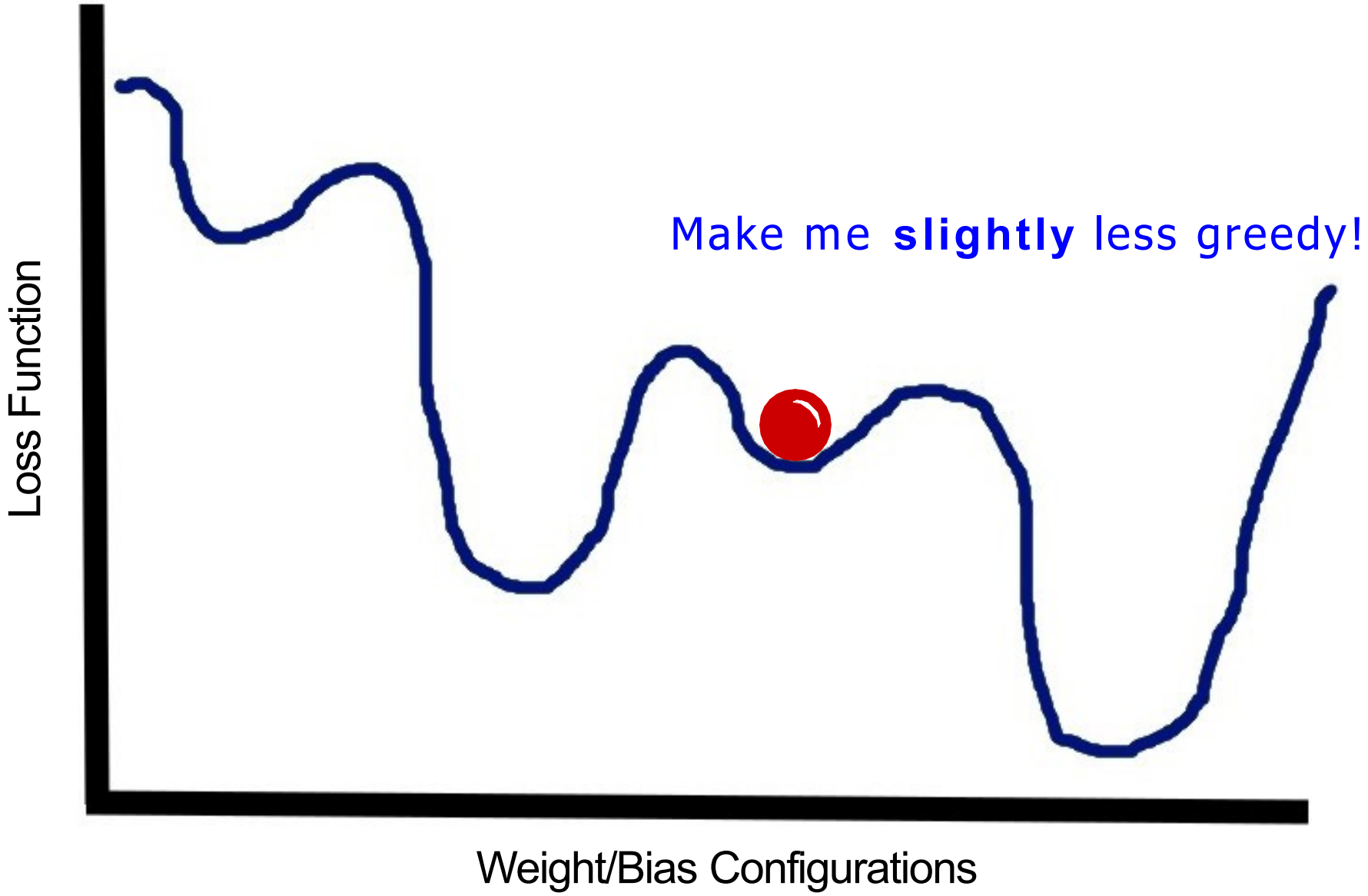

Loss Function
Weight/Bias Configurations

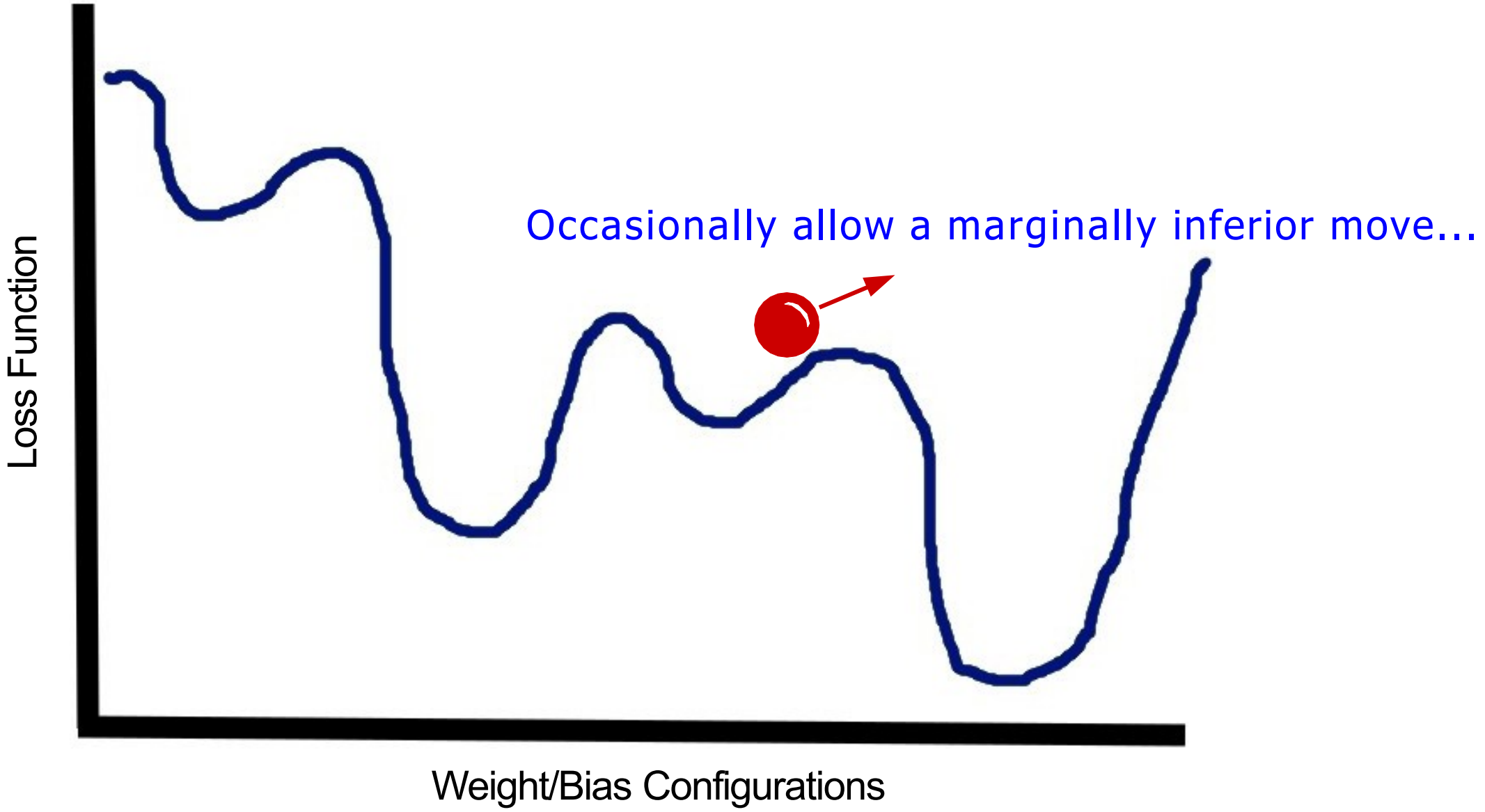

Loss Function
Greedy algorithm gets stuck
