*How much money will you make by spending more dollars on your Machine Learning training? Will the person who took out a loan with you - pay back the loan or not? I live in South Florida and recently, there was a new Hard Rock Hotel Casino that came up … in the shape of a guitar. What’s going to happen at the blackjack table there tomorrow? These are all problems that supervised learning helps us with.*

With supervised learning, you are going to insert the output of your algorithm into the system as a Model. What that means is that in supervised learning, the machine already knows the output of the algorithm before it starts working on it or learning it. That is a key point to understand. A basic example of this concept would be you learning machine learning from me as your instructor. You, as the student, already knows what you are learning from the course.

With the output of the algorithm known, all that a system needs to do is to work out the steps or the process needed to reach from the input to the output. In other words, you need to discover the best algorithm that maps inputs and outputs and you have both inputs and outputs already. The algorithm is being taught through a training data set that guides the machine. If the process goes bananas and the algorithms come up with results that are completely different than what should be expected, then the training data does its part to guide the algorithm back towards the right path.

Supervised Machine Learning currently makes up most of the ML that is being used by systems across the world. The input variable (x) is used to connect with the output variable (y) through the use of an algorithm. All of the input, the output, the algorithm, and the scenario are being provided by humans. We can understand supervised learning in an even better way by looking at it through two types of problems.

**Classification**: Classification problems categorize all the variables that form the output. Examples of these categories formed through classification would include demographic data such as marital status, sex, or age. One of the most common models used for this type of service status is the support vector machine. The support vector machines set forth to define the linear decision boundaries. We will discuss SVMs later in the course.

**Regression**: Problems that can be classified as regression problems include types where the output variables are set as a real number. The format for this problem often follows a linear format.

Let’s go back and review before we move forward again. In supervised learning problems, you always start with a data set that contains **training examples** with associated correct **labels**. Look at kaggle.com, DataLens.org or even University of California Irvine… they all have great datasets for us to start our journey with. Think about when we want to train an algorithm to learn to classify handwritten digits, a supervised learning algorithm takes thousands and thousands of pictures of handwritten digits along with labels containing the correct number that each image represents. The MNIST dataset is pretty much the best standard for a labeled dataset of images of alphanumeric characters. The algorithm will then learn the relationship between the images and their associated labeled numbers, and apply that learned relationship to classify completely new images (without labels) that the machine has not seen before. The next time you deposit a check at your bank through your phone - remember that this is what they are doing.

Let’s look at how supervised learning works by checking out the problem of **predicting the annual income of a population** based exclusively on the number of years of higher education someone has completed. If I said this like a math nerd, I’d phrase it like this: “I’m looking to build a model that generalizes or approximates the relationship ***f*** between the number of years of higher education **X** and corresponding annual income **Y**.”



In this use case: **X is the (input)** which is the years of higher education

**Y is the (output)** which in this case is the annual income

***Now pay attention to this part… f… f*** is the function that is describing the relationship between X and Y. In many supervised learning algorithms … this is what we are trying to discover. If we can solve for f and learn this function then we can make predictions. Do not miss this point.

**Finally, take ϵ (epsilon)** and that equals the random error term (positive or negative) with mean zero. Said differently *epsilon… epsilon* ϵ *represents the* ***irreducible error*** *in the model, which is a theoretical limit around the performance of your algorithm based on the inherent noise in the environment or the phenomenon that you are trying to explain.*

One way that we could predict income might be to create an inflexible rules-based model for how income and education are related. For instance: *“I’d estimate that for every additional year of higher education, annual income increases by $5,000.”*

The ultimate equation would look like:

income = ($5,000 \* years\_of\_education) + baseline\_income

*If we did this - then we’d say that we* ***engineered*** *a solution (as opposed to* ***learning*** *a solution, like you could with linear regression algorithms). That is a huge point. Make sure you understand the difference between a human engineering a solution and a machine learning a solution.*

If you wanted to, you could probably engineer a complicated model that includes some rules about additional features like degree type, years of work experience, school (and if it is a top tier school), etc. For instance - we could say: *“If they completed a Bachelor’s degree or higher, give the income estimate a 1.5x multiplier.”*

While this type of thinking works well with small datasets… by creating classic rule-based programming -- it absolutely does not work well with complex or even complicated datasets. Can you invision trying to design an image classification algorithm made of if-then statements describing the combinations of pixel brightnesses that should be labeled “person” or “not person”.

Supervised machine learning crushes this problem by getting the computer to *do the work for you. Specifically by getting GPUs to do the work for you. Supervised learning* identifies patterns in the data that is provided. The machine is able to form heuristics. Let me say that again - the machine is able to form heuristics. To truly understand what that means - Let’s look at the definition of heuristics:

A heuristic technique, or simply a heuristic for short, is any approach to problem solving or self-discovery that uses a practical method that is not guaranteed to be optimal, perfect or rational, but… and here is the important part… the approach is nevertheless sufficient for reaching an immediate, short-term goal. The main difference between this and human learning is that machine learning runs on computer hardware and is best understood through the paradigm of computer science, philosophy, mathematics, probability and statistics, while human learning happens in our wetware that we call our biological brain. (In the end, both approaches accomplishes the same goals).

In a supervised learning use case, the machine is attempting to learn the relationship between income and education *from scratch. Think about that… we know what our goal is - learn the relationship between income and education? How can we do this by just providing historical data? We do this* by running **labeled training data** through a **learning algorithm.** This learning function can then be used to estimate the income of people whose income Y is unknown, just so long as we have years of education X as inputs. In other words, we can apply our model to the **unlabeled test data** to estimate Y which is our prediction. We take a mathematical equation and we train it with historical labeled data and we output a “trained model. This is what we call programming by example as opposed to programming by providing instructions.

Again, The goal of supervised machine learning is to **predict what we are calling Y as accurately as possible** when we are given new examples where X is known and Y is unknown. Let’s take a look at some of the more common approaches to doing so.

# **There are two primary goals of supervised learning: either regression or classification. We want to predict a real number ...regression… or - we want to predict what class an entity belongs to. In other words, is this flower a rose or a tulip based on the input provided.**

**Defined explicitly, Regression means that we will** predict a continuous numerical value. *How much will that car, or real estate lot, or any really, sell for?*

**If we define Classification then I’d say that it is basically a thing labeler. We want to** assign a label to a thing. *Is this a picture of a cat or a dog or a horse or a person… or maybe a person who looks like a horse… like my ex-girl friend*

The next episode - we are going to focus on Supervised Learning with Regression. Again, my name is Ernesto Lee from Ernesto.Net. Thank you and I’ll see you on the other side!

-------------------------------------

# **Hello, my name is Ernesto Lee and in this episode we are going to be discussing Supervised Machine Learning with Regression. This means that we are going to be predicting a continuous value**

Regression predicts a **continuous** **target variable that we are going to call Y as in Yellow**. Regression allows you to estimate a value...a real value... like a price - maybe real estate prices or human lifespan, based on some input data that we will call X as in X-ray. Let’s predict how long a human will live based on the input data of diet for instance or family history of illnesses.

When I say **target variable** - I mean the unknown variable that we are trying to predict. When I say c**ontinuous** - I mean there are no gaps (which mathematicians call discontinuities) in the value that our predicted value Y can take on. **Continuous Variables** would (literally) take forever to count. In fact, you would get to “forever” and never finish counting them. For example, take age. You can NEVER count “age”. **Sounds crazy right… until I explain Why not?** Because it would literally take forever. For example, you could be:

25 years, 10 months, 2 days, 5 hours, 4 seconds, 4 milliseconds, 8 nanoseconds, 99 picosends…and so on.

A **Discrete** variable by contract, can only take on a finite number of values — for example, the number of kids somebody has is a discrete variable. You *could* turn age into a discrete variable and then you could count it. For example:

* A person’s age in years.
* A baby’s age in months. Once your baby gets over 1 year...don’t count your babies age in months. You baby is not a cheese… Just tell me he or she is 1 or 2.

Continous is similar to a real number. Discrete is like an integer… for instance - there are only 10 numbers if you count from 1 to 10.

One of the oldest use cases in ML is Predicting income. This is a classic regression problem. Your **input data X will** include all of the relevant information about individuals in the data set that can be used to predict income… things like your years of education, years of work experience, job title, maybe zip code. The attributes I just listed are called **features. Features can** be **numerical** (for instance: years of work experience) or features could be **categorical** (you know - like your job title or your field of study). If it is categorical, then the best we can do is count the categories but there is no implicit “math” in the category itself.

In supervised ML with regression, You will want to have as many training observations as you can get. These should all be relating these features to the target output prediction of Y. This is so that your model can learn the relationship *f* between X and Y. Again, said differently… if I have enough training data then I can simply use the data to create an equation that I’m going to call f and this function f will show the relationship between X (the input features) and Y (the predicted output). In English… I am going to give my algorithm many many examples of what I want to see. The more examples I give my algorithm, the better it can learn and make predictions when it encounters a new scenario.

