REGRESSION TASKS

We will define the regression task in terms of **how the regression algorithm processes a given input example**.

TASK DEFINITION

We will define a regressor as a function:

$$f: \Re^n \rightarrow \Re^m$$

where n is the dimensionality of the input space and m is the dimensionality of the output space. In general, we will denote the input vector with $\boldsymbol{x} \in \Re^n$, the ground truth output vector with $\boldsymbol{y} \in \Re^m$ and the predicted output vector with $\hat{\boldsymbol{y}} \in \Re^m$. We can predict a value from $\boldsymbol{x}$ as follows:

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x})$$

In this context, $\boldsymbol{x}$ is often called **the independent variable** and y the **dependent variable.**

FINDING THE REGRESSION FUNCTION F

We will use the **paradigm of machine learning to learn** a suitable function f. To this aim, once we defined our task, we need to define a suitable set of data to train and test our algorithms. We will discuss performance measures later.

We will assume to have a dataset D of labeled data pairs:

$$D = \left\{ \left(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)} \right) \right\}_j$$

Each data pair $\left(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)} \right)$ is composed of **an input sample** $\boldsymbol{x}^{(j)} \in \Re^n$ and a **ground truth label** $\boldsymbol{y}^{(j)} \in \Re^m$. We will assume that the dataset has been divided into three different sets:

- A training set $TR = \left\{ \left(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)} \right) \right\}_j$.
- A validation set $VA = \left\{ \left(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)} \right) \right\}_j$.
- A test set $TE = \left\{ \left(\boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)} \right) \right\}_j$.

PERFORMANCE MEASURES

According to the machine learning paradigm, we need to define a measure to evaluate the performance of a given regression algorithm. Similar to the case of classification, we will consider the set of ground truth test labels:

$$Y_{TE} = \left\{ \boldsymbol{y}^{(i)} | \left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \right) \in TE \right\}$$

And the set of predicted test labels:

$$\hat{Y}_{TE} = \left\{ f\left(\boldsymbol{x}^{(i)} \right) | \left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \right) \in TE \right\}$$

Ideally, we would like the predicted labels to match the ground truth labels, i.e., $\hat{Y}_{TE} = Y_{TE}$. In practice, we will define a performance measure to assess how close our predictions are to the ground truth values.

MEAN SQUARED ERROR (MSE)

Consider a ground truth label $\boldsymbol{y} \in \Re^m$ and a predicted label $\hat{\boldsymbol{y}} \in \Re^m$. Since both values are m-dimensional vectors, a natural way to measure if $\hat{\boldsymbol{y}}$ is a good approximation of $\boldsymbol{y}$ is to simply measure their Euclidean distance:

$$\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2 = \sqrt{\sum_{i=1}^{m} (\hat{y}_i - y_i)^2}$$

In practice, **we often use the squared Euclidean distance to penalize more large errors**:

$$error(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2 = \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

We can compute the average error over the whole test set to obtain a performance estimator, which is usually called Mean Squared Error (MSE):

$$MSE(Y_{TE}, \hat{Y}_{TE}) = \frac{1}{|TE|} \sum_{j=1}^{|TE|} \left\|\hat{\boldsymbol{y}}^{(j)} - \boldsymbol{y}^{(j)}\right\|_2^2$$

This performance measure is actually an **error measure**. A good regressor will obtain a **small error.**

SPECIAL CASES OF REGRESSION

While regression can be in general defined as the problem of mapping vectors to vectors with a function

$$f: \Re^n \to \Re^m,$$

some special cases obtained by considering values of n and m are particularly common to encounter in practice and for this reason they have special names. We summarize them in the following:

SIMPLE REGRESSION

When $m = n = 1$ the regression task consists in mapping scalar numbers to scalar numbers with a function

$$f: \Re \to \Re$$

In this case, the task is referred to as **simple regression**.

EXAMPLE

Predict house prices from surface in square meters.

MULTIPLE REGRESSION

When $m = 1$ and $n > 1$, the regression task consists in mapping vectors to scalar numbers with a function:

$$f: \Re^n \to \Re$$

In this case, the task is referred to as **multiple regression**.

EXAMPLE

Predict house prices from surface in square meters and number of rooms. In this case, $n = 2$.

MULTIVARIATE REGRESSION

When $m > 1$ and n is arbitrary, the regression task consists in mapping vectors (or scalars if $n = 1$) to vectors with a function:

$$f: \Re^n \rightarrow \Re^m$$

In this case, the task is referred to as **multivariate regression**.

EXAMPLE

Predict the GPS position from which an image has been captured. In this case, $m = 2$.

LINEAR REGRESSION

SIMPLE LINEAR REGRESSION

We will start by considering the case of simple regression, i.e., our regression function will have the form

$$f: \Re \rightarrow \Re$$

To this aim, we will first consider an example.

EXAMPLE DATASET

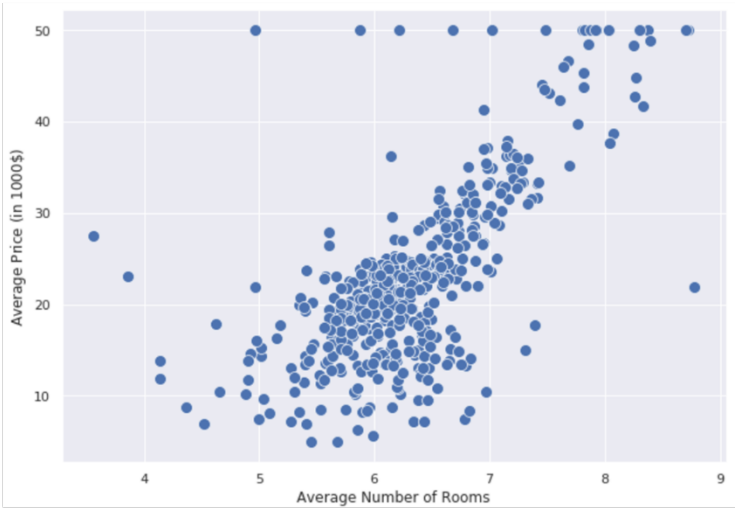
We will consider a simple dataset. The data consists in pairs of numbers (x, y) reporting:

- The **average number of rooms** of houses in the suburb x;
- The **average price** of the houses in the suburb in 1000$s y (i.e., $y = 30$ means 30000$).