Weight/Bias Configurations

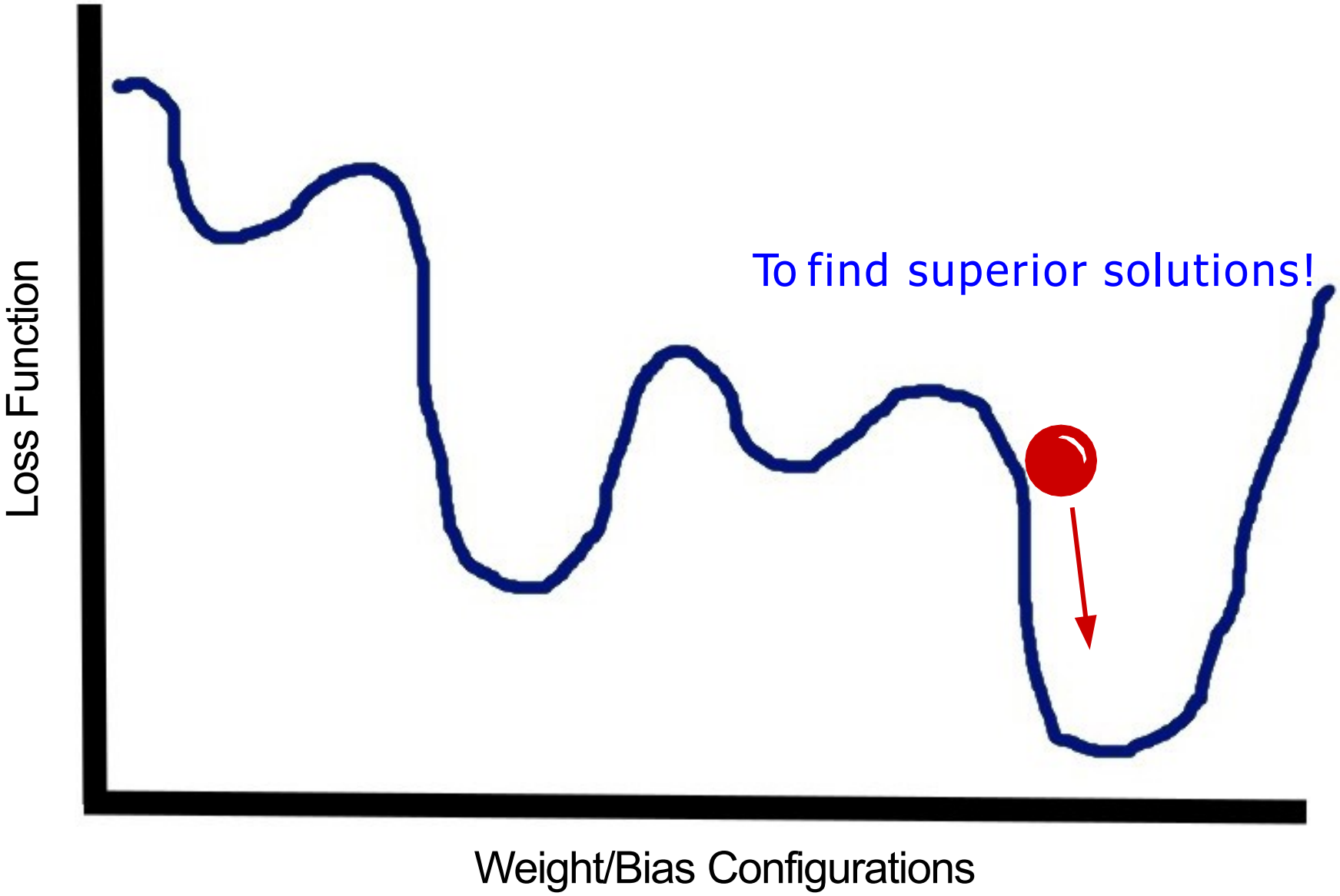

Loss Function
We really want to be here
Weight/Bias Configurations

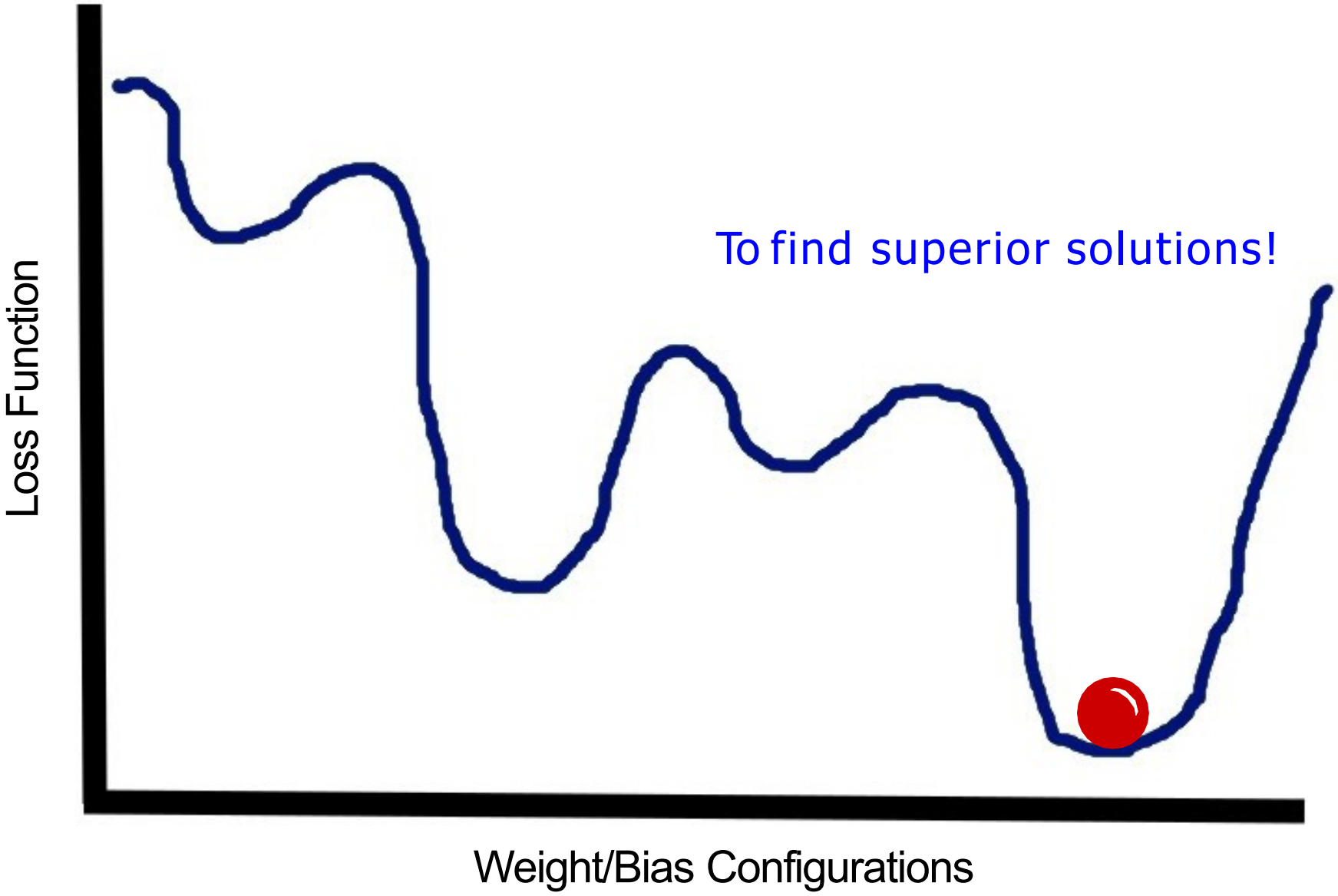