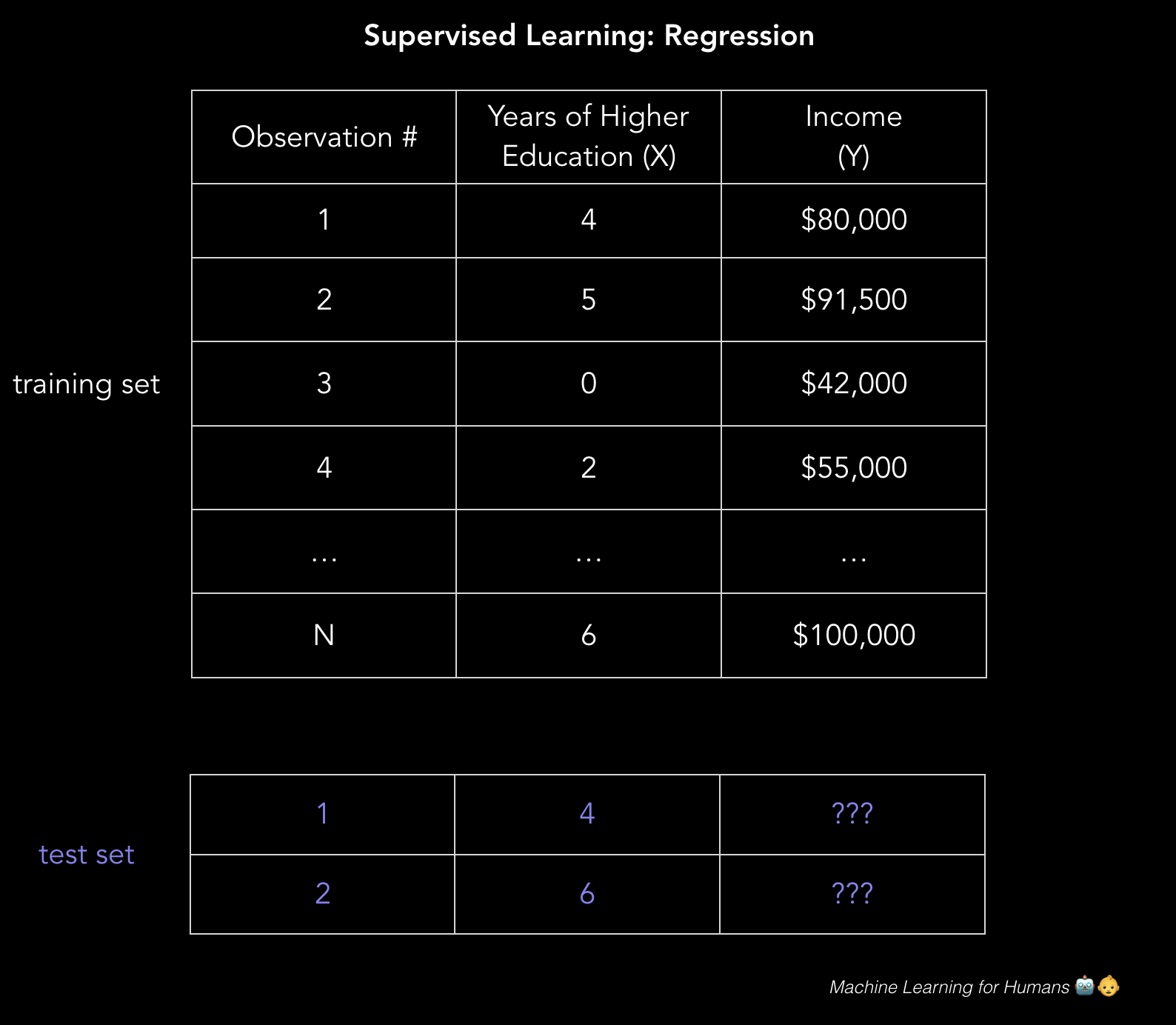
When it comes to the data that you have, the data is always split into a **training data set** and a **test data set in for Supervised Learning Machine Learning models**. The training set must have labels, so your model can learn from these labeled examples. The training set has the features and the associated predicted values so that we can train the algorithm by giving it examples of what we are looking for. The test set by contrast has the features but it does *not* have labels, i.e. you don’t yet know the value you’re trying to predict so your test data set does not have the answers… it does not have the answer so that we can give the algorithm (or model) an opportunity to predict. It’s important that your model can generalize to situations it hasn’t encountered before so that it can perform well on the test data. Again, I am calling this out as an important point. You do NOT want to train the model so that it learns how to map the data to a high degree. You want it to create a generalized model so you can make predictions over many data sets… not just the training data set. If it learns the relationships in the training data too well then it will absolutely suck with test data and unknown data. You want a generalized model so that it works well with many situations. We will go back to this when we talk about overfitting and underfitting the model.

**Here is the general equation for Linear Regression**

**Y = f(X) + ϵpsilon**, where X = (x1, x2 … all the way to xn)

*As a side Note - please know that the input value X can be a* ***tensor*** *with an any number of dimensions. Remember,* A 1 Dimensional tensor is a vector (1 row, many columns), a 2D - 2 dimensional tensor is a matrix (many rows, many columns), and then you can have tensors with 3, 4, 5 or more dimensions (e.g. a 3D tensor with rows, columns, and depth). I would recommend everyone who is planning on being a data engineer or a data scientist needs to review Linear Algebra.

In our super simple 2 dimensional example, we could take the input data as a .csv (comma separated file) where each row contains a person’s education level and income. Add more columns with more features and you’ll have a more complex, but possibly more accurate, model.



# **So here is the big question… how do we solve these types of problems where we want to predict the income based on education but we only have examples based on historical data?**

How do we build machine learning models that make accurate, useful predictions in the real world? We do this by using **supervised learning algorithms.**

**Now let’s get to the fun part: getting to know the algorithms.** We’ll explore some of the ways to approach regression and classification and illustrate key machine learning concepts throughout.

The next episode - we are going to focus on Ordinary Least Squares to help us with Linear Regression use cases. Again, my name is Ernesto Lee from Ernesto.Net. Thank you and I’ll see you on the other side!

--------------------------------

# **Hi my name is Ernesto Lee and we are going to go over Linear regression (we are going to learn this using ordinary least squares initially)**

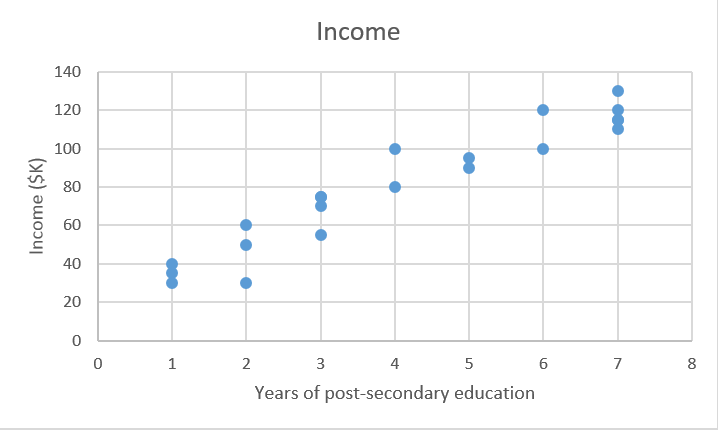
*“So think of this - if you simply look at the data and it is roughly “linear” or - if it looks roughly has a shape like a line… then use the equation for a line… y=mx + b. Then simply Draw the line and use the equation to make predictions. Yes, this counts as machine learning and yes… it is that easy.”*

First, we’ll focus on the use case of predicting income using linear regression.

We have our data set X*,* and corresponding target values or predictions Y. The goal of **ordinary least squares (OLS)** regression is to learn a linear model that we can use to predict a new predicted value of *y* given a previously unseen input value *x* with as little error as possible. We can make terrible predictions but what good what that do? The idea is to make predictions with a minimal error and a large degree of accuracy. We are trying to predict how much income someone earns based on how many years of education they received by using that data only. To do this, We need to learn this relationship.

X years of college = [4, 5, 0, 2, …, 6] #as a tensor for training

Y income based on college = [80, 91.5, 42, 55, …, 100] # as a tensor for training



Linear regression is what we call a **parametric method**. A parametric method means it makes an assumption about the form of the function relating X and Y. In other words, the data looks like a line so we’re assuming that it is linear and we use y = mx + b (we’ll cover examples of non-parametric methods later). This may or may not be a good assumption. Often people will look at a financial instrument that does well initially and the price over time looks to be steadily increasing… it looks linear… until one day it doesn’t and you’re living in a cardboard box wondering what happened. Consider how bitcoin must have looked linear until one day it didn’t. Anyway, back to our model… Our model will be a function that predicts *ŷ (y hat)* given a specific input value of *x*:



In this case, we make the explicit assumption that there is a **linear relationship** between X and Y — that is, for each one-unit increase in X, we see a constant increase (or decrease) in our prediction Y.

*Beta sub zero* is the y-intercept (and we often set that to zero so we don’t need to display it) and *Beta sub 1* is the slope of our line, i.e. how much income increases (or decreases) with one additional year of education. Beta sub 1 is the coefficient of X. It is what we are trying to solve for.

Our goal is to learn the **model parameters… it is to learn the coefficients** (in this case, *Beta sub zero* and *Beta sub 1*) that minimize the error in the model’s predictions.