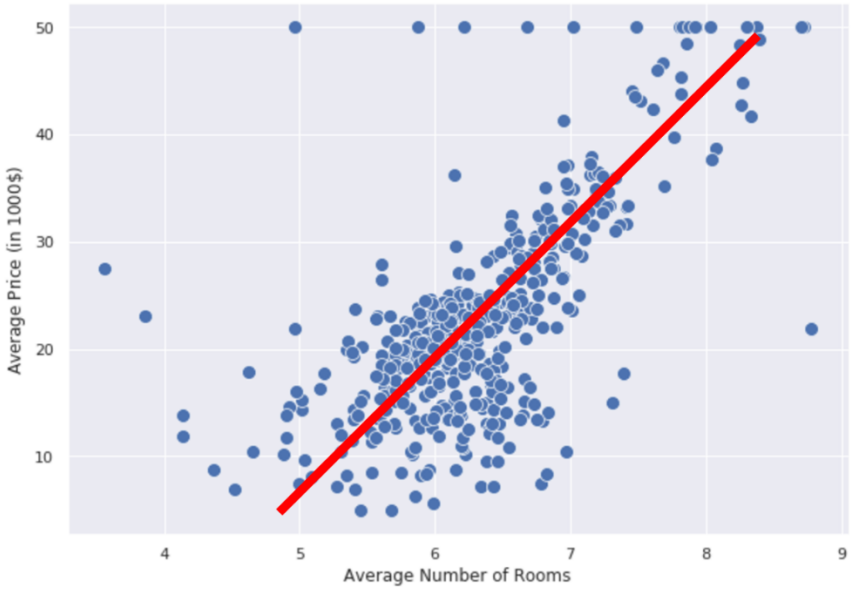
The figure below shows a plot of the (x, y) pairs. We want to find a function f which allows us to map the average number of rooms x to the average price y.

	Rooms	Price (1000$s)
0	6.575	24.0
1	6.421	21.6
2	7.185	34.7
3	6.998	33.4
4	7.147	36.2
5	6.430	28.7
6	6.012	22.9
7	6.172	27.1
8	5.631	16.5
9	6.004	18.9

...



If we observe the points in the plot, we observe that they follow a **linear trend**, meaning that they are approximately distributed along a line as shown in the following plot:



DEFINITION

In the previous example, we noted that points are approximately distributed in a linear way. We can hence think to define our regression function f using the **analytical formulation of a line**:

$$f(x) = \theta_0 + \theta_1 x$$

where θ_1 is the coefficient and θ_0 is the intercept.

We will call the above defined function f **a linear model** or a **linear regressor**. θ_0 and θ_1 are **the parameters of the model.**

Learning a linear regressor means finding values for the parameters θ_0 and θ_1 such that f(x) is a good predictor of y. We will see later an algorithm to find such values.

EXAMPLE LINEAR MODEL

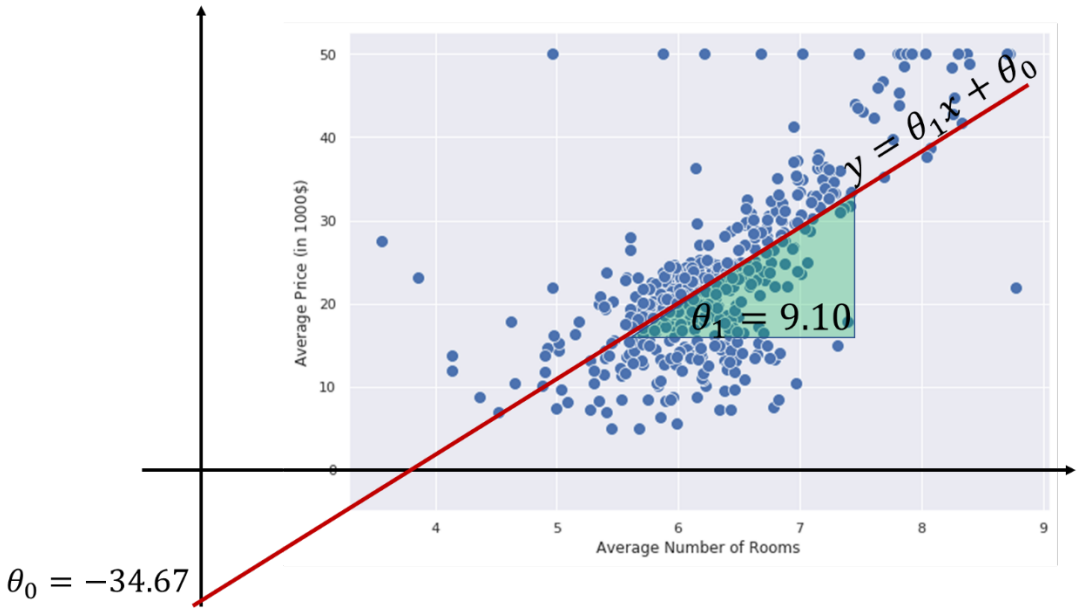
Let us consider our previous simple regression example of the predicting average price from the average number of rooms. We will see how to find these variables from data in a few slides. For the moment, just imagine that someone told us that a pair of good values to solve our regression problem are:

- $\theta_0 = -34.67$;
- $\theta_1 = 9.10$

Our linear model will be:

$$f(x) = 9.1 \cdot x - 34.67$$

If we try to plot this line on the data, we obtain the following figure:



MULTIPLE LINEAR REGRESSION

We can easily extend the simple linear regression model to the case of multiple regression, i.e., the case in which we want to find a function:

$$f: \Re^n \rightarrow \Re$$

In this case, we will use a parameter for each of the dimensions of the input variable, plus a parameter for the intercept:

$$f(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

The parameters of the model are $\boldsymbol{\Theta} = (\theta_0, \theta_1, \ldots, \theta_n)$. **Learning the regressor means finding suitable values of the parameters $\boldsymbol{\Theta}$.**

To simplify the notation, we can set $x_0 = 1$ and write the linear model as follows:

$$f(\boldsymbol{x}) = \sum_{i=0}^{n} \theta_i x_i$$

The expression above can also be written as:

$$f(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{x}$$

Note that this model is essentially a perceptron without any thresholding operation!

MULTIVARIATE LINEAR REGRESSION

Let us now consider the general case of multivariate regression in which we want map vectors to vectors:

$$f: \Re^n \rightarrow \Re^m$$

Multivariate linear regression solves the problem by defining m independent multiple regressors $f_i(\boldsymbol{x})$ which process the same input $\boldsymbol{x}$, but are allowed to have different weights:

$$\begin{pmatrix} y_1 \\ \ldots \\ y_m \end{pmatrix} = \begin{pmatrix} f_1(\boldsymbol{x}) \\ \ldots \\ f_m(\boldsymbol{x}) \end{pmatrix}$$

Each regressor f_i has its own parameters and their optimizations are carried out independently.

LEARNING

We will focus on the general form of multiple linear regression:

$$f_\Theta(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

Where f_Θ indicates that f depends on the parameters $\Theta = (\theta_0, \ldots, \theta_n)$. Learning the regressor f means finding values for the parameters Θ which minimizes the error on the **training set**. We can quantify the overall error of the regressor on the test set defining the following **cost function**:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{\mathbf{m}} \left(f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2$$

Note that the above expression is similar to **the MSE error that we obtain on the test set for a given choice of the parameters $\boldsymbol{\Theta}$**. As such, this is a function of the parameters Θ. Ideally, we would like to have a cost equal to zero, which would mean that we have an MSE equal to zero on the training set.

