# **Under the Hood**

## **Plan**

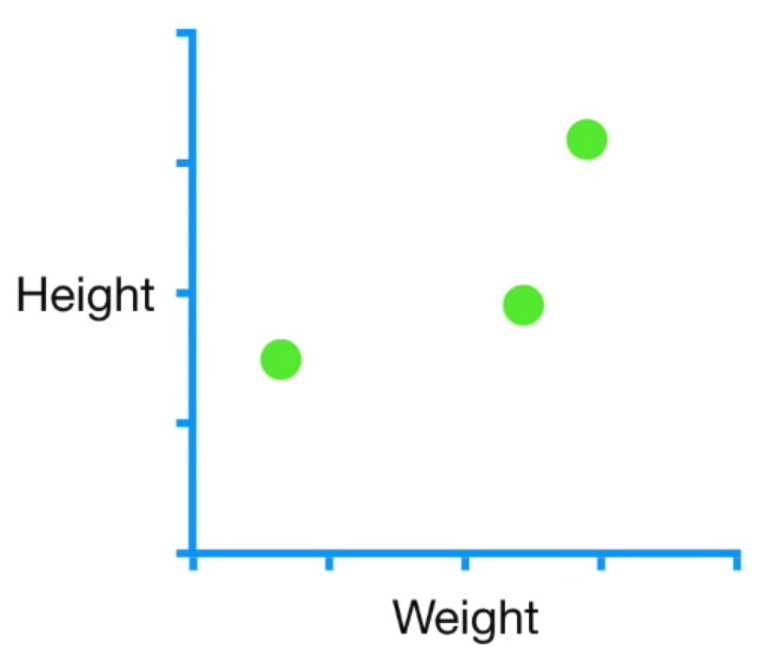
1. What happens behind .fit()?
2. Gradient Descent
3. Other Solvers
4. Loss Functions

# **1. What happens behind .fit()?**

Consider the following data:

data

|  | **weight** | **height** |
| --- | --- | --- |
| **0** | 0.7 | 1.5 |
| **1** | 2.4 | 1.8 |
| **2** | 2.8 | 3.2 |



Train a Linear Regression model:

**from** **sklearn.linear\_model** **import** LinearRegression

*# Instantiate Linear model*

model = LinearRegression()

*# Train Linear Model*

model.fit(data[['weight']], data['height'])

LinearRegression()

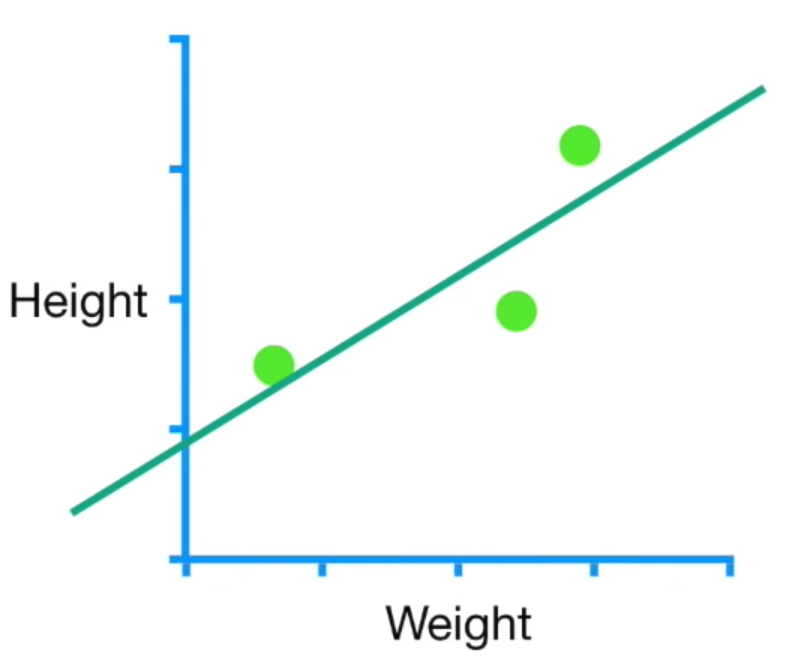
The model is now trained and its optimal parameters can be accessed.

print('beta\_0 (intercept) =', model.intercept\_)

print('beta\_1 (slope) =', model.coef\_[0])

beta\_0 (intercept) = 0.9434316353887398

beta\_1 (slope) = 0.6219839142091154



### **What happens during .fit()?**

Any model can be expressed as

y

=

h

(

X

,

β

)

+

e

r

r

o

r

* h
* is called our **hypothesis** function
* h
* (
* X
* ,
* β
* )
* is called our **prediction** (
* ^
* y
* )
* h
* (
* X
* ,
* β
* )
* =
* β
* 0
* +
* β
* 1
* X
* 1
* in our example

👉 .fit() finds parameters

β

0

and

β

1

which **minimize** the

e

r

r

o

r

(

X

,

y

,

β

)

❓ Which **norm** is used to measure error in

R

numbers?

**The Loss Function L**

.fit() minimizes

L

(

e

r

r

o

r

)

L

O

L

S

=

∥

e

r

r

o

r

∥

2

=

∥

y

−

β

0

−

β

1

X

1

∥

2

We often write:

β

=

arg

min

β

L

(

β

,

X

,

y

,

h

)

**There are numerous "solvers" to minimize L(**

β

**)** beyond Gradient Descent

* Exact mathematical resolution 👉 matrix inversion, often too complex, thus only used in "simple" ML models like [SVD in Linear Regression](https://sthalles.github.io/svd-for-regression/)
* Iterative approaches

*# In Sklearn, these methods are called "solvers"*

LogisticRegression(solver='newton-cg')

✏️ **Let's try to think about our solver**

Imagine that we already know the value of the ideal slope (

β

1

=

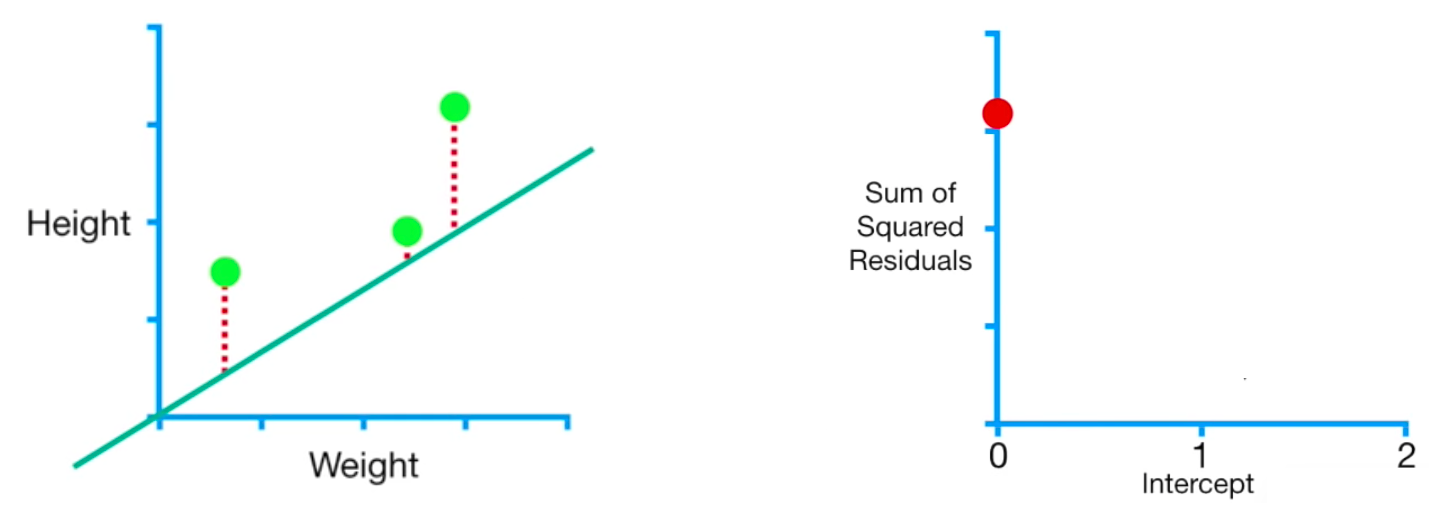
0.64

), and need to find the optimal intercept (

β

0

):

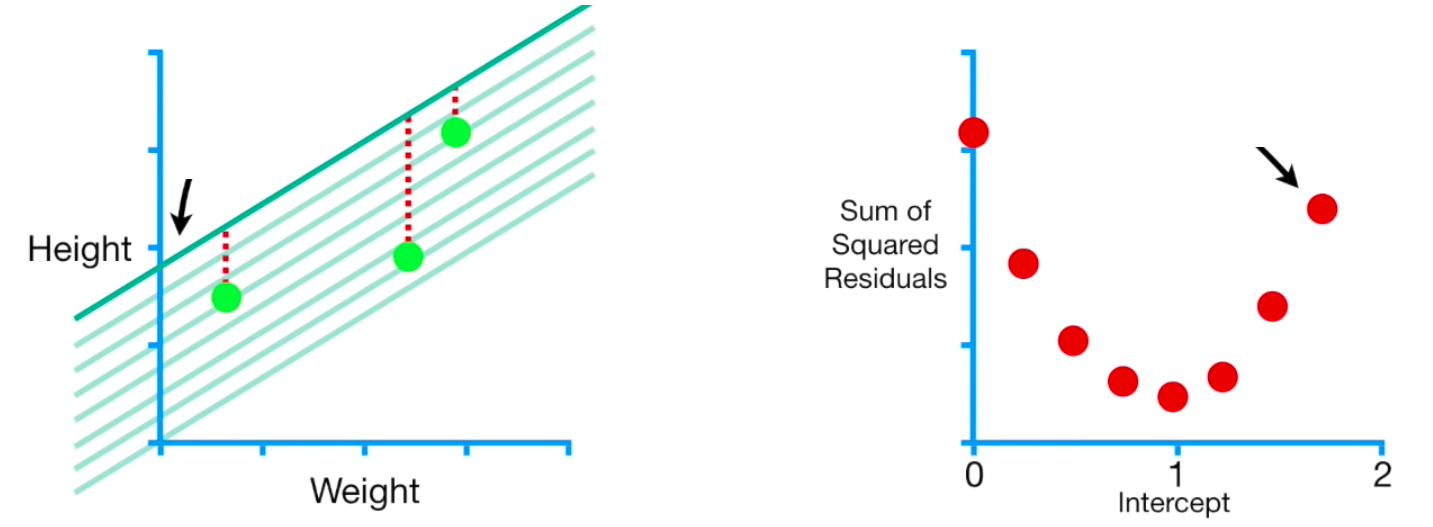


Steps:

1. Randomly initialize an **intercept**, say at 0
2. Compute the **loss** at that intercept value; here, the loss is the **Sum of Squared Residuals (SSR)**

3. Change the **intercept** and repeat the process until we find the smallest loss

If we look at the Loss Function, we see that it has a convex shape 👇



⚠️ Problems:

* We could miss the exact minimum if our steps are too large
* We don't know the best
* β
* 1
* to start with

👉 We need to tweak both

β

0

and

β

1

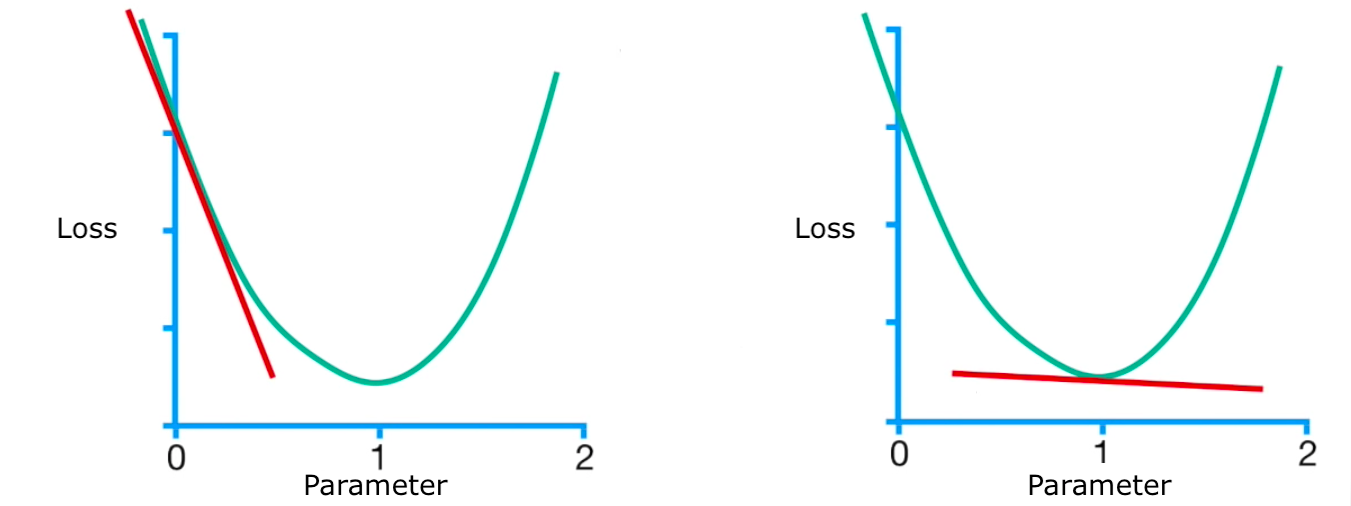
simultaneously in each iteration

So let's discover the most basic but very powerful iterative method: the **Gradient Descent**

# **2. Gradient Descent**

## **2.1 1D Descent Step-by-Step**

* Uses the **slope (gradient)** of the Loss Function as an indicator
* As the slope approaches zero, the loss approaches its minimum



The slope is equal to the **partial derivative** of the Loss Function with respect to the parameter of interest:

∂

L

o

s

s

F

u

n

c

t

i

o

n

∂

p

a

r

a

m

e

t

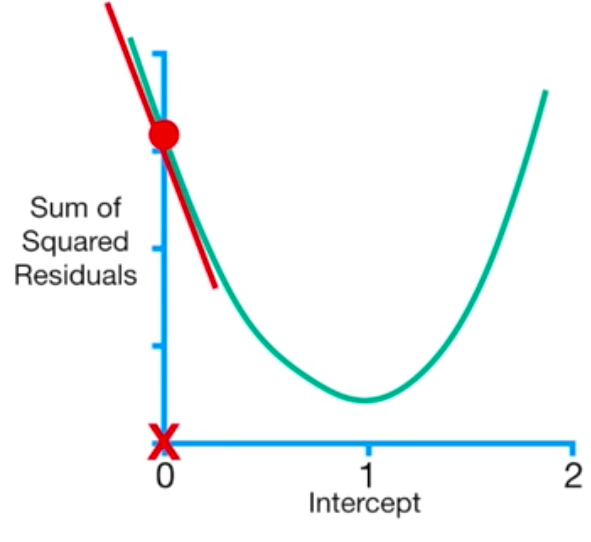
e

r

✏️ Let's go back to our example

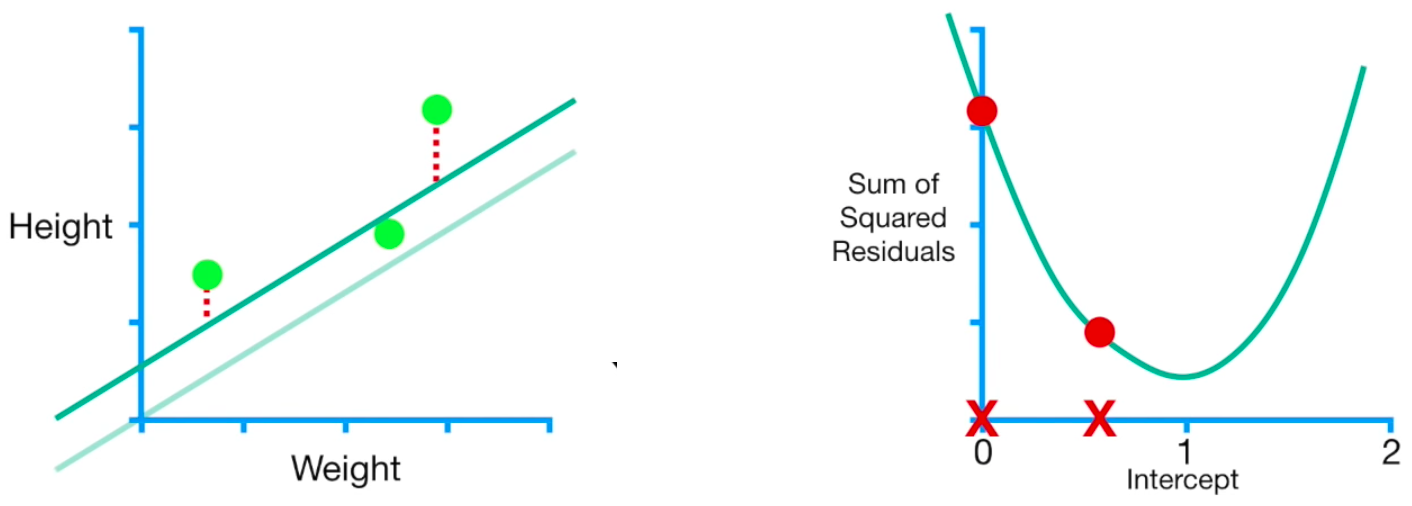
**Step One**

* Initialize a random parameter value, say
* β
* 0
* =
* 0
* Calculate the derivative of the Loss Function at that point 👉
* ∂
* S
* S
* R
* ∂
* β
* 0
* (
* 0
* )



**Step Two**

* move in the opposite direction of the derivative by one **step**

****

**Note:**

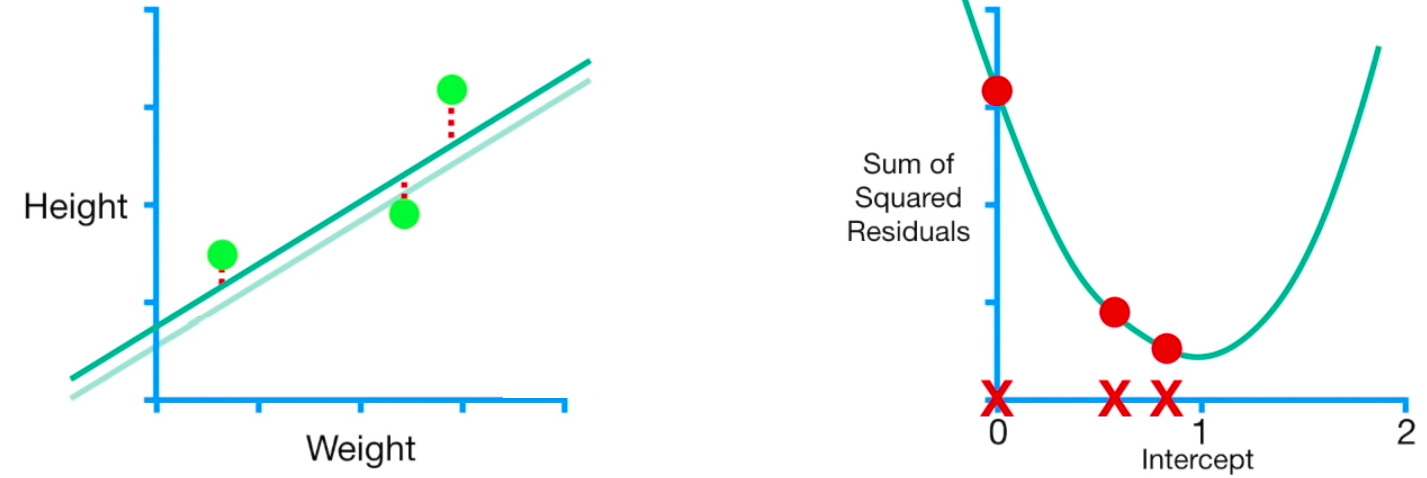
* the step size is **proportional** to the derivative's value
* it moves according to a chosen **Learning Rate** =
* η
* (eta)
* β
* (
* 1
* )
* 0
* =
* 0
* −
* η
* ∂
* L
* ∂
* β
* 0
* (
* 0
* )

We just did one

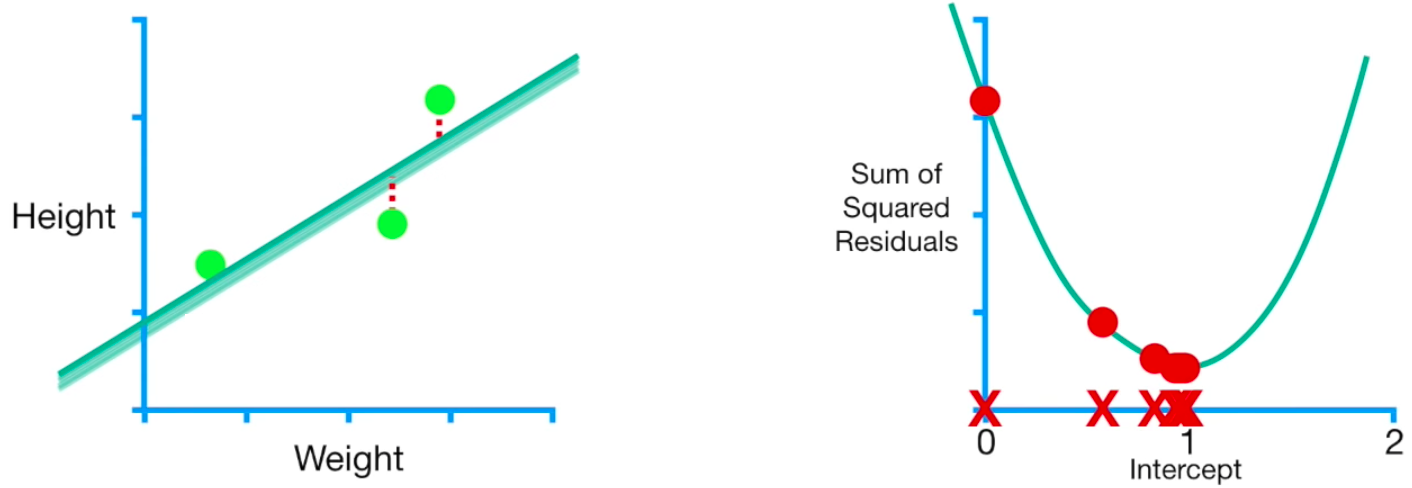
epoch

! Now we repeat the process.

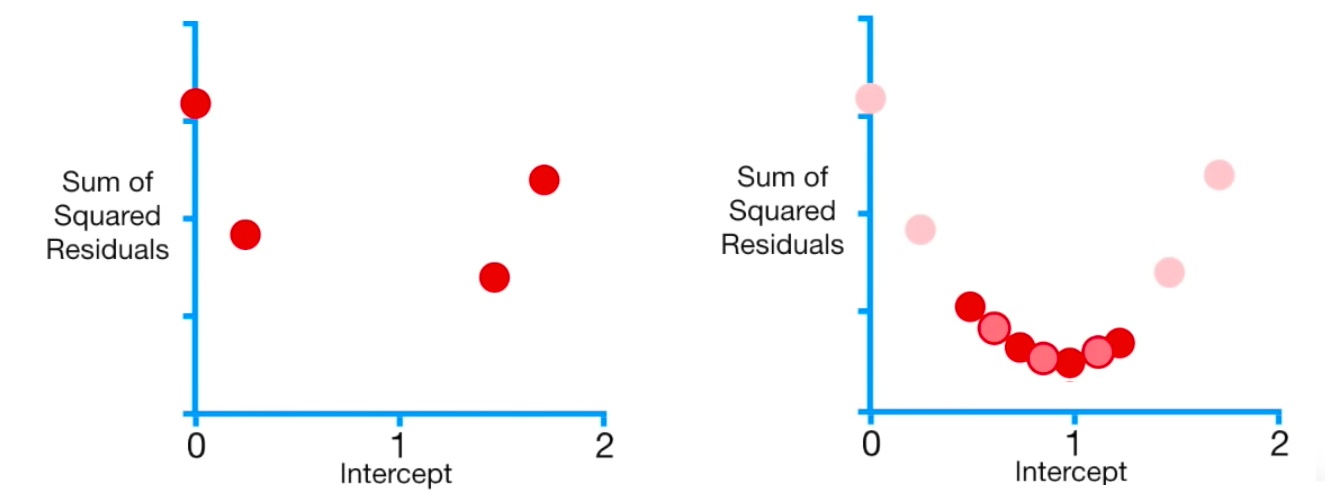
The updated intercept value is plugged back into the derivative of the Loss Function, and we repeat the process



As the **loss** approaches its minimum, the derivative gets smaller, and so do the steps



This makes the **Gradient Descent** computationally efficient. It does few calculations far away from the minimum, and more calculations as it approaches the minimum of the Loss Function.



#### **When does it stop?**

The Gradient Descent algorithm can have different **stopping criteria**:

* **Minimum Step Size** (e.g. 0.001). When the step size is smaller than this threshold, the Gradient Descent has converged, and the corresponding intercept is the optimal value
* **Maximum Number of steps** (e.g. 1000)

**1D Descent Summary**

β

(

k

+

1

)

0

=

β

(

k

)

0

−

η

∂

L

∂

β

0

(

β

(

k

)

0

)

1. Randomly initialize the parameter value
2. β
3. (
4. 0
5. )
6. 0
7. Compute the derivative of the Loss Function at that point
8. Update the parameter value according to the step size
9. η
10. Go back to step 2 with the updated value of the parameter

Repeat steps 2 to 4 until the Gradient Descent hits the stopping criterion of your choice (either Minimum Step Size or Maximum Number of Steps)

### **Analytical Solution**

Can we compute

∂

S

S

R

∂

β

0

for our example?

S

S

R

(

β

)

=

n

∑

i

=

1

(

y

i

−

^

y

i

)

2

=

n

∑

i

=

1

(

y

i

−

(

β

0

+

β

1

X

(

i

)

1

)

)

2

We can compute its partial derivative with respect to

β

0

∂

S

S

R

∂

β

0

=

n

∑

i

=

1

−

2

(

y

i

−

(

β

0

+

β

1

X

(

i

)

1

)

)

=

n

∑

i

=

1

−

2

(

y

i

−

^

y

i

)

💡

f

(

g

)

′

=

g

′

∗

f

′

(

g

)

### **✏️ We can now code the Gradient Descent for our example**

X = data['weight']

y = data['height']

b1 = 0.64

eta = 0.1 *# Learning Rate*

*# Hypothesis function h*

**def** h(x, b0):

**return** b0 + b1 \* x

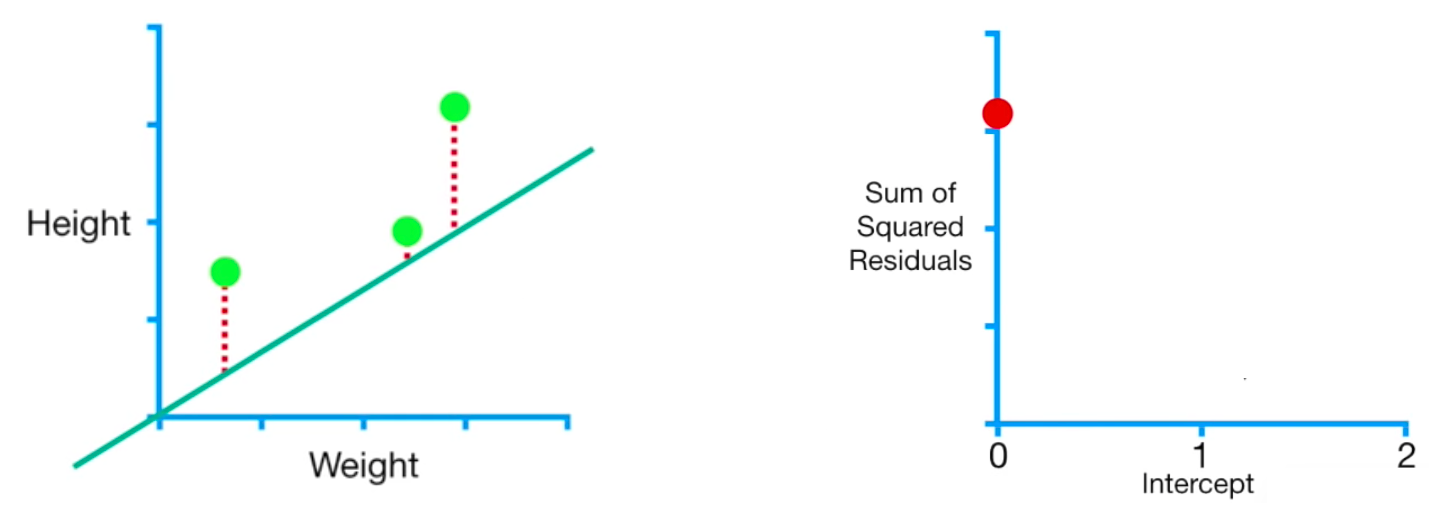
*# Initialize intercept at 0 for this example*

b0\_epoch0 = 0

*# L(b0\_epoch\_0)*

np.sum((y - h(X, b0\_epoch0)) \*\* 2)

3.1588640000000012

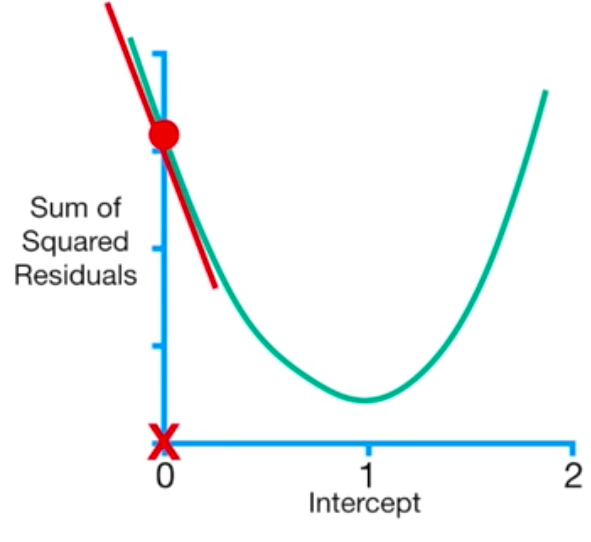


*# Step 1: compute the derivative of the Loss Function at b0\_epoch\_0*

derivative = np.sum(-2 \* (y - h(X, b0\_epoch0)))

derivative

-5.448

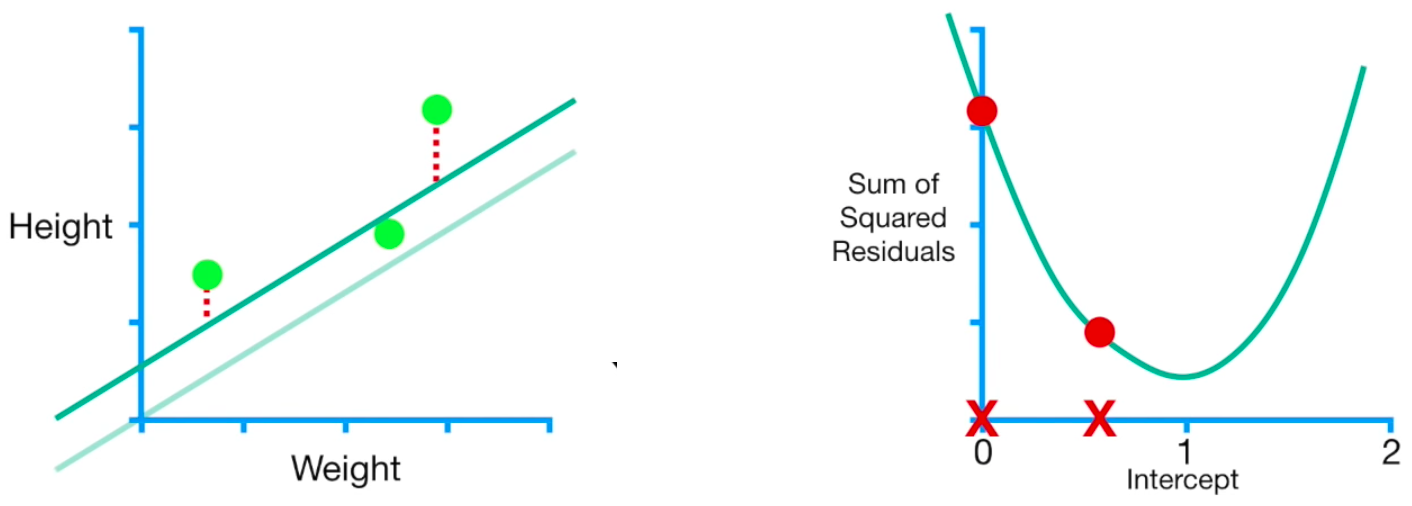


*# Step 2: update the intercept*

b0\_epoch1 = b0\_epoch0 - (eta \* derivative)

b0\_epoch1

0.5448000000000001



#### **Repeat**

That was one **Epoch**!

Let's do a second one!

*# Step1: compute the new derivative at b0\_epoch1*

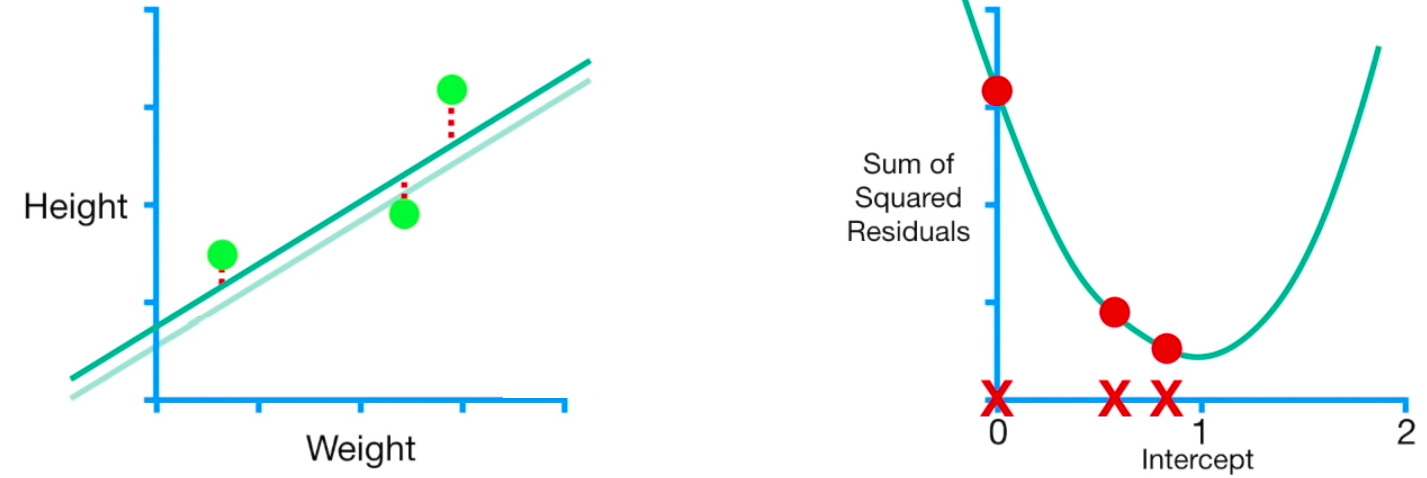
derivative = np.sum(-2 \* (y - h(X, b0\_epoch1)))

*# Step2: update the previously updated intercept*

b0\_epoch2 = b0\_epoch1 - eta \* derivative

b0\_epoch2

0.7627200000000002



Keep going until it converges to the minimum!

## **2.2 2D Descent: How to Co-Optimize**

## β

## 0

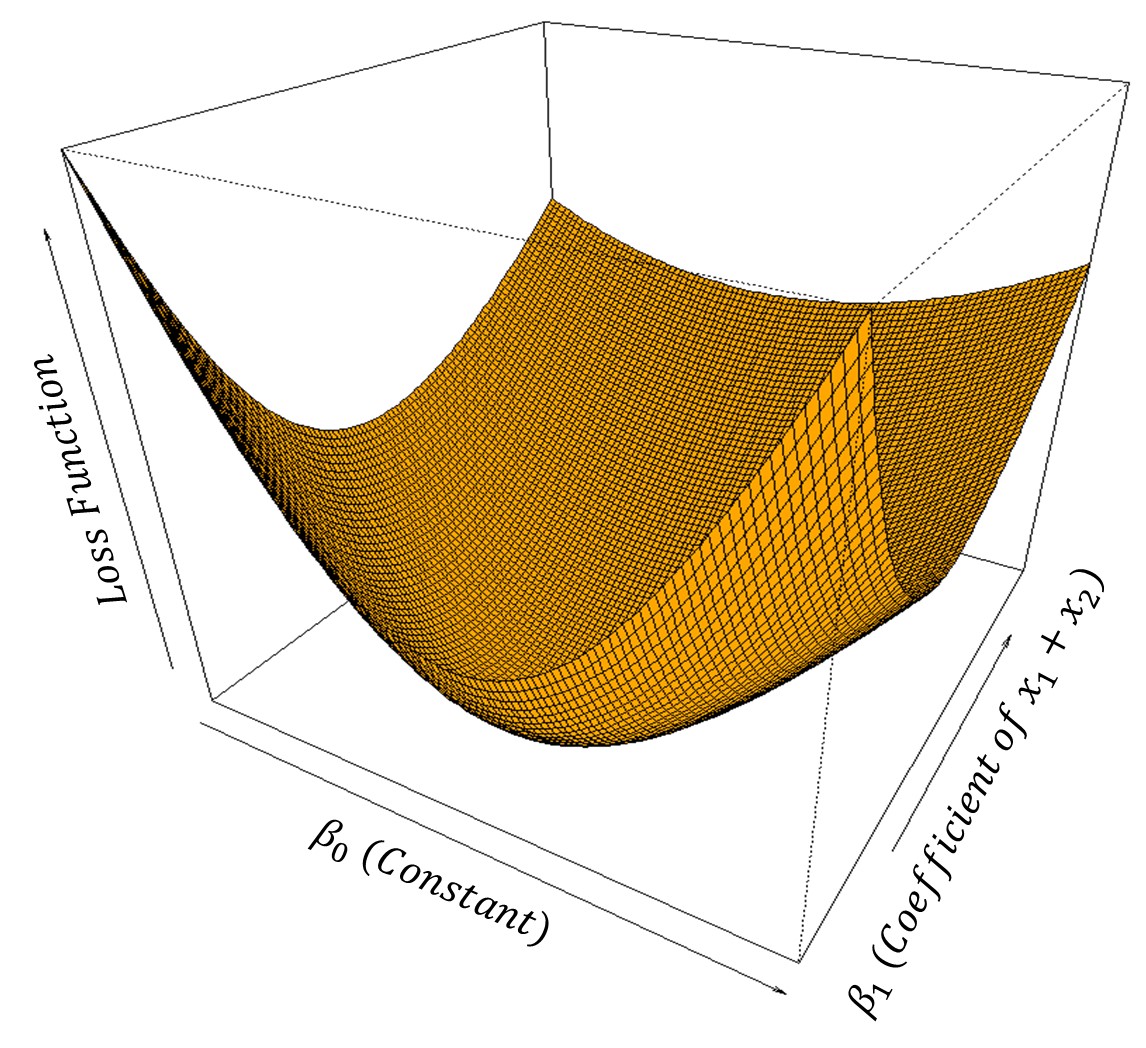
## **and**

## β

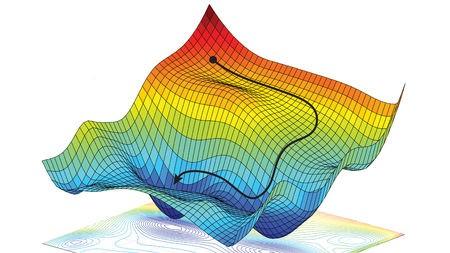
## 1

## **?**

The Loss Function would be represented in a 3-dimensional space and look something like this:



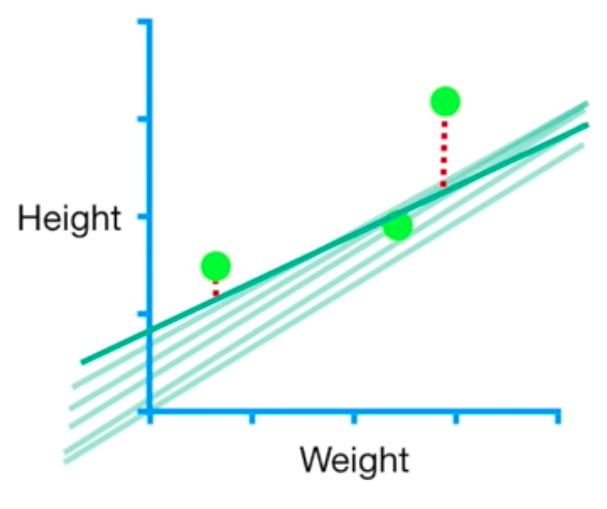
Or like this in more complex problems:



☝️ This is called the **energy landscape** of the Loss Function

☝️ Notice the projected **2D contour plot** below

We re-iterate the same procedure for both parameters **simultaneously**.