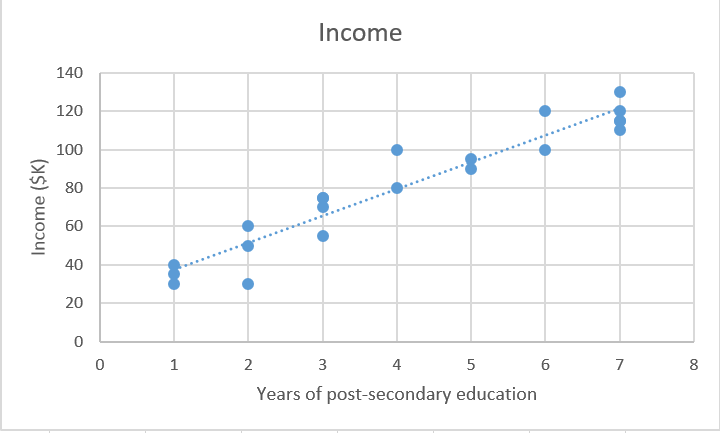
If you get that - then you need to understand that the idea is To find the best parameters that create a generalized model. I have a background in Physics and we are use to finding “the solution… the single solution” to a problem With Data Science - there is a lot of Art to this Science. You must find the BEST set of coefficients in supervised learning use cases that make the best predictions. There could be many possible solutions. So how do we find the BEST coefficients?:

*1. First you want to Define a* ***cost function****, (also called a* ***loss function)****. Cost functions measures how inaccurate your model’s predictions are.*

*2. Next you want to Find the parameters that* ***minimize loss****, i.e. make our model as accurate as possible.*

Think of this in two dimensions and this looks like a line with the best fit of the existing data. In three dimensions, we would draw a plane, and so on and so forth with higher-dimensional hyperplanes. With multi-dimensional space…. We’d create hyperplanes.

*Side bar… I am not going to go off the deep end when it comes to dimensionality: our example is two-dimensional on purpose and for simplicity. In the real world - you will almost assuredly have more features (x’s) and coefficients (betas) in your model, e.g. when adding more relevant variables to improve the accuracy of your model predictions. You may want to add the feature / dimension of location… or school ratings… or many other features that can be used to predict the price of a house. The same principles can be generalized to multiple dimensions, though things get much harder to visualize beyond three dimensions. And by much harder - I mean anything over 3 spatial and 1 time dimension becomes impossible for our human brain to visualize. Don’t even try.*

**

Back to Ordinary Least Squares method of Linear Algebra… From a Mathematical perspective, we look at the difference between each real data point (*y*) and our model’s prediction (yhat). We Square those differences simply to avoid negative numbers and we also square those numbers so that we can penalize larger differences. Again, we take a look at what the model predicted (we are calling that yhat) and we look at the actual value (since this is supervised learning and we are training … we have access to the real data at this stage)... and then we take the difference between those two numbers. Finally, we then add them up and take the average. This is the quantitative measure of how well our data fits the line. Take a look at the Cost equation:



n = # of observations. Using 2\*n instead of n makes the math work out more cleanly when taking the derivative to minimize loss, though some stats people say this is blasphemy. When you start having opinions on this kind of stuff, you know that you are definitely missing the point. Again - in this cost function… we know everything except for Beta sub 1 and Beta sub 0… that is what we are solving for.

For a simple problem like this, we can compute a closed form solution using calculus to find the optimal beta parameters that minimize our loss function. But as a cost function grows in complexity, finding a closed form solution with calculus is no longer feasible or practical. This is the motivation for an iterative approach called **gradient descent**, which allows us to minimize a complex loss function. So let me summarize - for simpler machine learning problems - we can use Ordinary Least Squares to train the model and take a look at the deltas between the real values and what was predicted and use that to square… and then sum… and then take the average and we will then have a measure of how well our model will perform. We keep doing this over and over (and using multiple cpu/gpu cycles) until we come up with a minimal number that we can accept. This is iterative until we find the coefficients that give us a best fit line.

If I didn’t mention this before… this is a concept that you want to conquer but there are libraries in place that do this work for you.

The next episode - we are going to focus on Gradient Descent to help us with Linear Regression use cases. Again, my name is Ernesto Lee from Ernesto.Net. Thank you and I’ll see you on the next episode!

------------------------------------------------------------

# **Hi, my name is Ernesto Lee and now we are going to go over Gradient descent as another way other than Ordinary Least Squares to learn the parameters or coefficients.**

*“Imagine that you have on a blindfold and you are at the top of a hill. You have been asked to find your way to the bottom of the hill. You would solve this by taking a step downhill. You’ll keep doing this until you have nowhere to go but up. This is what Gradient Descent is all about”*

Gradient descent will come up over and over again, especially in neural networks. Machine learning libraries like [scikit-learn](http://scikit-learn.org/stable/) and [TensorFlow](https://www.tensorflow.org/) use it in the background everywhere, so it’s worth understanding the details. It is worth understanding the details but you will rarely have to code it from scratch since it is in every machine learning and deep learning library.

The goal of gradient descent is to find the minimum of our model’s loss function by iteratively getting a better and better approximation of it. In English that means that we have discussed that a large part of parametric algorithms involve making an assumption about the solution and then optimizing the coefficients. Gradient Descent is a method that allows us to quickly find the coefficients.

Imagine yourself walking through a valley with a blindfold on. Your goal is to find the bottom of the valley. How would you do it?

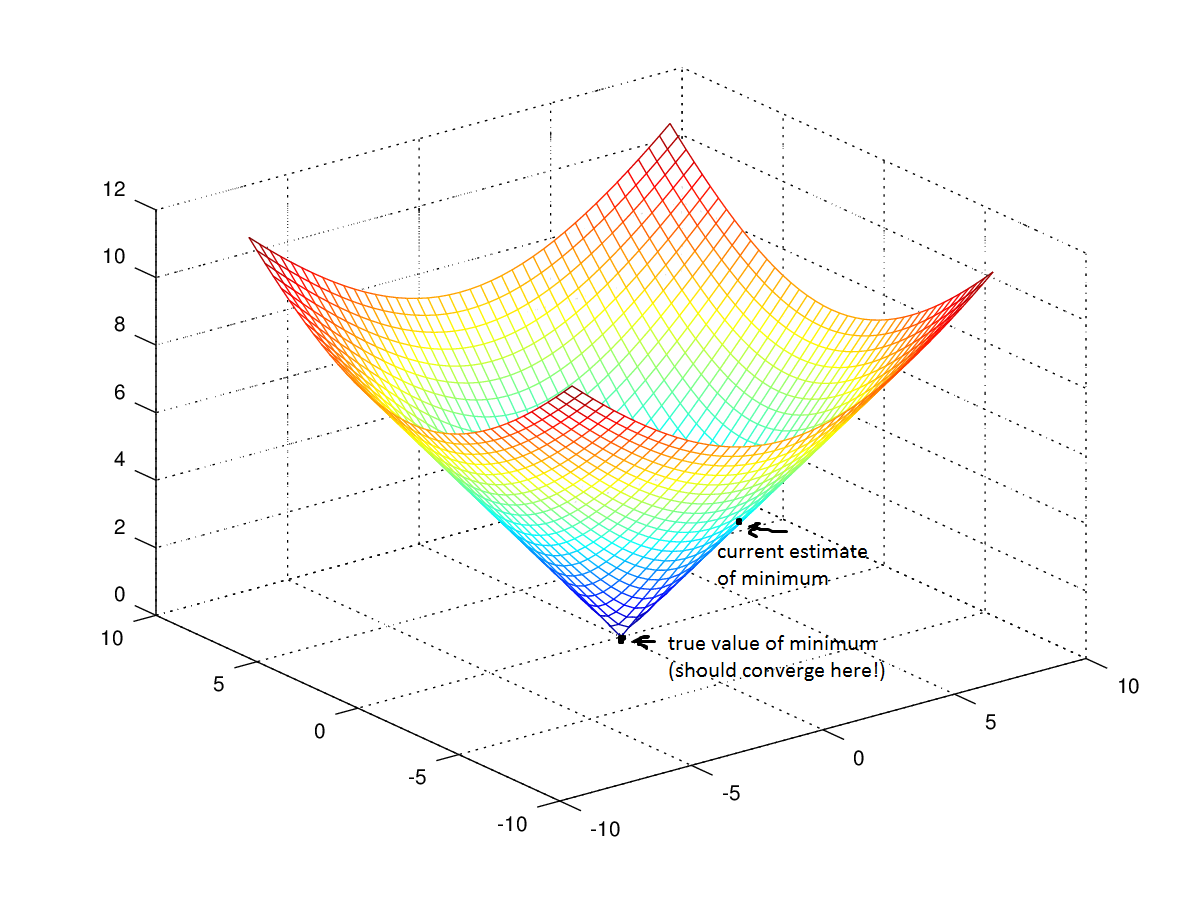
A reasonable approach would be to touch the ground around you and move in whichever direction the ground is sloping down most steeply. Take a step and repeat the same process continually until the ground is flat. Then you know you’ve reached the bottom of a valley; if you move in any direction from where you are, you’ll end up at the same elevation or you will be moving uphill.

Going back to mathematics, the ground becomes our loss function, and the elevation at the bottom of the valley is the minimum of that function. Using logic… this means that we have discovered the best coefficients for the data set that we have.

Let’s take a look at the loss function we saw in regression:



We see that this is really a function of two variables: *Beta sub 0* and *Beta sub 1*. All the rest of the variables are determined, since X, Y, and *n* are given during training. We want to try to minimize this function.



The function is f(β0,β1)=z gives a value for the loss function. For Beta sub 0 and Beta sub 1 - we continually change those values so that we can minimize the loss function. This also has the benefit of giving us the best coefficients for our data set.

