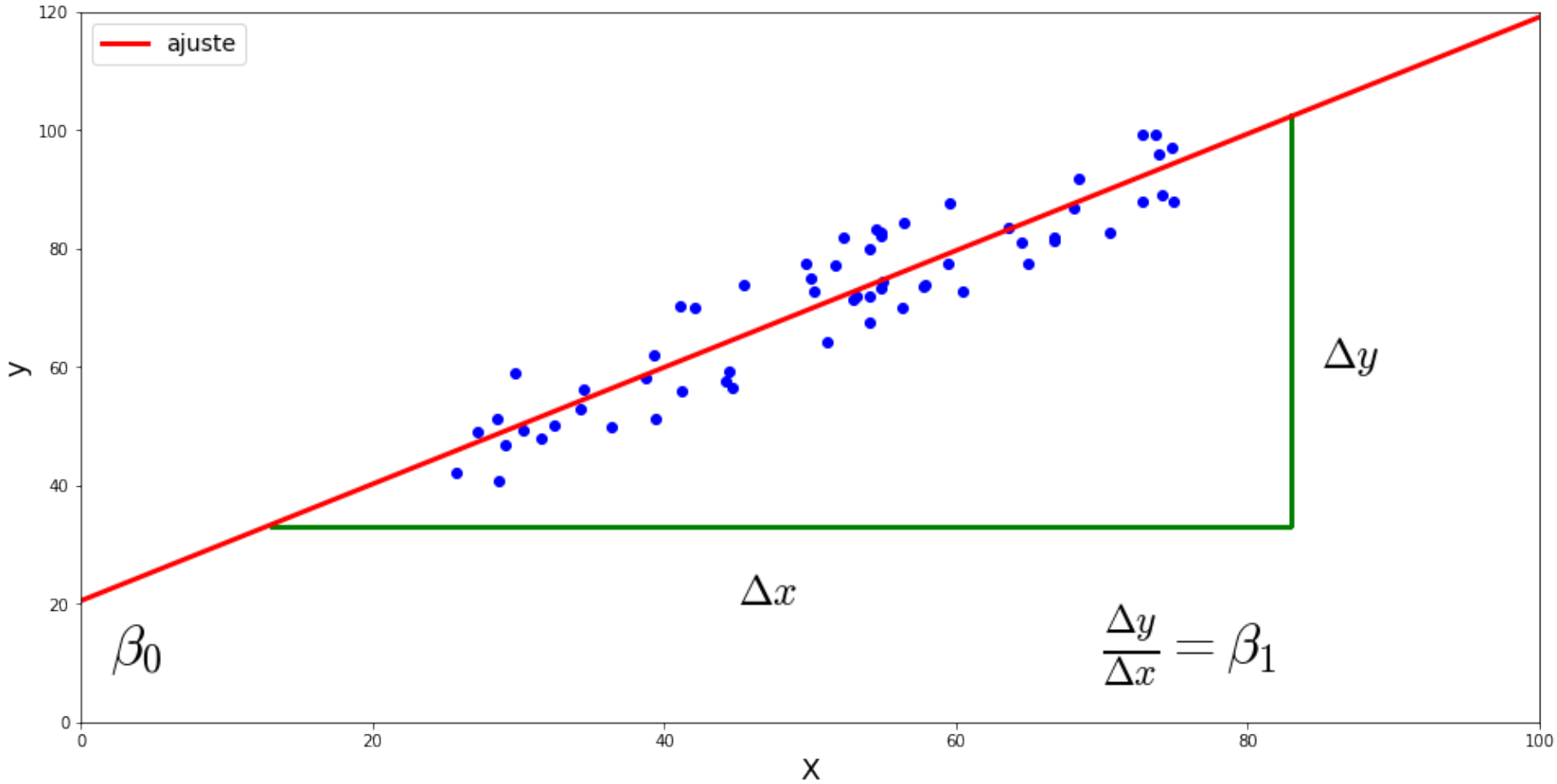
Linear regression

Carl McBride Ellis

carl.mcbride@u-tad.com

Regresión lineal: Ejemplo en 1D (*"regresión lineal univariante"*):



The linear regression model:

$$\hat{y} = \beta_1 x + \beta_0 + \epsilon \text{ we assume that } \mathbb{E}(\epsilon) = 0, \text{ p.ej. } \sim \mathcal{N}(0, \sigma^2)$$

For each *'feature'* x there is an adjustable parameter, β_1, which corresponds to its slope or gradient (often called the **weight**) as well as a intersection term β_0 (often called the **bias**).

This is a 'parametric' model; here we assume the relation between the dependent and the independent variables is linear, and the noise is from a (Gaussian) parametric distribution.

Our data, especially when working with tabular data, is usually stored in a `DataFrame` (`df`):

	feature_1	feature_2	feature_3	feature_4	target
0	-0.285624	-1.109467	-0.704222	1.311941	1.011175
1	0.115344	-0.582376	0.850097	-0.078389	-1.017000
2	0.168309	0.781371	0.564461	1.798512	-0.339078
3	-0.364210	0.427111	1.909429	1.822293	-0.702626
4	-0.054081	2.215833	-0.384593	0.634965	-1.837298
5	-0.104199	0.828819	0.553767	-0.325654	0.885174
6	-0.143291	-1.508430	-0.183994	0.063684	0.234775
7	-1.076164	-0.233883	0.404790	-1.287040	-1.522206
8	0.209559	-0.778077	-0.207529	0.560157	-0.074932
9	0.222729	-0.549980	-0.459486	1.581879	0.921162

(In deep learning is it often stored in numpy arrays or in tensors).

Solver: Ordinary least squares (OLS)

$$y = \beta_1 x + \beta_0$$

$$\beta_1 = \frac{\sum((x_i - \overline{x})(y_i - \overline{y}))}{\sum((x_i - \overline{x})^2)}$$

$$\beta_0 = \overline{y} - \beta_1 \overline{x}$$

(The technique of least squares was invented by Carl Friedrich Gauss and Adrien-Marie Legendre around 1802-1809)

Practical session

Practical session: linear regression with python

`notebook_ML_OLS_numpy.ipynb`

Objective: convert the equations for ordinary least squares into python code, and visually inspect the result

Our `DataFrame` is basically an augmented coefficient matrix:

🍏	🍇	total
3🍏	3🍇	21€
5🍏	2🍇	20€
8🍏	6🍇	46€

(Note: this example is overdetermined)

Say we were paying by card, and there was also a fixed transacion fee of 2€

🍏	🍇	🪙	total
3🍏	3🍇	1🪙	23€
5🍏	2🍇	1🪙	22€
8🍏	6🍇	1🪙	48€

This constant term would be our β_0

We can treat our `DataFrame` as if it were a system of linear equations where the values of x_{mn} represent the coefficients:

$$\begin{cases} \beta_1.x_{0,1} & + & \beta_0,1 & = & y_0 \\ \beta_1.x_{1,1} & + & \beta_0,1 & = & y_1 \\ \beta_1.x_{2,1} & + & \beta_0,1 & = & y_2 \end{cases}$$

where we wish to find the β_0 and β_1 associated with each column.

(The constant column can be created using `statsmodels add_constant`)

Written as matrices:

$$\mathbf{X} = \begin{bmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix}$$

($\mathbf{X}$ is known as the '*design matrix*')

In more dimensions

Multiple linear regression (now with n features):

$$\mathbf{X} = \begin{bmatrix} 1 & x_{01} & x_{02} & \cdots & x_{0n} \\ 1 & x_{11} & x_{12} & \cdots & x_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix}$$

and in matrix notation:

$$\mathbf{X}\beta = \mathbf{y}$$

A solver for system of linear equations based on linear algebra is the *"Normal equation"*.
This is the closed-form solution for the matrix equation $\mathbf{X}\beta = \mathbf{y}$:

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

where $(\mathbf{X}^T\mathbf{X})$ is known as the *"Gram"* or *"normal"* matrix.