What about a **3D** problem?

β

=

(

β

0

,

β

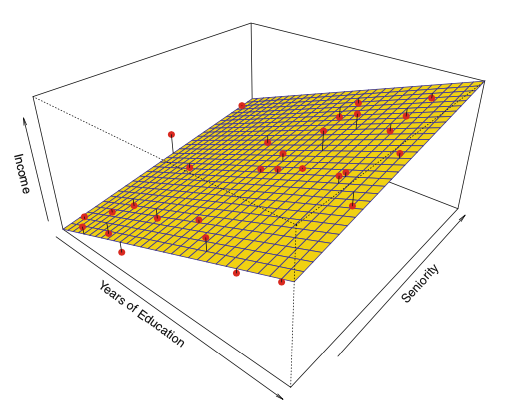
1

,

β

2

)

Below is the 3D plot of some sample **observations** 👇  


The associated **energy landscape** of the Loss Function is in **4D** 🤯

### **Vectorial Formulation (*N Dimensions*)**

1. Start with random values for
2. β
3. 0
4. and
5. β
6. 1
7. (epoch 0)
8. At each **epoch**
9. k
10. , update both
11. (
12. β
13. (
14. k
15. +
16. 1
17. )
18. 0
19. ,
20. β
21. (
22. k
23. +
24. 1
25. )
26. 1
27. ) in the direction of the "downward-pointing gradient"

β

(

k

+

1

)

0

=

β

(

k

)

0

−

η

∂

L

∂

β

0

(

β

(

k

)

)

β

(

k

+

1

)

1

=

β

(

k

)

1

−

η

∂

L

∂

β

1

(

β

(

k

)

)

* with a learning rate
* η
* (eta)

This vector of partial derivatives is called the **gradient** vector

∇

∇

L

(

β

)

=

⎡

⎢

⎢

⎢

⎢

⎣

∂

L

∂

β

0

(

β

)

⋮

∂

L

∂

β

p

(

β

)

⎤

⎥

⎥

⎥

⎥

⎦

.

💡 Hence the name *Gradient Descent*

Gradient Descent - vector formula

β

(

k

+

1

)

=

β

(

k

)

−

η

∇

L

(

β

(

k

)

)

The

b

l

u

e

a

r

r

o

w

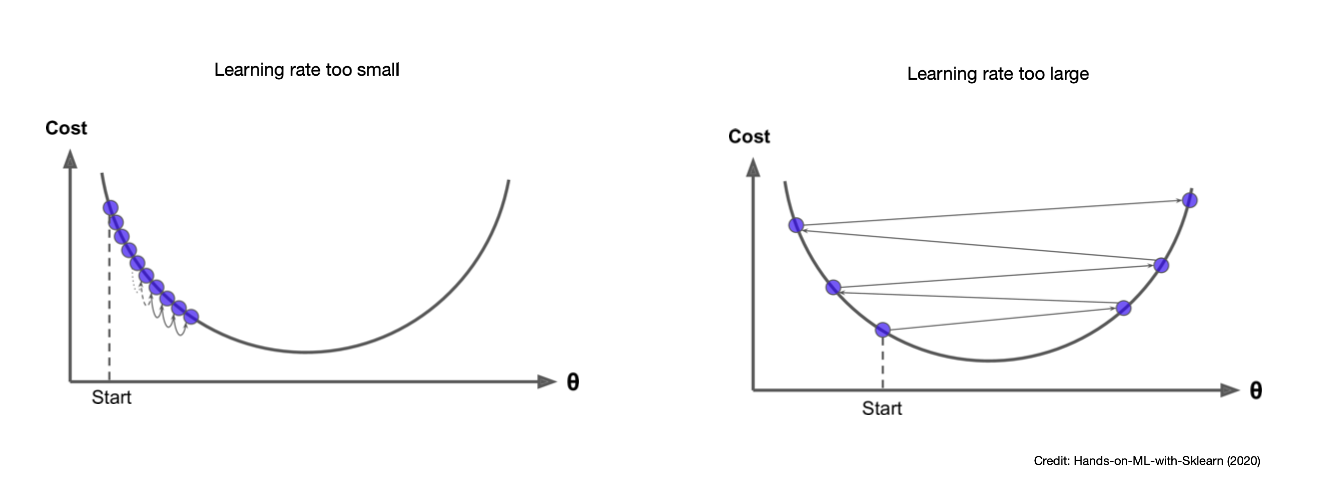
s

indicate the direction and strength of the gradient of this energy landscape; it is null at the loss minimum  


### **What is the Effect of Learning Rate**

### η

### **?**

****

📚 [Distill](https://distill.pub/2017/momentum/)

**Small Learning Rate**

* a shorter path to the minimum
* requires more epochs
* may get stuck at local minima

**Large Learning Rate**

* requires fewer epochs
* may never converge!

💡 **The Gradient Descent algorithm always converges faster when features are scaled!** [**Why?**](https://datascience.stackexchange.com/questions/55656/why-does-feature-scaling-improve-the-convergence-speed-for-gradient-descent)

### **Analytical Gradient for OLS Regression (Linear Regression + SSR Loss)**

We can also compute the following partial derivatives:

∂

S

S

R

∂

β

0

(

β

)

=

n

∑

i

=

1

−

2

(

y

i

−

^

y

i

)

∂

S

S

R

∂

β

1

(

β

)

=

n

∑

i

=

1

−

2

X

(

i

)

1

(

y

i

−

^

y

i

)

and more generally speaking:

∇

S

S

R

(

β

)

=

−

2

X

T

(

y

−

^

y

)

∇

S

S

R

(

β

)

=

−

2

X

T

(

y

−

X

β

)

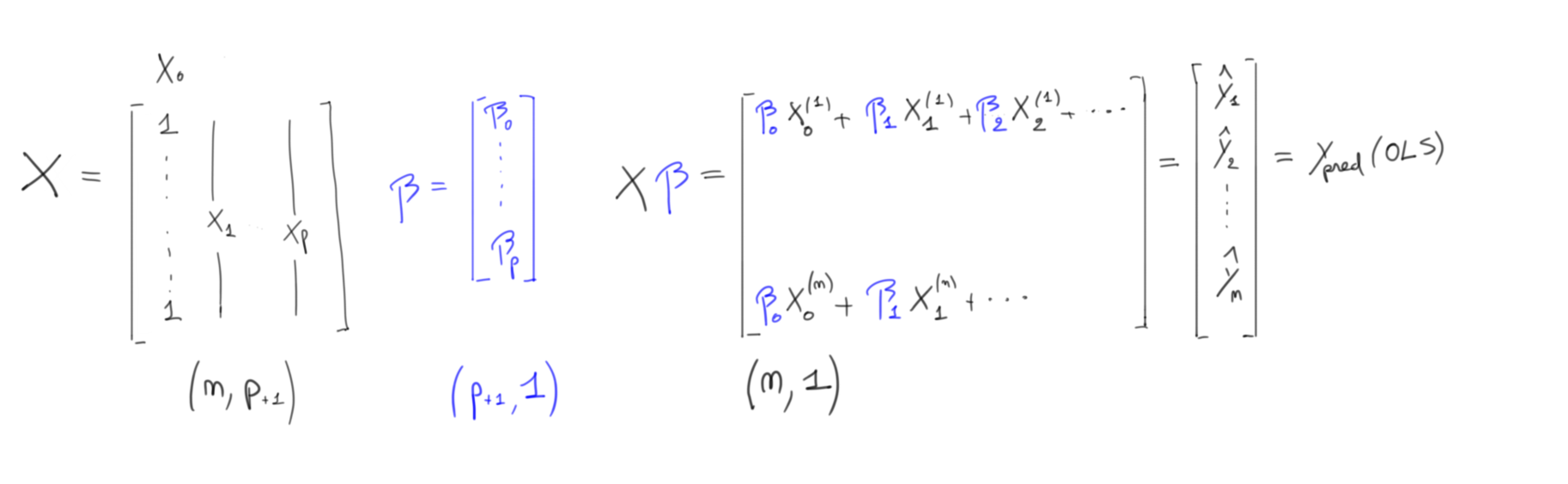
☝️ Because the formula for the gradient of the loss is so easy to compute, Gradient Descent is very efficient for OLS regressions