Back to the math… this is the Gradient Descent process… To begin gradient descent, you make some guess of the parameters β0 and β1 that minimize the function.

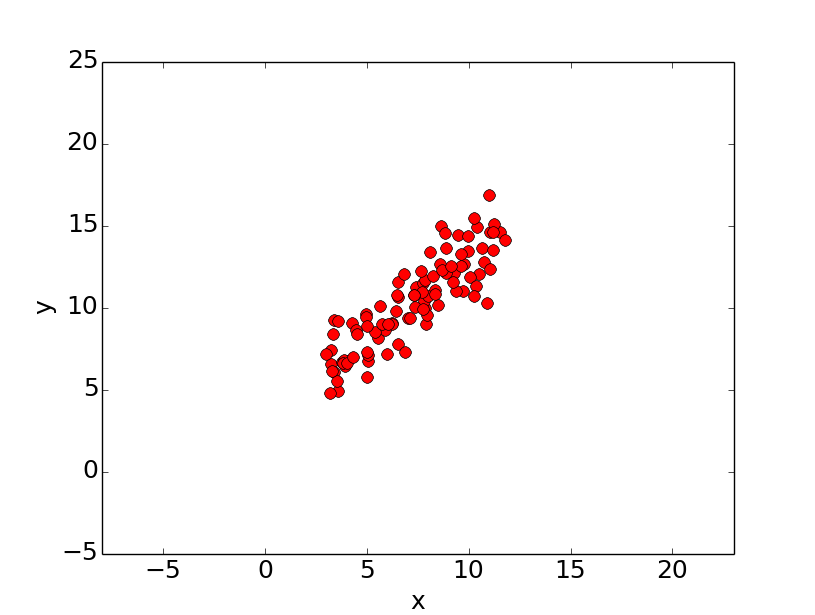
Next, you find the [partial derivatives](https://en.wikipedia.org/wiki/Partial_derivative)of the loss function with respect to each beta parameter: [*dz/dβ0, dz/dβ1*]. A **partial derivative** indicates how much total loss is increased or decreased if you increase *β0* or *β1* by a very small amount.

Put another way, how much would increasing your estimate of annual income … and we are assuming zero higher education (*β0*)... increase the loss (or the inaccuracy) of your model? You want to go in the *opposite* direction so that you end up walking *downhill* and minimizing loss. Let me say that again bc that is the heart and soul of what Gradient Descent is all about and it is a very very powerful tool that is used extensively in Machine Learning. If we are trying to learn the optimal coefficients for a particular dataset… we first find the form of the equation with unknown coefficients. Next we put it in the form of the loss function which is a measure of the inaccuracy of your model. We want to find coefficients that give us the most accurate model. We take the partial derivative of each coefficient and observe the result which will tell us if the new coefficient values makes our model MORE or LESS accurate. We move in the direction that makes our model more accurate. We do this repeatedly until we can’t find values that make our model any more accurate and you have at that point… minimized the loss function. You have found the most accurate values for the coefficients.

Let’s look at this using the real world example of predicting income based on years of college. If you increase your estimate of how much each incremental year of education affects income (*βeta sub* 1), how much does this increase loss (*z*) or the accuracy of the model? If the partial derivative of the loss function over Beta sub 1 (or *dz/β1)* is a *negative* number, then *increasing* *βeta sub 1* is good because it will reduce total loss. If it’s a *positive* number, you want to *decrease* *βeta sub 1 because that value decreases your models accuracy*. If it’s zero, don’t change *β1 because it means you’ve reached an optimum*.

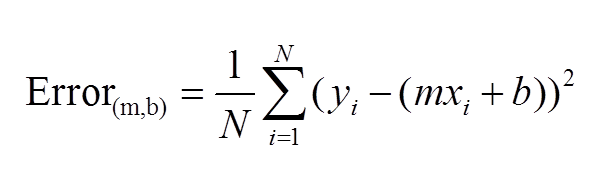
Keep doing that until you reach the bottom, i.e. the algorithm **converged** and loss has been minimized. There are lots of tricks and exceptional cases beyond the scope of this class, but generally, this is how you find the optimal **parameters** for your **parametric** model.

Do not proceed until you have an intuitive understanding of Gradient Descent. Gradient Descent is used extensively in machine learning and deep learning. To further explain Gradient Descent, Let’s suppose we want to model the points in the diagram with a line.

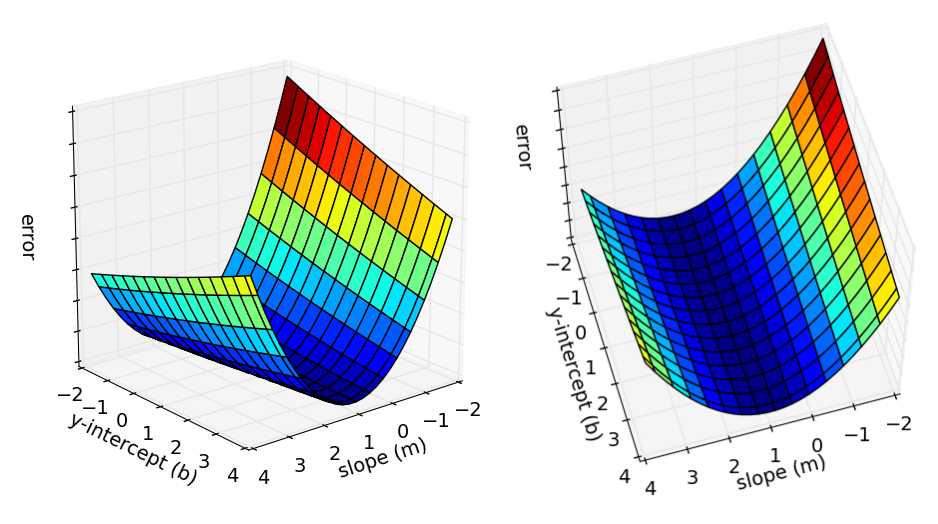


To do this we’ll use the standard y = mx + b line equation where m is the line’s slope and b is the line’s y-intercept. To find the best line for our data, we need to find the best set of slope m and y-intercept b values.

A standard approach to solving this type of problem is to define an error function (also called a cost function) that measures how “good” a given line is. This function will take in a (m,b) pair and return an error value based on how well the line fits our data. To compute this error for a given line, we’ll iterate through each (x,y) point in our data set and sum the square distances between each point’s y value and the candidate line’s y value (computed at mx + b). It’s conventional to square this distance to ensure that it is positive and to make our error function differentiable. This is Ordinary Least Squares that we discussed.

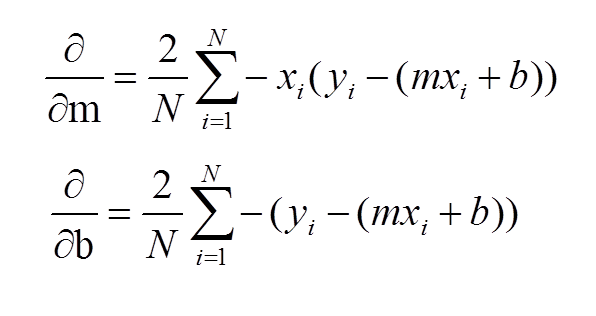


Lines that fit our data better (where better is defined by our error function) will result in lower error values. If we minimize this function, we will get the best line for our data. Since our error function consists of two parameters (m and b) we can visualize it as a two-dimensional surface. This is what it looks like for our data set:



Each point in this two-dimensional space represents a line. The height of the function at each point is the error value for that line. You can see that some lines yield smaller error values than others (i.e., fit our data better). When we run gradient descent search, we will start from some location on this surface and move downhill to find the line with the lowest error.

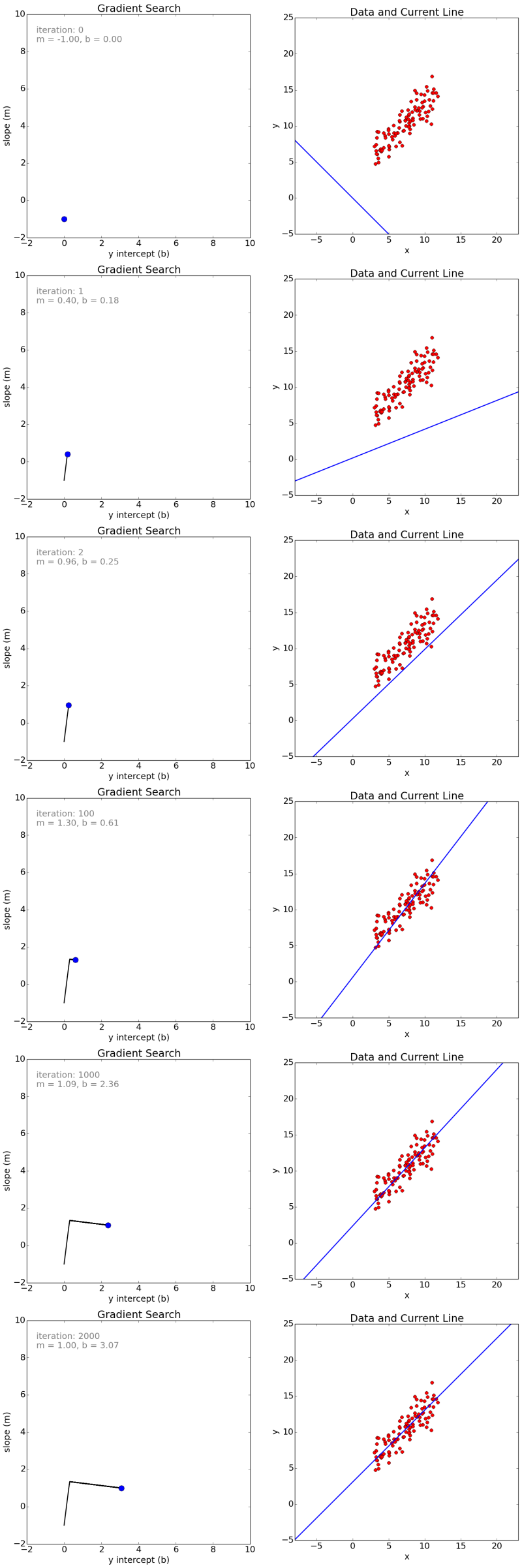
To run gradient descent on this error function, we first need to compute its gradient. The gradient will act like a compass and always point us downhill. To compute it, we will need to differentiate our error function. Since our function is defined by two parameters (m and b), we will need to compute a partial derivative for each. These derivatives work out to be:



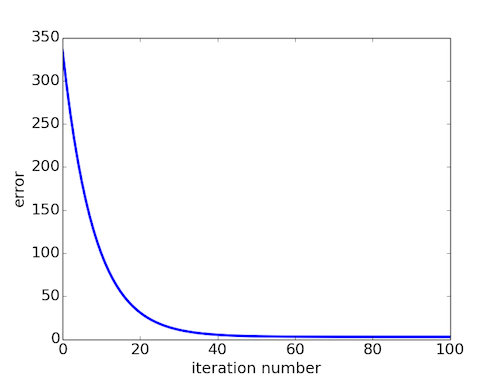
We now have all the tools needed to run gradient descent. We can initialize our search to start at any pair of m and b values (i.e., any line) and let the gradient descent algorithm march downhill on our error function towards the best line. Each iteration will update m (the slope) and b (the y-intercept) to a line that yields slightly lower error than the previous iteration. The direction to move in for each iteration is calculated using the two partial derivatives.

The learningRate variable controls how large of a step we take downhill during each iteration. If we take too large of a step, we may step over the minimum. However, if we take small steps, it will require many iterations to arrive at the minimum.

Below are some snapshots of gradient descent running for 2000 iterations for our example problem. We start out at point m = -1 b = 0. Each iteration m and b are updated to values that yield slightly lower error than the previous iteration. The left plot displays the current location of the gradient descent search (blue dot) and the path taken to get there (black line). The right plot displays the corresponding line for the current search location. Eventually we ended up with a pretty accurate fit.



We can also observe how the error changes as we move toward the minimum. A good way to ensure that gradient descent is working correctly is to make sure that the error decreases for each iteration. Below is a plot of error values for the first 100 iterations of the above gradient search.



We’ve now seen how gradient descent can be applied to solve a linear regression problem. While the model in our example was a line, the concept of minimizing a cost function to tune parameters also applies to regression problems that use higher order polynomials and other problems found around the machine learning world.

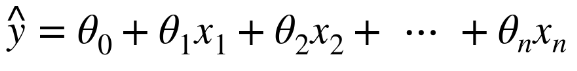
-----------------Gradient Descent Part 2 Next

So Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea (…) is to tweak parameters iteratively in order to minimize a cost function. Or, said differently, gradient descent will help us find optimal coefficients or weights for our models in a fast manner.

Let’s do a part 2 of Gradient descent because it is so important. Let’s look at an explanation of Gradient Descent that uses something real as opposed to just being abstract. But first, let’s go back to the **Cost Function.**

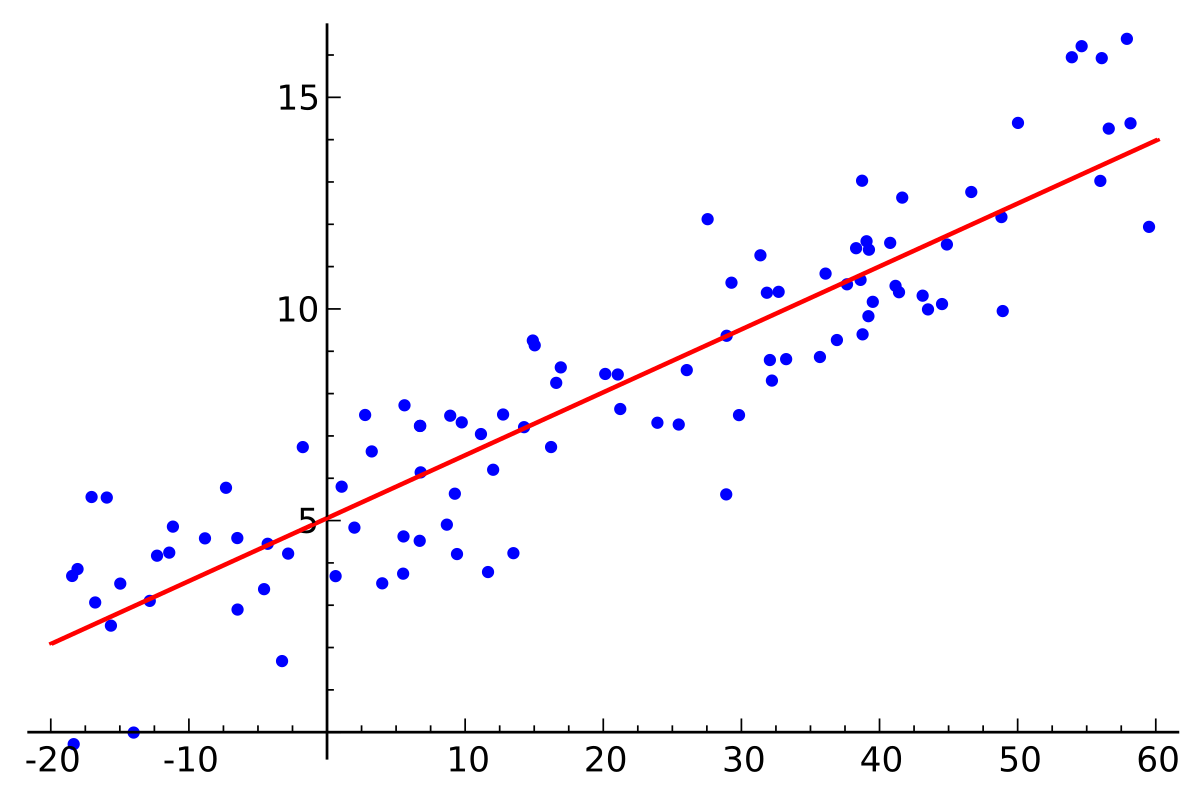
## **So What is a Cost Function?**

Let’s start this definition by taking the simplest example of a Machine Learning algorithm: Linear Regression. Linear Regression is used to estimate linear relationships between continuous or/and categorical data and a continuous output variable.



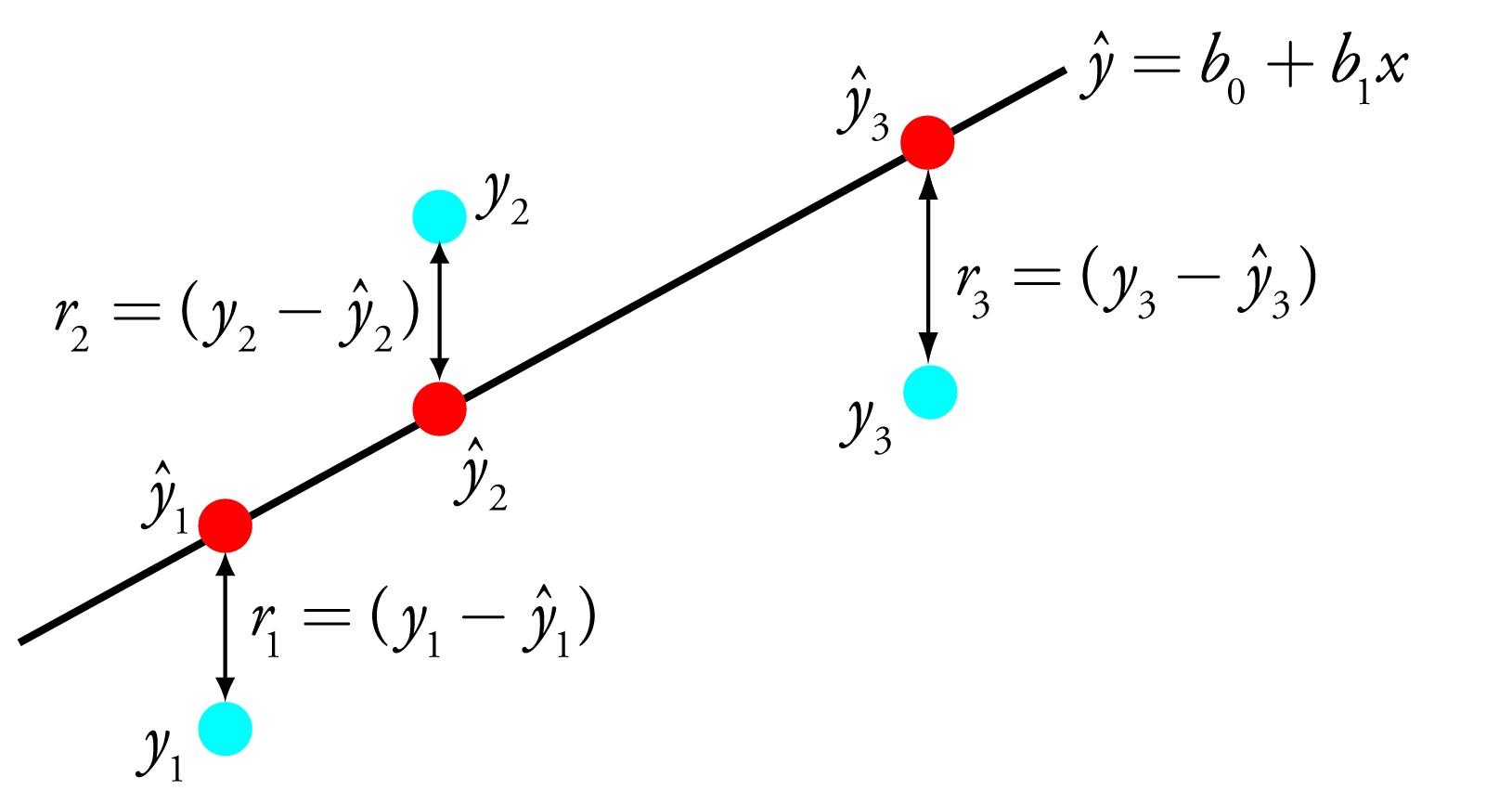
* *Y hat is the predicted value*
* *n is the number of features*
* *X sub* i *is the ith feature value*
* *(Omega sub j) is the jth model parameter (including the bias term θ sub 0 and the feature weights θ1, θ2, ⋯, θn).*