In python code:

```
X1 = np.column_stack(( X, np.ones(len(X)) ))
betas = np.linalg.inv(X1.T @ X1) @ X1.T @ y
print("beta_n,...,beta_0", betas)
```

Polynomial regression

Not always (...basically never!) is the relationship between x and y is a straight line. We can add power terms *i.e.*

$$\hat{y}(\mathbf{X}) = \beta_2 \mathbf{X}^2 + \beta_1 \mathbf{X} + \beta_0$$

for example via

```
df["x_squared"] = df["x"]**2
```

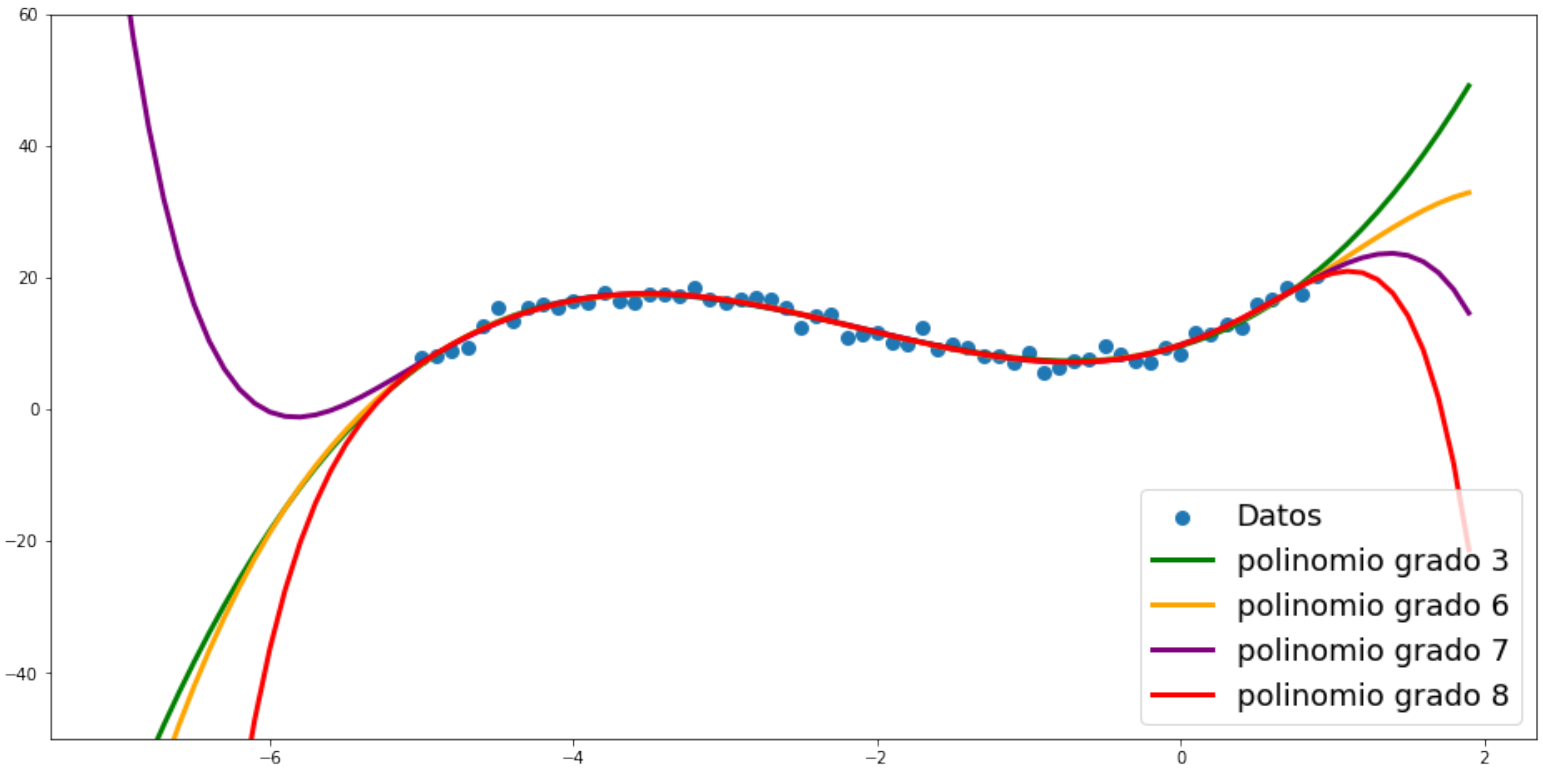
and with this new feature we introduce curvature into our model

Note: This is still a linear function in terms of β, *i.e.* the fruits

🍍	🍏	🍇	total
9 🍍	3 🍏	3 🍇	21€
25 🍍	5 🍏	2 🍇	20€
64 🍍	8 🍏	6 🍇	46€

Extrapolation

Warning! Regression with high degree polynomials have unstable 'tails':



(Interpolation: within the convex hull, and extrapolation: outside)

Linear regression using the scikit-learn

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression(fit_intercept=True)
# fit the model
regressor.fit(X, y)
# make predictions
y_pred = regressor.predict(X_test)
```

(The `fit_intercept=True` option will create the column of 1's needed to calculate β_0)

To tablulate the β's

```
import eli5
feature_names = X.columns.tolist()
eli5.show_weights(regressor,
                  top=None,
                  feature_names = feature_names )
```

In Colab you will need to: `!pip install -q eli5`

and using the statsmodels library

(In Colab you will need to: `!pip install -q statsmodels`)

