Under the Hood

Plan

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

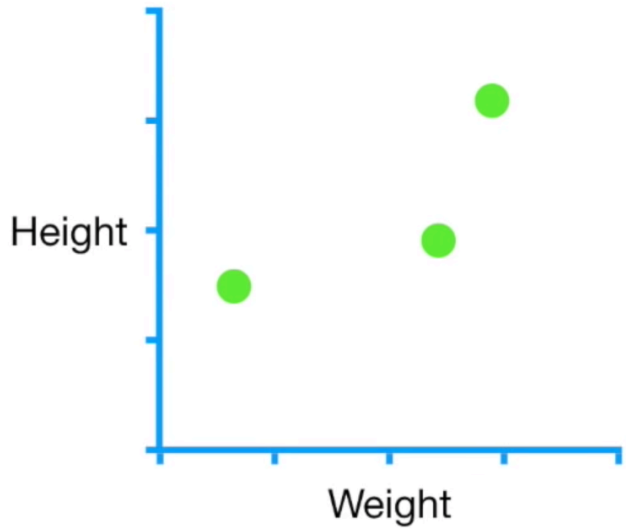
1. What happens behind `.fit()` ?

Consider the following data:

In [ ]:
```
data
```

Out[ ]:

	weight	height
0	0.7	1.5
1	2.4	1.8
2	2.8	3.2



Train a Linear Regression model:

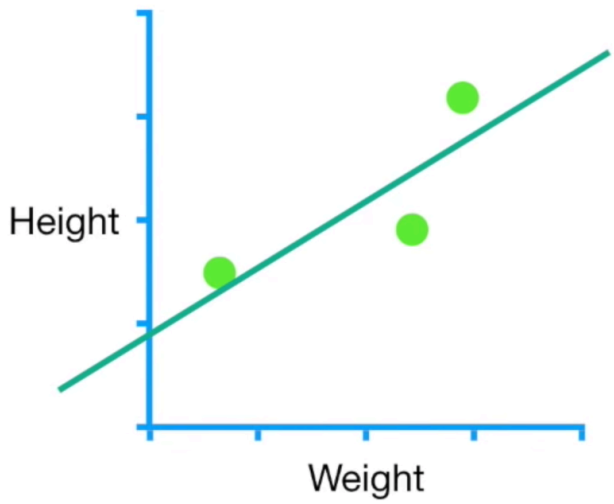
In [ ]:
```
from sklearn.linear_model import LinearRegression

# Instantiate Linear model
model = LinearRegression()

# Train Linear Model
model.fit(data[['weight']], data['height'])
```