To make things simpler, let’s explain this with real world constructs. Imagine that our predicted value y is sales while X is advertising dollars. We want to create a model that can estimate how sales is a function of advertising budget.



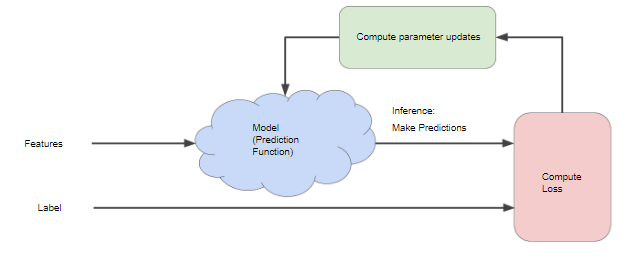
The data points on your screen show X as budget and the Y axis as sales… as you can see … Our linear model (THE LINE) generalizes all of the data available to us (blue dots). [Wikipedia](https://en.wikipedia.org/wiki/Linear_regression)

Now, what our model tried to do is to best estimate the parameter -Omega sub j**-** that would generalise for future advertising budget data, allowing us to make good predictions on the sales return. **Cost Functions** are used to measure how wrong the model is in terms of its ability to estimate the relationship between our advertising dollars X and our target prediction variable, sales y. Cost functions are the “score” that we give to how accurate our model is. Usually, the cost function is described as the difference between the predicted value given by our model and the true value.



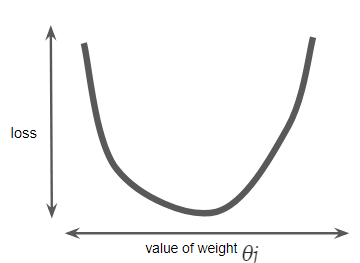
As you can see on your screen - it is about the Difference or distance between the predicted value (red) and the actual value (blue)[. Source](https://towardsdatascience.com/simple-linear-regression-2421076a5892)

*Therefore, the goal of a Machine Learning algorithm is to find the best parameters -Omega sub j -* *which will minimise the cost function. It is about finding the Optimal coefficient for the data and the cost function is our measure for how we can concretely say what is the best optimization.*

**

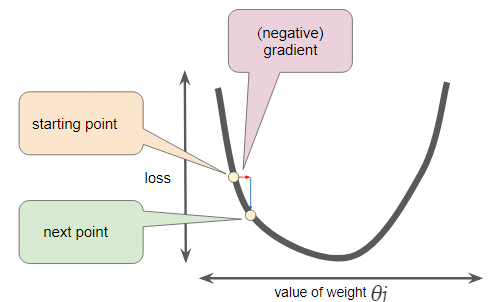
## **Gradient Descent is used to find the optimal coefficients … or weights… for a dataset which therefore : Minimises the Cost Function and maximizes the accuracy.**

Hopefully, you’re starting to see why the cost function is and how Gradient Descent can be used in Machine Learning. Gradient descent is an efficient optimisation algorithm that attempts to find what is called, the global minima of a cost function. It attempts to find the “BEST” coefficient value. According to [Aurélien](https://medium.com/u/c939be75faee?source=post_page-----235a6c8d26b0----------------------), the gradient descent *“measures the local gradient of the error function with regards to the parameter vector* ***Omega sub j****, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached the minimum.”* For most regression problems, the resulting plot of *“Cost Function vs the Parameter Omega “θ”* will always be convex and therefore having only a single minimum where the slope is exactly zero and also where the cost function converges.



Calculating the cost function parameters for all Omega sub j’s over the entire data set would be an inefficient way of finding the convergence point. **This is where gradient descent does its magic! But how does it work?**

You start by randomly fill the parameter *θ* with random values (also known as random initialisation). At this point, the gradient descent algorithm calculates the gradient of the loss curve at the starting point, which is the derivative (slope) of the curve. In the end, it gives you the direction of your next step. It serves as a compass.



You repeat this process incrementally, each step at a time, trying to reduce the cost function until your algorithm converges to a minimum as shown in this diagram on your screen. When it finds this… this is the best value for your model.



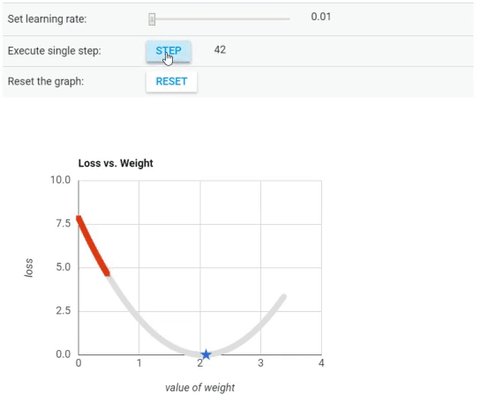
Cost Function vs Value of Weight (parameter θ). [Source](https://developers.google.com/machine-learning/crash-course/fitter/graph)

At this point (local minimum) the model has optimised the parameters such that it minimise to a minimum the cost function.

**Let’s talk about the Importance of the Learning Rate**

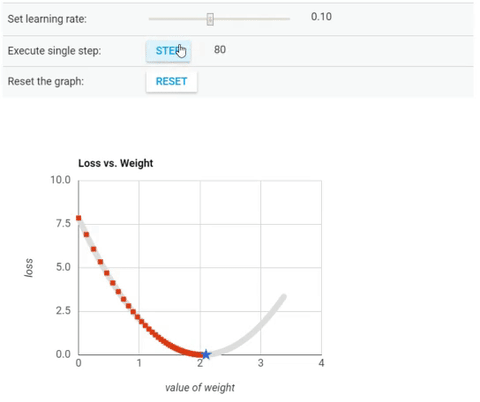
We’ve seen so far that the gradient vector has both a direction and a magnitude (red arrow). The gradient descent algorithm multiplies the gradient by a scalar known as **learning rate** (or step size). So, the learning rate is the hyperparameter that the algorithm uses to converge either by taking small steps (much more computational time) or larger steps. See the following images provided by google on your screen to understand the impact of selecting different learning rates.

* **Learning rate = 0.10**

****

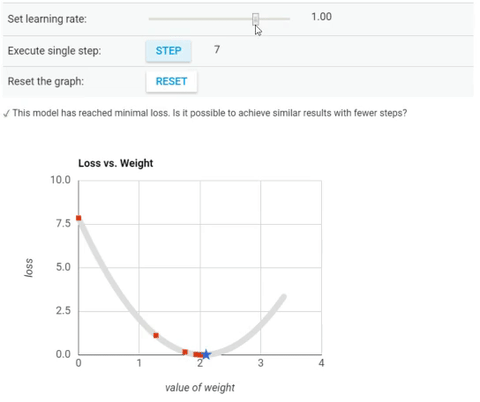
Cost Function vs Value of Weight (parameter θ). Learning Rate = 0.10. [Source](https://developers.google.com/machine-learning/crash-course/fitter/graph)