Hence, a good choice for Θ would be the one which **minimizes the cost function $\boldsymbol{J}(\boldsymbol{\Theta})$**, i.e.,

$$\boldsymbol{\theta}^* = \arg_\theta \min J(\boldsymbol{\theta})$$

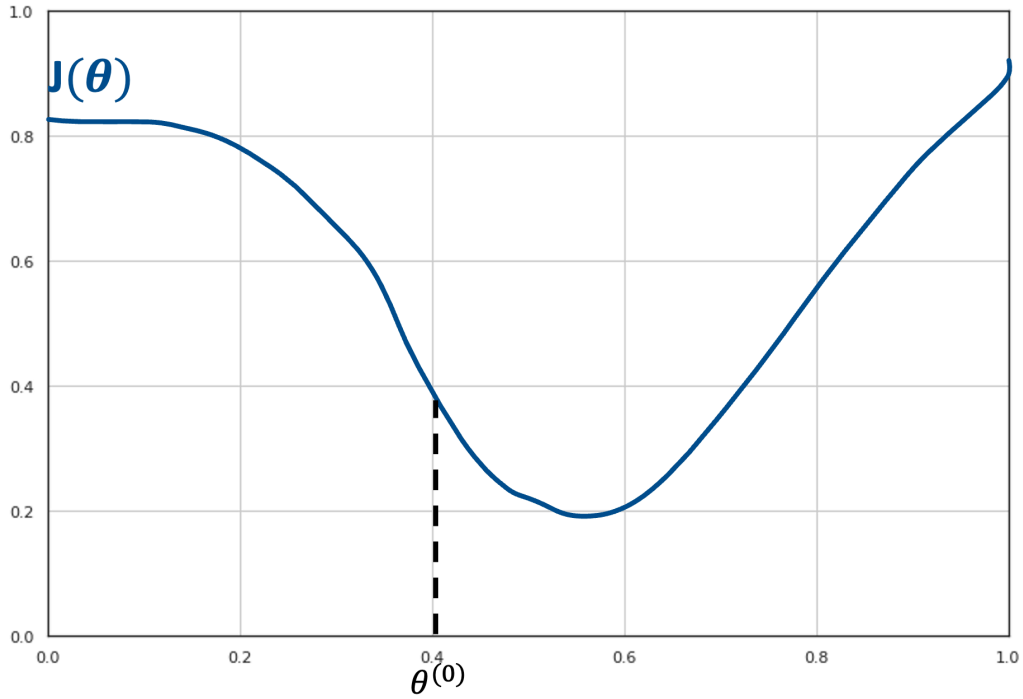
THE GRADIENT DESCENT ALGORITHM

We have seen that we can find suitable values $\boldsymbol{\theta}$ by solving the optimization problem $\boldsymbol{\theta} = \arg_{\boldsymbol{\theta}} \min J(\boldsymbol{\theta})$. However, it is still unclear how to perform such choice. If we know the partial derivatives of the function with respect to all variables, we can define an equation system by setting all such derivatives to zero, however this is not always feasible when, in general, we may have several parameters and we may not know the analytical form of all derivatives. Alternatively, we could **compute $\boldsymbol{J}(\boldsymbol{\Theta})$ for all possible values of $\boldsymbol{\Theta}$ and choose the values of $\boldsymbol{\Theta}$ which minimizes the cost**. However, this option is unfeasible in practice as Θ may assume an infinite number of values. Hence, we need a way to **find the values of $\boldsymbol{\Theta}$ which minimize $\boldsymbol{J}(\boldsymbol{\Theta})$ without computing $\boldsymbol{J}(\boldsymbol{\Theta})$ for all possible values of $\boldsymbol{\Theta}$.**

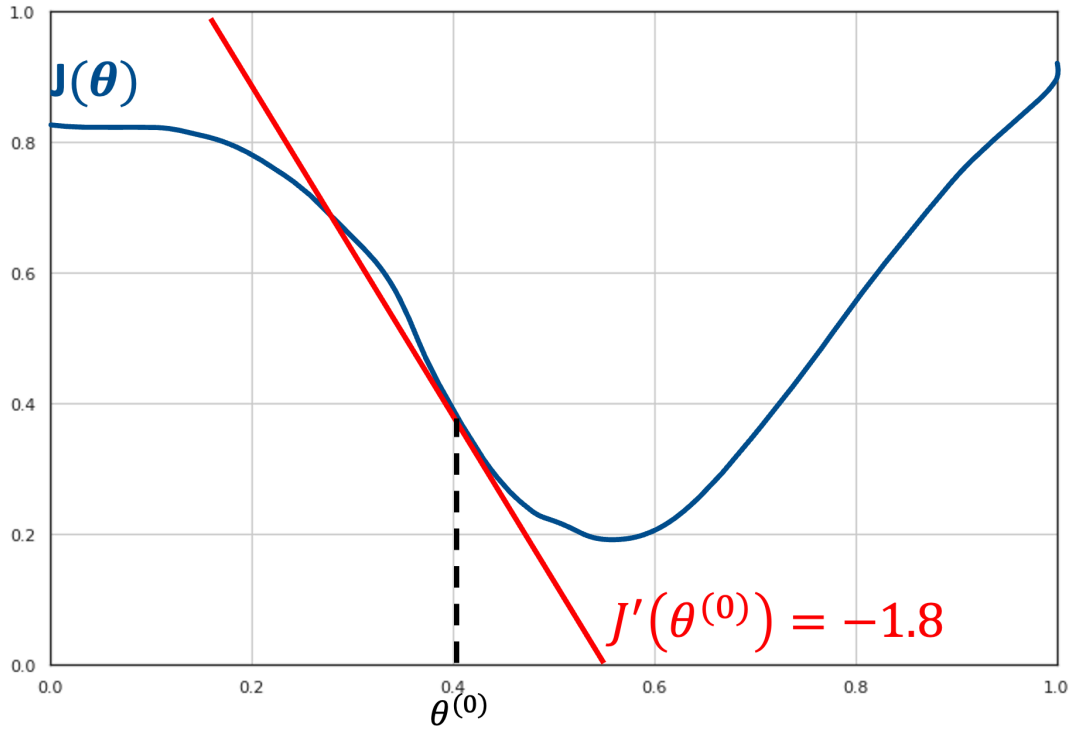
To solve this problem, we can use different optimization strategies. We will see a numerical optimization strategy called gradient descent, which allows to **minimize differentiable functions** with respect to their parameters.

We will introduce the gradient descent algorithm considering initially the problem of minimizing a function of a single variable $J(\theta)$. We will then extend to the case of multiple variables.