Out[ ]:
```
LinearRegression()
```

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

In [ ]:
```
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
$y = h(X, \beta) + error$

- h
  is called our **hypothesis** function
- $h(X, \beta)$
  is called our **prediction** (
  $\hat{y}$
)
- $h(X, \beta) = \beta_0 + \beta_1 X_1$
  in our example

👉 `.fit()` finds parameters
β_0
and
β_1
which **minimize** the
$error(X, y, \beta)$

❓ Which **norm** is used to measure error in $\mathbb{R}$ numbers?

The Loss Function L

`.fit()` minimizes $L(error)$

$$L_{OLS} = \|error\|^2 = \|y - \beta_0 - \beta_1 X_1\|^2$$

We often write:
$$\beta = \arg\min_{\beta} L(\beta, X, y, h)$$

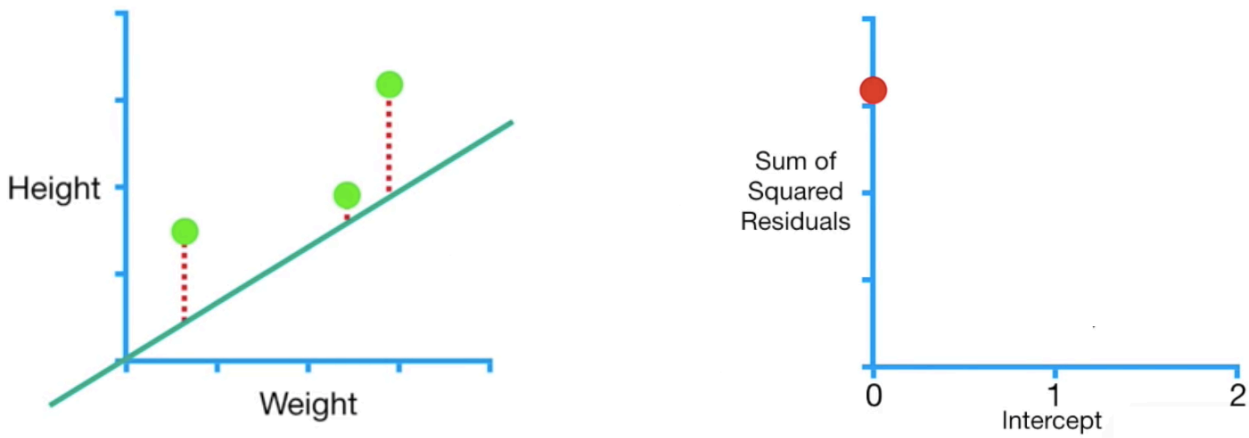
There are numerous "solvers" to minimize L(β) beyond Gradient Descent

- 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/)](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 ($\beta_1 = 0.64$), and need to find the optimal intercept (β_0):

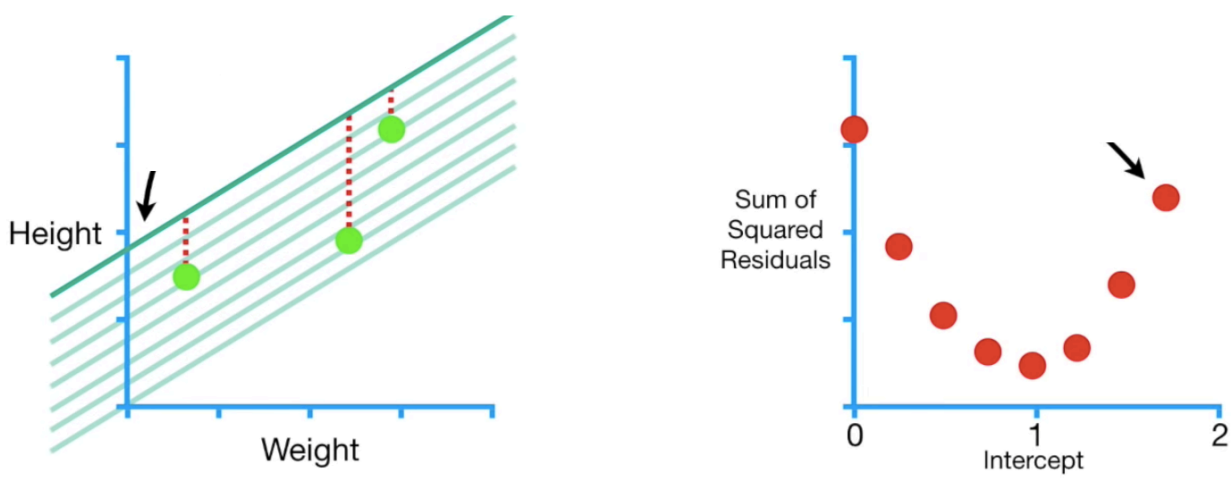


Steps:

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

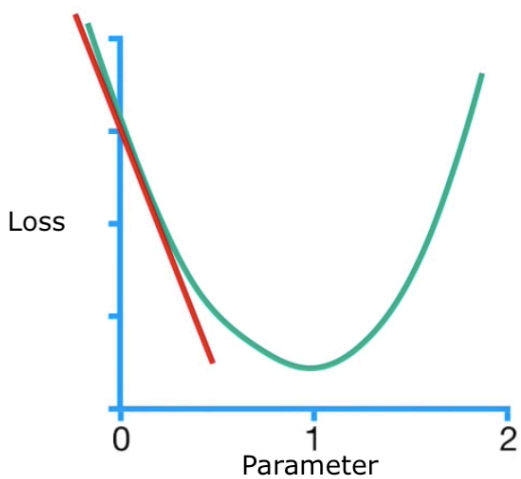
👉 We need to tweak both
β_0
and
β_1
simultaneously in each iteration

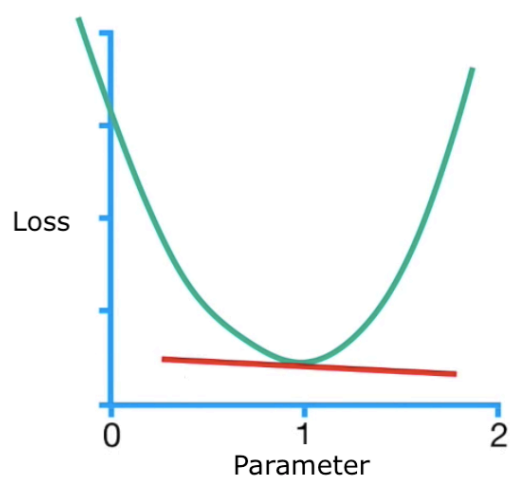
So let's discover the most basic but very powerful iterative method: the **`Gradient Descent`**

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





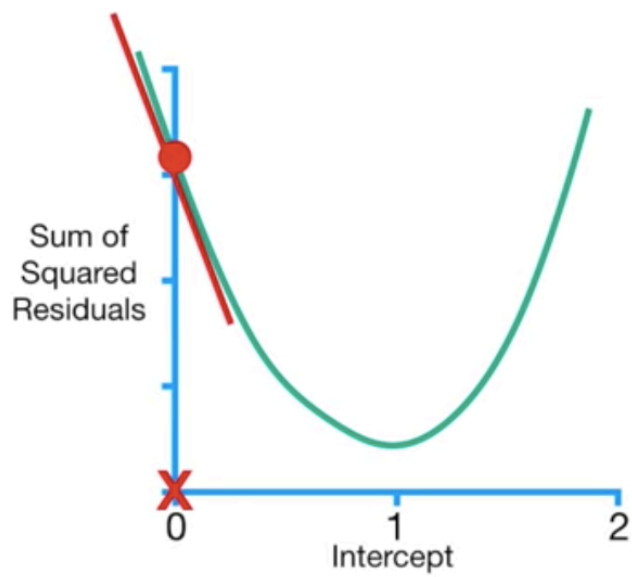
The slope is equal to the **partial derivative** of the Loss Function with respect to the parameter of interest:

$$\frac{\partial\ Loss\ Function}{\partial\ parameter}$$

✏️ Let's go back to our example

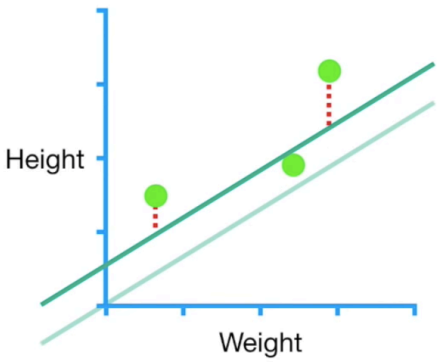
Step One

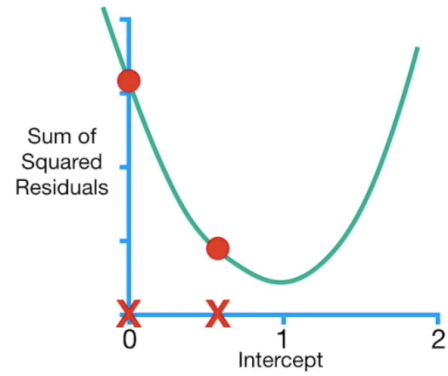
- Initialize a random parameter value, say $\beta_0 = 0$
- Calculate the derivative of the Loss Function at that point 👉 $\frac{\partial SSR}{\partial \beta_0}(0)$



Step Two