Loss Function
Or even here
Weight/Bias Configurations

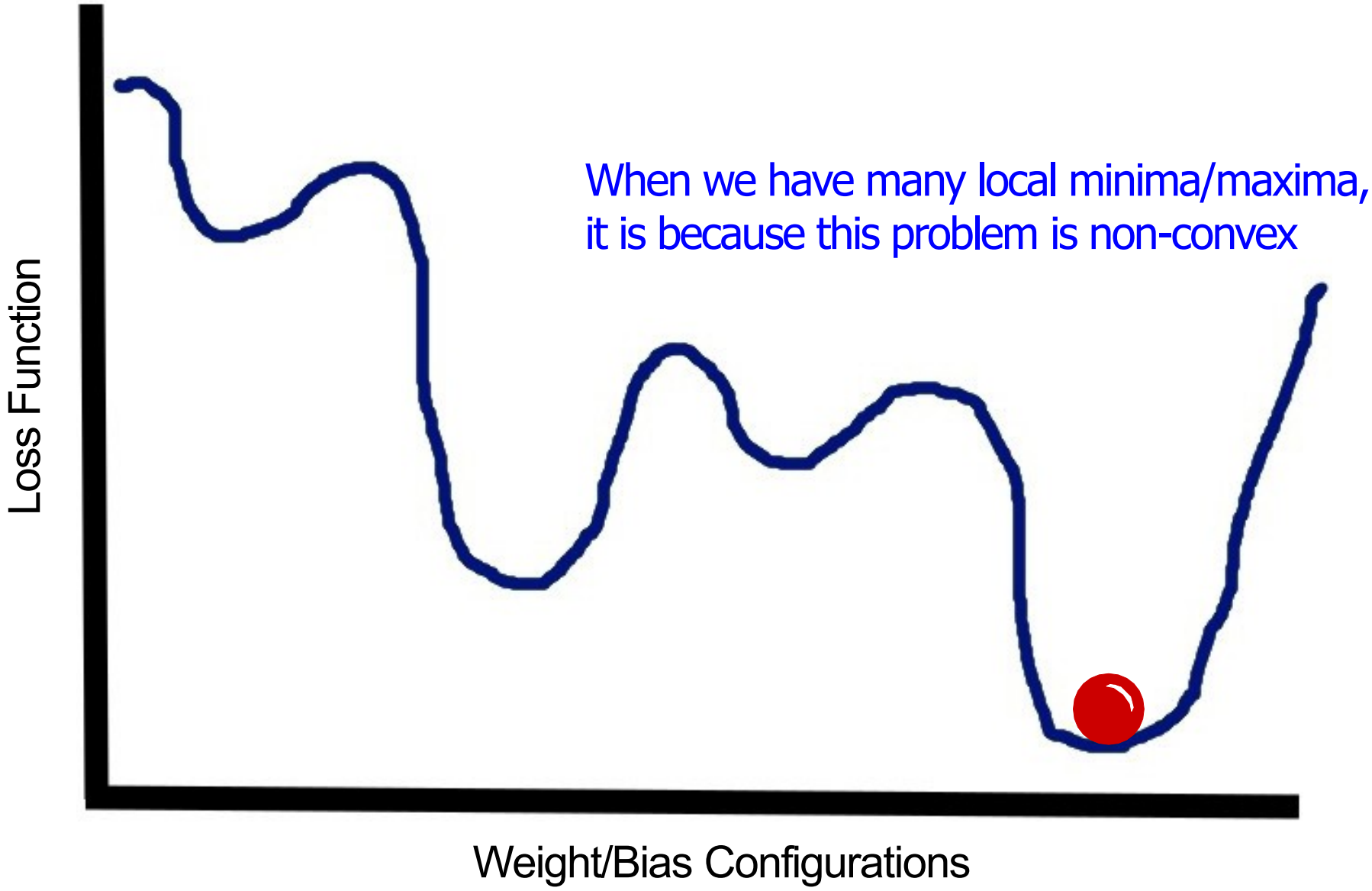

How do we escape?
Loss Function
Weight/Bias Configurations


Make me **slightly** less greedy!
Loss Function
Weight/Bias Configurations


Occasionally allow a marginally inferior move...
Loss Function
Weight/Bias Configurations