```
import statsmodels.api as sm

X = sm.add_constant(X) # add the column of 1's
model = sm.OLS(y,X)
results = model.fit()
print(results.summary())
```

Explainability

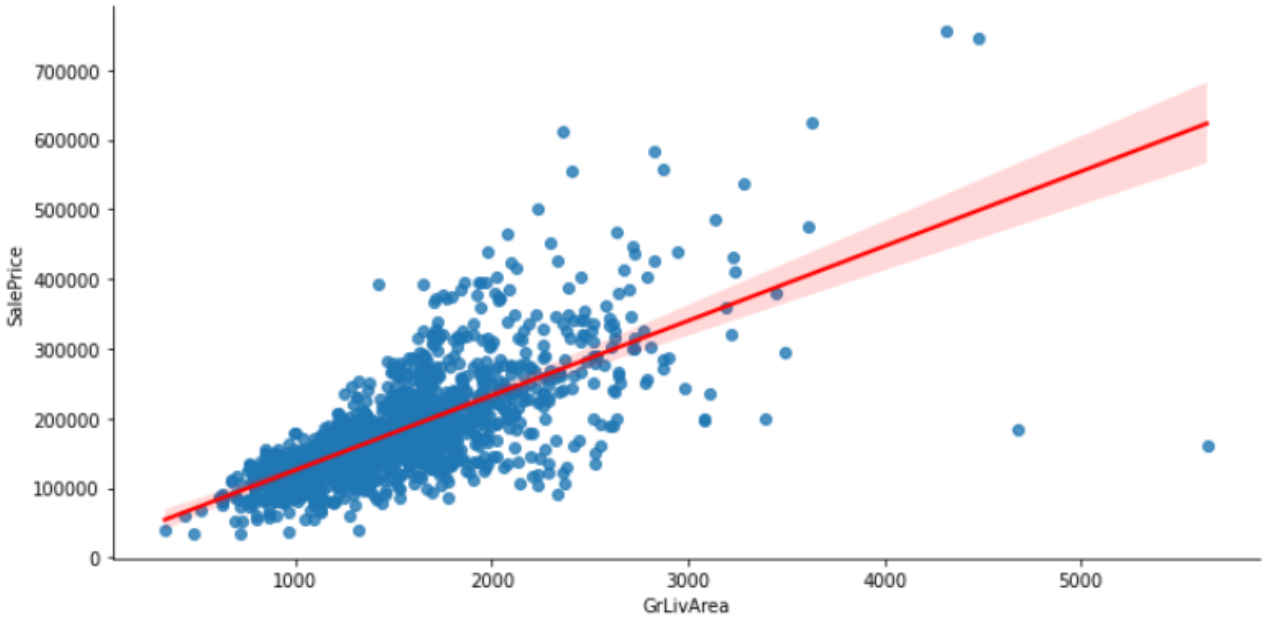
The β_1, i.e. the inclinations, provide valuable information regarding the relative importance (weight) of each feature.

For example, if a feature has $\beta_1 = 0$, then this feature does not form part of the model, and has no influence when it comes to predicting $\hat{y}$.

Note: For a correct interpretation of the relative importance of the regression coefficients (β) it is important that all of the features are on the same scale.

One can calculate the confidence intervals (CI) of the coefficients *i.e.* by using `statsmodels.regression.linear_model.OLS` with `conf_int`

To assist with our visual inspection ('EDA') we can use the `seaborn.lmplot`

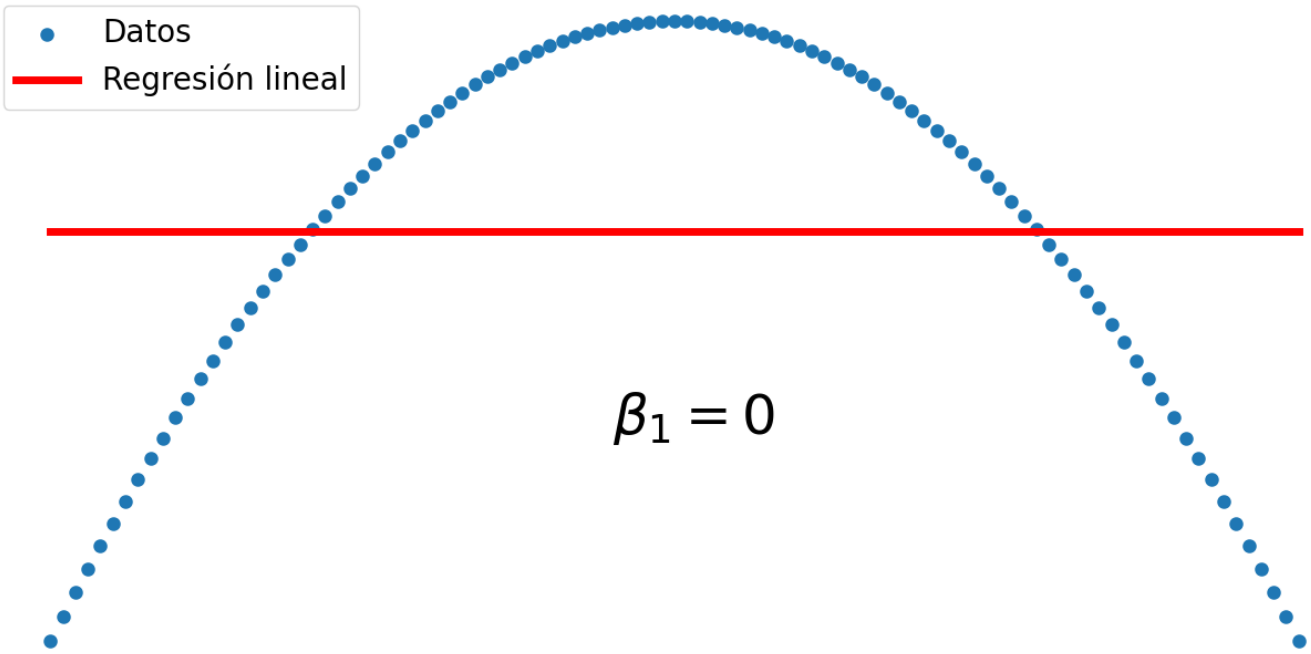


which can provide the confidence interval (`ci`) of the fit

```
sns.lmplot(x=X, y=y, data=data, ci=95)
```

The four basic assumptions of a good linear regression:

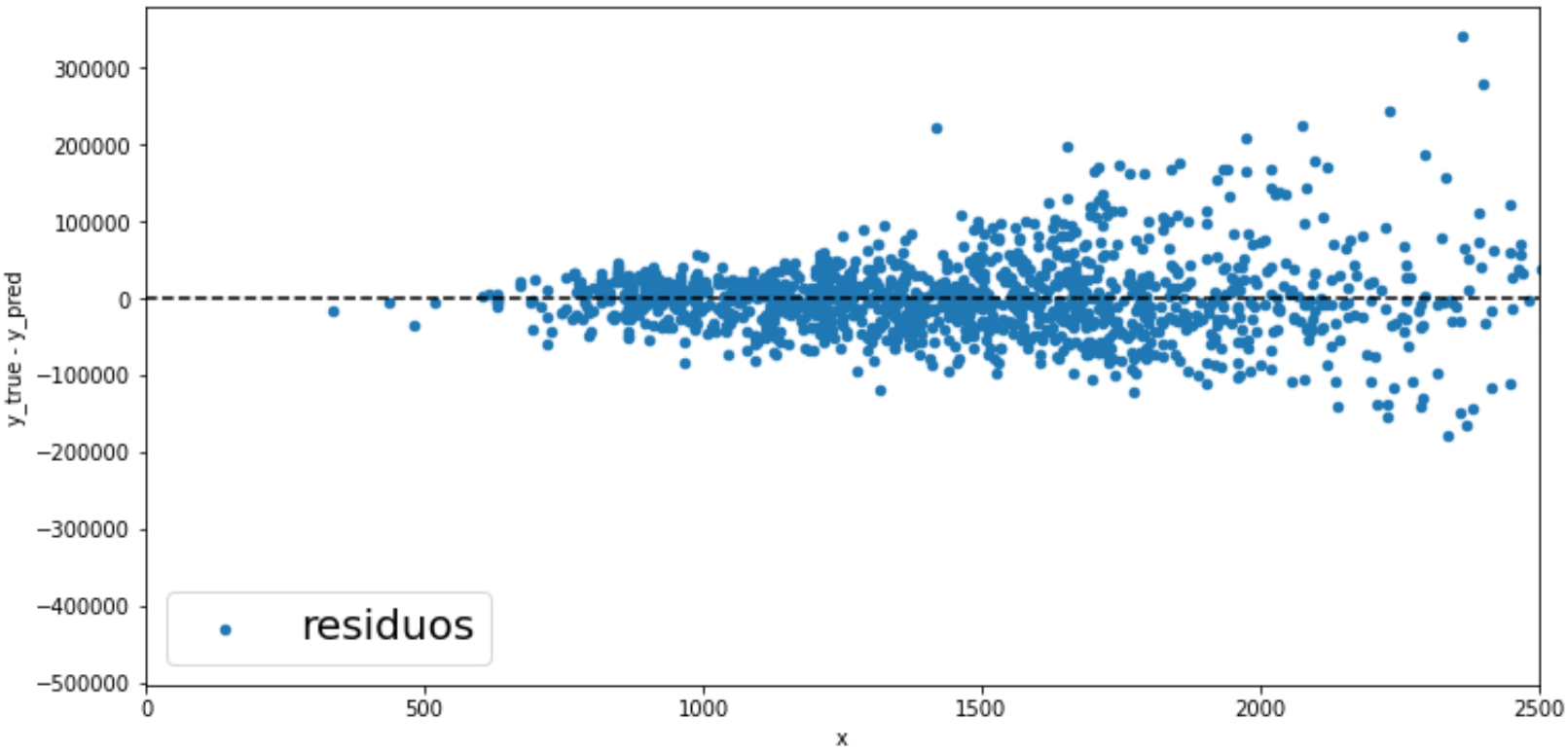
- Suffice to say, linear regression works best when the relation between the independent variable (x) and the dependant variable(y) is actually linear



furthermore, to obtain reliable results for the regression coefficients and the confidence intervals the residuals (errors) should be:

- **i.i.d**: each error is independent of the other errors (no autocorrelation)
- **belong to a Gaussian distribution** $(\forall\ i \in n, \varepsilon_i \sim \mathcal{N}(0, \sigma^2))$
- **homoscedasticity**: the errors have a constant variance $(\sigma^2 \neq f(x))$

Example of heteroscedasticity



Possible 'solution': apply a transformation, for example take the log or square root of the dependent variable (y)

(Note: this is not such a good idea if there is more than one independent variable)

Loss and cost

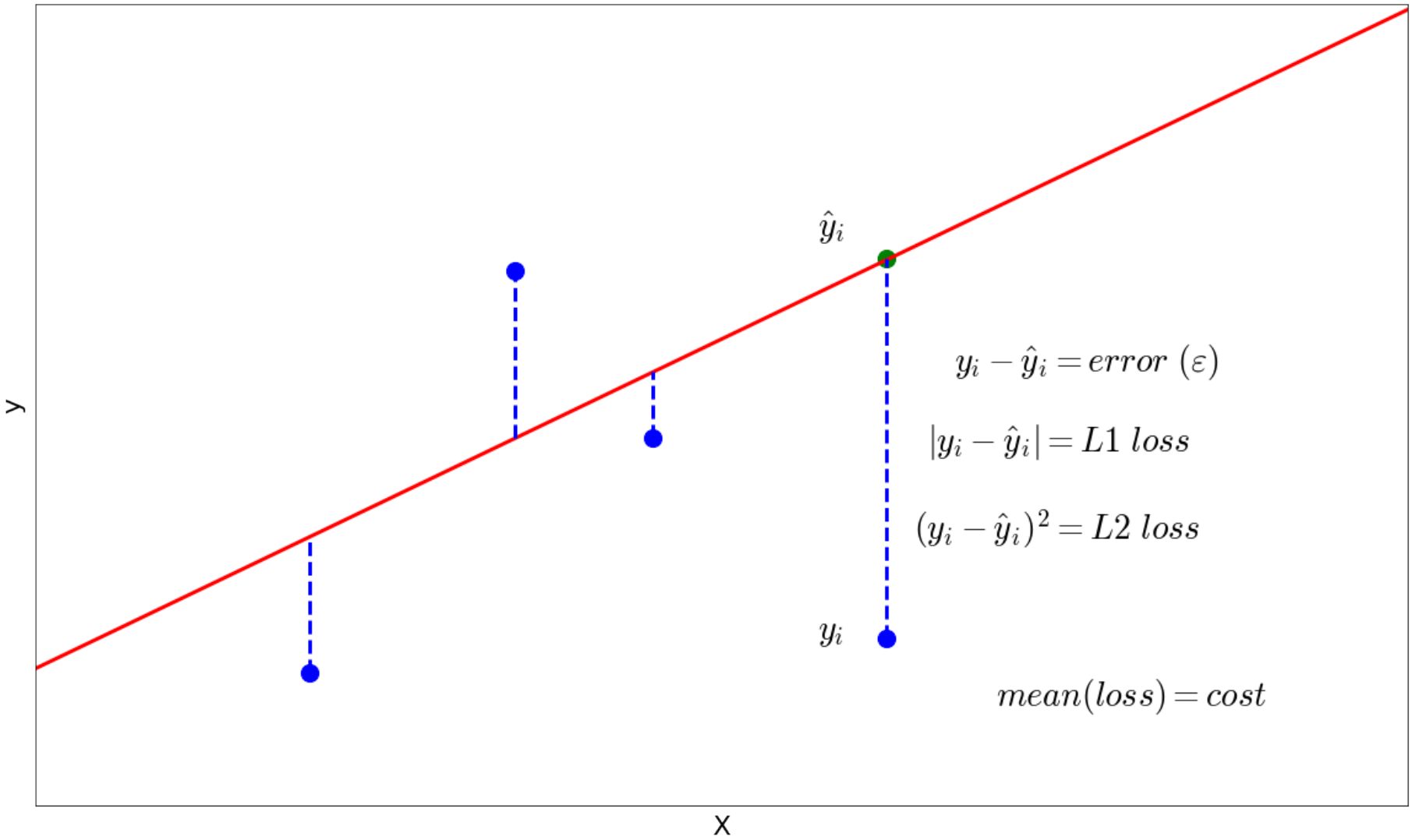
How does the algorithm find the optimal solution?:
The *loss* ($\mathcal{L}$) and *cost* (J) functions

- *loss* is a distance measure* between the points $\hat{y}$ and y
- *cost* is the mean of all of the losses: $\bar{\mathcal{L}}$

(sometimes the loss function is known as the *error function*, in other words how bad are we doing

and the cost functions is known as the *objective function*)

*(Advanced material: norms: p-norm and Lebesgue spaces)


$\hat{y}_i$
y_i
$y_i - \hat{y}_i = error\ (\varepsilon)$
$|y_i - \hat{y}_i| = L1\ loss$
$(y_i - \hat{y}_i)^2 = L2\ loss$
$mean(loss) = cost$
y
X

A valid loss function should have the following properties; it should be symmetric, *i.e.*

$$L(\hat{y}, y) = L(y, \hat{y})$$

it should be positive

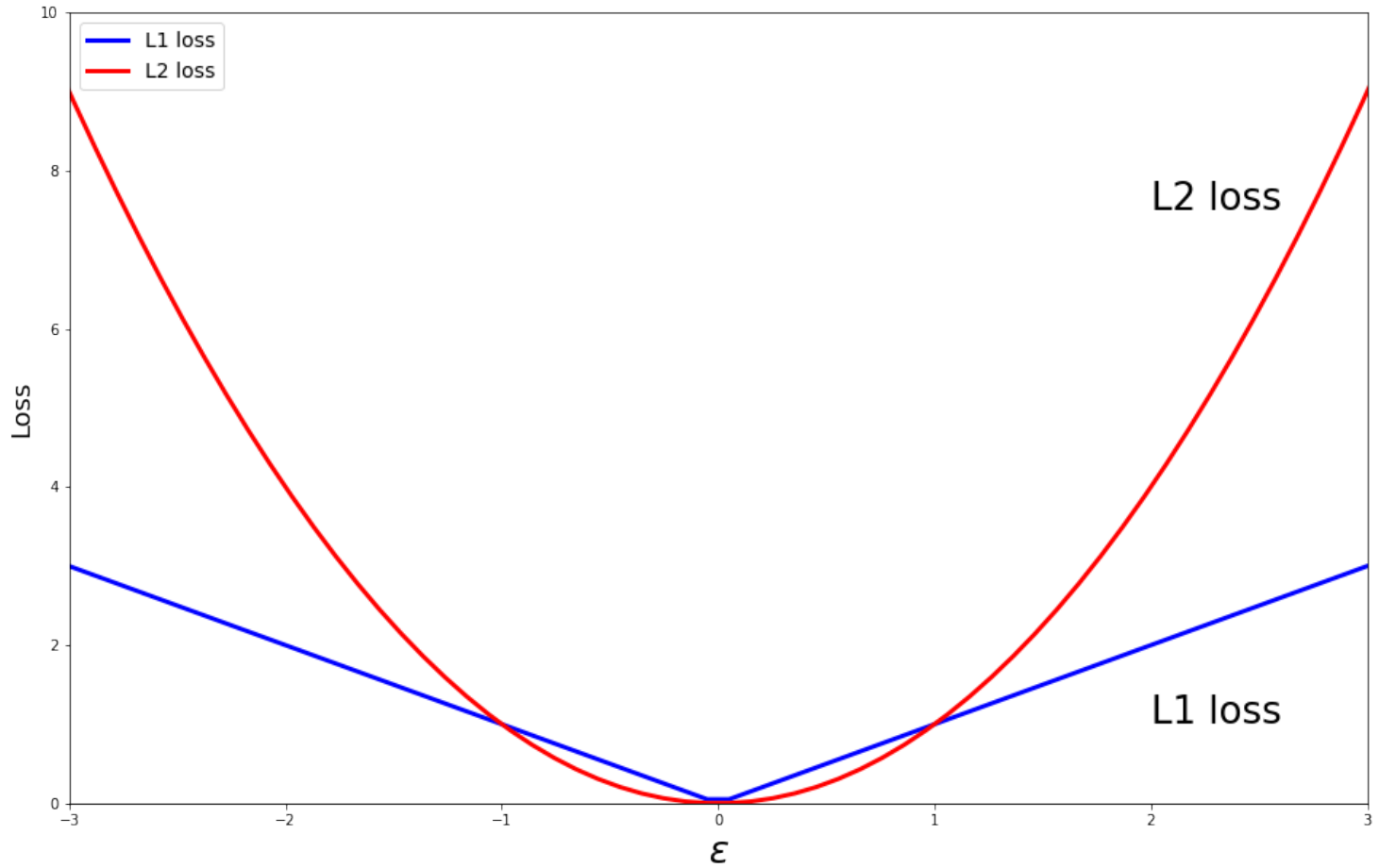
$$L(\hat{y}, y) > 0, \qquad \hat{y} \neq y$$

and should staisfy the triangle inequality

$$L(\hat{y}, y) \leq L(\hat{y}, z) + L(z, y)$$

i.e. $L(\hat{y}, y)$ is the shortest distance between $\hat{y}$ and y.

What do the loss functions look like?



- L1 loss is not very sensitive to 'outliers'
- L2 loss is sensitive to outliers as the distance is squared

In code:

$y \rightarrow$ `y_true`

$\hat{y} \rightarrow$ `y_pred`

`(y_true - y_pred)` is known as the error (ε) or "*residual*".

- L1 loss = AE (absolute error) = `abs(y_true - y_pred)`
- L1 cost = MAE (mean absolute error) = $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- L2 loss = SE (squared error) = `(y_true - y_pred)**2`
- L2 cost = MSE (mean squared error) = $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

RMSE (root mean squared error) = $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$

We wish to minimize the cost (or *'objective function'*) J

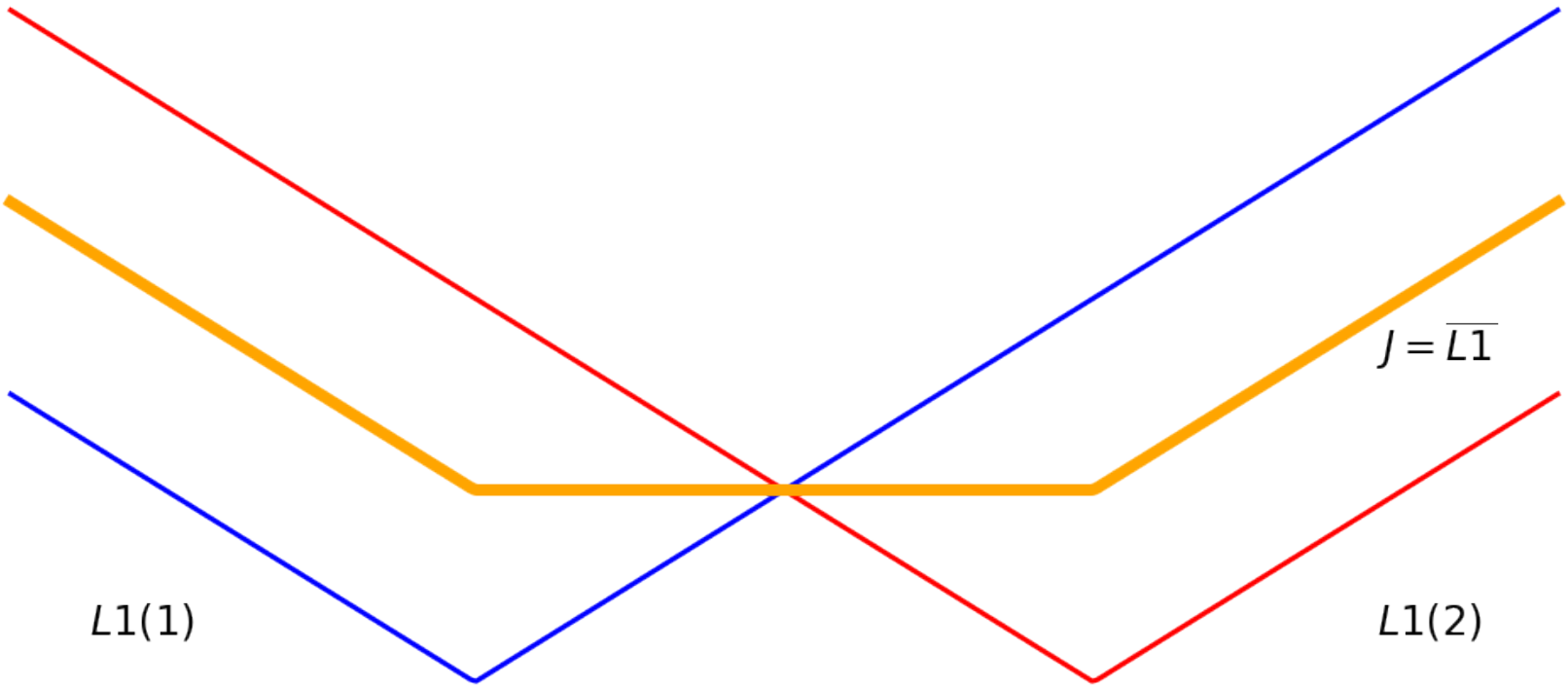
For example, in linear regression we use ordinary least squares, in other words the L2 loss and the L2 cost:

$$\begin{aligned} J(\beta) &= MSE \\ &= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \\ J(\beta_1, \beta_0) &= \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta_1 \mathbf{X}_i + \beta_0)^2 \end{aligned}$$

The optimal values of β_1 and β_0 are given by argmin(J)

(If we use the L1 loss we perform *median regression*, see quantile regression)

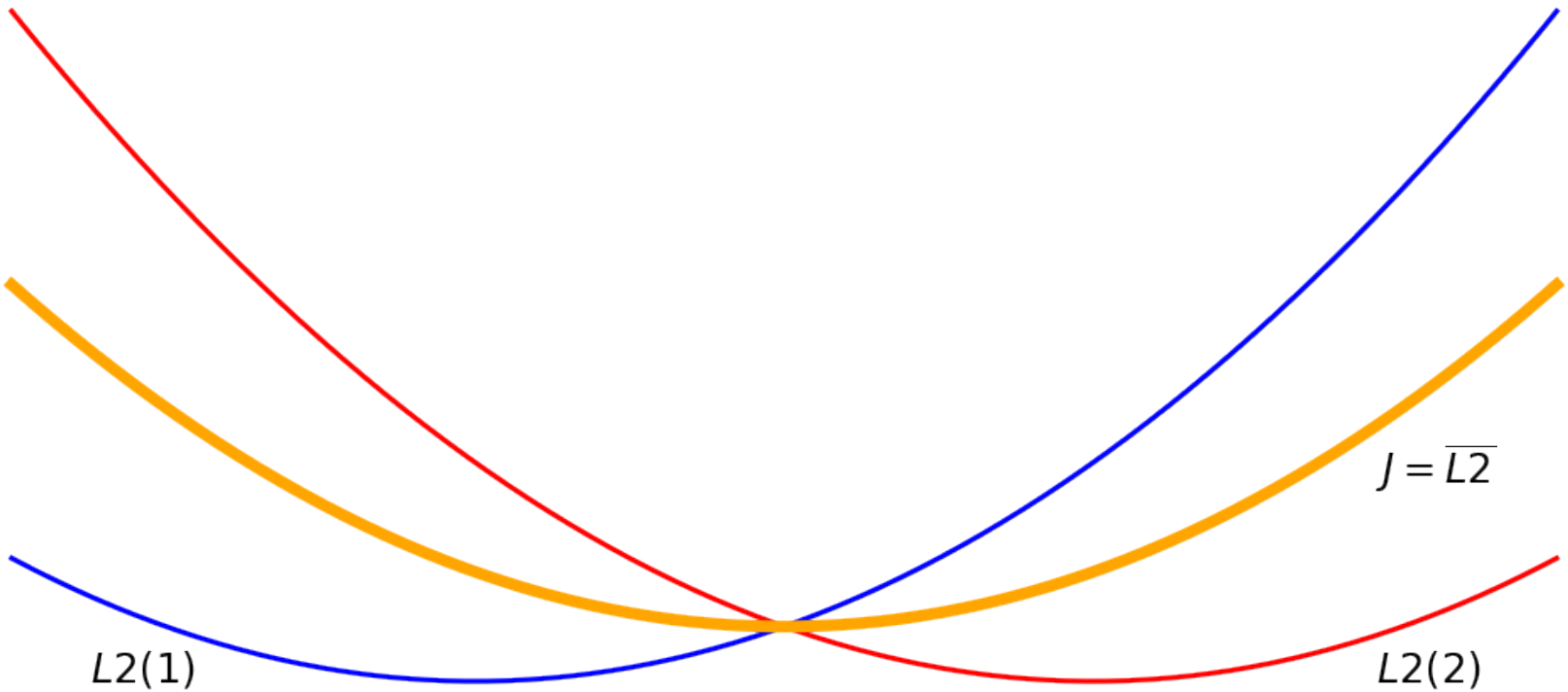
What form does the cost (J) have for the L1 loss for two points?



- J is piecewise-continuous
- the minima is multivalued

(The mean absolute error is a new member of the collection of objectives in XGBoost (October 2022))

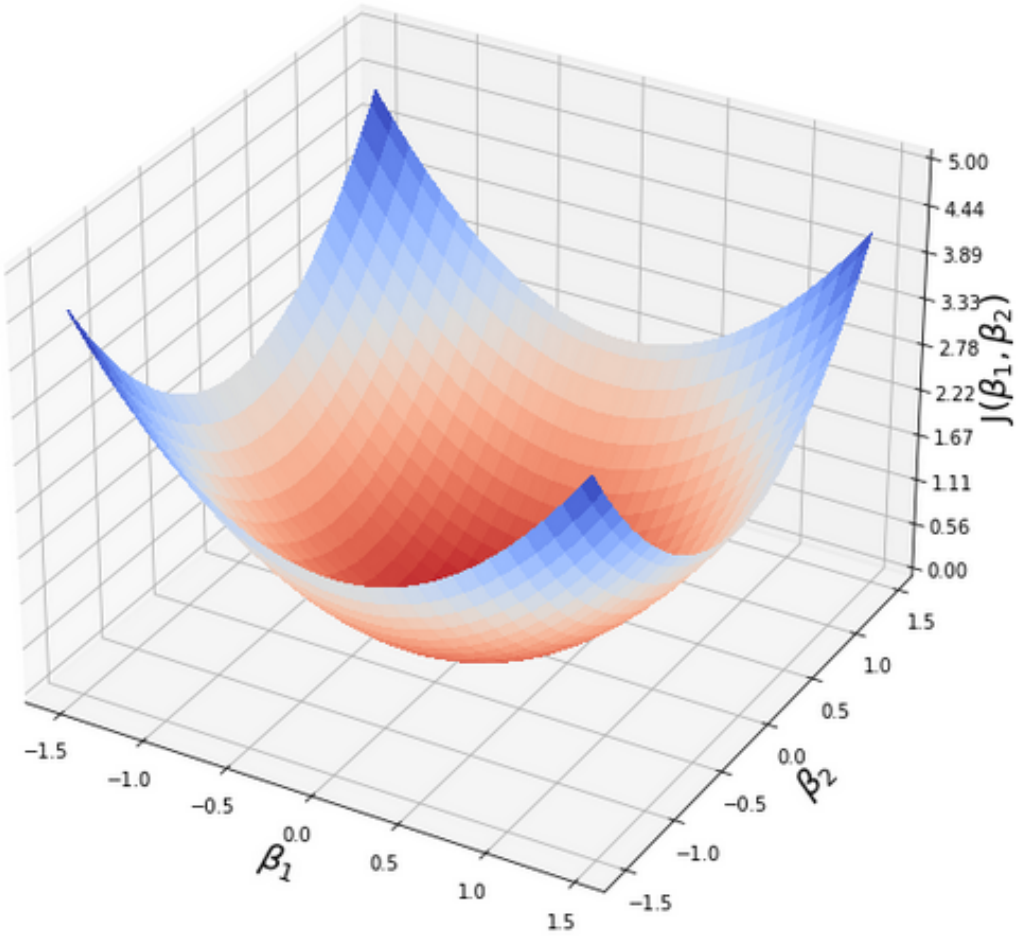
What form does the cost (J) have for the L2 loss for two points?



- J is smooth
- it is a convex function $\rightarrow$ has a unique global minima.

What form does the cost (J) have for the L2 loss in 2d?

J is now a parabolic surface with the same dimension as the number of parameters:



Gradient descent

Solver: gradient descent (Augustin Louis Cauchy, 1847)
The gradient (G) of a surface is given by

$$G = \frac{\partial J(\beta)}{\partial \beta}$$

where β are the parameters
Gradient descent *'update rule'*

$$\beta \leftarrow \beta - \eta \frac{\partial J(\beta)}{\partial \beta}$$

where η is the *"learning rate"* parameter

We wish to locate the local minima, or better still the global minima

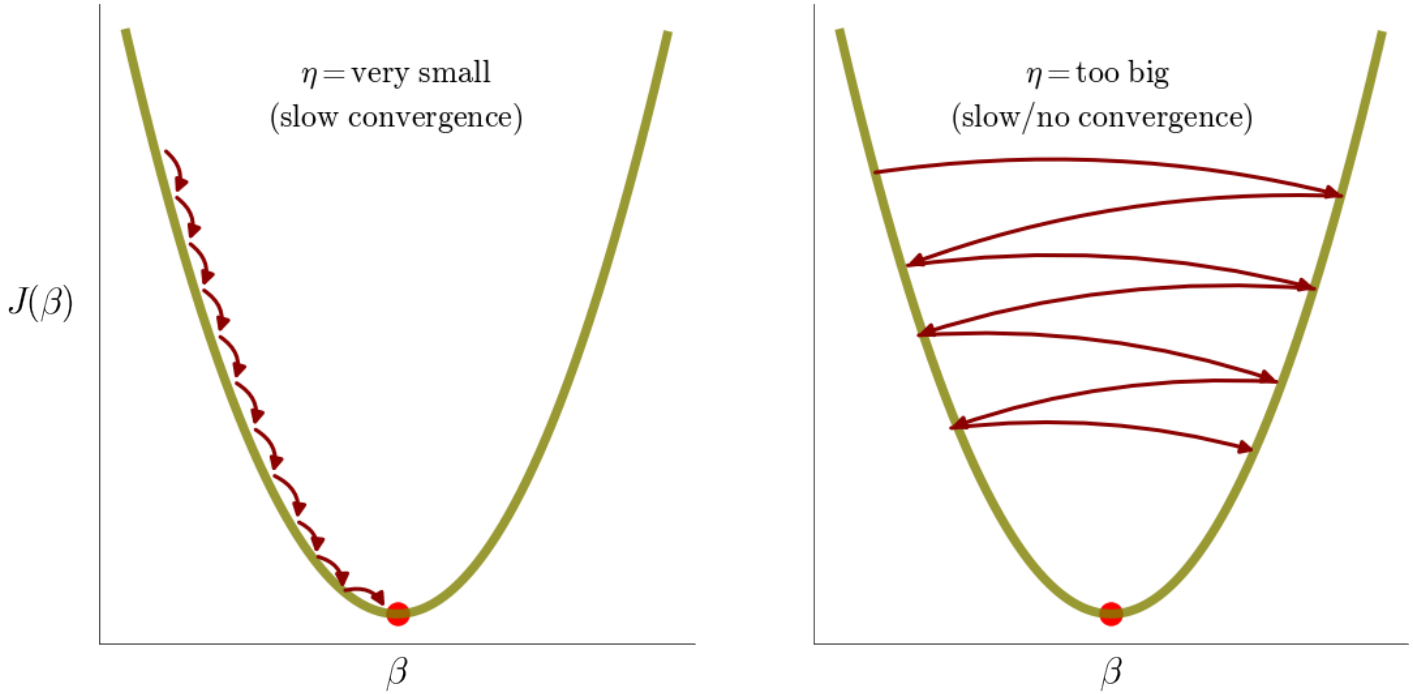
(Paper: *"An overview of gradient descent optimization algorithms"*)

algorithm:

first choose a random starting point for β

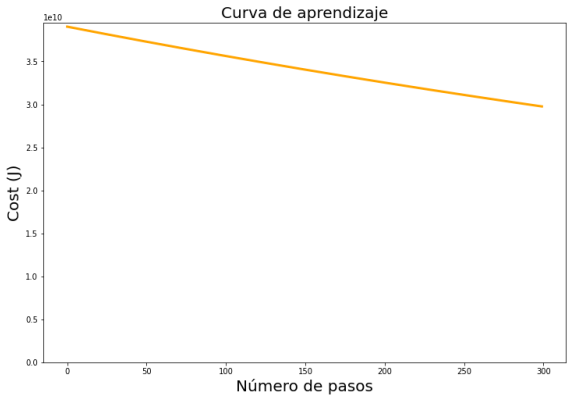
1. calculate the gradient $(\partial J/\partial \beta)$
2. multiply by -1 to change the sign, hence direction
3. scale by η
4. this is the new value of β
5. repeat, and when β hardly changes, stop

Effect of the learning rate:

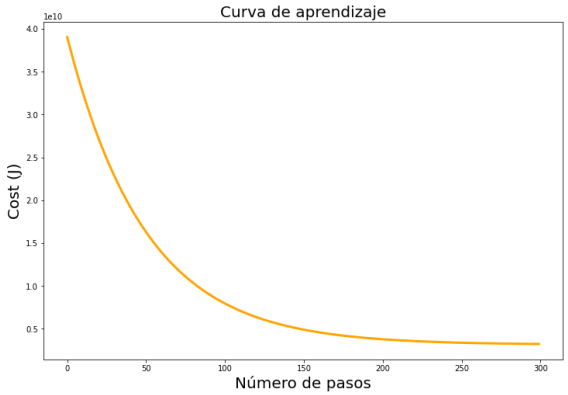


One can find a good η by plotting the cost step by step.
Early stopping: for example if $\Delta J < 0{,}001$
If the cost rises then there is a big problem, and one must use a smaller η

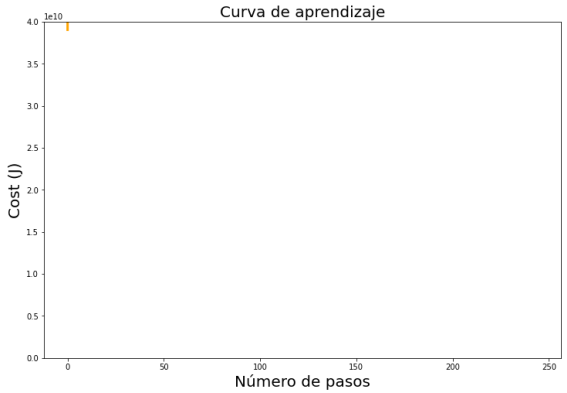
η too small



good η



η too big



- Adagrad - Adaptive Gradient Algorithm: changes η on the fly

Gradient descent for linear regression:

$$\beta \leftarrow \beta - \eta \frac{\partial J(\beta)}{\partial \beta}$$

where

$$\begin{aligned} J(\beta) = \text{MSE} = \frac{1}{n}\left(\varepsilon^T \varepsilon\right) &= \frac{1}{n}\left(\hat{\mathbf{y}} - \mathbf{y}\right)^T \left(\hat{\mathbf{y}} - \mathbf{y}\right) \\ &= \frac{1}{n}\left(\beta\mathbf{X} - \mathbf{y}\right)^T \left(\beta\mathbf{X} - \mathbf{y}\right) \end{aligned}$$

leading to

$$\frac{\partial J(\beta)}{\partial \beta} = \frac{1}{n}\left(\left(\beta\mathbf{X} - \mathbf{y}\right)\mathbf{X}\right)$$

giving

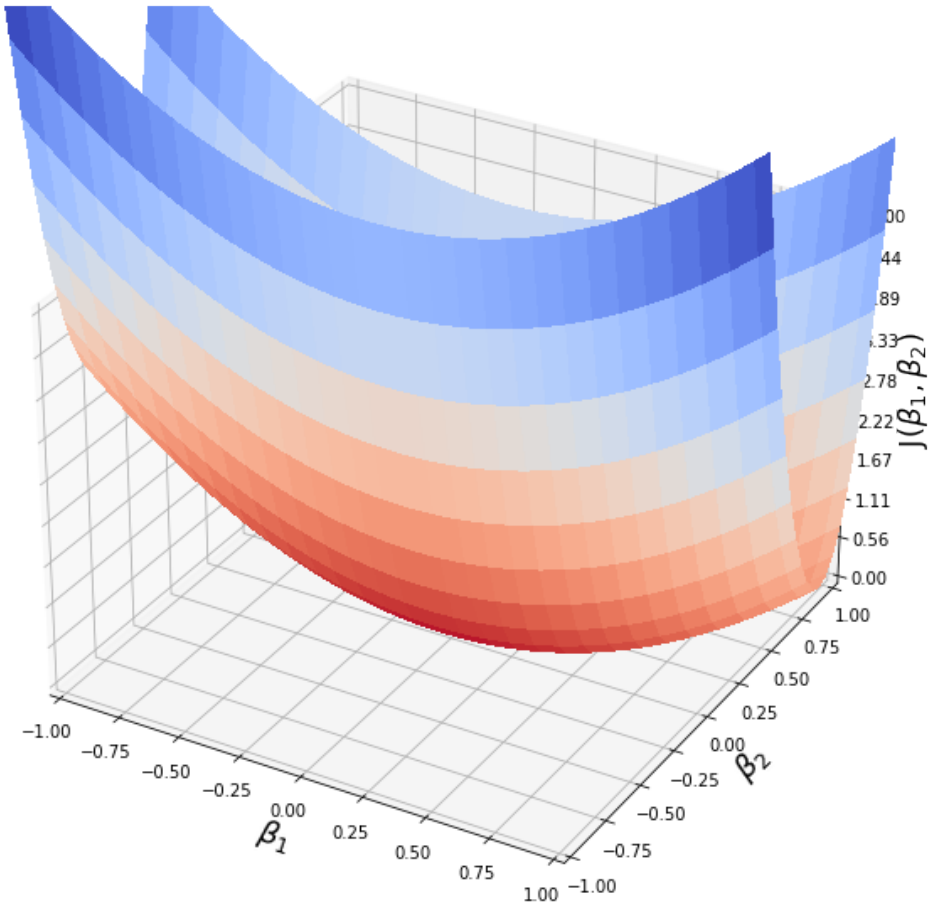