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





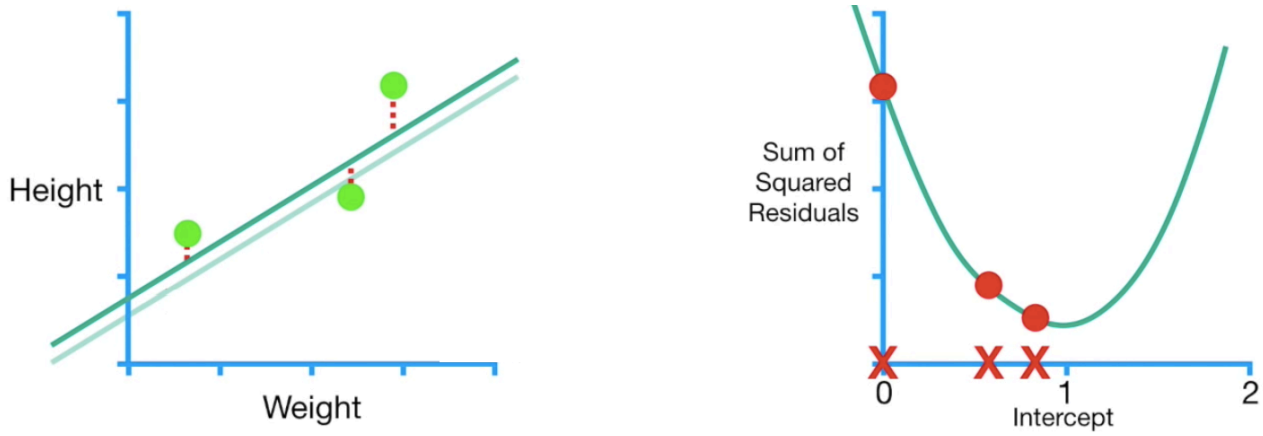
Note:

- the step size is **proportional** to the derivative's value
- it moves according to a chosen **Learning Rate** = η (eta)

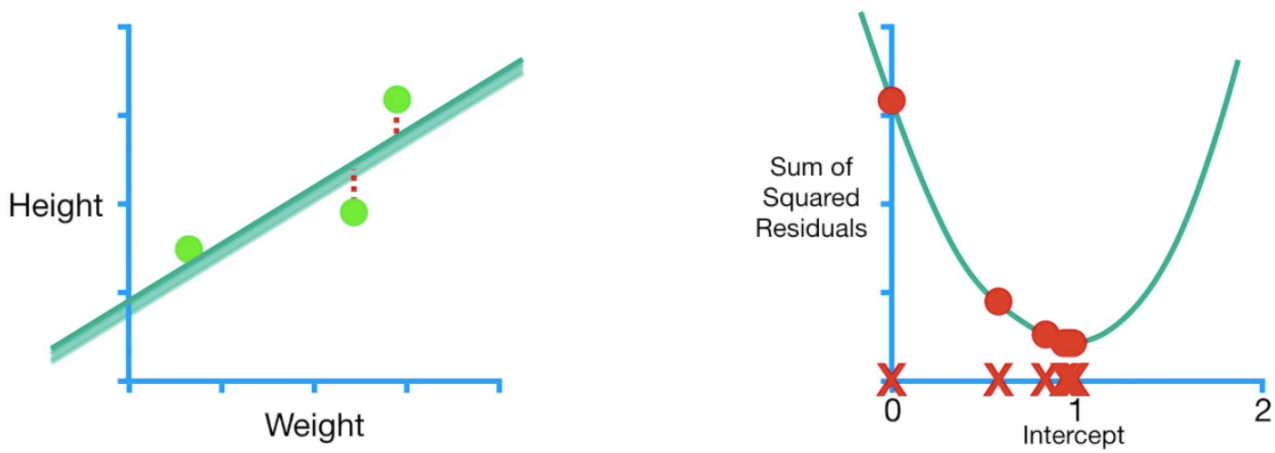
$$\beta_0^{(1)} = 0 - \eta \frac{\partial L}{\partial \beta_0}(0)$$

We just did one epoch! Now we repeat the process.

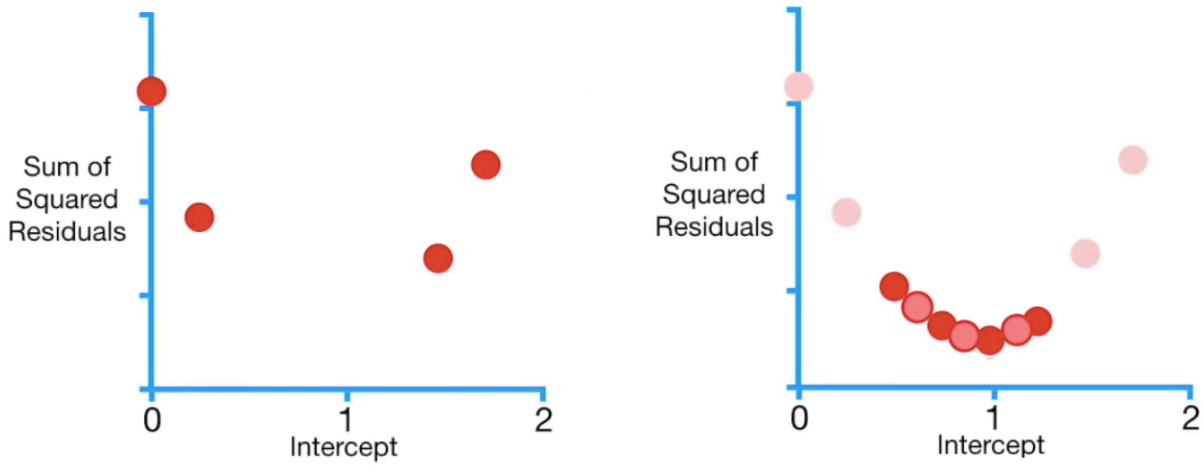
The updated intercept value is plugged back into the derivative of the Loss Function, and we repeat the process



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

$$\beta_0^{(k+1)} = \beta_0^{(k)} - \eta \frac{\partial L}{\partial \beta_0}(\beta_0^{(k)})$$

1. Randomly initialize the parameter value $\beta_0^{(0)}$
2. Compute the derivative of the Loss Function at that point
3. Update the parameter value according to the step size η
4. 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
$\frac{\partial SSR}{\partial \beta_0}$
for our example?

$$SSR(\beta) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 X_1^{(i)}))^2$$

We can compute its partial derivative with respect to
β_0

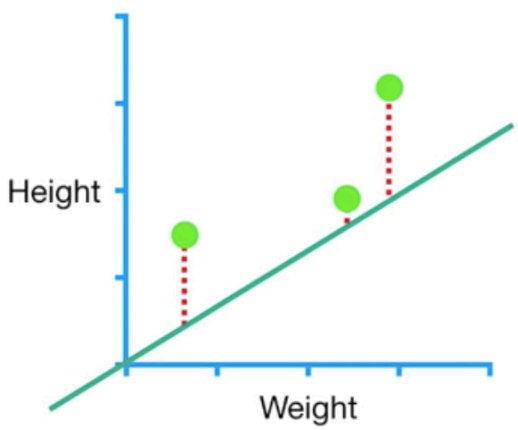
$$\frac{\partial SSR}{\partial \beta_0} = \sum_{i=1}^{n} -2(y_i - (\beta_0 + \beta_1 X_1^{(i)})) = \sum_{i=1}^{n} -2(y_i - \hat{y}_i)$$

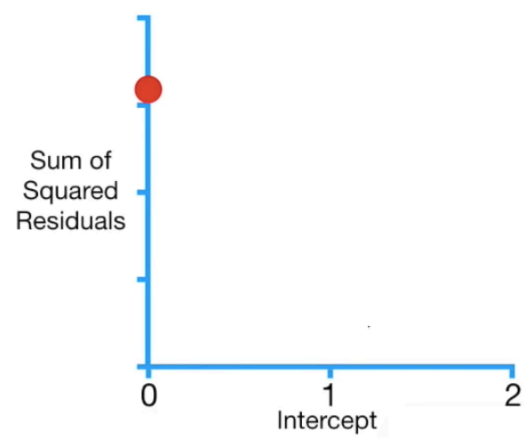
💡
$f(g)' = g' * f'(g)$

✏️ We can now code the Gradient Descent for our example

In [ ]:
```
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)
```

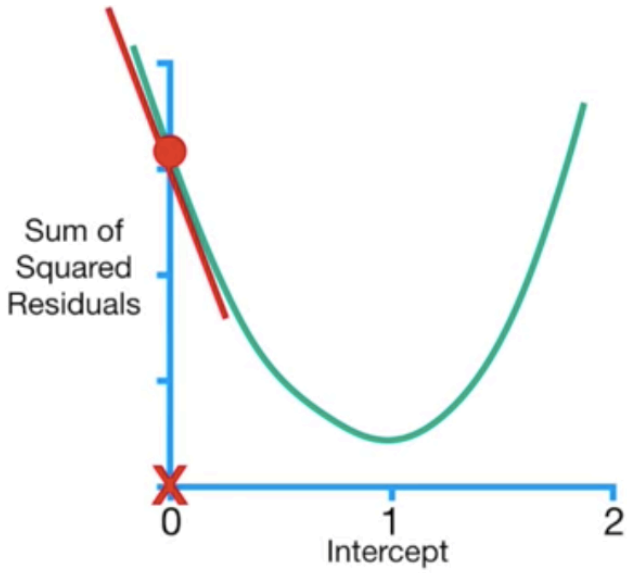
Out[ ]: 3.1588640000000012





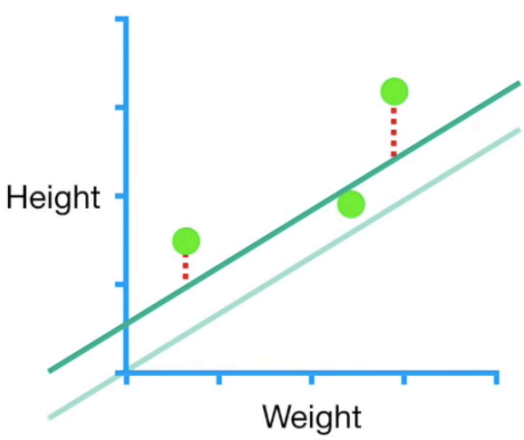
In [ ]:
```
# Step 1: compute the derivative of the Loss Function at b0_epoch_0
derivative = np.sum(-2 * (y - h(X, b0_epoch0)))
derivative
```

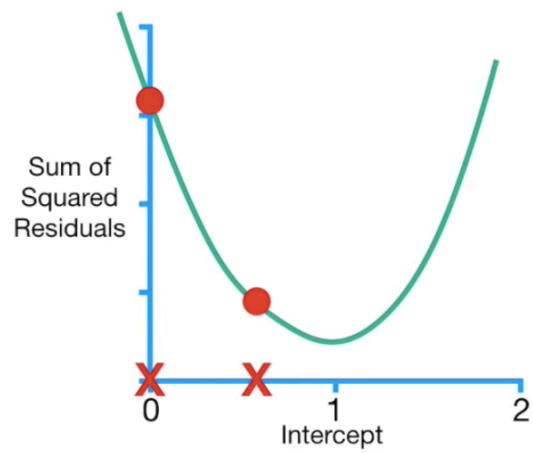
Out[ ]: -5.448



In [ ]:
```
# Step 2: update the intercept
b0_epoch1 = b0_epoch0 - (eta * derivative)
b0_epoch1
```

Out[ ]: 0.5448000000000001





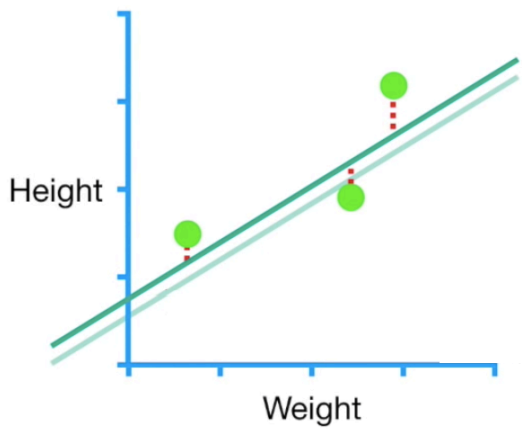
Repeat

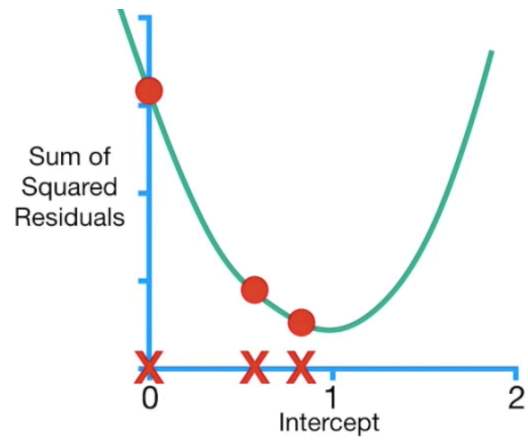
That was one **Epoch**!

Let's do a second one!

In [ ]:
```
# 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
```

Out[ ]: 0.7627200000000002

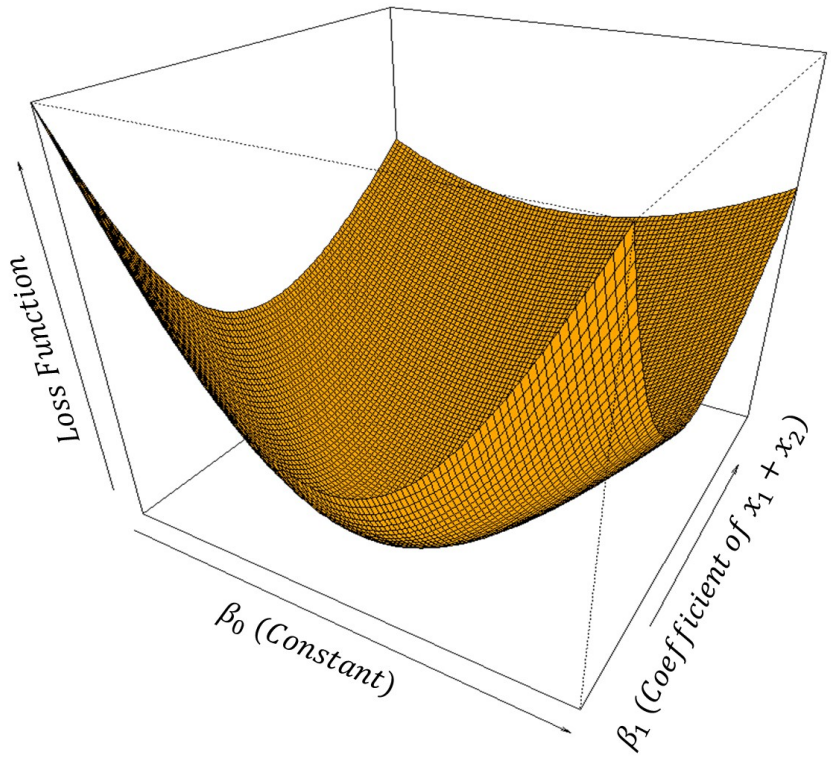




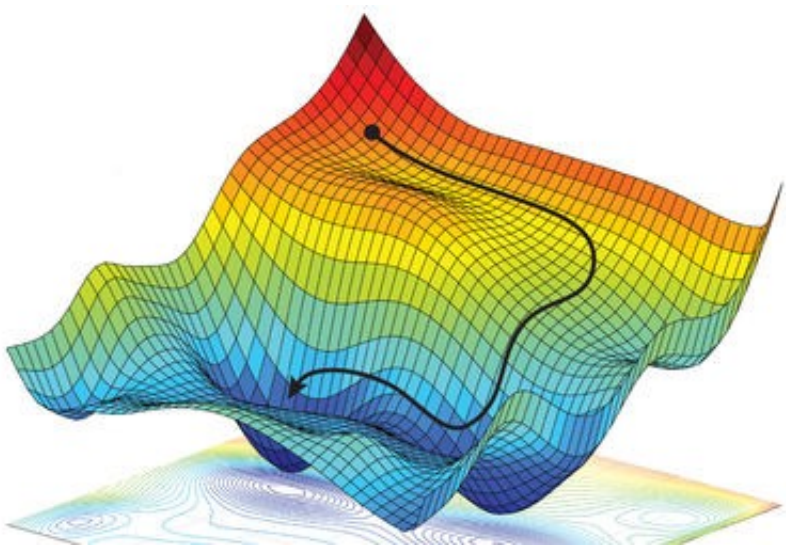
Keep going until it converges to the minimum!

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:



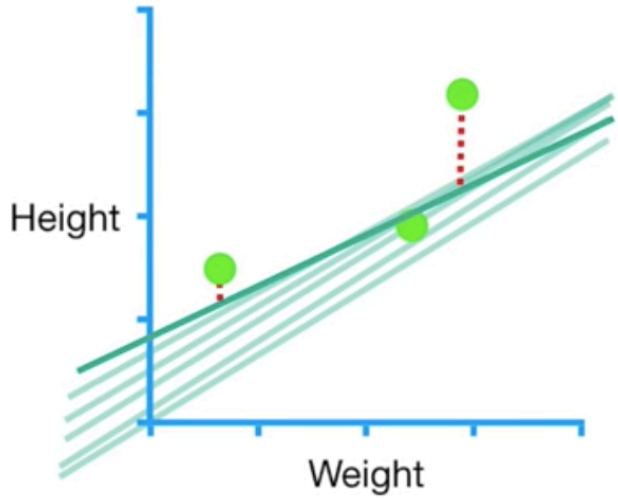
Or like this in more complex problems:



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

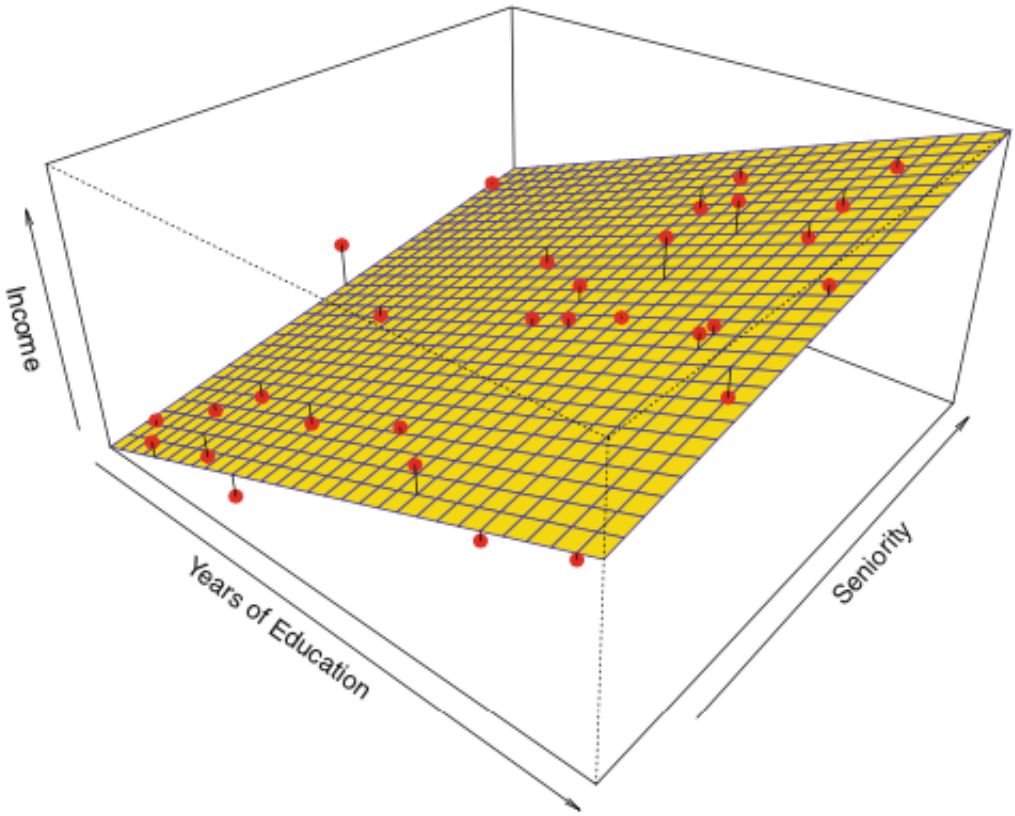
☝️ Notice the projected **2D contour plot** below

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



What about a **3D** problem?
$\beta = (\beta_0, \beta_1, \beta_2)$

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
   β_0
   and
   β_1
   (epoch 0)
2. At each **epoch**
   k
   , update both
   $(\beta_0^{(k+1)}, \beta_1^{(k+1)}$
) in the direction of the "downward-pointing gradient"

$$\beta_0^{(k+1)} = \beta_0^{(k)} - \eta \frac{\partial L}{\partial \beta_0}(\beta^{(k)})$$

$$\beta_1^{(k+1)} = \beta_1^{(k)} - \eta \frac{\partial L}{\partial \beta_1}(\beta^{(k)})$$

- with a learning rate
  η
  (eta)

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

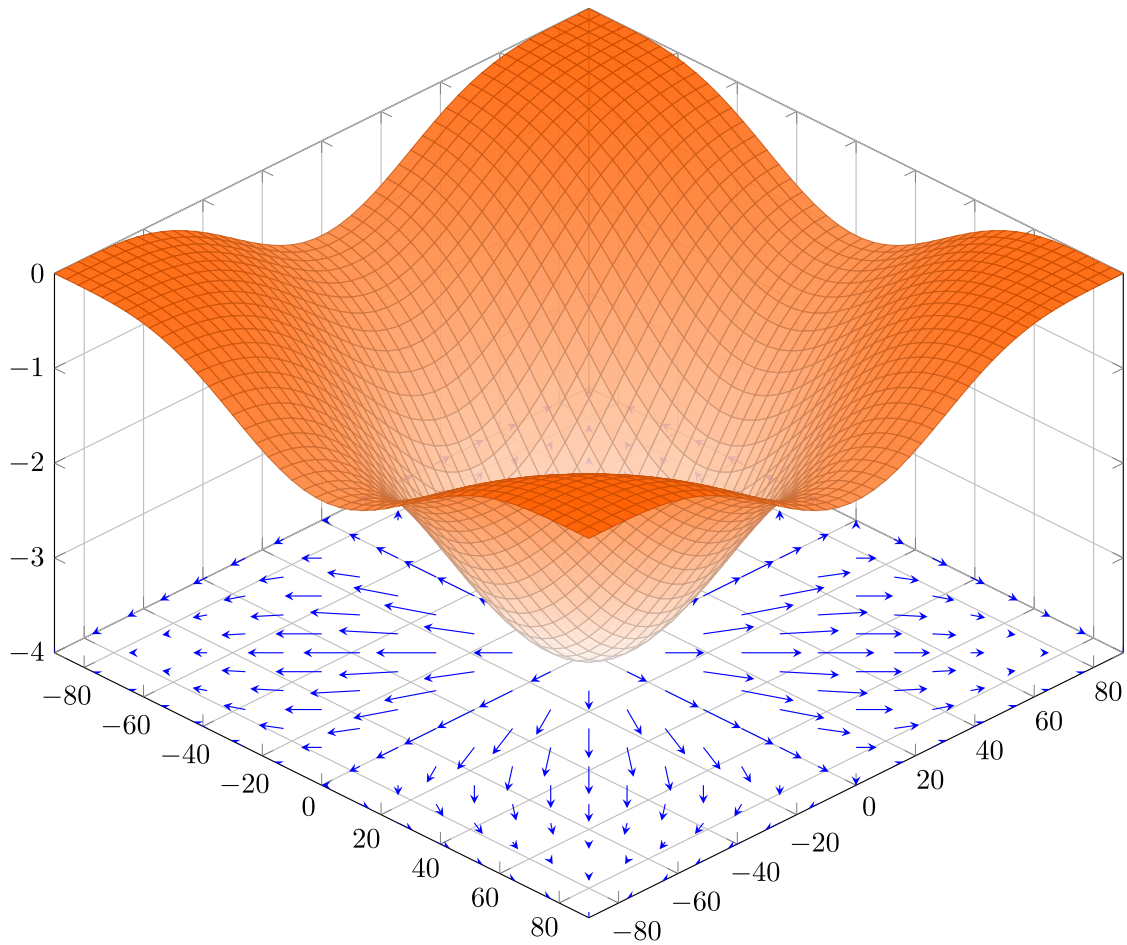
$$\nabla L(\beta) = \begin{bmatrix} \frac{\partial L}{\partial \beta_0} & (\beta) \\ \vdots & \\ \frac{\partial L}{\partial \beta_p} & (\beta) \end{bmatrix}$$

💡 Hence the name *Gradient Descent*

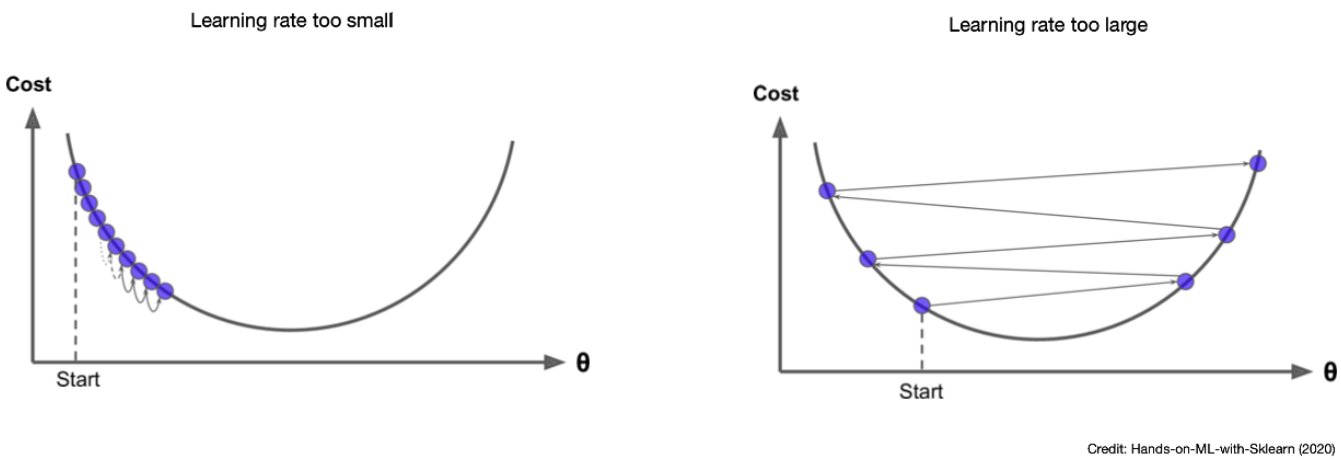
$$\text{Gradient Descent - vector formula}$$

$$\beta^{(k+1)} = \beta^{(k)} - \eta \, \nabla L(\beta^{(k)})$$

The
blue
arrows
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/)](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)](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:

$$\frac{\partial SSR}{\partial \beta_0}(\beta) = \sum_{i=1}^{n} -2(y_i - \hat{y}_i)$$

$$\frac{\partial SSR}{\partial \beta_1}(\beta) = \sum_{i=1}^{n} -2X_1^{(i)}(y_i - \hat{y}_i)$$

and more generally speaking:

$\nabla SSR(\beta) = -2X^T(y - \hat{y}$

$$\nabla SSR(\beta) = -2X^T(y - X\beta)$$

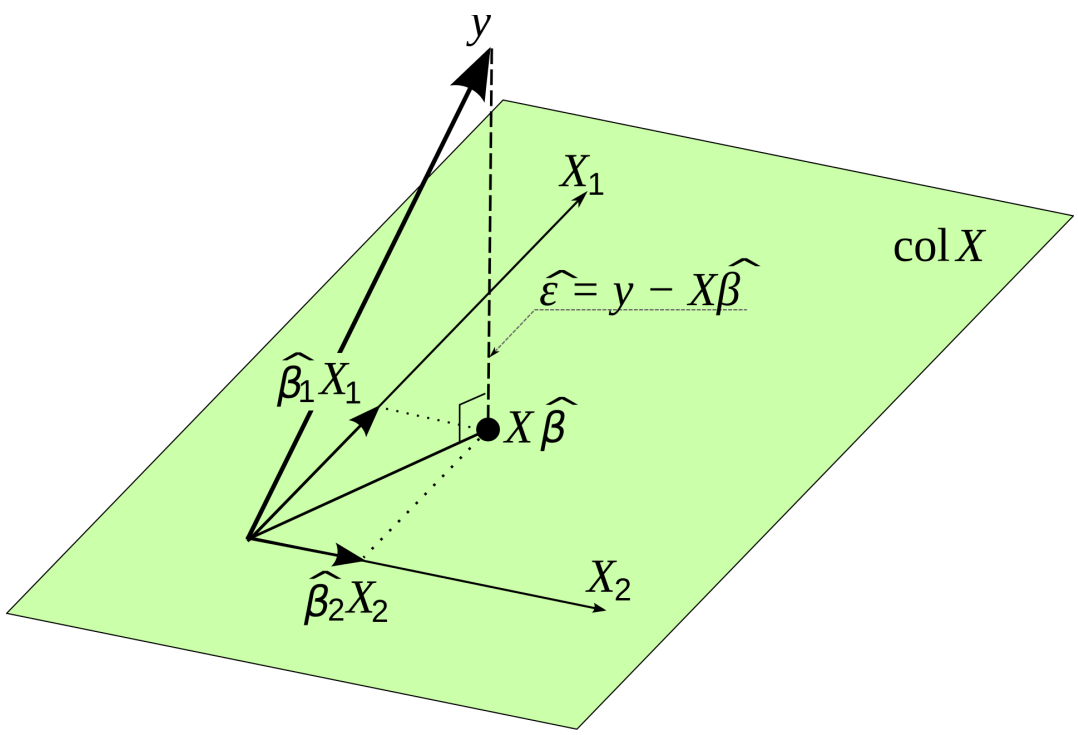
☝️ Because the formula for the gradient of the loss is so easy to compute, Gradient Descent is very efficient for OLS regressions

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

(optional) Can we get a geometric intuition about the best β in OLS? 📐

$$X = \begin{array}{c} \begin{array}{cccc} X_0 & & & \end{array} \\ \begin{bmatrix} 1 & | & & | \\ \vdots & X_1 & \cdots & X_p \\ 1 & | & & | \end{bmatrix} \end{array} \qquad \beta = \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_p \end{bmatrix} \qquad X\beta = \begin{bmatrix} \beta_0 X_0^{(1)} + \beta_1 X_1^{(1)} + \beta_2 X_2^{(1)} + \cdots \\ \\ \beta_0 X_0^{(m)} + \beta_1 X_1^{(m)} + \cdots \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_m \end{bmatrix} = y_{pred}(OLS)$$

$$(m, p+1) \qquad (p+1, 1) \qquad (m, 1)$$



$\hat{y}$
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
$(y - \hat{y}$
❓

💡 Pythagoras tells us that the shortest path is

> the **orthogonal projection** of
> y_{true}
> into the hyperplane (col X)

3. Other Solvers

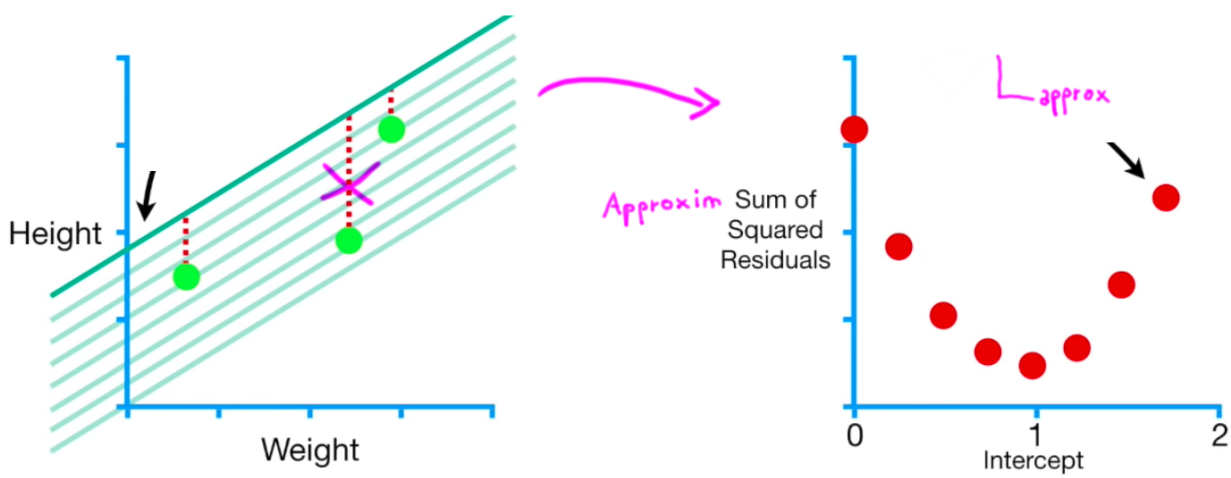
Let's recall the definition of the gradient
(for OLS)

$$\nabla L(\beta) = \begin{bmatrix} \frac{\partial L}{\partial \beta_0}(\beta) \\ \vdots \\ \frac{\partial L}{\partial \beta_p}(\beta) \end{bmatrix}$$

👎 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_{mini} (e.g. the first 16 observations):
  1. Compute the gradient of the mini-batch ∇L_{mini}
  2. Use this gradient to update $\beta^{(k+1)} = \beta^{(k)} - \eta \, \nabla L_{mini}(\beta^{(k)})$
  3. Move to next X_{mini} (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
```

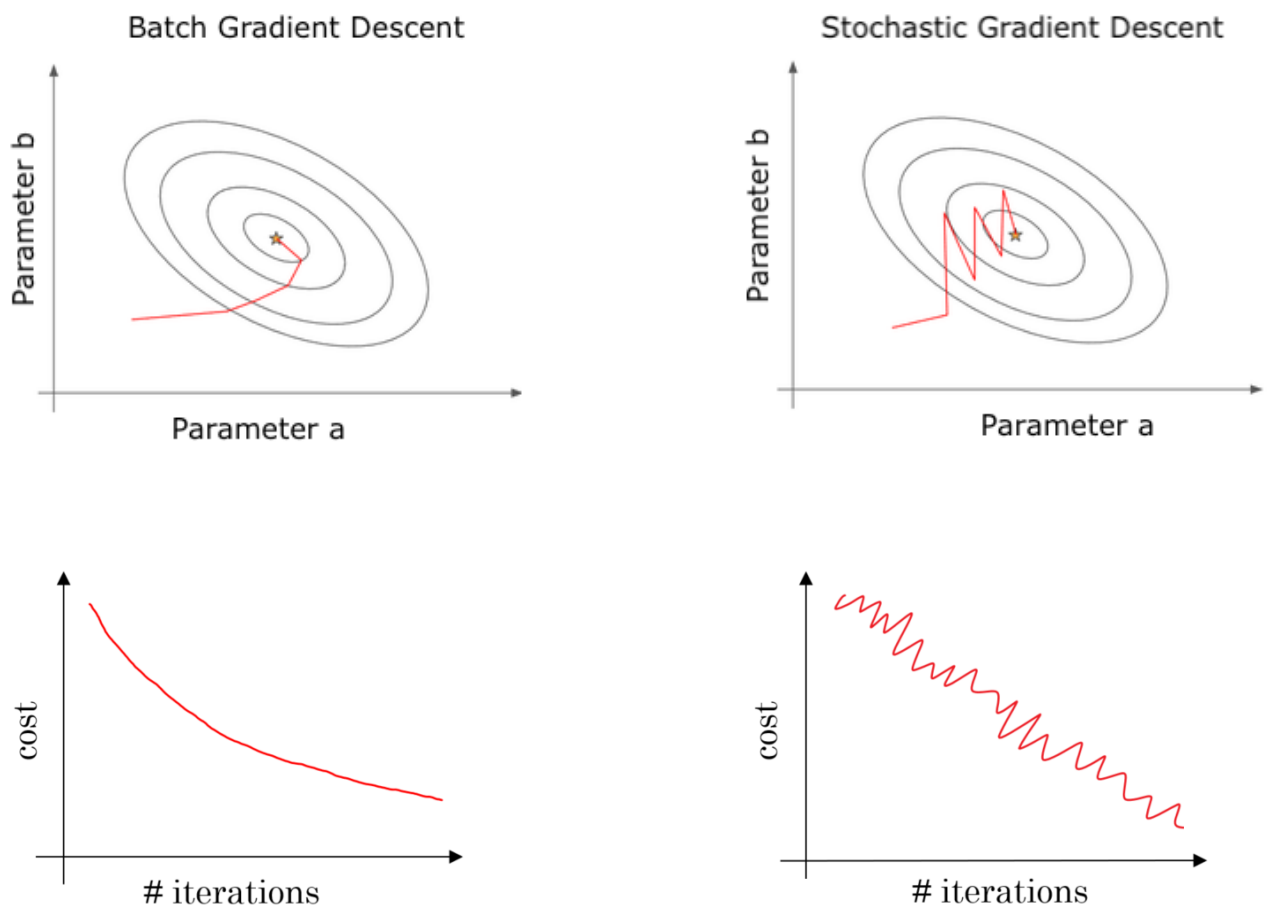
- Loop one-by-one over all n observations
  - Select a **single, randomly selected data point**
  - Compute the loss/gradient for this single point
  - Update β
- Once all n observations have been viewed, repeat another epoch

💻 Let's code it for our OLS

In [ ]:
```
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)](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)](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.

In [ ]:
```
from sklearn.linear_model import SGDRegressor, LinearRegression

lin_reg = LinearRegression() # OLS solved by matrix inversion (SVD met
hod)

lin_reg_sgd = SGDRegressor(loss='squared_error') # OLS solved by SGD
```

In [ ]:
```
from sklearn.datasets import make_regression

# Create a "fake problem" to solve
X, y = make_regression(n_samples=10000, n_features=1000)
```

In [ ]:
```
%%time
lin_reg.fit(X,y)
```

```
CPU times: user 8.47 s, sys: 403 ms, total: 8.88 s
Wall time: 2.07 s
```

Out[ ]:
```
LinearRegression()
```

In [ ]:
```
%%time
lin_reg_sgd.fit(X,y)
```

```
CPU times: user 182 ms, sys: 2.85 ms, total: 184 ms
Wall time: 189 ms
```

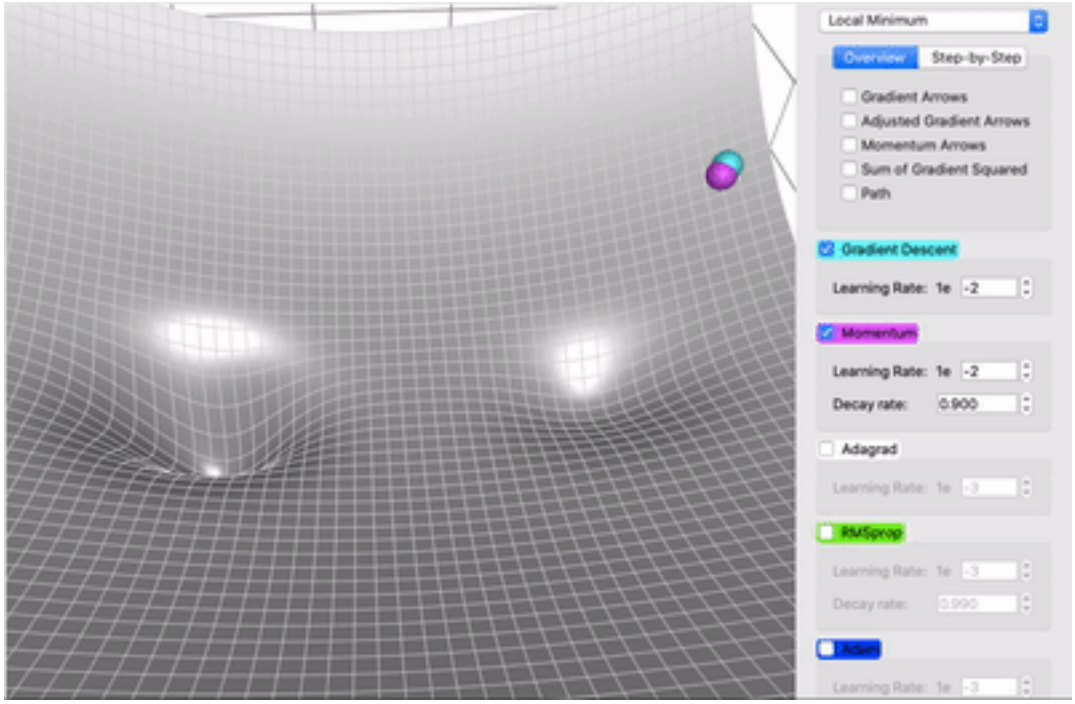
Out[ ]:
```
SGDRegressor()
```

✅ Gradient Descent performs better than matrix inversion when feature number p is large

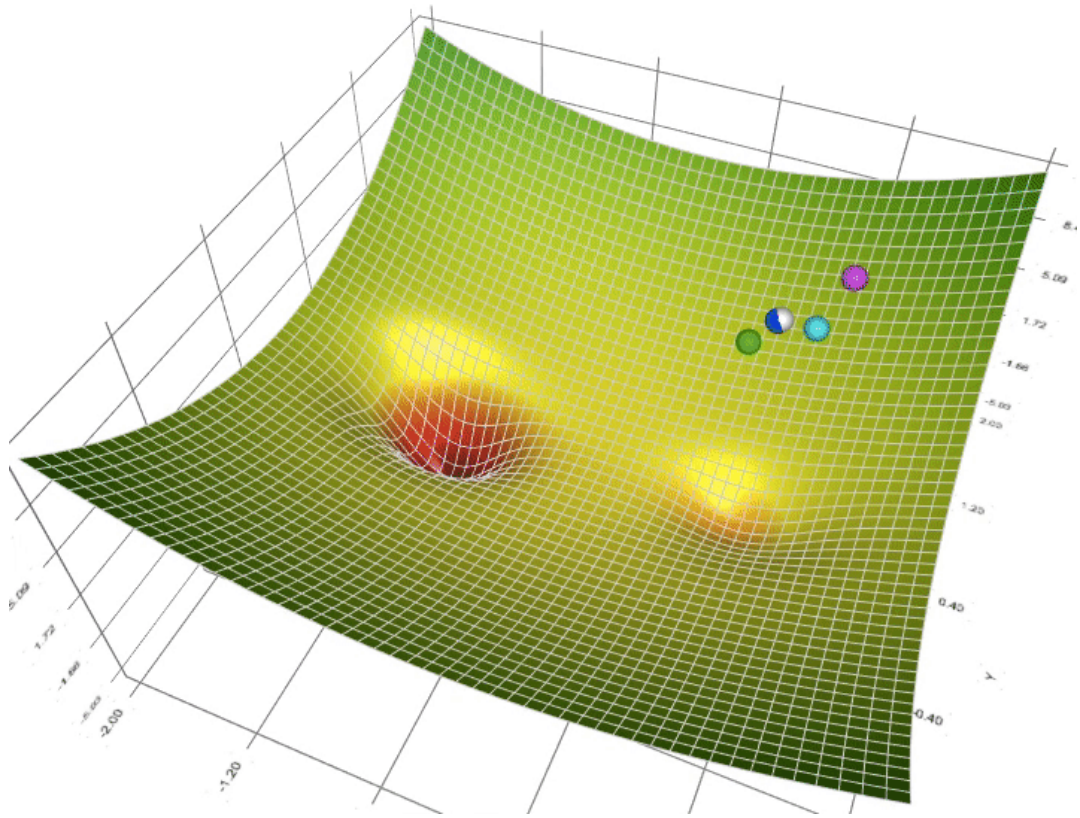
✅ SGD scales even better when the number of observations n is large

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)](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')
```

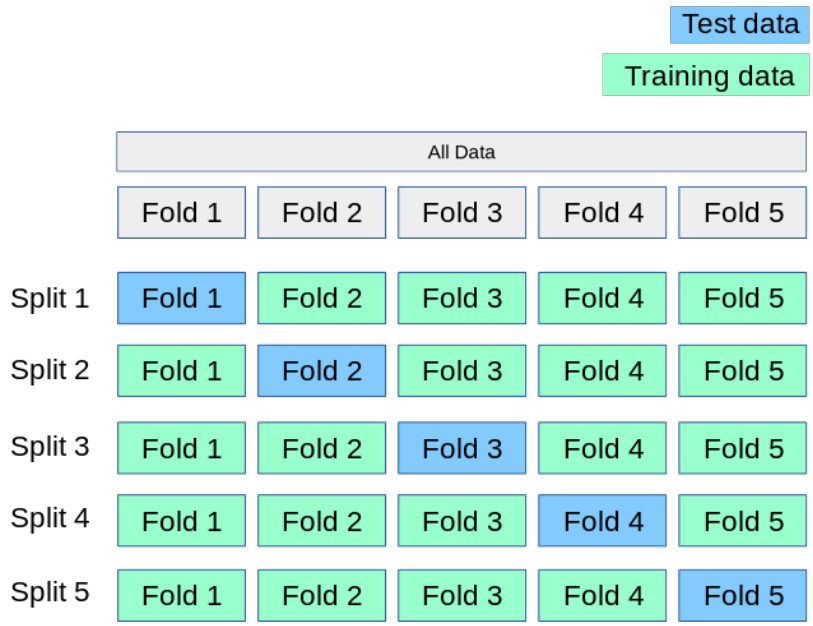
- The same model class on Sklearn can be instantiated with various attributes for `loss`

```
SGDClassifier(loss='log')
SGDClassifier(loss='hinge')
```

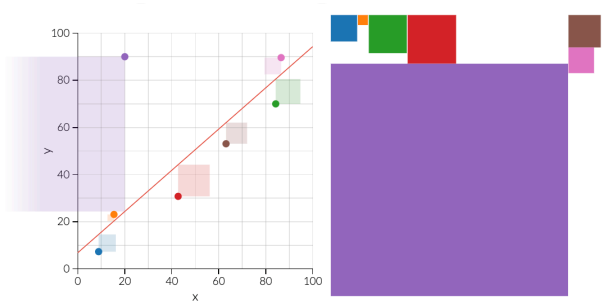
- Classification and Regression have different Loss Functions by nature

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.)

	Predicted **0**	Predicted **1**
Actual **0**	TN	FP
Actual **1**	FN	TP

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)

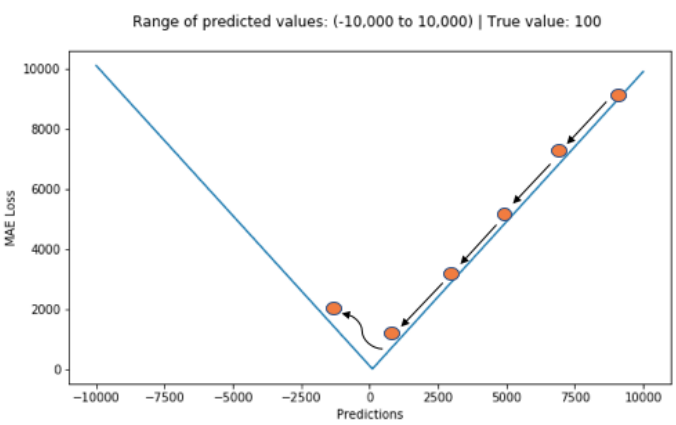
👉 "Accuracy" for instance can never be used as a loss metric

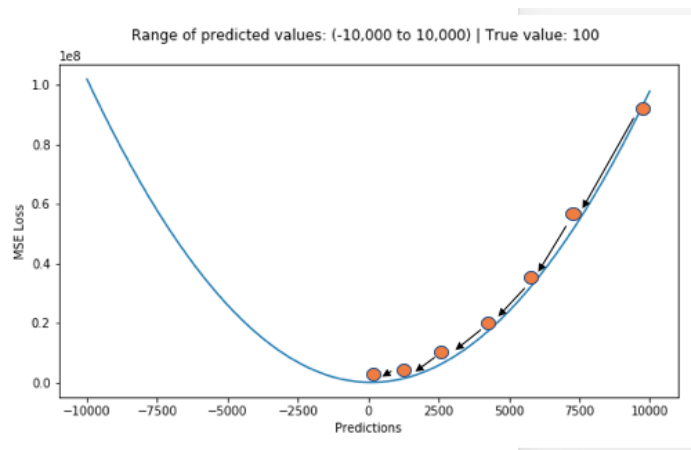
3.1 Regression Loss Functions

L1 Loss (MAE) vs L2 Loss (MSE)