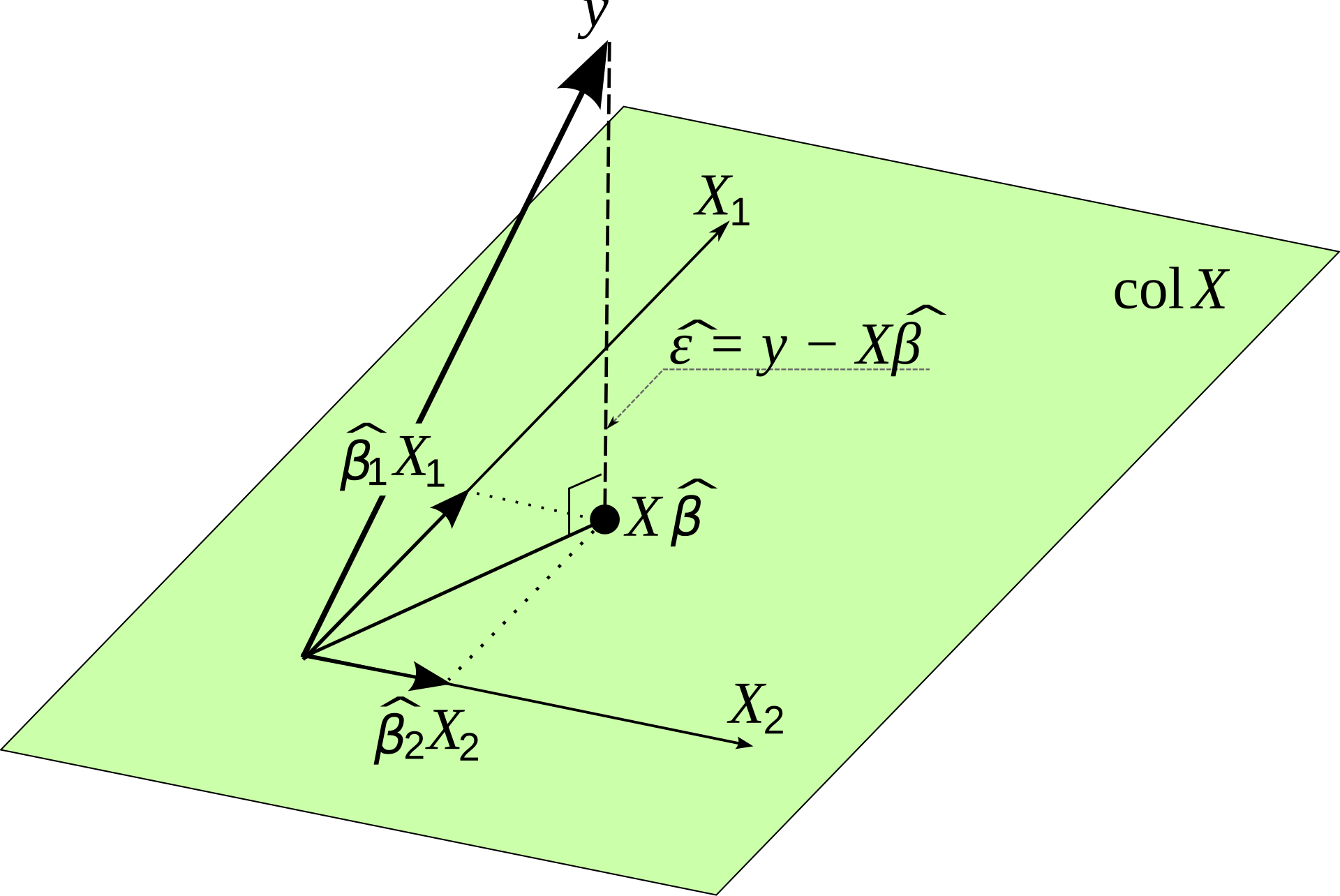
💪 You will implement this iterative Gradient Descent in today's challenges

### **(optional) Can we get a geometric intuition about the best**

### β

### **in OLS? 📐**

****

****

^

y

=

X

β

must lie somewhere in the [**column space**](https://www.omnicalculator.com/math/column-space) of

X

(the hyperplane defined by the span of all possible linear combinations of features

X

i

)

❓ What is the position (defined by choice of

β

) that minimizes the OLS loss

(

y

−

^

y

)

2

❓

💡 Pythagoras tells us that the shortest path is

the **orthogonal projection** of

y

t

r

u

e

into the hyperplane (col X)

# **3. Other Solvers**

Let's recall the definition of the gradient

(for OLS)

∇

L

(

β

)

=

⎡

⎢

⎢

⎢

⎢

⎣

∂

L

∂

β

0

(

β

)

⋮

∂

L

∂

β

p

(

β

)

⎤

⎥

⎥

⎥

⎥

⎦

=

⎡

⎢

⎢

⎣

∑

n

i

=

0

.

.

.

⋮

∑

n

i

=

0

.

.

.

⎤

⎥

⎥

⎦

=

−

2

X

T

(

y

−

^

y

)

👎 Gradient Descent is **computationally expensive** on big datasets:

* At **each epoch**, evaluating
* ∇
* L
* requires using all
* n
* observations, for each of those
* p
* features

❓ Couldn't we use **less than all**

n

**observations** to compute an "approximate loss"?



### **Mini-Batch Gradient Descent**

At each iteration, compute an **"approximate loss"** and take one step against its gradient

* Choose a mini-batch size (e.g. 16)
* Loop over your
* n
* observations in mini-batches, and for each mini-batch
* X
* m
* i
* n
* i
* (e.g. the first 16 observations):
  1. Compute the gradient of the mini-batch
  2. ∇
  3. L
  4. m
  5. i
  6. n
  7. i
  8. Use this gradient to update
  9. β
  10. (
  11. k
  12. +
  13. 1
  14. )
  15. =
  16. β
  17. (
  18. k
  19. )
  20. −
  21. η
  23. ∇
  24. L
  25. m
  26. i
  27. n
  28. i
  29. (
  30. β
  31. (
  32. k
  33. )
  34. )
  35. Move to next
  36. X
  37. m
  38. i
  39. n
  40. i
  41. (e.g. the 17-32 obs)
* Once all
* n
* observations have been viewed, repeat another **epoch**

### **Stochastic Gradient Descent (SGD)**

SGD

⇔

Mini-Batch of size 1

* Loop one-by-one over all
* n
* observations
  + Select a **single, randomly selected data point**
  + Compute the loss/gradient for this single point
  + Update
  + β
* Once all
* n
* observations have been viewed, repeat another epoch

💻 Let's code it for our OLS

b0 = 0

eta = 0.1

n\_epoch = 5 *# we have to choose when to stop*

**for** epoch **in** range(n\_epoch):

*# Loop randomly over all 3 data points in our example*

**for** i **in** np.random.permutation(3):

*# Select a mini-batch (of size 1)*

X\_mini = X[i]

*# Compute the gradient of the loss at b\_0*

y\_pred = h(X\_mini, b0)

y\_true = y[i]

derivative = -2 \* (y\_true - y\_pred)

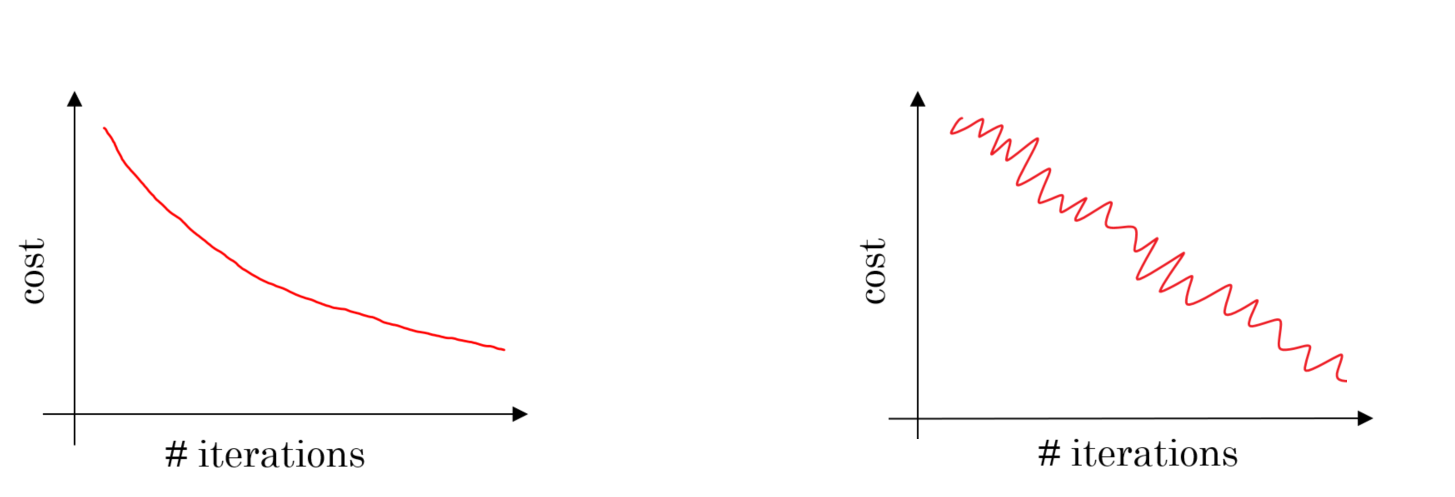
*# Update b\_0*

b0 = b0 - eta \* derivative

print(f'b0 epoch **{**epoch**}**:', b0)

Due to working on a single point rather than the dataset average, the SGD is less stable.

* The loss **fluctuates** more from epoch to epoch and does not necessarily decrease
* As a result, the **steps** taken toward the minimum are **less direct**

****

Pros

* SGD is faster for very large datasets
* Jumps out of local minima!
* Greatly reduces RAM load (see Deep Learning)

Cons

* Needs more epochs
* Never exactly converges (careful when to stop)
* Maybe slower for small
* n
* datasets with many features
* p

👉 **Use when**

* The number of observations in your dataset has 6 digits or more
* You want to get "un-stuck" from a local minimum
* By default?

#### **Sklearn** [**SGDRegressor**](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html) **and** [**SGDClassifier**](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html)

* SGDRegressor is a Linear Model (Linear Regression) that uses the Stochastic Gradient Descent as a solver to minimize its Loss Function (MSE)
* SGDClassifier is a Linear Model (Logistic Regression) that uses the Stochastic Gradient Descent as a solver to minimize its Loss Function (Log Loss)

**Note:** We'll talk about these Loss Functions in detail in section 4.

**from** **sklearn.linear\_model** **import** SGDRegressor, LinearRegression

lin\_reg = LinearRegression() *# OLS solved by matrix inversion (SVD method)*

lin\_reg\_sgd = SGDRegressor(loss='squared\_error') *# OLS solved by SGD*

**from** **sklearn.datasets** **import** make\_regression

*# Create a "fake problem" to solve*

X, y = make\_regression(n\_samples=10000, n\_features=1000)

%%time

lin\_reg.fit(X,y)

CPU times: user 8.47 s, sys: 403 ms, total: 8.88 s

Wall time: 2.07 s

LinearRegression()

%%time

lin\_reg\_sgd.fit(X,y)

CPU times: user 182 ms, sys: 2.85 ms, total: 184 ms

Wall time: 189 ms

SGDRegressor()

✅ Gradient Descent performs better than matrix inversion when feature number

p

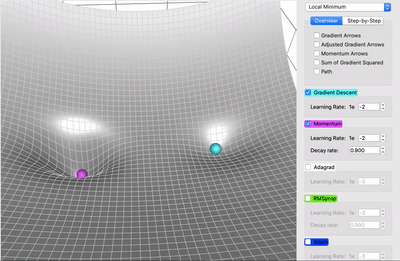
is large  
✅ SGD scales even better when the number of observations

n

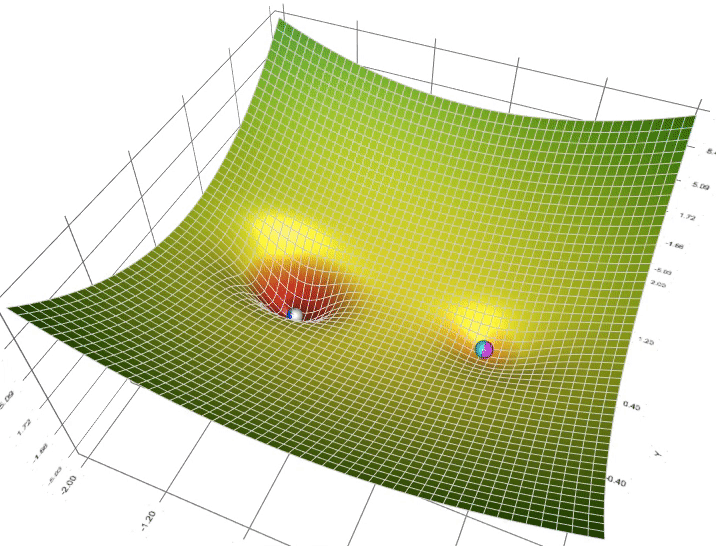
is large

### **Other Solvers?**

#### **1) Enhanced Gradient Descents**

****

* Gradient
* Momentum (adds inertia)



* Gradient
* Momentum (adds inertia)
* AdaGrad (adaptative
* η
* per feature - prioritize weakly updated params)
* RMSProp (adds decay - only recent gradient matters)
* Adam (all combined)

📚[Credits](https://medium.com/towards-data-science/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c)

#### **2) Second-Order Partial Derivative Methods (Hessian Matrix)**

In each iteration, one approximates

h

(

x

)

using a **quadratic** function instead of a "slope"

* Newton’s Method
* L-BFGS (approx. Hessian)

Pros: Converges with far fewer epochs

Cons: Computationally expensive

👉 Used for "easy" ML problems. Default solver for Sklearn's LogisticRegression

# **4. Loss Functions L**

Squared Loss is not the only Loss Function that you can minimize to fit a regression  
SGDRegressor(loss='squared\_loss')

SGDRegressor(loss='huber')

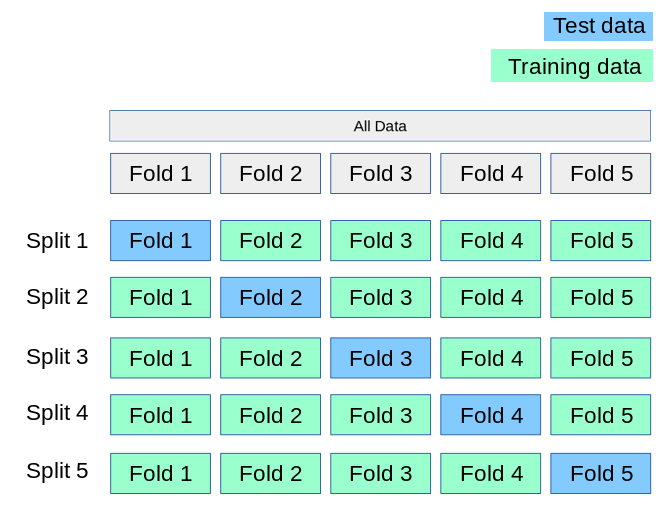
The same model class on Sklearn can be instantiated with various attributes for loss  
SGDClassifier(loss='log')

SGDClassifier(loss='hinge')

* Classification and Regression have different Loss Functions by nature

### **3.1 Loss ≠ Performance Metrics**

Performance metrics are computed **after** the model is fitted

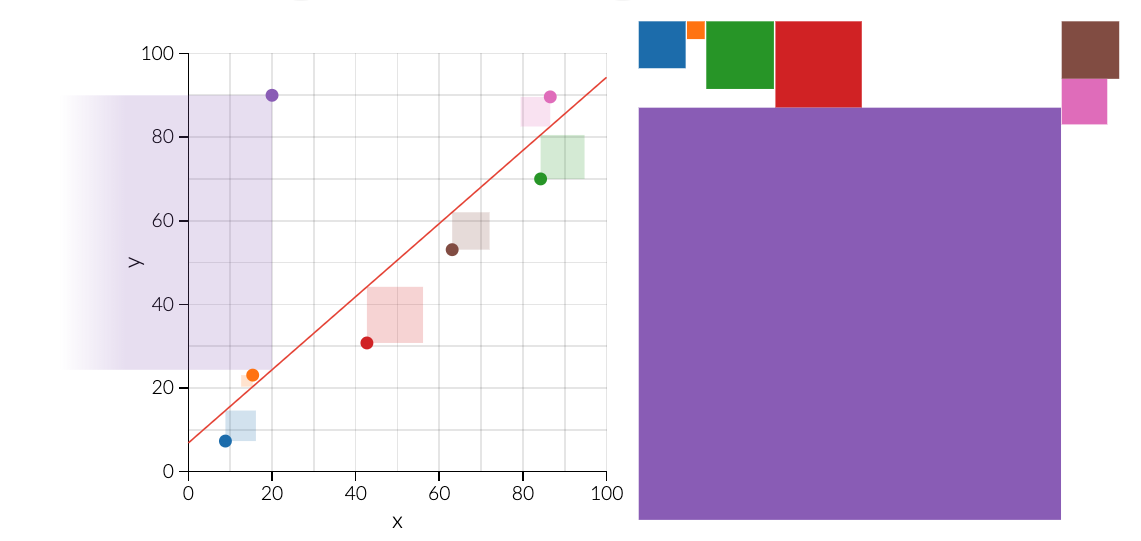


**Regression performance metrics** (MSE, RMSE, RMSLE, MAE,

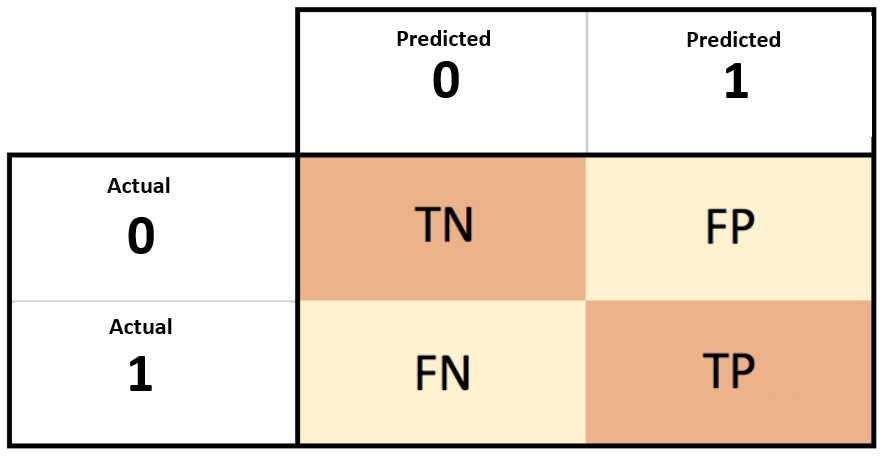
R

2

, etc.)



**Classification Metrics** (Accuracy, Precision, Recall, F1, etc.)



**Loss** is used to **fit** the model

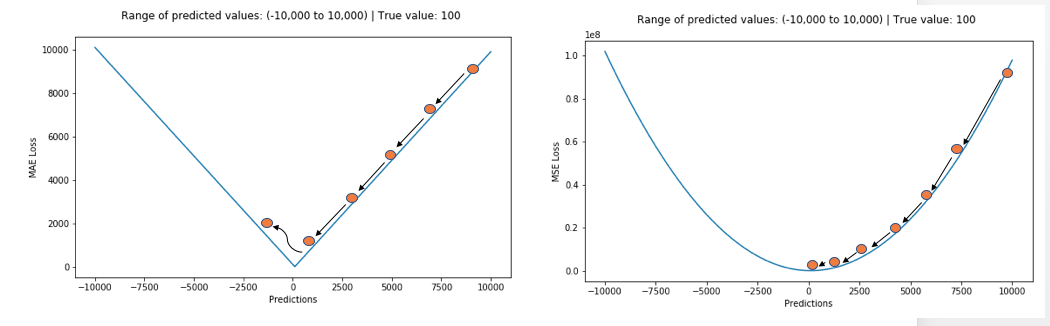
* Sometimes loss and performance metrics may be the same (e.g. MSE)
* But loss needs to be **(sub)differentiable** (ie. smooth)

👉 "Accuracy" for instance can never be used as a loss metric

## **3.1 Regression Loss Functions**

### **L1 Loss (MAE) vs L2 Loss (MSE)**

| L  1  =  M  A  E  =  1  n  n  ∑  i  =  1  |  ^  y  i  −  y  i  | |  |  |  | L  2  =  M  S  E  =  1  n  n  ∑  i  =  1  (  ^  y  i  −  y  i  )  2 |
| --- | --- | --- | --- | --- |



* MSE is very sensitive to outliers, MAE is less strict
* MAE requires a Learning Rate
* η
* which decreases at every epoch

### **Huber Loss (mix of L1 and L2 losses, also called *Smooth Absolute Loss*)**

* MAE which becomes MSE when error is small, typically adjustable by hyperparameter [epsilon](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html#sklearn.linear_model.SGDRegressor:~:text=The%20verbosity%20level.-,epsilon,-float%2C%20default%3D0.1)
* Adjustable for outliers
* Slope can be used as an indicator of reaching minima

L

δ

=

{

1

2

(

y

−

^

y

)

2

i

f

|

(

y

−

^

y

)

|

<

δ

δ

(

|

(

y

−

^

y

)

|

−

1

2

δ

)

o

t

h

e

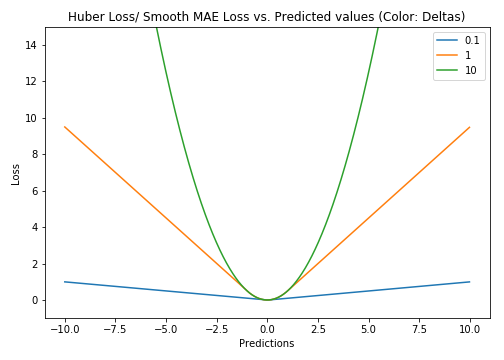
r

w

i

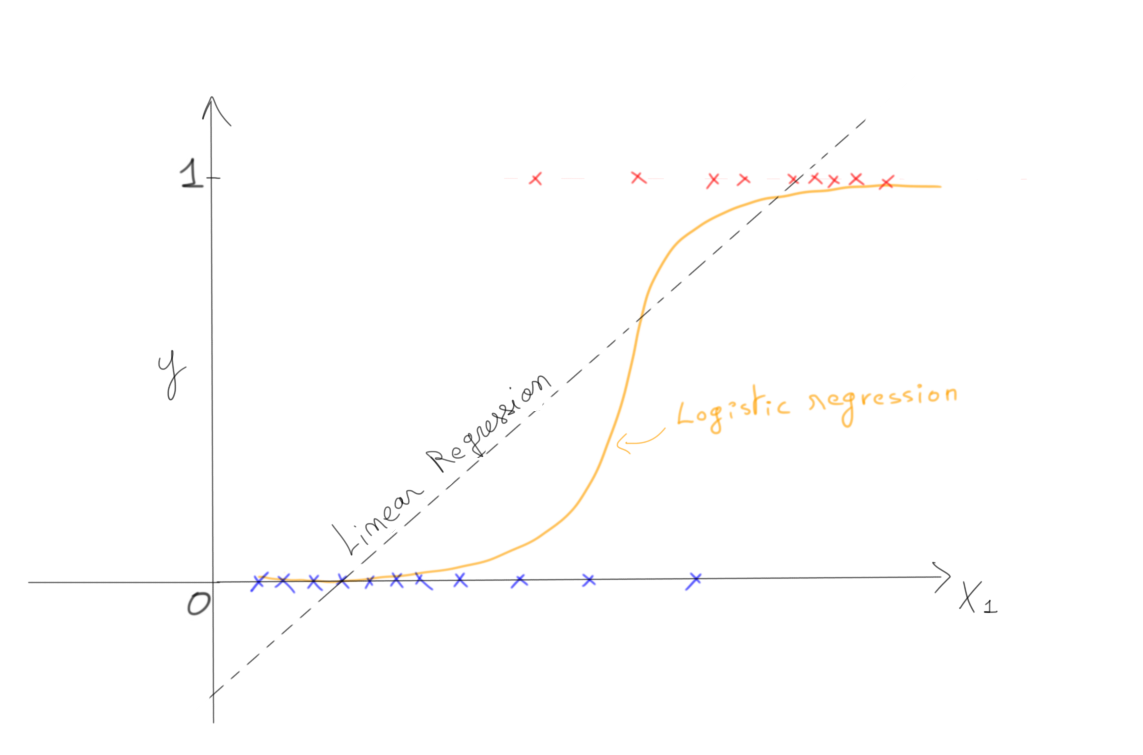
s

e



## **3.2 Classification Losses**

### **Logistic Classifiers**

****

* We want to predict a binary vector
* y
* =
* [
* 0
* ,
* 0
* ,
* 1
* ,
* 0
* ,
* .
* .
* .
* ,
* 1
* ]
* of size
* n
* We model it by a vector
* ^
* y
* =
* [
* 0.1
* ,
* 0.3
* ,
* .
* .
* .0
* .8
* ]
* =
* h
* (
* X
* ,
* β
* )

❓ **What loss to compute** between

^

y

and

y

?

Logistic classifiers want to maximize this **product**

∏

i

when

y

i

=

1

^

y

i

∏

i

when

y

i

=

0

(

1

−

^

y

i

)

* ^
* y
* i
* **close to 1** when
* y
* i
* =
* 1
* (
* 1
* −
* ^
* y
* i
* )
* **close to 1** when
* y
* i
* =
* 0
* for all **independent** observations
* i

👉 This is the **combined probability** of observing all

y

i

, if each were sampled randomly from a binary probability distribution p =

^

y

i

👉 Called the **Likelihood** of observing the true

y

under some hypothesis function

h

(See [Logistic Regression lecture](https://kitt.lewagon.com/camps/1917/lectures/content/04-Decision-Science_04-Logistic-Regression.slides.html?title=Logistic-Regression&program_id=10#/2/5/1))

It's easier to maximize the log instead (**log-likelihood**)

l

o

g

(

∏

y

i

=

1

∏

y

i

=

0

)

=

∑

i

when

y

i

=

1

l

o

g

(

^

y

i

)

+

∑

i

when

y

i

=

0

l

o

g

(

1

−

^

y

i

)

=

n

∑

i

=

0

y

i

l

o

g

(

^

y

i

)

+

(

1

−

y

i

)

l

o

g

(

1

−

^

y

i

)

### **Log Loss (a.k.a Cross-Entropy Loss)**

L

o

g

L

o

s

s

=

−

1

n

n

∑

i

=

0

y

i

l

o

g

(

^

y

i

)

+

(

1

−

y

i

)

l

o

g

(

1

−

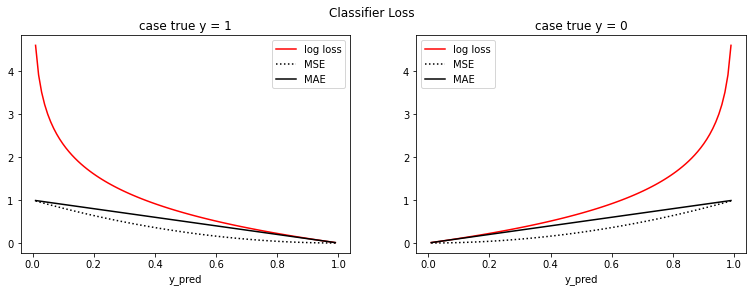
^

y

i

)

* y
* =
* 1
* ⇒
* L
* o
* g
* L
* o
* s
* s
* =
* −
* l
* o
* g
* (
* ^
* y
* )
* y
* =
* 0
* ⇒
* L
* o
* g
* L
* o
* s
* s
* =
* −
* l
* o
* g
* (
* 1
* −
* ^
* y
* )



☝️ Infinitely penalize wrong predictions

💡 Cross-Entropy name comes from [Shanon Theory of information](https://www.youtube.com/watch?v=ErfnhcEV1O8)

🤔 The gradient of the **log-loss** of the **sigmoid** function is simple in vectorial form

∇

L

o

g

L

o

s

s

s

i

g

m

o

i

d

=

−

2

n

X

T

(

y

−

^

y

)

Exact **same formula** as that of the MSE loss of a Linear Regression

∇

M

S

E

l

i

n

e

a

r

=

−

2

n

X

T

(

y

−

^

y

)

😎 Think **vectorial** whenever possible. OLS/Logit gradient descent = ~4 lines in NumPy

⚠️ These gradients **do not have the same value** of course as:

^

y

s

i

g

m

o

i

d

=

1

1

+

e

−

X

β

^

y

l

i

n

e

a

r

=

X

β

### **Other (non-logistic) classifiers exist!**

* Naive Bayes
* Support Vector Machine Classifier (SVC)
* ...

All have different losses!

# **5. Summary**

**Problem setting**

* X
* = features
* y
* = target =
* h
* (
* X
* ,
* β
* )
* +
* e
* r
* r
* o
* r
* h
* = hypothesis function (Linear, Sigmoid, Neural Network, etc.)

**Parameters of the model:**

β

* Computed automatically during .fit()
* by minimizing
* L
* (
* β
* )

**Hyperparameters of the model** :

*(chosen manually)*

* Loss function
* L
* (MSE, MAE, Log-Loss, etc.)
  + Parameters of the loss itself (learning\_rate, etc.)
* Solver = method used to minimize
* L
* ('newton', 'sdg', etc.)
* Model specificities ('n\_neighbors', etc.)

"**Model**" is a **loosely defined** term.

sklearn models generally refer to the hypothesis function

h

(

X

,

β

)

**Regressors**

LinearRegressor() *# OLS regression*

KNeighborsRegressor() *# KNN*

SVR() *# Support Vector Regressor*

**Classifiers**

LogisticRegressor() *# Logit regression*

KNeighborsClassifier() *# KNN*

SVC() *# Support Vector Classifier*

⚠️ An exception: SGDClassifier and SGDRegressor refer to a linear hypothesis function

h

, fitted using an SGD algorithm.

SGDRegressor(loss='squared\_loss') *# eq. to OLS regression*

SGDRegressor(loss='huber') *# non-OLS linear regression*

SGDClassifier(loss='log') *# eq. to Logit*

SGDClassifier(loss='hinge') *# eq. to SVC*

Finally, once a model is selected, it can be further "fine-tuned" by selecting **hyper-parameters**manually as model class attributes

SGDRegressor(loss='squared\_loss', learning\_rate=0.1, eta0=0.01)

KNeighborsRegressor(n\_neighbors=5)

LogisticRegression(solver='newton')

...and then fitted .fit()

## **Bibliography**

* 📺 [StatsQuest - Gradient Descent](https://www.youtube.com/watch?v=sDv4f4s2SB8)
* 📚 [Hands-On ML with SKLearn - Chapter 4](https://github.com/yanshengjia/ml-road/blob/47cadb02faa756f85fd2f058e31221cc8223b97a/resources/Hands%20On%20Machine%20Learning%20with%20Scikit%20Learn%20and%20TensorFlow.pdf)
* 📚 [Andrew NG - Linear Models CS229 Notes](https://cs229.stanford.edu/main_notes.pdf)
* 📚 [Derivative of log-loss function](https://medium.com/analytics-vidhya/derivative-of-log-loss-function-for-logistic-regression-9b832f025c2d)

## **Your turn! 🚀**