The gradient descent algorithm is based on the observation that, if a function $J(\theta)$ is defined and differentiable in a neighborhood of a point $\theta^{(0)}$, then $J(\theta)$ decreases fastest if one goes from $\theta^{(0)}$ towards the direction of the negative derivative of J computed in $\theta^{(0)}$. Consider the function $J(\theta)$ shown in the plot below:



Let us assume that we are at the initial point $\theta^{(0)}$. From the plot, we can see that we should move to the right part of the x axis in order to reach the minimum of the function.



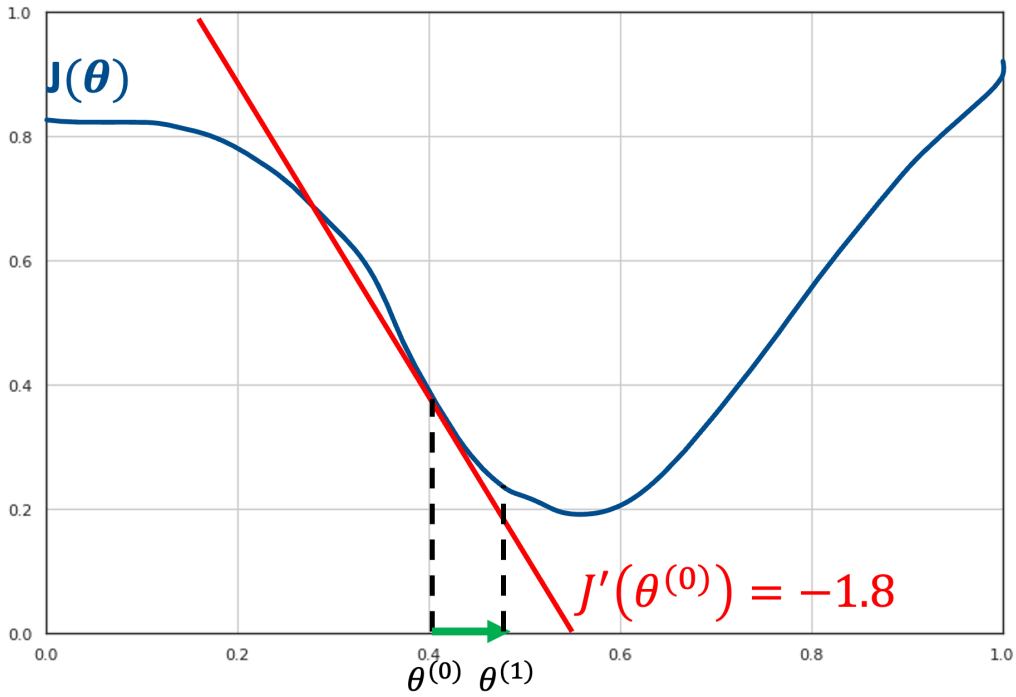
The first derivative of the function in that point $J'(\theta^{(0)})$ will be equal to the angular coefficient of the tangent to the curve in the point $(\theta^{(0)}, J(\theta^{(0)}))$. Since the curve is decreasing in a neighborhood of $\theta^{(0)}$, the tangent line will also be decreasing. Therefore, its angular coefficient $J'(\theta^{(0)})$ will be negative. If we want to move to the right, we should follow in the *inverse direction* of the derivative of the curve in that point.

The gradient descent is an iterative algorithm; hence we are not trying to reach the minimum of the function in one step. Instead, we would like to move to another point $\theta^{(1)}$ such that $J\left(\theta^{(1)}\right) < J(\theta^{(0)})$. If we can do this for every point, we can reach the minimum in a number of steps.

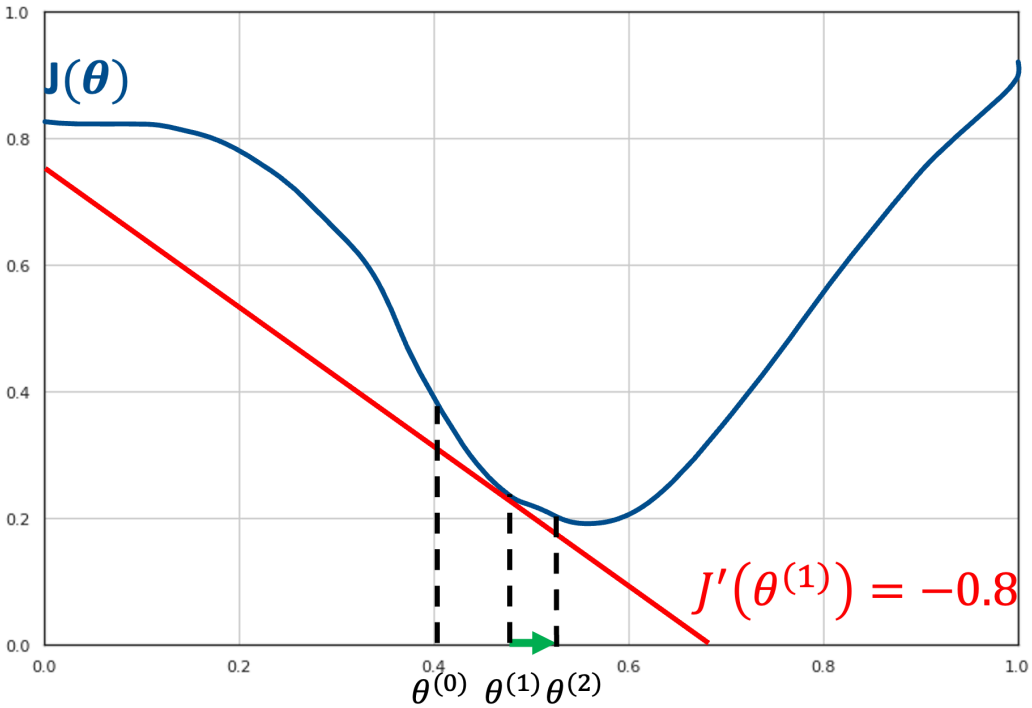
At each step, we will move proportionally to the value of the derivative. This is based on the observation that larger absolute values of the derivative indicate steeper curves. If we choose a multiplier factor γ, we will move to the point:

$$\theta^{(1)} = \theta^{(0)} - \gamma J'(\theta^{(0)})$$

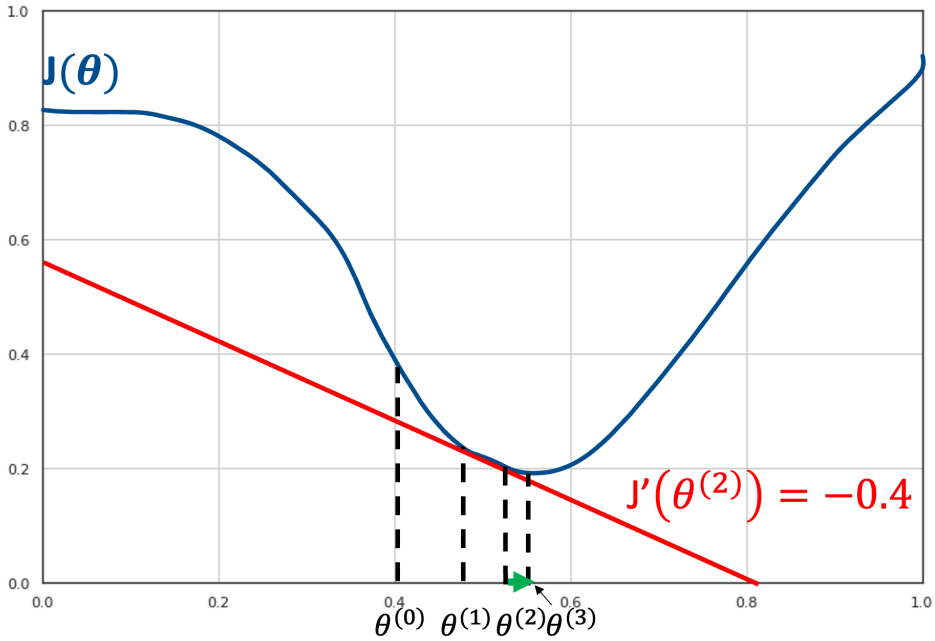
For instance, if we choose $\gamma = 0.02$, we will move to point $\theta^{(1)} = 0.4 + 0.02 \cdot 1.8 = 0.436$. The procedure works iteratively until the derivative is so small that no movement is possible, as shown in the following figure.



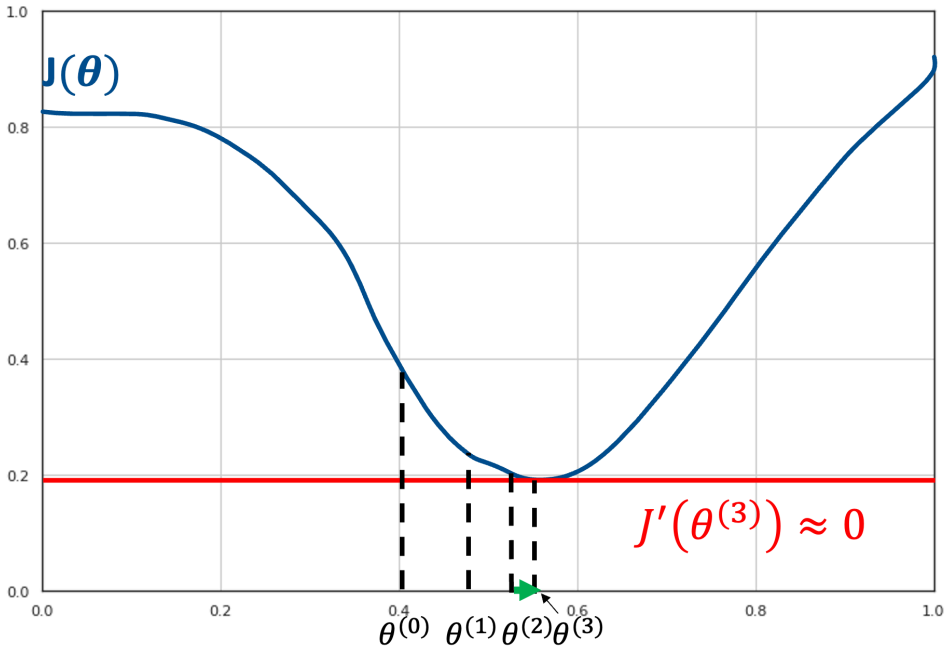
In the next step, we compute the derivative of the function in the current point $J\left(\theta^{(1)}\right) = -0.8$ and move to point $\theta^{(2)} = \theta_1 - \gamma J'(\theta^{(1)})$.



Next, we compute the derivative of the function in the current point $f(\theta^{(2)}) = -0.4$ and move to point $\theta^{(3)} = \theta^{(2)} - \gamma J'(\theta^{(2)})$:



We then compute the derivative of the current point $J(\theta^{(3)}) \approx 0$:



This derivative is so small that we cannot advance further. We are in a local minimum. The optimization terminates here. We have found the value $\theta^{(3)} = \arg_\theta \min J(\theta)$.

In practice, the algorithm is terminated following a **given termination criterion**. Two common criteria are:

- A maximum number of iterations is reached.
- The value $\gamma J'(\theta)$ is below a given threshold.

ONE VARIABLE

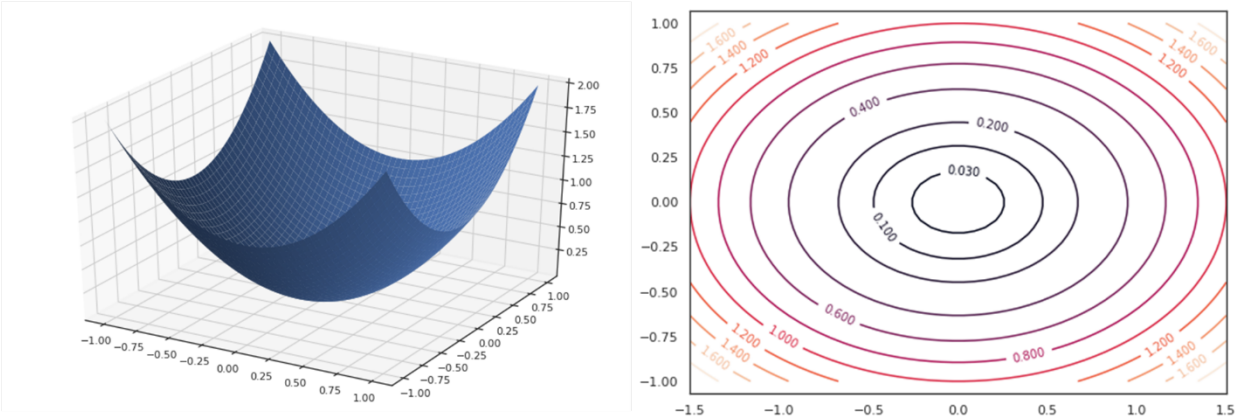
The gradient descent algorithm can be written in the following form in the case of one variable:

1. Choose an initial random point θ;
2. Compute the first derivative of the function J' in the current point θ: $J'(\theta)$;
3. Update the position of the current point using the formula $\theta = \theta - \gamma J'(\theta)$;
4. Repeat 2-3 until some termination criteria are met.

MULTIPLE VARIABLES

The gradient descent algorithm generalizes to the case in which the function J to optimize depends on multiple variables $J(\theta_1, \theta_2, \dots, \theta_n)$.

For instance, let's consider a function of two variables $J(\theta_1, \theta_2)$. We can plot such function as a 3D plot (left) or as a contour plot (right). In both cases, our goal is to reach the point with the minimum value (the 'center' of the two plots). Given a point $\boldsymbol{\theta} = (\theta_1, \theta_2)$, the direction of steepest descent is the **gradient** of the function in the point.