$$L_1 = MAE = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i|$$

$$L_2 = MSE = \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$



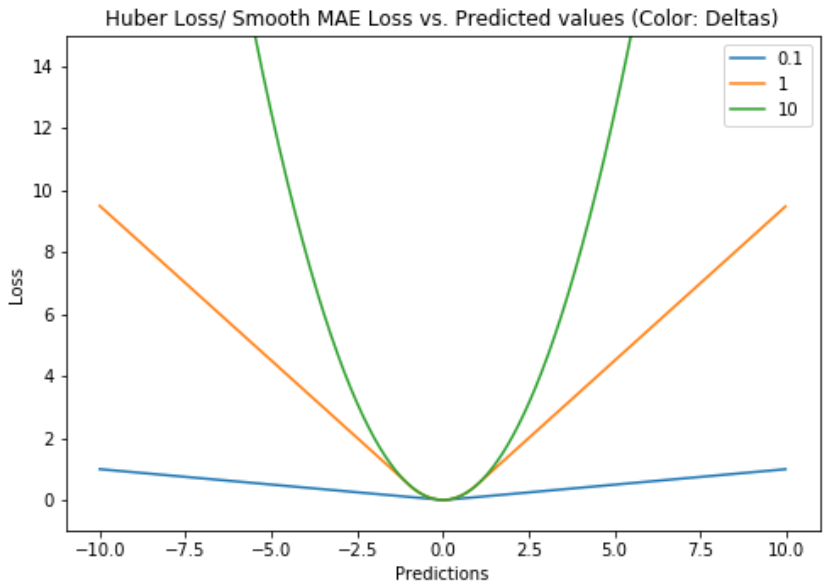


- MSE is very sensitive to outliers, MAE is less strict
- MAE requires a Learning Rate
  η
  which decreases at every epoch

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

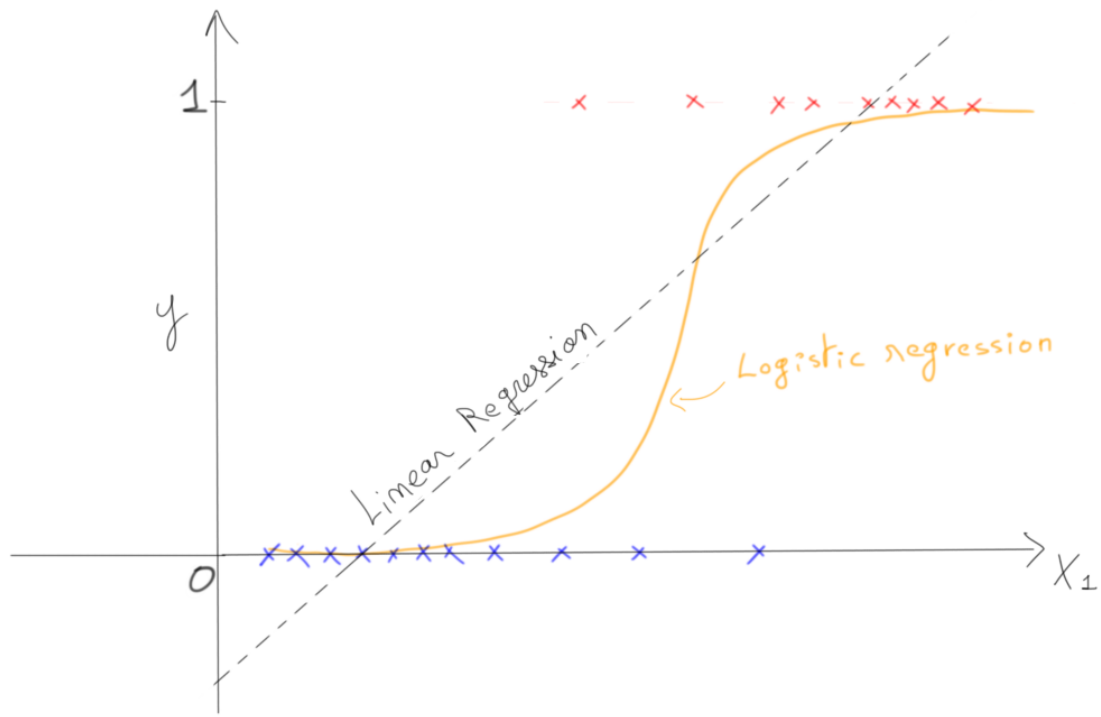
- MAE which becomes MSE when error is small, typically adjustable by hyperparameter `epsilon` (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html#sklearn.linear_model.SGDF float%2C%20default%3D0.1)
- Adjustable for outliers
- Slope can be used as an indicator of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

$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y}^2 & if\ |(y - \hat{y}| < \delta \\ \delta(|(y - \hat{y}| - \frac{1}{2} & otherwise \end{cases}$$



3.2 Classification Losses

Logistic Classifiers



- We want to predict a binary vector $y = [0, 0, 1, 0, \ldots, 1]$ of size n
- We model it by a vector $\hat{y}$

❓ **What loss to compute** between $\hat{y}$ and y ?

Logistic classifiers want to maximize this **product**

$$\prod_{i \text{ when } y_i = 1} \hat{y}_i$$

- $\hat{y}_i$ **close to 1** when $y_i = 1$
- $(1 - \hat{y}_i$ **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 = $\hat{y}_i$

👉 Called the **Likelihood** of observing the true y under some hypothesis function h

(See [Logistic Regression lecture (04-Decision-Science_04-Logistic-Regression.slides.html?title=Logistic-Regression&program_id=10#/2/5/1)](04-Decision-Science_04-Logistic-Regression.slides.html?title=Logistic-Regression&program_id=10#/2/5/1))

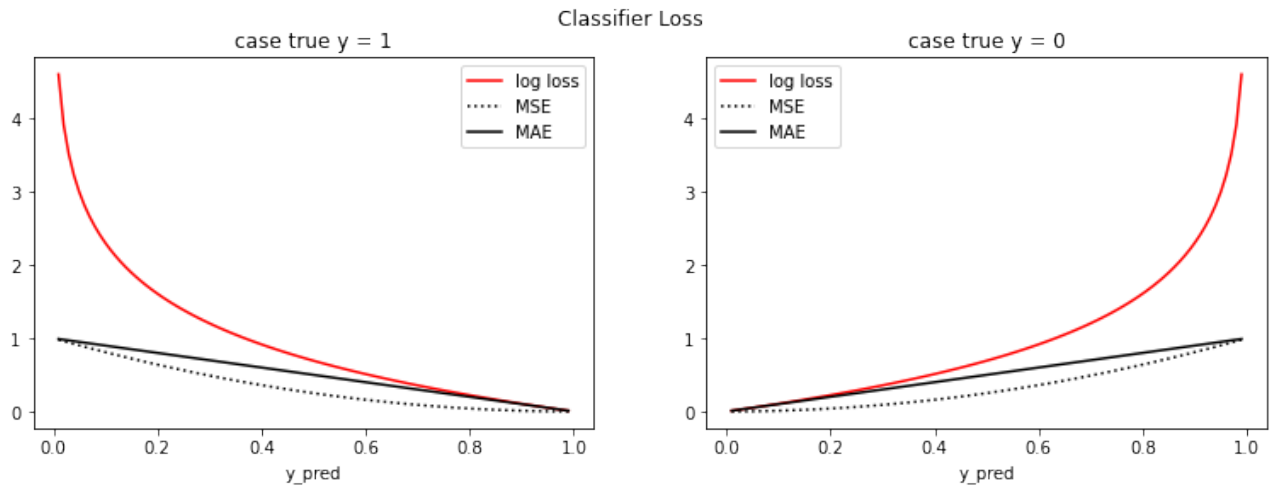
It's easier to maximize the log instead (**log-likelihood**)

$$log(\prod_{y_i=1}\prod_{y_i=0}) = \sum_{i \text{ when } y_i=1} log(\hat{y}_i) + \sum_{i \text{ when } y_i=0} log(1-\hat{y}_i) = \sum_{i=0}^{n} y_i log(\hat{y}_i) + (1-y_i) log(1-\hat{y}_i)$$

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

$$Log\ Loss = -\frac{1}{n}\sum_{i=0}^{n} y_i log(\hat{y}_i) + (1-y_i)log(1-\hat{y}_i)$$

- $y = 1 \Rightarrow Log\ Loss = -log(\hat{y}$
- $y = 0 \Rightarrow Log\ Loss = -log(1-\hat{y}$



☝️ Infinitely penalize wrong predictions

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

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

$\nabla LogLoss_{sigmoid} = -\frac{2}{n}X^T(y-\hat{y}$

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

$\nabla MSE_{linear} = -\frac{2}{n}X^T(y-\hat{y}$

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

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

$\hat{y}_{sigmoid} = \frac{1}{1+e^{-X\beta}}$

Other (non-logistic) classifiers exist!

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

All have different losses!

5. Summary

Problem setting

- X
  = features
- y
  = target =
  $h(X, \beta) + error$
- h
  = hypothesis function (Linear, Sigmoid, Neural Network, etc.)

Parameters of the model:
β

- Computed automatically during `.fit()`
- by minimizing
  $L(\beta)$

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
$h(X, \beta)$

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%20Lea
- 📚 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! 🚀