$$\beta \leftarrow \beta - \frac{\eta}{n}\left(\varepsilon\mathbf{X}\right)$$

Practical session

Let us look at gradient descent in our notebook

`notebook_ML_regression_GD.ipynb`

What happens if the features have very different scales?



Feature scaling or normalization

With parametric estimators the solvers have better numerical stability if the data is *"normalized"*

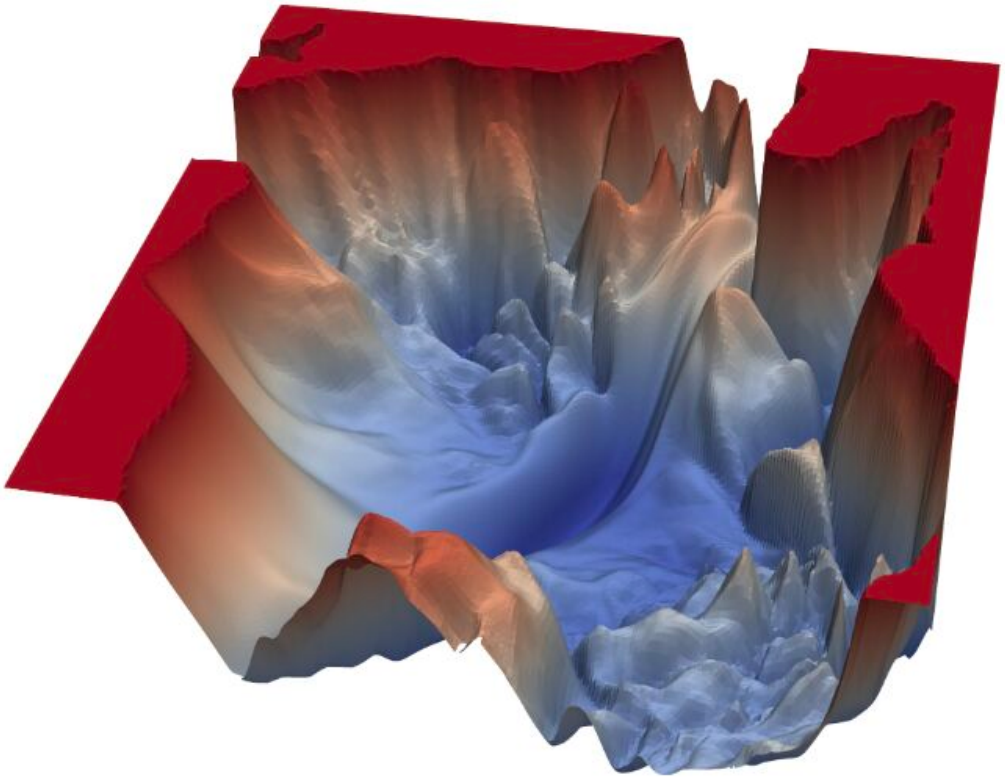
Here are some linear transformations:

- `sklearn.preprocessing.MinMaxScaler` $[0, 1]$
- `sklearn.preprocessing.StandardScaler` $\rightarrow \mu = 0, \sigma = 1$
- `sklearn.preprocessing.RobustScaler` in case there are outliers
- (for neural networks there is the new `SquashingScaler` available in the `skrub` library)

Usage example:

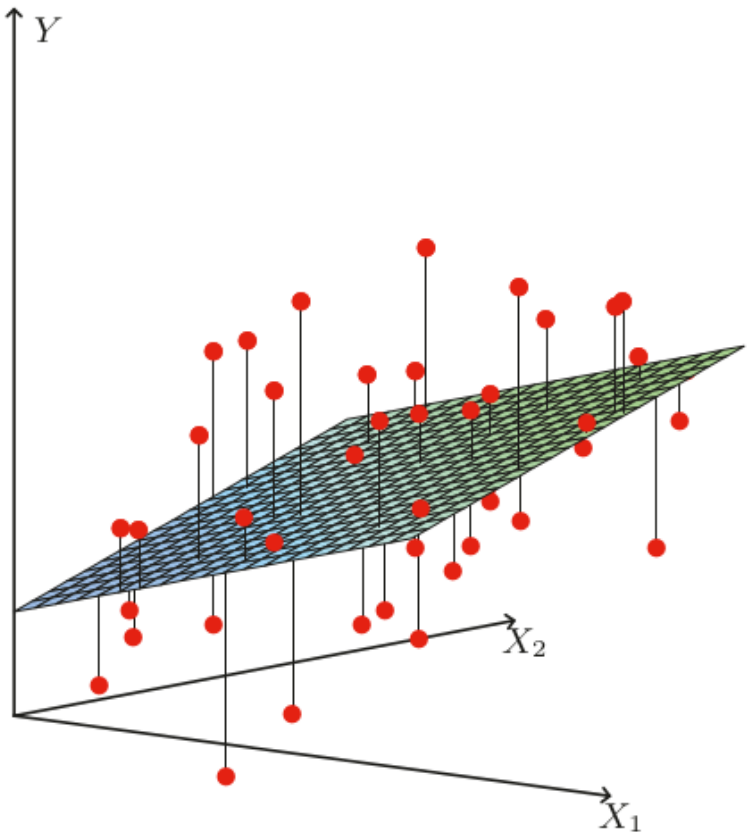
```
from sklearn.preprocessing import RobustScaler

RS = RobustScaler()

X_train = RS.fit_transform(X_train)
X_test  = RS.transform(X_test)
```

An example of a real neural network loss surface



(Source: *"Visualizing the Loss Landscape of Neural Nets"*)

Linear regression in two dimensions, *i.e.* having two features:



The line is now a plane.

Further reading

Free books on linear regression

- "The Truth about Linear Regression" by Cosma Rohilla Shalizi
- "Lecture notes on Ridge regression" by Wessel N. van Wieringen
- "Linear Model and Extensions" by Peng Ding
- "Regression and Other Stories" by Gelman, Hill, and Vehtari