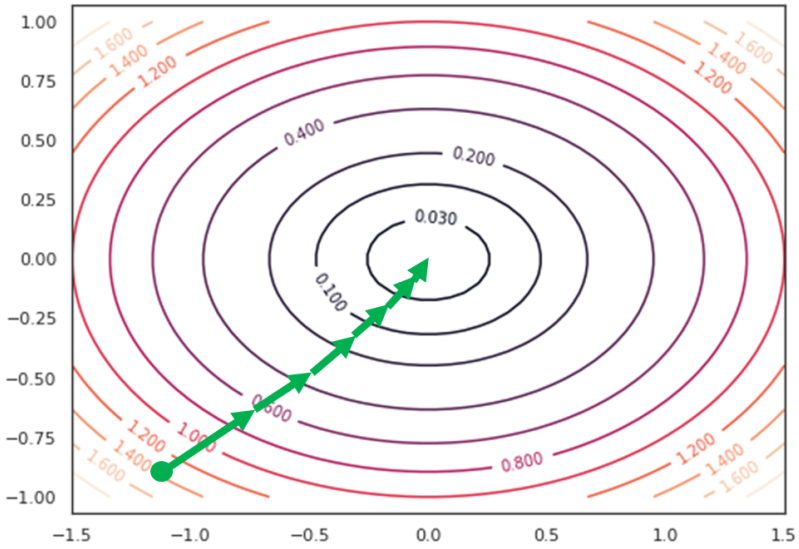


The gradient is a multi-variable generalization of the derivative. The gradient of a function of n variable computed in a point $\boldsymbol{\theta}$ is a vector whose i^{th} variable is given by the partial derivative of the function with respect to the i^{th} variable:

$$\nabla J(\boldsymbol{\theta}) = \begin{pmatrix} J_{\theta_1}(\boldsymbol{\theta}) \\ J_{\theta_2}(\boldsymbol{\theta}) \\ \dots \\ J_{\theta_n}(\boldsymbol{\theta}) \end{pmatrix}$$

In the case of two variables, the gradient will be a 2D vector (the gradient) indicating the direction to follow. Since in general we want to optimize multi-variable functions, the algorithm is called 'gradient descent'.

The following figure shows an example of an optimization procedure to reach the center of the curve from a given starting point:



The pseudocode of the procedure, in the case of the multiple variables is as follows:

1. Initialize $\boldsymbol{\theta} = (\theta_1, \theta_2, \dots, \theta_n)$ randomly.
2. For each variable x_i:
    - Compute the partial derivative at the point:
      $\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta})$
    - Update the current variable using the formula:
      $\theta_i = \theta_i - \gamma \frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta})$
3. Repeat 2 until the termination criteria are met.

GRADIENT DESCENT AND LINEAR REGRESSION

We will use the gradient descent algorithm to optimize our linear regression problem, i.e., to find $\widehat{\boldsymbol{\theta}} = \arg_{\boldsymbol{\theta}} \min J(\boldsymbol{\theta})$. This is done by first initializing $\boldsymbol{\theta}$ randomly (this will provide the starting point for the gradient descent) and then applying the following update rule repeatedly for each variable:

$$\theta_j = \theta_j - \gamma \cdot \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta})$$

To implement this, we need to compute the partial derivatives of the cost function with respect to each parameter. Let us first write the cost function explicitly in terms of the parameters θ:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{N}\left(f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2 = \frac{1}{2}\sum_{i=1}^{N}\left(\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)} - y^{(i)}\right)^2$$

Where we have introduced the $x_o^{(i)} = 1$ terms as previously discussed. We can compute the partial derivative of the cost function with respect to the j^{th} variable as follows:

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{N} 2\left(f_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)\frac{\partial}{\partial \theta_j}\left(\theta_0 x_0^{(i)} + \cdots + \theta_n x_n^{(i)} - y^{(i)}\right)$$

We note that:

$$\frac{\partial}{\partial \theta_j}\left(\theta_0 x_0^{(i)} + \cdots + \theta_n x_n^{(i)} - y^{(i)}\right) = x_j^{(i)}$$

So, we obtain:

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \sum_{i=1}^{N}\left(f_{\theta}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

The update rule hence can be written as follows:

$$\theta_j = \theta_j - \gamma \sum_{i=1}^{N}\left(f_{\theta}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Note that the update rule above is essentially the ADELINE algorithm!

FEATURE SCALING

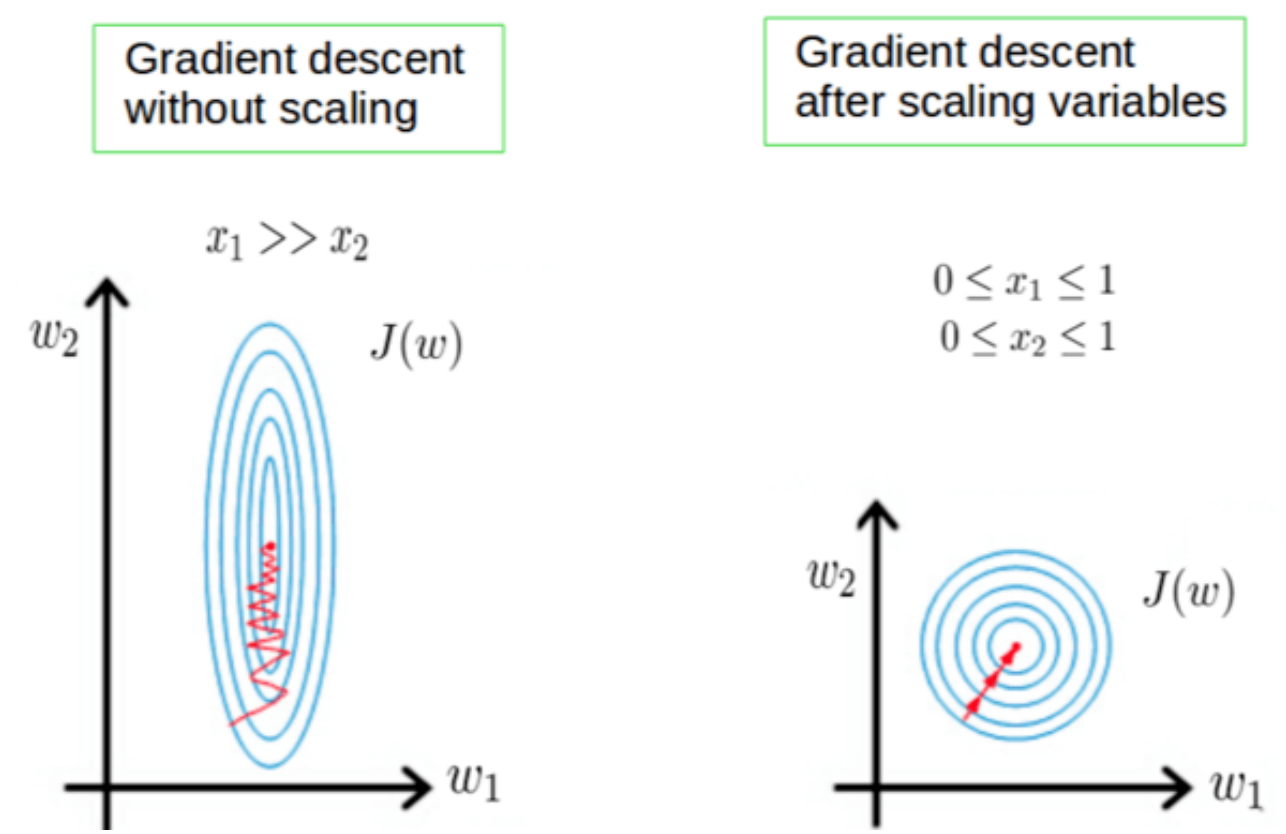
Note that, for the gradient descent algorithm to work well, it is necessary to scale the features so that they lie in similar ranges. Feature scaling can be performed in a variety of ways, the most common way to do this is by **z-scoring**, which consists in subtracting each feature the mean of that feature in the training set and dividing it by the standard deviation of that feature in the training set:

$$x_j = \frac{x_j - \mu_j}{\sigma_j}, \quad where \quad \mu_j = \frac{1}{N}\sum_{i=1}^{N} x_j \quad and \quad \sigma_j = \frac{1}{N}\sum_{i=1}^{N}\left(x_j - \mu_j\right)^2$$

Another possible approach is the min-max scaling:

$$x_j = \frac{x_j - x_j^{min}}{x_j^{max} - x_j^{min}}, \quad where \quad x_j^{min} = \min\{x_j\}_j \quad and \quad x_j^{max} = \max\{x_j\}_j$$

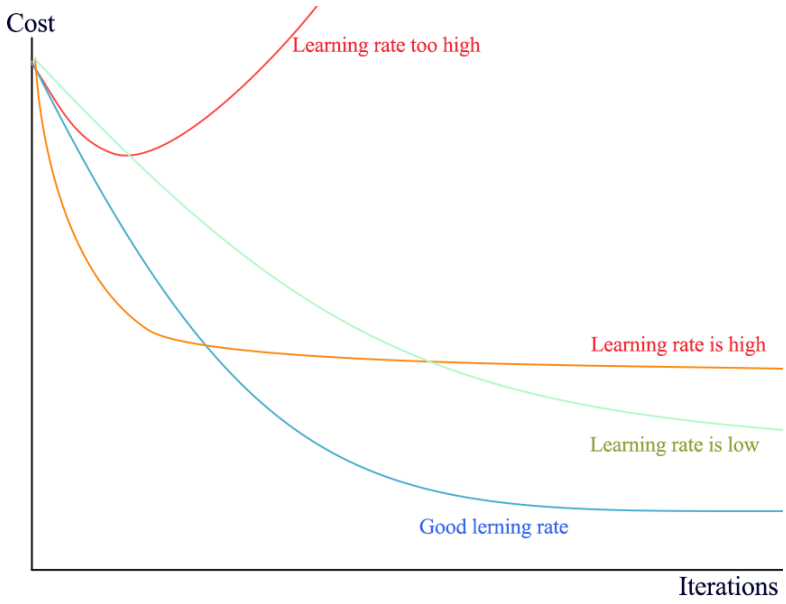
These feature scaling (or normalization) techniques generally accelerate learning and avoid divergence.



https://medium.com/analytics-vidhya/why-do-feature-scaling-overview-of-standardization-and-normalization-machine-learning-3e99d16eeca8

DEBUGGING GRADIENT DESCENT

In practice, choosing a good learning rate (or other set of parameters) can be tricky. Hence, it is usually good to look at how the loss function computed on the training set varies with time. This can be done by plotting (often interactively, during training) the values of the loss as a function of the number of iterations. By looking at the curve, different behaviors can be observed:



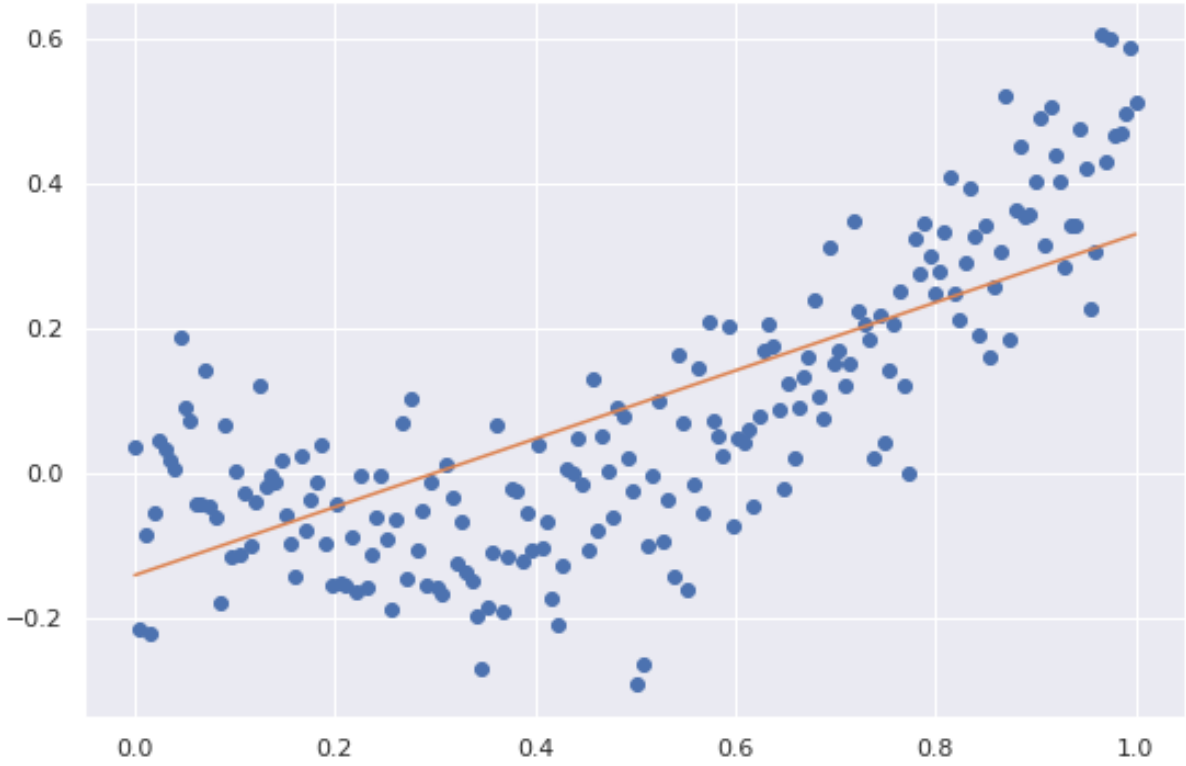
https://jonathan-hui.medium.com/debug-a-deep-learning-network-part-5-1123c20f960d

When one of those situations are encountered, the training can be repeated changing the learning rate.