Loss Function
To find superior solutions!
Weight/Bias Configurations

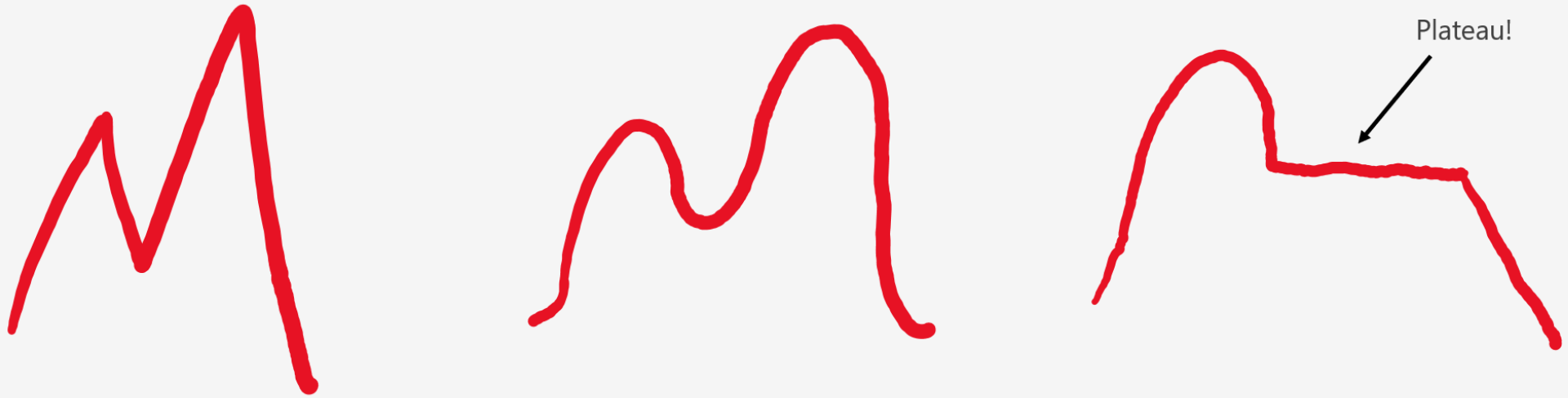

To find superior solutions!
Loss Function
Weight/Bias Configurations


When we have many local minima/maxima,
it is because this problem is non-convex
Loss Function
Weight/Bias Configurations

Convex versus Non-Convex

Convex

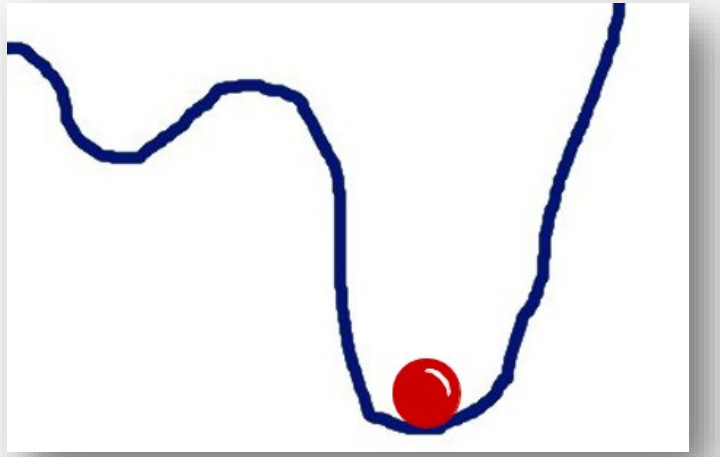
Convex versus Non-Convex



Non-Convex

The Hard Part of Deep Learning

A lot of the machine learning algorithms we have learned to this point were convex, meaning they have a single local minimum.



However, deep learning problems often have several local minima, and it can be challenging to end up in one that is optimal.

Like any machine learning algorithm, we also have to be careful to not overfit so finding the global minimum may not be a good thing!

In another O'Reilly Online Training *Intro to Mathematical Optimization*, we cover other techniques like simulated annealing and gradient descent to better address this problem.

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?

The Practicality of Neural Networks

Despite some seemingly intelligent applications like image recognition, neural networks have drawbacks:

- Requires **LOTS** of labeled data fortraining.
- Difficult for the data scientist to understand convoluted layers and nodes.
- Has a quick diminishing return for problem spaces outside of image and natural language processing.

Neural networks generate excitement because many believe they will generalize solutions to most problems one day.

- In reality, specialized and incumbent algorithms will outperform neural networks for most practical problems.

Problems with Deep Learning

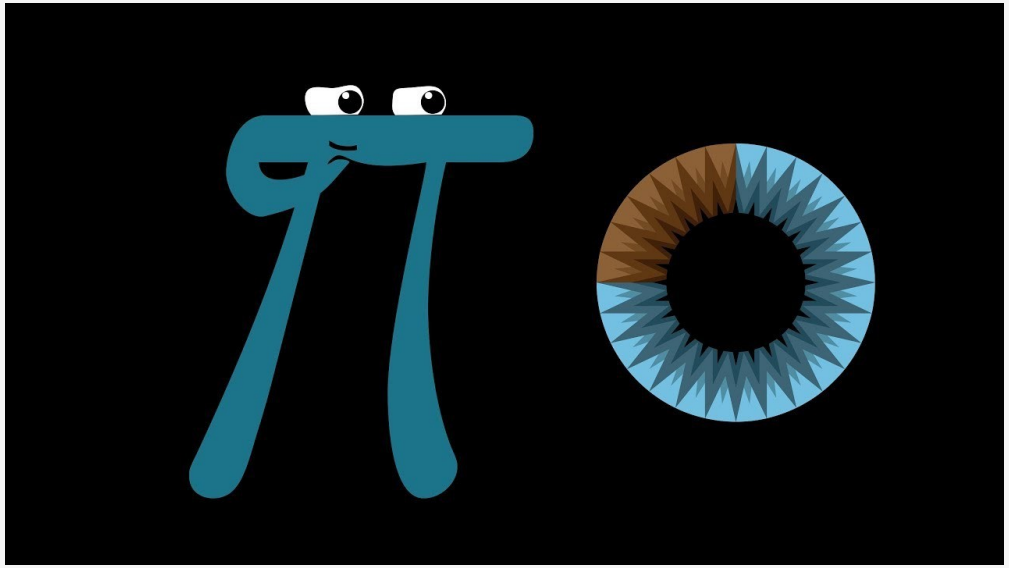
Machine learning algorithms, especially deep learning, are not perfect and easily error-prone especially with outliers. **Correlation is not Causation!**

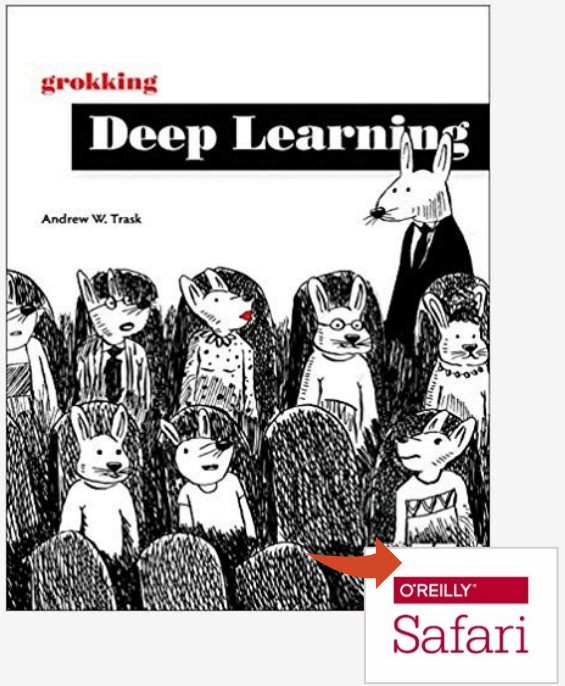
To the right are examples of a well-trained neural network unable to recognize images correctly due to objects in abnormal positions

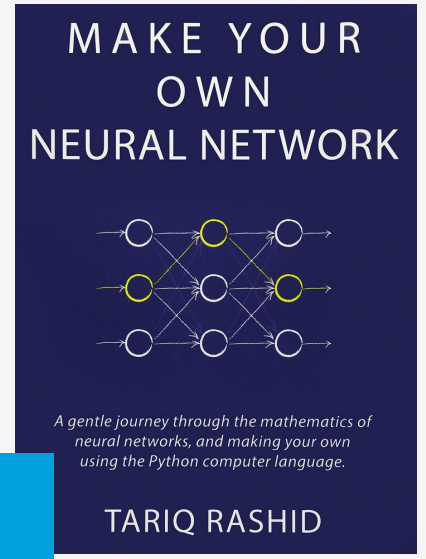
SOURCE: https://arxiv.org/abs/1811.11553

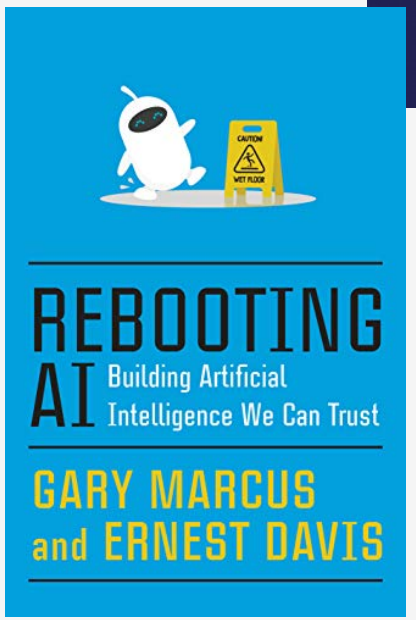
Learn More About Neural Networks

3Blue1Brown - YouTube









Section VII
Going Forward

Final Thoughts

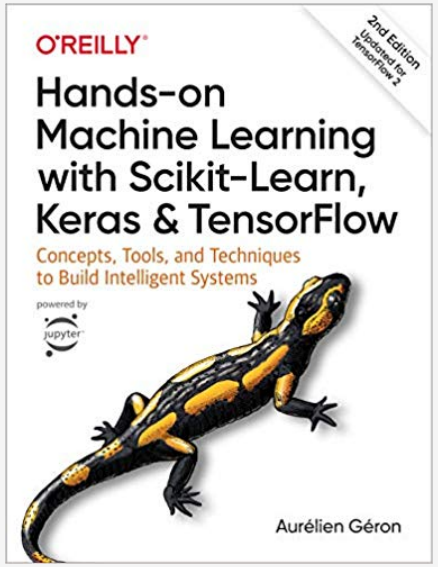
Machine Learning is a powerful tool but is by no means a silver bullet or panacea.

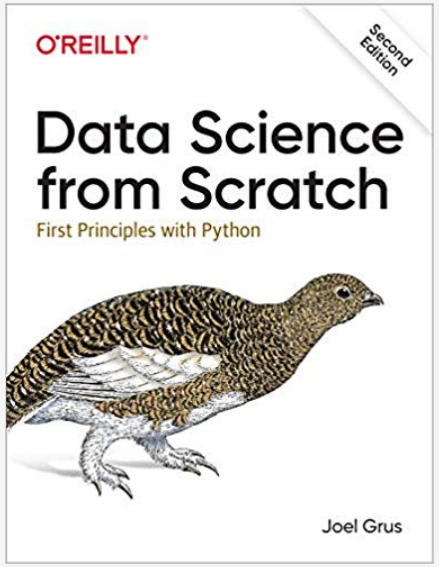
- "When all you have is a hammer, everything starts to look like a nail"
- Data and data-driven models (like machine learning) are useful, but do not use them to solve problems that are better solved with more explicit models
- Do not dismiss explicit models, business rules, and classical "AI" models like search algorithms to solve your problem too.

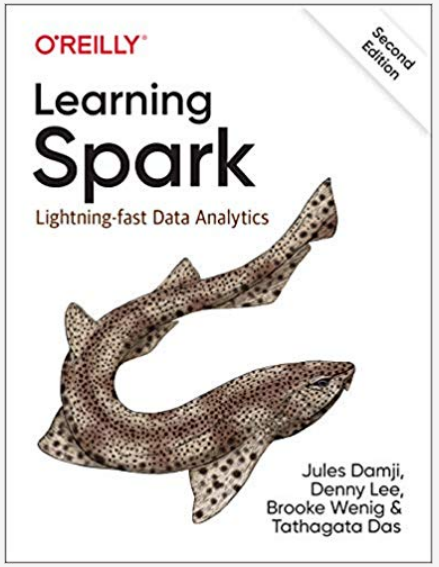
Be strategic about scaling your machine learning applications.

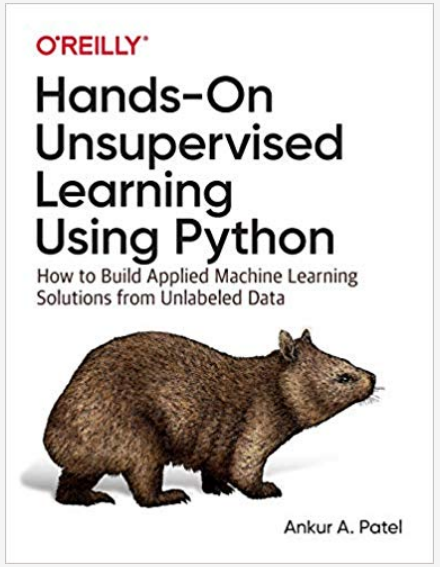
- Learn NumPy so you can write more efficient Python machine learning programs, as this class does not teach how to code efficient models.
- You can also learn faster programming platforms like C, C++, Go, Java, Kotlin, Scala, etc.

Great Books to Learn More About Machine Learning





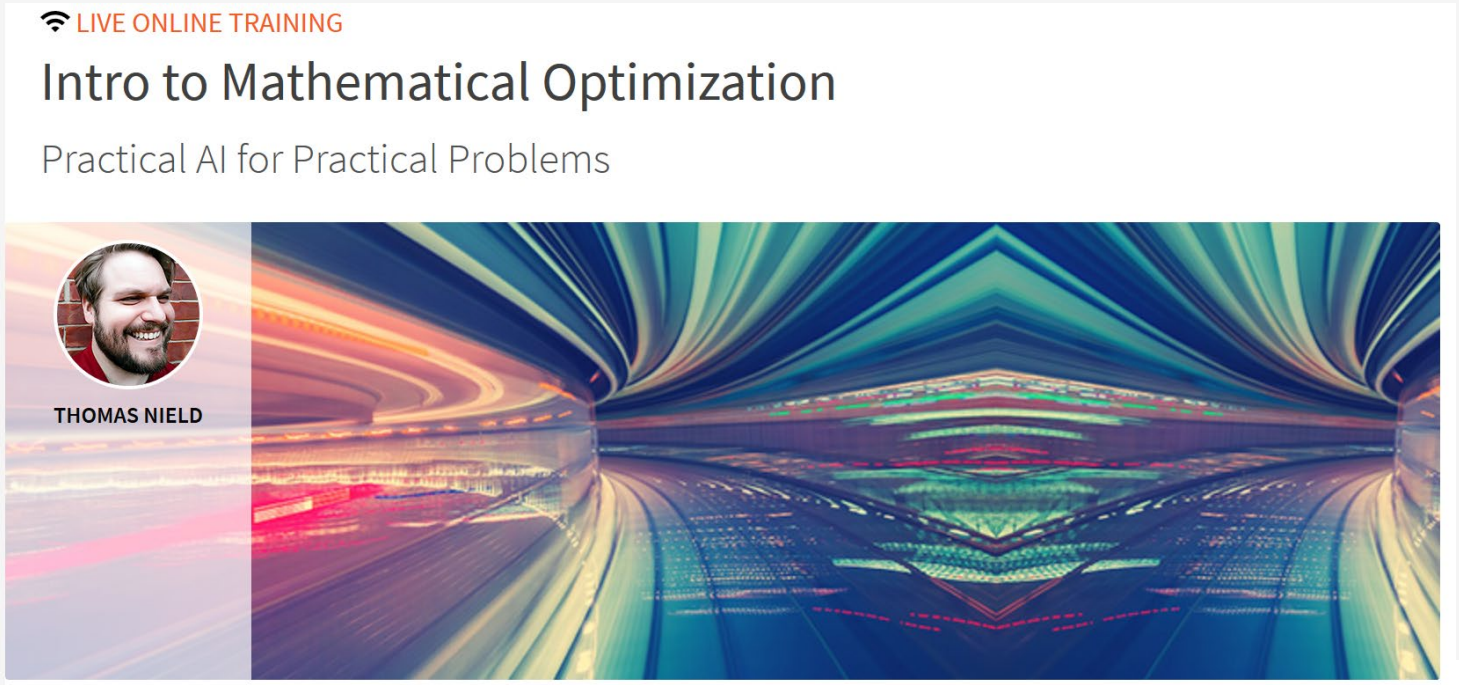




Continuing Your Journey

If you enjoyed this online training, please take the sister class *Intro to Mathematical Optimization*.

- Optimization is the heart of machine learning and the black box few explore.
- There is far more to cover beyond hill climbing and optimization is intuitive, hard, and rewarding all at the same time.



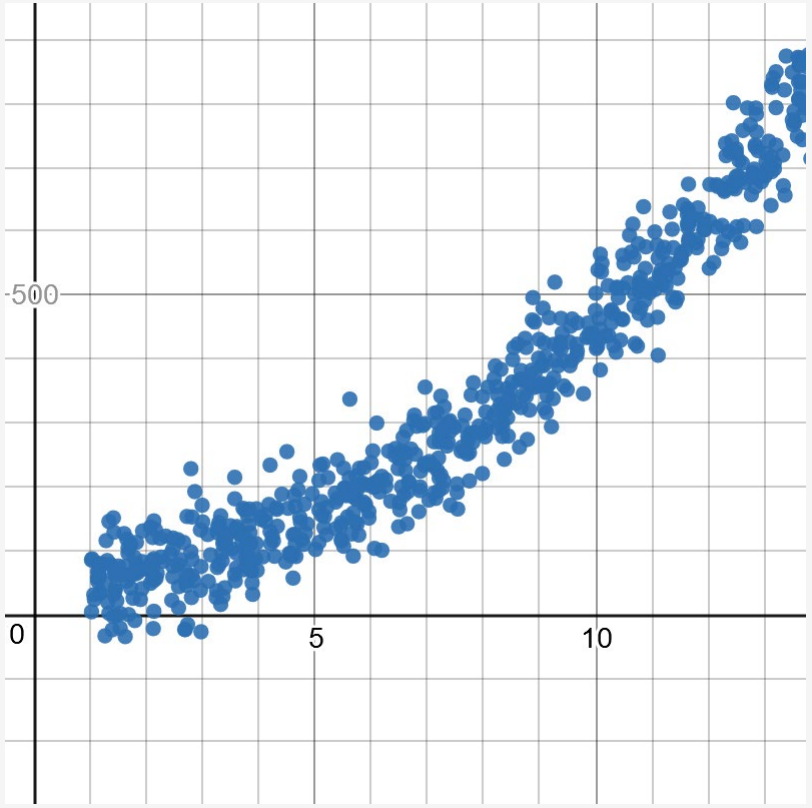
Homework Assignments (Answers Are In “code” Folder)

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.

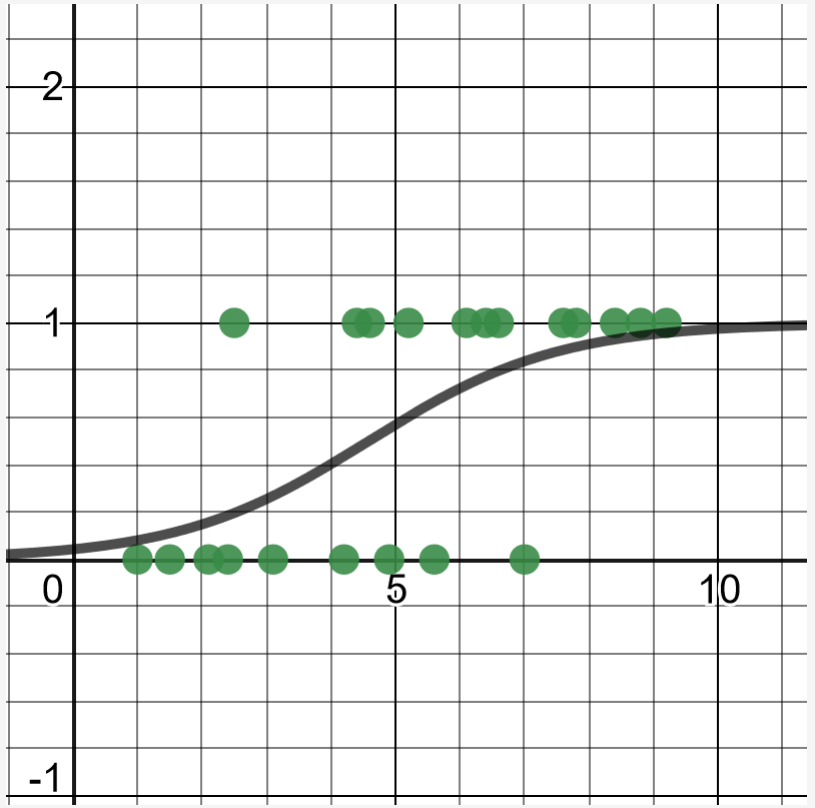


https://www.desmos.com/calculator/canygmx67n

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*

Homework #2

Take the RGB background color data (accessible at https://tinyurl.com/y2qmhfsr) 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?