* **Learning rate = 1.0**

****

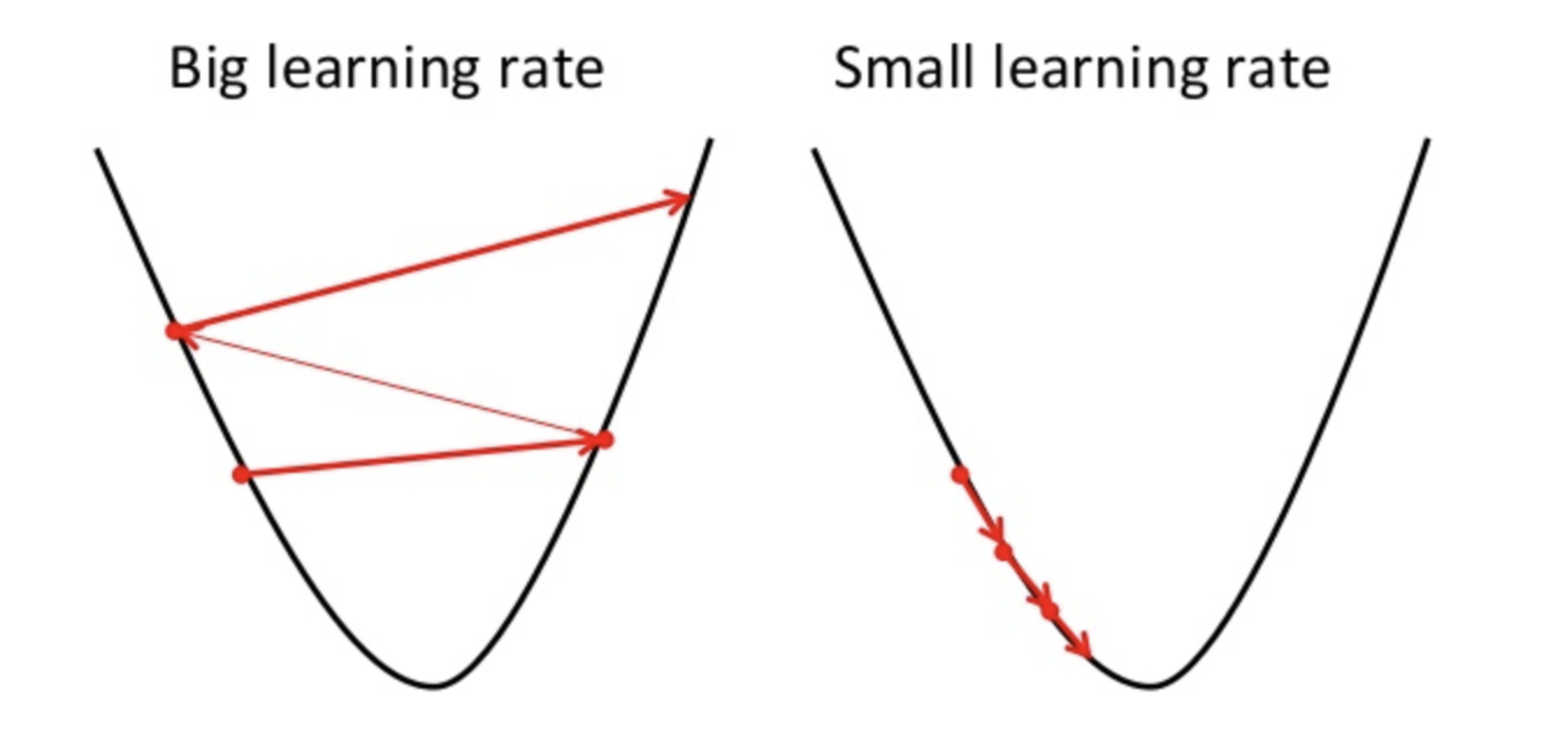
Cost Function vs Value of Weight (parameter θ). Learning Rate = 1. [Source](https://developers.google.com/machine-learning/crash-course/fitter/graph)

* **Learning rate = 1.6**

****

Cost Function vs Value of Weight (parameter θ). Learning Rate = 1.6. [Source](https://developers.google.com/machine-learning/crash-course/fitter/graph)

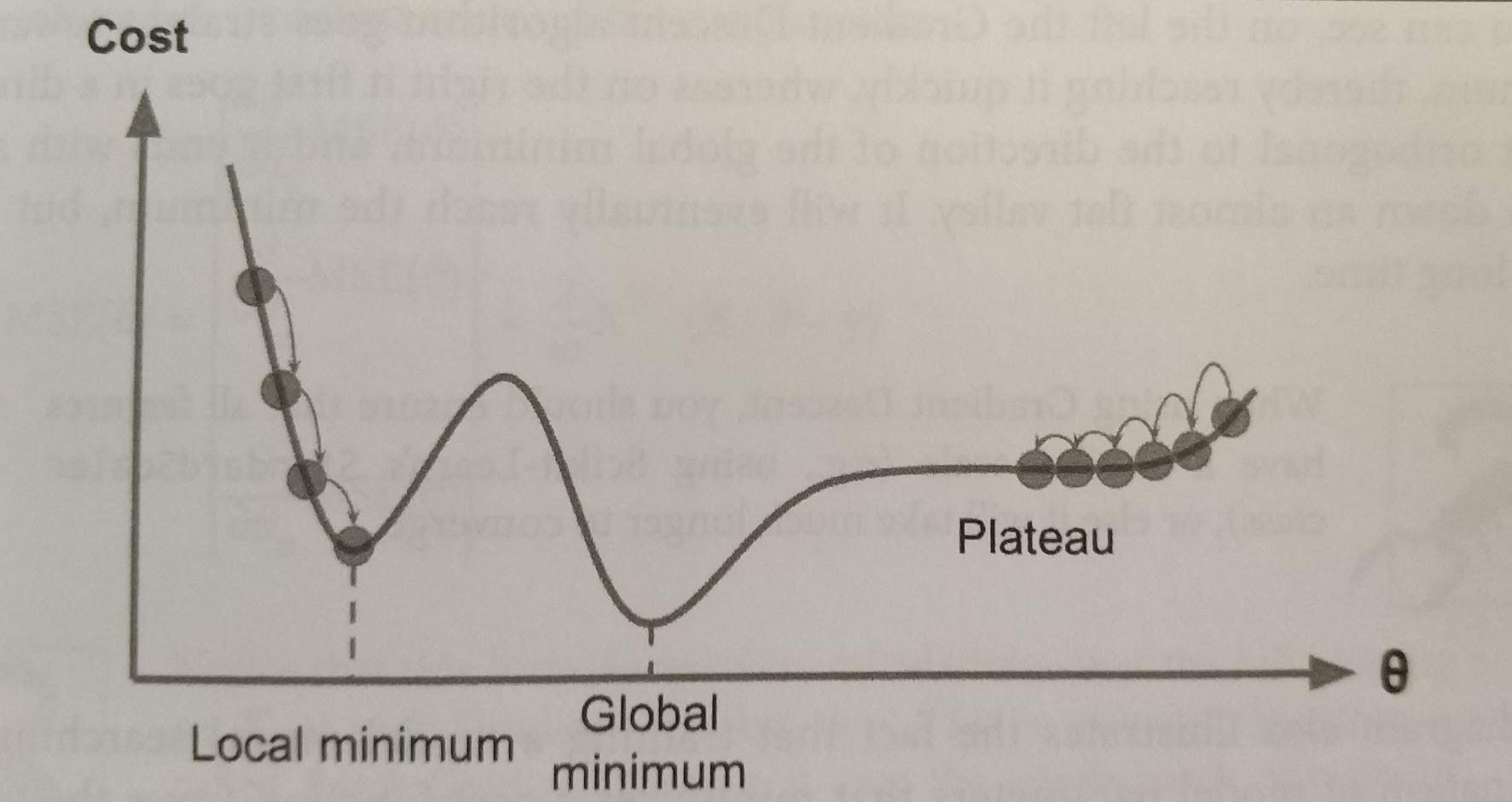
The examples show great learning rates that allowed us to achieve the cost function convergence point slower or faster. Nevertheless, just be careful when choosing higher learning rates since this might cause the algorithm to diverge, with larger and larger values, failing to find a good solution. Take a look at the image on your screen for an example.



[Source](https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0)

Moreover, you should also have in mind that not all cost functions look like nice regular bowls. That usually only happens for Linear Regression, and it is great because regardless of your initial assumption for parameter -Omega sub j***-*** it will always converge. With other Machine Learning algorithms such as Neural Networks you might find cost functions with irregularities making it hard to converge.

Take into consideration the image on your screen. If our random initialisation for our parameter ***Omega sub j*** starts on the left then our algorithm will converge to a minimum which is not the best. However, if it starts on the right it will take longer to do the plateau and eventually it might reach the right minimum if you do not stop early.



Hands-on Machine Learning with Scikit-Learn & TensorFlow

*An important fact you should have in mind is that all features should have similar scales in order to ensure the algorithm will not take more than necessary to converge. This means data normalization or standardization is critical as with all things AI.*

## **The types of Gradient Descent**

There are three popular gradient descent types that mainly differ for the amount of data they handle.

* **Batch Gradient Descent**

In gradient descent, the **batch** is the total number of examples you use to calculate the gradient in a single iteration. This algorithm does its calculations over the full training dataset, at each step of the gradient descent. Therefore, it uses the whole dataset at every step, making it very very very slow for large datasets.

**Pros:** Computational efficient, since it produces a stable error gradient and a stable convergence.

**Cons:** Requires that the training dataset is in memory and available to the algorithm. It is slow since it uses the whole training set to compute the gradient at every learning step.

* **Stochastic Gradient Descent**

This alternative to Batch Gradient Descent, it is on the other extreme of the idea, using a single example (batch of 1) per each learning step. Of course, due to this methodology ,the algorithm is much faster since it has less data to manipulate at every iteration, however it can return very noisy gradients which can cause the error rate to jump around. Therefore, due to this jumping around when the algorithm stops instead of finding the optimal fit it ends up obtaining only good results.

**Pros:** Constant update allows us to have a detailed rate of improvement. Allows usage of huge datasets since only one instance is needed to be allocated to the memory at each step.

When the cost function is very irregular such as in the last image, due to the jumping around this algorithm behaves better.

**Cons:** Due to its stochastic (i.e. random) nature, the algorithm is much less regular than the previous one. Instead of slowly decreasing the cost function until it reaches the desired value, the Stochastic bounces up and down only decreasing on average. Jumping around is good to escape local minimum but bad because hardly the algorithm will ever settle at the global minimum.

* **Mini Batch Gradient Descent**

**This is the go-to method!** Contrary to the two last gradient descent algorithms that we saw, instead of using the full dataset or a single instance, the Mini Batch, as the name indicates ,computes the gradients on small random sets of instances called mini batches.

This algorithm is able to reduce the noise from the Stochastic and still be more efficient than full-batch. Basically, it splits the training dataset into small batches ranging from 10 to 1,000 examples, chosen at random.

This algorithm is the best option when you are training a neural network and it is the most common within the Deep Learning field.

When it comes to Gradient Descent - Efficient is the word we’re looking for, not effective.

Gradient descent is a primitive algorithm. It merrily walks down the hill in the steepest direction until it gets to a bottom. Then it just stops.

Optimization is usually a delicate balancing act between exploitation and exploration, but gradient descent doesn’t care about exploration. It keeps walking down without bothering to observe its surroundings in the search space.

This simplistic approach has the unfortunate consequence that global convergence can’t be guaranteed. On the other hand, it makes gradient descent fast.

There are many heuristic optimization algorithms with better convergence guarantees than gradient descent, but they aren’t nearly as computationally efficient.

Until we have faster computers, gradient descent is a good compromise for training large neural networks.

And it’s not all that bad. We have simple techniques to introduce some randomness to the optimization process, like random sampling and dropout.

Gradient descent is an efficient optimization algorithm that allows us to train neural networks on huge datasets in reasonable time, and it has proven itself to work well in practice.

The next episode - we are going to focus on Overfitting and Hyperparameters to help us with Linear Regression use cases. Again, my name is Ernesto Lee from Ernesto.Net. Thank you and I’ll see you on the next episode!

-------------------Overfitting Next

# **Hi, my name is Ernesto Lee and now we are going to go over** **Overfitting and Hyperparameters**

***When you think of Overfitting consider:*** *“You can often tell when someone is lying because they are too specific. They add details that are completely random. Their story sounds rehearsed.*

***Regularization simply means:*** *“Don’t overcomplicate things, You will be punished for extra complexity. Make your models as simple as possilble and absolutely no simler. Again - you will be punished for extra complexity.*

***When I say Hyperparameter (λ):*** *“I am talking about how much I will be punishing you for extra complexity.*

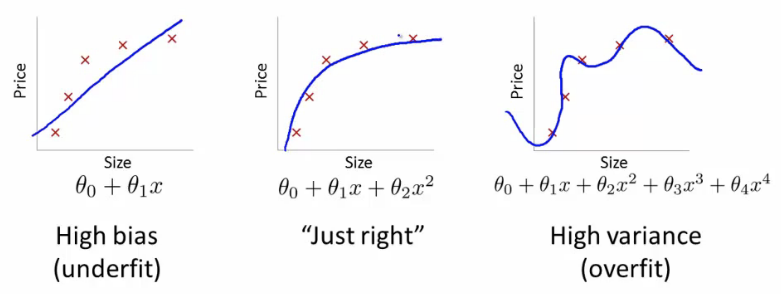
A common problem in machine learning is **overfitting**: learning a function that perfectly explains the training data that the model learned from, but does not generalize well to unseen test data. Overfitting happens when a model *overlearns* from the training data to the point that it starts picking up idiosyncrasies that aren’t representative of patterns in the real world. This becomes especially problematic as you make your model increasingly complex. *Underfitting* is a related issue where your model is not complex enough to capture the underlying trend in the data.

**This is where the Bias-Variance Tradeoff comes in**

**Bias** is the amount of error introduced by approximating real-world phenomena with a simplified model.

**Variance** is how much your model's test error changes based on variation in the training data. It reflects the model's sensitivity to the idiosyncrasies of the data set it was trained on.

As a model increases in complexity and it becomes more wiggly (**flexible)**, its bias decreases (it does a good job of explaining the training data), but variance increases (it doesn't generalize as well). **Ultimately, in order to have a good model, you need one with low bias *and* low variance.**

****

Source: Coursera’s [ML course](https://medium.com/@v_maini/machine-learning-for-humans-part-1-why-machine-learning-matters-965dfadf19d4), taught by Andrew Ng

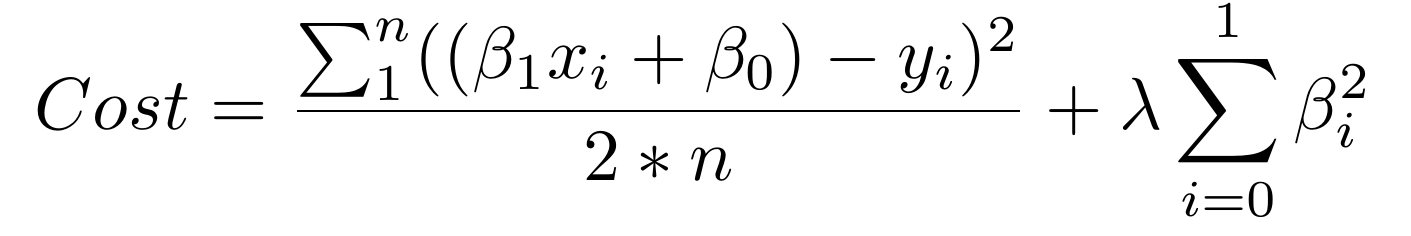
Remember that *the only thing we care about is how the model performs on test data*. You want to predictwhich emails will be marked as spam before they’re marked, not just build a model that is 100% accurate at reclassifying the emails it used to build itself in the first place. Hindsight is 20/20 — the real question is whether the lessons learned will help in the future.

The model on the right has zero loss for the *training data* because it perfectly fits every data point. But the lesson doesn’t generalize. It would do a horrible job at explaining a new data point that isn’t yet on the line.

Two ways to combat overfitting:

**1. Use more training data**. The more you have, the harder it is to overfit the data by learning too much from any single training example.

**2. Use regularization**. Add in a penalty in the loss function for building a model that assigns too much explanatory power to any one feature or allows too many features to be taken into account.



The first piece of the sum above is our normal cost function. The second piece is a **regularization term** that adds a penalty for large beta coefficients that give too much explanatory power to any specific feature. With these two elements in place, the cost function now balances between two priorities: explaining the training data and preventing that explanation from becoming overly specific.

The **lambda** coefficient of the regularization term in the cost function is a **hyperparameter:** a general setting of your model that can be increased or decreased (i.e. **tuned**)in order to improve performance. A higher lambda value will more harshly penalize large beta coefficients that could lead to potential overfitting. To decide the best value of lambda, you’d use a method called **cross-validation** which involves holding out a portion of the training data during training, and then seeing how well your model explains the held-out portion. We’ll go over this in more depth in future episodes.

# **Woo! We made it.**

Here’s what we covered in this section:

* How **supervised machine learning** enables computers to learn from labeled training data without being explicitly programmed
* The tasks of supervised learning: **regression** and **classification**
* **Linear regression,** a bread-and-butter **parametric** algorithm
* Learning **parameters** with **gradient descent**
* **Overfitting** and **regularization**

In the next section — [Part 2.2: Supervised Learning II](https://medium.com/@v_maini/supervised-learning-2-5c1c23f3560d) — we’ll talk about two foundational methods of classification: **logistic regression** and **support vector machines**.

# **Practice materials & further reading**

## **2.1a — Linear regression**

*For a more thorough treatment of linear regression, read chapters 1–3 of* [*An Introduction to Statistical Learning*](http://www-bcf.usc.edu/~gareth/ISL/index.html)*. The book is available for free online and is an excellent resource for understanding machine learning concepts with accompanying exercises.*

*For more practice:*

* *Play with the* [*Boston Housing dataset*](http://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html)*. You can either use software with nice GUIs like Minitab and Excel or do it the hard (but more rewarding) way with* [*Python*](https://medium.com/@haydar_ai/learning-data-science-day-9-linear-regression-on-boston-housing-dataset-cd62a80775ef) *or* [*R*](https://datascienceplus.com/linear-regression-from-scratch-in-r/)*.*
* *Try your hand at a* [*Kaggle*](http://www.kaggle.com/) *challenge, e.g.* [*housing price prediction*](https://www.kaggle.com/c/house-prices-advanced-regression-techniques)*, and see how others approached the problem after attempting it yourself.*

## **2.1b — Implementing gradient descent**

*To actually implement gradient descent in Python, check out* [*this tutorial*](https://spin.atomicobject.com/2014/06/24/gradient-descent-linear-regression/)*. And* [*here*](http://eli.thegreenplace.net/2016/understanding-gradient-descent/) *is a more mathematically rigorous description of the same concepts.*

*In practice, you’ll rarely need to implement gradient descent from scratch, but understanding how it works behind the scenes will allow you to use it more effectively and understand why things break when they do.*