POLYNOMIAL REGRESSION

LIMITS OF LINEAR REGRESSION

Linear regression is limited in those cases in which the relationship between the dependent variables and the independent variable is clearly nonlinear. Consider for instance the set of points in the following plot.



These points clearly follow a nonlinear relationship. Hence, any attempts to fit them using a linear model will have limited performance. This limited performance is due to the fact that our model has a **reduced capacity**: i.e., we can model only linear functions, while the relationship between input and output is not linear. In these cases, we usually say that the model is **underfitting**.

POLYNOMIAL MODEL

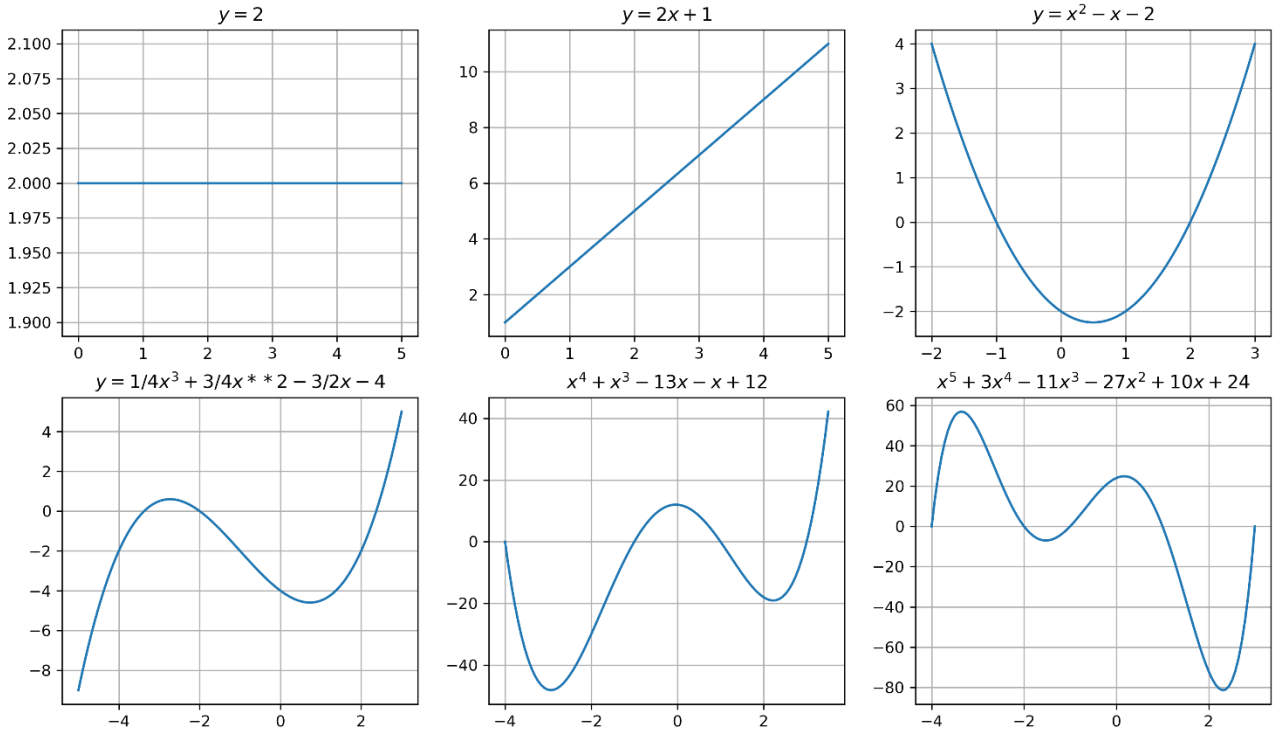
Rather than using a linear function

$$f(x) = \theta_0 + \theta_1 x$$

to fit the data, we could introduce $d - 1$ additional parameters and use a polynomial model of degree d such as the following one:

$$f(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_d x^d$$

Polynomial models of higher degrees allow to represent functions which are nonlinear. Consider for instance the examples reported in the following figure.



As can be seen, higher order polynomials allow to obtain more complex curves than lower order polynomials. A polynomial of degree 1 is a linear function.

We can easily observe that, while the function is **not linear with respect to the variable** x, it is **linear with respect to the exponentiated variables:** $x, \dots, x^d$. Indeed, if we are given the values of the x^i variables, finding optimal values for the θ_i parameters is still a linear regression problem, which can be solved with the gradient descent algorithm.

We hence observe that we can turn a linear regression problem into a polynomial regression problem by **replicating the features and taking powers up to the d-th**. For example:

Input Scalar

$[x]$

(degree 2 polynomial)

Input Vector

$[x, x^2]$

$$f(x) = \theta_0 + \theta_1 x$$

$$f(x) = \theta_0 + \theta_1 x + \theta_2 x^2$$

EXAMPLE

If we solve the problem seen before with a polynomial regressor of degree 2

$$f(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$$

we obtain the results shown in the following plot:

This is a much better fit as compared to the previous linear fit.

MULTIPLE FEATURES AND GENERAL IMPLEMENTATION

If the regression is multiple, we repeat the same process for each of the features **and include interaction terms**. For instance, the multiple regression problem:

$$f(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

turns into:

$$f(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1 x_2 + \theta_5 x_2^2$$

Due to the presence of the interaction terms, the computation can become very heavy when we have many input features.

From a coding perspective, the easiest way to implement Polynomial regression is to simply replicate features. For instance, if we choose a polynomial of degree 2, the features $[x_1, x_2]$ are transformed into $[x_1, x_2, x_1^2, x_1 x_2, x_2^2]$. Thus, a design matrix of size $m \times 2$ becomes a matrix of size $m \times 5$. This can require a lot of memory and computational power if we are in the presence of many input features. The process is exemplified in the following figure:

$$[x_1, x_2] \xrightarrow{\text{(degree 2)}} [x_1, x_2, x_1^2, x_1 x_2, x_2^2]$$

$$\begin{pmatrix} 0.1 & \cdots & 2.2 \\ \vdots & \ddots & \vdots \\ 3.7 & \cdots & 1.2 \end{pmatrix} \xrightarrow{\text{(degree 2)}} \begin{pmatrix} 0.7 & \cdots & 0.8 \\ \vdots & \ddots & \vdots \\ 9.2 & \cdots & 0.1 \end{pmatrix}$$

$$[m \times 2] \qquad\qquad [m \times 5]$$

REFERENCES

Section 3.1 of [1];

[1] Bishop, Christopher M. *Pattern recognition and machine learning*. springer, 2006. https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf