Machine Learning from scratch

Santiago Toro

May 21, 2025

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

Part I
Preliminaries

1 Introduction

What is ”Learning”? Types of learnings, etc.

2 Multivariate Calculus

3 Probability Theory

3.1 Probabilities

3.2 Probability Distributions

Part II
Supervised Learning

4 Regression

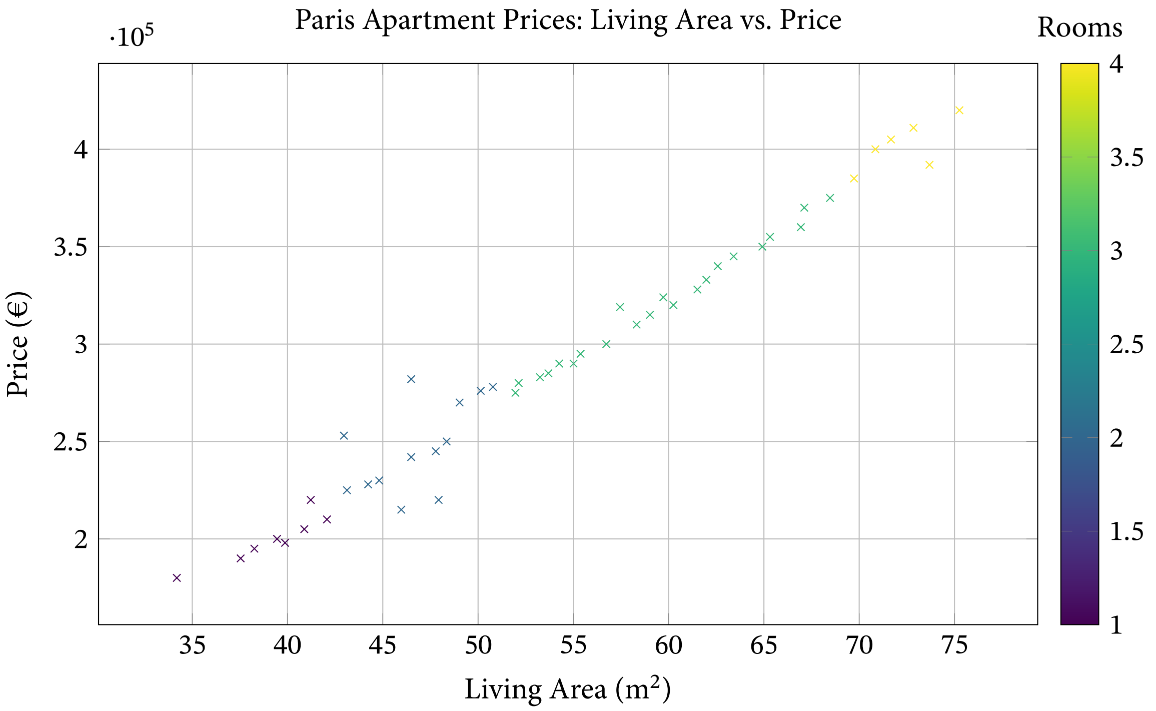
4.1 Linear Regression

In statistical modeling, regression analysis consists of a set of statistical processes and techniques for estimating the relationships between a dependent variable (often called the outcome or response variable, or a label in machine learning) and one or more error-free independent variables (often called regressors, predictors, covariates, explanatory variables or features).

The goal of regression is then to predict the value of one or more continuous response variables y given the values of a m-dimensional vector $\vec{x}$ of *input* (features) variables. Usually we are given a training data set which contains a fixed number N of observations $\{\vec{x_n}\}_{n=1,2,...,N}$ together with their corresponding outcome values $\{y_n\}$. The goal is then to predict the value of y for a new given value of $\vec{x} \in \mathbb{R}^m$. For instance, suppose we have a dataset giving the living areas, number of rooms, the district number location and prices of 50 departments in Paris, France:

Living Area (m^2)	Number of Rooms	District	Price (€)
57.45	3	9	319000
47.93	2	15	220000
59.72	3	13	324000
72.85	4	1	411000
46.49	2	7	282000
46.49	2	9	242000
73.69	4	1	392000
61.51	3	12	328000
42.96	2	8	253000
58.14	3	11	327000
43.05	2	19	225000
43.01	2	17	245000
53.63	3	8	300000
21.30	1	3	136000
24.13	1	3	117000
⋮	⋮	⋮	⋮

We can plot this data to observe the Price according to the Living Area:



Given data like this, we would like to learn to predict the prices of other apartments in Paris, as a function of the input features (Living Area, Rooms, District). Let us first introduce some notation: We set $\vec{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, x_3^{(i)})$ to be the vector in $\mathbb{R}^3$ whose entries correspond to the features (Living Area, Rooms, District) of the i-th apartment in the training set (see Table). For instance, according to the Table

$$\vec{x}^{(1)} = (57.45, 3, 9)$$

Let $y^{(i)}$ be the output or *target* variable that we are trying to predict; in our case it represents the Price. For instance $y^{(1)} = 319000$. The dataset that we will use to learn consist then of the list of $N = 50$ training examples:

$$\left\{ \left(\vec{x}^{(i)}, y^{(i)}\right) \ : \ i = 1, 2, ..., N \right\}$$

This last list will be then called *training set*.

Remark 4.1. In general when constructing a learning problem we have freedom over which features to use so one may decide to collect more data and include other features. We will say more about feature selection later.

To achieve the goal of predicting the value of y for new given vectors of $\vec{x} \in \mathbb{R}^m$, we will formulate a function $y(\vec{x}, \vec{w})$ whose values are the predictions for new inputs $\vec{x}$, and where $\vec{w}$ represents a vector of parameters that will be learned from the training set.

The simplest model for the function $y(\vec{x}, \vec{w})$ is the one that consist of a linear combination of the inputs:

(4.2) {{Eq.Sec.Lin.Reg.1}}
$$y(\vec{x}, \vec{w}) = w_0 + w_1 x_1 + \cdots + w_m x_m$$

Remark 4.3. The parameter w_0 is usually called a *bias parameter*. The key property of this model is that it is a linear function of the parameters $w_0, ..., w_m$ and also of the input variables $x_1, ..., x_m$.

4.1.1 Non linear models

The class of linear models defined by Equation (4.2) can be easily extended by considering linear combinations of non-linear functions of the input features $\vec{x}$. More precisely, suppose $\phi_0, ..., \phi_m : \mathbb{R}^m \longrightarrow \mathbb{R}$ are a collection of *basis* functions. Then, we can consider models of the form:

$$y(\vec{x}, \vec{w}) = w_0 + \sum_{i=1}^{m} w_i \phi_i(\vec{x})$$

For convenience we will set ϕ_0 to be the constant function 1 so that the last equation becomes:

(4.4) {{Eq.Sec.Lin.Reg.2}}
$$y(\vec{x}, \vec{w}) = \sum_{i=0}^{m} w_i \phi_i(\vec{x}) = \vec{w}^T \phi(x)$$

where $\vec{w} = (w_0, ..., w_m)^T$ and $\phi = (\phi_0, ..., \phi_m)^T$.

Remark 4.5. Models like Equation (4.4) are called *linear models* because they are linear in the parameters $\vec{w}$, nevertheless the model $y(\vec{w}, \vec{x})$ can be a non-linear function of the inputs $\vec{x}$ (it suffices to consider non-linear basis functions). Moreover, notice that if each ϕ_j is defined as the projection onto the j-th coordinate then Equation (4.4) recovers Equation (4.2).

Before deep learning it was common practice in machine learning to use some form of pre-processing of the input variables x, also known as *feature extraction*, expressed in terms of a set of basis functions. The goal was to find a suitable set of basis functions in a way that the resulting learning task could be solved using a simple network model. This is very difficult task. Deep Learning avoids this problem by learning the required nonlinear transformations of the data from the data set itself. To put it in more simpler terms, Deep Learning can help in the task on finding new suitable features that will help to build a more robust and accurate model. This turns out to be very useful for problems about image recognition. We will see more about this in Part III. In the following table we provide a list of some common useful choices of the basis functions (assuming a single feature x):

Basis functions $\phi_j(x)$	Model equation $y(x, \vec{w})$	Model nature
x^j	$w_0 + w_1 x + w_2 x^2 + \cdots + w_m x^m$	Polynomial
$\exp\left(-\frac{(x-\mu_j)^2}{2s^2}\right)$	$\sum w_i \exp\left(-\frac{(x-\mu_j)^2}{2s^2}\right)$	Gaussian basis functions
$\sigma\left(\frac{x-\mu_j}{s}\right)$	$\sum w_i \sigma\left(\frac{x-\mu_i}{s}\right)$	Sigmoidal basis functions

where σ denotes the sigmoid function defined by

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

4.1.2 Maximum likelihood

There are different ways to estimate the parameters $\vec{w}$ for a given model $y(\vec{x}, \vec{w})$ using the dataset $\{(\vec{x}^{(i)}, y^{(i)}) : i = 1, 2, ..., N$
Perhaps the most common method consist of finding the parameters $\vec{w}$ that maximize a certain function

called *likelihood function*. To derive such function we must make some assumptions about the relation between the target variables $y^{(i)}$ and the inputs $\vec{x}^{(i)}$.

Let us then assume that the target variable y is given by a deterministic function as in 4.4 $y(\vec{x}, \vec{w})$ with some additive Gaussian noise. This means that the target values and the inputs are related by an equation

(4.6) {{Eq.Sec.Lin.Reg.3}}
$$y = y(\vec{x}, \vec{w}) + \epsilon$$

where ϵ is a random variable that captures either unmodeled effects (*e.g.,* missing important features) or random noise. We assume further that ϵ is a zero-mean Gaussian with variance σ^2. We can write this last assumption as $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Therefore, the probability density of ϵ is given by

$$p(\epsilon) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{\epsilon^2}{2\sigma^2}\right)$$

By substituting in 4.6 we then get $p(y|\vec{x}; \vec{w}, \sigma) \simeq \mathcal{N}(y(\vec{x}, \vec{w}, \sigma))$. In other words,

$$p(y|\vec{x}; \vec{w}, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(y - \vec{w}^T\phi(\vec{x}))^2}{2\sigma^2}\right) \tag{4.7}$$

Finally we can then consider the *design matrix* whose rows correspond to the vectors $\vec{x}^{(i)}$ in the dataset together with the vector or target values $\vec{y}$, *i. e.*,

There's an error here in this matrix, one has to include phi

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots & x_m^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots & x_m^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & x_3^{(N)} & \dots & x_m^{(N)} \end{bmatrix}_{Nxm} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \ddots \\ y^{(N)} \end{bmatrix}$$

Remark 4.8. Once we start using our dataset we are working under the assumption that each $y^{(i)}$ satisfies a relation as in 4.6 and moreover these of the $\epsilon^{(i)}$ are themselves independent (and hence also the $y^{(i)}$'s given the $\vec{x}^{(i)}$'s).

From the last assumption we obtain an expression for the likelihood function:

$$L(\vec{w}) = L(\vec{w}; X, \vec{y}) = p(\vec{y}|X, \vec{w}, \sigma^2) = \prod_{i=1}^{N} p(y^{(i)}|\vec{x}^{(i)}, \vec{w}, \sigma) \tag{4.9}$$

The principle of *maximum likelihood* implies that we should choose the parameters $\vec{w}$ so that the data has the highest possible probability. In other words, we should choose $\vec{w}$ so that the maximum likelihood function $L(\vec{w})$ is maximum. To simplify calculations we can, equivalently, maximize the *log likelihood* $\ell(\vec{w}) = \ln(L(\vec{w}))$. A simple calculation shows that

$$\ell(\vec{w}) = \sum_{i=1}^{N} \ln\left(\frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))^2}{2\sigma^2}\right)\right) \tag{4.10}$$

$$= -\frac{N}{2}\ln(2\pi) - \frac{N}{2}\ln(\sigma^2) - \frac{1}{\sigma^2} \cdot \underbrace{\frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))^2}_{J(\vec{w})} \tag{4.11}$$

The first two quantities in the last equality are constant and therefore, maximizing the log likelihood $\ell(\vec{w})$ is equivalent to minimizing the *sum-of-squares error function*, also called the *cost function* $J(\vec{w})$:

(4.12) {{Eq.Sec.Lin.Reg.4}}
$$J(\vec{w}) = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))^2$$

We will observe two ways of minimizing the cost function J.

4.1.3 Gradient Descent and LMS algorithm

Recall we want to find the parameters $\vec{w}$ so that the cost function $J(\vec{w})$ is minimized. To do this we can use the *gradient descent* algorithm. The idea is to start with an initial guess, say, $\vec{w} = (w_0, w_1, ..., w_m)$ and then iteratively update the parameters using the following rule:

(4.13) {{Eq.Sec.Lin.Reg.5}}
$$\vec{w} \leftarrow \vec{w} - \eta\nabla J(\vec{w})$$

where η is a (usually small) positive number called the *learning rate* and $\nabla J(\vec{w})$ is the gradient of the cost function $J(\vec{w})$. The gradient is a vector whose components are given by the partial derivatives of J with respect to each parameter w_j:

(4.14) {{Eq.Sec.Lin.Reg.6}}
$$\nabla J(\vec{w}) = \begin{bmatrix} \frac{\partial J}{\partial w_0} \\ \frac{\partial J}{\partial w_1} \\ \vdots \\ \frac{\partial J}{\partial w_m} \end{bmatrix} = - \begin{bmatrix} \sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))\phi_0(\vec{x}^{(i)}) \\ \sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))\phi_1(\vec{x}^{(i)}) \\ \vdots \\ \sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))\phi_m(\vec{x}^{(i)}) \end{bmatrix}$$

By replacing in Equation (4.13) and grouping all coordinates we obtain the following update vector rule for the parameters:

(4.15)
$$\vec{w} \leftarrow \vec{w} + \eta \cdot \sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))\phi(\vec{x}^{(i)})$$

This rule is called the *Least Mean Squares update rule*, sometimes also refered as (LMS) algorithm. The LMS algorithm is a stochastic gradient descent algorithm that updates the parameters $\vec{w}$ using the gradient of the cost function $J(\vec{w})$ at each iteration. The learning rate η controls the step size of the update and can be adjusted to improve convergence. The full pseudo code for the LMS algorithm reads as follows:

Algorithm 1 Least Mean Squares (LMS) Algorithm

```
Require: Training set $\{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, learning rate $\eta$
Require: Feature functions $\phi_j$, $j = 0, 1, \ldots, m$
Initialize $\vec{w} \leftarrow \vec{0}$ or random values
repeat
    $\vec{w} \leftarrow \vec{w} + \eta \sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))\phi(\vec{x}^{(i)})$
until convergence or maximum iterations return $\vec{w}$
```

Remark 4.16. This method is called *batch gradient descent* because it uses the entire training set to compute the gradient at each iteration. In practice, we can also use *stochastic gradient descent* (SGD) which updates the parameters using only one training example at a time. This is done by replacing the sum in Equation (4.13) by a single term $(y^{(i)} - \vec{w}^T \phi(\vec{x}^{(i)}))\phi(\vec{x}^{(i)})$ and iterate over each i. The SGD algorithm is usually faster than batch gradient descent and can be used for large datasets.

Finally notice also that the LMS algorithm is a special case of the more general *gradient descent* algorithm. The main difference is that in the LMS algorithm we use the gradient of the cost function $J(\vec{w})$ to update the parameters $\vec{w}$, while in gradient descent we can use any function to update the parameters. In general gradient descent is susceptible to capture local minima of a function, while the LMS algorithm will capture one global minimum. This is because the cost function $J(\vec{w})$ is a convex function, which means that it has only one global minimum. In other words the LMS will always converge to the same solution regardless of the initial guess $\vec{w}$.

4.1.4 The Normal Equations

The LMS algorithm is a very powerful method to minimize the cost function $J(\vec{w})$ but it can be slow to converge. In some cases, we can find the optimal solution for the parameters $\vec{w}$ in closed form. This is done by setting the gradient of the cost function $J(\vec{w})$ to zero and solving for $\vec{w}$. This gives us the so-called *normal equations*:

4.1.5 Regularization

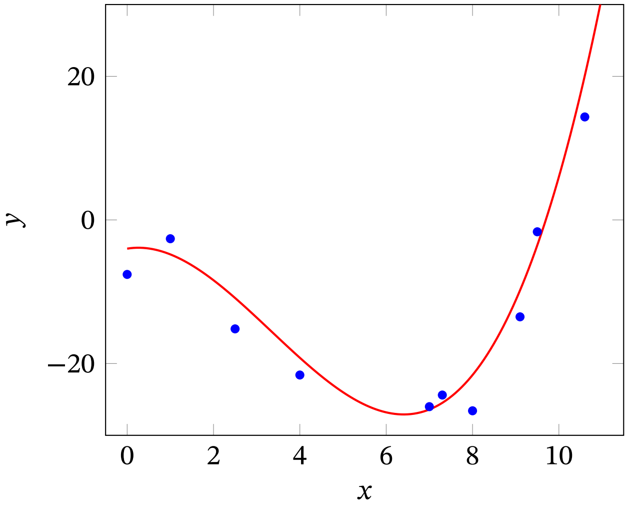
Let us consider a linear model (of polynomial nature) of the form:

$$\text{\{\{Eq.Sec.Lin.Reg.9\}\}}\; y(x, \vec{w}) = \sum_{j=0}^{m} w_j \phi_j(x) = w_0 + w_1 x + w_2 x^2 + \cdots + w_m x^m \tag{4.17}$$

where $\phi_j(x) = x^j$. Notice that we are considering a single feature x and in this case m is no longer the number of features but the degree of the polynomial $y(x, \vec{w})$. Sometimes this kind of model is also refered as a model having polynomial features $x, x^2, x^3, ..., x^m$. We will stick to the first terminology for simplicity.

In this situation we want to use the polynomial model given by Equation (4.19) to fit or approximate a dataset $\{(x^{(i),y^{(i)}}) : i = 1, ..., N\}$. Notice that we are not using the notation $\vec{x}^{(i)}$ because we are only using one feature x. The degree of the polynomial model now becomes a hyperparameter of the model. That is, our goal now is to find the parameters $\vec{w}$ together with the degree m such that the polynomial $y(\vec{w}, x)$ is a good approximation of the dataset and such that the model will give suitable predictions for new data. The choice of the degree of the polynomial the becomes important. A high degree polynomial may fit well the dataset but it may not generalize well to new data. This situation is usually know as *overfitting*. Similarly, a low degree polynomial may not fit well the dataset but it may generalize well to new data. This is known as *underfitting*.

In the following figure we show four examples of the results of fitting polynomials on some given data points. The polynomials have degrees 1,2,3,4,5 and 7. The data points are shown in blue and the fitted polynomials are shown in red. The first two polynomials (degree 1 and 2) are underfitting the data, while the last two polynomials (degree 5 and 7) are overfitting the data. The polynomial of degree 3 is a good fit for the data.



4.2 Python Implementation

This section is dedicated to the implementation in python of the LMS algorithm via some problems and exercises taken from Andrew Ng CS229 course at Stanford University. We will use the numpy library which is a powerful library for numerical computing in Python, and the pandas library, a powerful library for data manipulation and analysis. We will also use the matplotlib library to plot the data and the results. In some other problems we will use the scikit-learn library which is a powerful library for machine learning in Python. These notes will contain the main functions and routines in Python. Additional code used to generate the figures and plots will be provided in the Implementations folder of the course repository.

4.2.1 Cost, Gradient and Gradient Descent

Cost function

Let us first implement a function that computes the cost function. Remember that the cost function is given by:

(4.18) {{Eq.Sec.Lin.Reg.8}}

$$J(\vec{w}) = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\phi(\vec{x}^{(i)}))^2$$

We will assume that the basis functions ϕ_J are given by projections onto the j-th coordinate, *i. e.* $\phi_j(\vec{x}) = x_j$. So that

$$\phi(\vec{x}^{(i)}) = \begin{bmatrix} 1 & x_1^{(i)} & x_2^{(i)} & \cdots & x_m^{(i)} \end{bmatrix}^T$$

In this case we can write the cost function as in Equation (4.2) as follows:

(4.19) {{Eq.Sec.Lin.Reg.9}}

$$J(\vec{w}) = \frac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\vec{x}^{(i)})^2$$

where $\vec{x}_0^{(i)} = 1$ for all $i = 1, ..., N$. The code to compute the cost function for a single training example $\vec{x}$ and target value y is then given by:

{code:compute_

```python
def compute_cost(X,y,w):
    """
    Compute the cost function for logistic regression.

    Parameters:
    X : numpy array (N,m). Data matrix where N is the number of samples and m is the number of features.
    y : numpy array (N,). Vector or target values.
    w : numpy array (M+1,). Weights vector including the bias term w_0.

    Returns:
      Cost value : float.
    """
    N,m = X.shape
    X = np.hstack((np.ones((N, 1)), X)) # Add dummy feature 1 term to X
    w = w.reshape(-1, 1) # Ensure w is a column vector
    y = y.reshape(-1, 1) # Ensure y is a column vector
    cost = (1/2)*np.sum((y - X@w)**2)
    return cost
```

Listing 1: Compute cost

Along all these notes the common conventions and practices will be to use vectorized or Linear algebra notation. In the last code snippet, we start by append to the left a column of ones to the data matrix X. This is done to include the bias term w_0 in the model. Next we ensure that both the weights vector $\vec{w}$ and the outputs vector $\vec{y}$ are both seen as column vectors. The compute_cost function takes $\vec{w}$ and $\vec{y}$ as 1D numpy arrays which are not the same as $N \times 1$ and $(m+1) \times 1$ vectors. Next, we can compute the cost function by simply computing the vector difference y - X@w:

$$\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} - \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots & x_m^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots & x_m^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(N)} & x_2^{(N)} & x_3^{(N)} & \dots & x_m^{(N)} \end{bmatrix} \cdot \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Finally, the function simply uses the np.sum function to compute the sum of the squares of the differences. The np.sum function is a vectorized function that computes the sum of all the elements of an array much faster than a for loop.

Gradient

The next step is to implement the gradient descent algorithm. Recall that we have to update the parameters $\vec{w}$ using the gradient of the cost function $J(\vec{w})$. First, we create a function to compute the gradient:

{code:compute_

```python
def compute_gradient(X,y,w):
    """
    Compute the gradient for linear regression.

    Parameters:
    X : numpy array (N,m). Data matrix where N is the number of samples and m is the number of features.
    y : numpy array (N,). Vector or target values.
    w : numpy array (m+1,). Weights vector including the bias term w_0.

```

```
10     Returns:
11       Gradient : numpy array (m+1,) containing at each entry the gradient with respect to the parameters w.
12     """
13     N,m = X.shape
14     X = np.hstack((np.ones((N, 1)), X)) # Add dummy feature 1 term to X
15     w = w.reshape(-1, 1) # Ensure w is a column vector
16     y = y.reshape(-1, 1) # Ensure y is a column vector
17     gradient = - (y-X@w).T @ X
18 
19     return gradient.flatten() # Return as a flat array for consistency with w shape
```

Listing 2: Compute Gradient

Remark 4.20. The expression in line 17 of the code snippet is a vectorized version of the expression in Equation (4.14). Formally we have a matrix product of $(\vec{y} - X \cdot \vec{w})^T$ with X . The former expression is a $1xN$ vector and the latter is a $Nx(m+1)$ matrix. The result is a $1x(m+1)$ vector containing the partial derivatives of the cost function with respect to each parameter w_j or in other words, the gradient of the cost function.

LMS algorithm

Finally we proceed to implement the LMS algorithm using gradient descent to minimize the sum-of-squares error function. We will use the compute_cost and compute_gradient functions to compute the cost and the gradient at each iteration.

{code:gradient_

```
def gradient_descent(X, y, w_in, cost_function, gradient_function,eta=0.01, num_iter=1000):
    """
    Batch gradient descent algorithm for linear regression. Updates the weights w by taking num_iters gradients steps with learning rate eta

    Parameters:
    X (ndarray (N,m))      :Data matrix with N samples and m features.
    y (ndarray (N,))      :Vector of target values.
    w_in (ndarray (m+1,))   :Initial weights vector including the bias term w_0.
    cost_function        :Function to compute the cost.
    gradient_function      :Function to compute the gradient.
    eta (float)         :Learning rate.
    num_iter (int)        :Maximum number of iterations.
    tol (float)         :Tolerance for convergence.

    Returns:
      w (ndarray (m+1,)) : Final weights vector.
      cost_history (list) : Cost history during training.
    """
    N,m = X.shape
    X = np.hstack((np.ones((N, 1)), X)) # Add dummy feature 1 term to X
    w = copy.deepcopy(w_in).reshape(-1, 1) # Ensure w is a column vector
    y = y.reshape(-1, 1) # Ensure y is a column vector

    cost_history = [] # To store the cost at each iteration for plotting or analysis

    for i in range(num_iter):
```

```python
        # Compute gradient using
        gradient = gradient_function(X,y,w)
        w -= eta * gradient

        # Save the cost at each iteration
        cost_history.append(cost_function(X,y,w))

        # Print cost and gradient information every 10 times or as many as num_iters if less than 10
        if i% math.ceil(num_iter / 10) == 0:
            print(f"Iteration {i:4d}: Cost = {cost_history[-1]:8.4f}, Gradient Norm = {np.linalg.norm(gradient):8.4f}")

    return w, cost_history
```

Listing 3: Gradient Descent

Observation 4.21. It is common practice to consider an average cost function, *i. e.*, we can divide the cost function by N in Equation (4.18) and obtain:

(4.22) {{Eq.Sec.Lin.Reg.10}}

$$J(\vec{w}) = \frac{1}{2N}\sum_{i=1}^{N}(y^{(i)} - \vec{w}^T\vec{x}^{(i)})^2$$

This is done to make the cost function independent of the number of training examples. Of course this has no effect in finding the minimum as N is a constant. However dividing by N makes our calculations more consistent, especially when working with big data sets. In this case we will have to divide the gradient by N as well. The gradient then turns out to be an ”average gradient” which again, makes updates more consistent regardless of the dataset size. We have not done this in the code snippet for simplicity but the reader is encouraged to do so.

4.2.2 House price prediction

Let us go back to our initial motivating example and use our previous implementation of the LMS algorithm. We will use a dataset of house prices stored in a txt file called houses.txt. The dataset contains prices of 100 houses with 4 features (size, number of bedrooms, floors and age). This dataset was derived from the *Ames Housing dataset* which is a popular dataset for regression tasks. The dataset is available at https://www.kaggle.com/datasets/shashanknecrothapa/ames-housing-dataset.

Remark 4.23. Each of the problems or practical exercises will have a separate Jupyter notebook where the reader can observe and play around with the code. The reader is encouraged to use the Jupyter notebooks to run the code and play around with the parameters. The Jupyter notebooks also are available in the 📂Implementations folder of the course repository.

The first four rows of the dataset are shown below:

Table 1: House Data Sample

{tab:house_data

Index	Size (sqft)	Number of bedrooms	Number of floors	Age	Price (in 1000’s)
0	952.0	2.0	1.0	65.0	271.5
1	1244.0	3.0	1.0	64.0	300.0
2	1947.0	3.0	2.0	17.0	509.8
3	1725.0	3.0	2.0	42.0	394.0

We can have a better understanding of our dataset by plotting each of the features against the target variable (price):

5 Classification and Logistic Regression

5.1 Logistic Regression

5.1.1 The perceptron algortihm

5.2 Python Implementation

6 Generalized Linear Models

Part III

Deep Learning

{part:deep_learn

7 Neural Networks

8 Convolutional Neural Networks

{sec:cnn}

9 Generative Adversarial Networks (GANs)

{sec:gans}

10 Transformers

{sec:transformer

10.1 Natural language processing (NLP)

{subsec:nlp_wor

Natural language processing is field of study of linguistics and machine learning that focuses on understanding the interaction between computers and human language. In particular, it is related to the task of enabling computers to understand, interpret and generate human language. NLP comprises a wide range of tasks and techniques, including:

- Text classification: Assigning predefined categories to text documents based on their content. Examples include spam detection, sentiment analysis, topic classification and determining if a sentence is grammatically correct or not.
- Named entity recognition (NER): Identifying and classifying named entities in text, such as people, organizations, locations and dates.
- Part-of-speech tagging: Assigning grammatical categories (e.g., noun, verb, adjective) to words in a sentence.
- Machine translation: Automatically translating text from one language to another.
- Text generation: Creating coherent and contextually relevant text based on a given input or prompt.
- Question answering: Given a question and a context, extracting the answer to the question based on the information in the context.

10.1.1 Word embeddings

{subsubsec:word

Many languages can be seen as a collection of sequential data where each element of the sequence is a word and they are separated by white spaces along with punctuation symbols. The first step in processing this kind of data is to find a way to convert word into a numerical representation that is suitable to be used as input to a deep neural network. It is worth mentioning that the most simple approach to this task is to use a one-hot encoding representation of the words. This means that each word is represented as a vector of zeros with a single one at the position corresponding to the index of the word in some predetermined dictionary. For example, if we have a vocabulary of 10,000 words, the word *cat* would be represented as a vector of size 10,000 with a one at the index corresponding to *cat* and zeros everywhere else. Of course, a realistic vocabulary or dictionary might have several hundred thousand entries, leading to very large (high dimensional) representation vectors. Also this representation does not capture any semantic information about the words. For example, the words *cat* and *dog* would be represented by two different vectors that are orthogonal to each other, even though they are semantically related to the word *animal*.

Both issues can be addressed by using *word embeddings*, which are dense vector representations of words that capture their semantic meaning. Word embeddings are typically learned from large corpora of text using unsupervised learning techniques. The idea is to represent words in a continuous vector space where semantically similar words are close to each other.

Word2vec

The embedding process can be described as follows: We suppose we have some dictionary of words of size m and we want to represent words in an embedding space of dimension N. Then for each one-hot encoded input vector x_i we can compute the corresponding embedding vector y_i as follows:

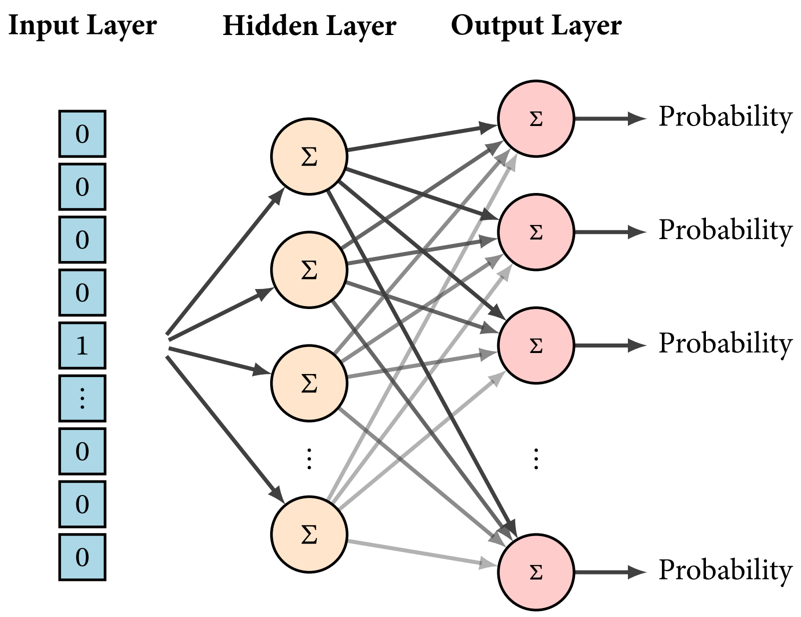
$$y_i = Ex_i \tag{10.1}$$

where E is an $N \times m$ matrix of learnable parameters. The matrix E is usually called the *embedding matrix*. The idea is to learn the parameters of the embedding matrix E in such a way that the resulting embedding vectors y_i capture the semantic meaning of the input words. The matrix E can be learned from a corpus (a large data set) of text. There are many techniques to learn the parameters of the embedding matrix. One of the most popular techniques is referred to as *word2vec* [Mik+13]. This is just one of many frameworks to learning word vectors.

Remark 10.2. Notice that since the input vector x_i has a one-hot encoding representation, the output vector y_i is simply the i-th column of the embedding matrix E. This means that the embedding matrix can be seen as a collection of embedding vectors, one for each word in the dictionary.

The overall architecture behind the word2vec framework consists of a simple two-layer neural network. We will first state two general considerations and later we will delve into two specific architectures used in this framework: *skip-gram* and *continuous bag of words* (CBOW).

- We assume a *distributional hypothesis*, which simply states that words that occur in similar contexts tend to have similar meanings. This means that if two words appear in the same context (e.g., in the same sentence or paragraph), they are likely to have similar meanings. This is a very strong assumption and it is not always true, but it is a good starting point for learning word embeddings.
- The architecture consists of a shallow neural network with a single hidden layer. The input layer will receive one-hot encoded vectors of dimension m (The length of our vocabulary). The hidden layer is a standard fully connected (Dense) layer whose weights are the word embeddings. The output layer will provide probabilities for the target words from the vocabulary. Notice that the number of units in the hidden layer is equal to the the length of the embedding vectors we want, *i. e.*, N.



10.2 Encoder - Decoder Architecture

{subsec:encoder

10.2.1 Encoder

{subsubsec:enco

10.2.2 Decoder

{subsubsec:deco

10.3 Attention mechanism

{subsec:attentio

The fundamental idea behind a transformer model is the *attention* mechanism, which allows the model to focus on different parts of the input sequence when making predictions. This mechanism arose from the need to improve the performance of recurrent neural networks (RNNs) for machine translation tasks [BCB16]. Later on, performance was improved considerably by eliminating the RNN architecture altogether and using a fully attention-based architecture, which is the basis of the transformer model [Vas+17].

Let us consider the following three sentences as an example:

> *I need to* ***run*** *to catch the bus!*
> *Paul decided to* ***run*** *for president.*
> *We had a* ***run*** *of bad luck.*

In each case, the word *run* has a different meaning depending on the context. The attention mechanism allows the model to focus on the surrounding words to determine the meaning of *run* in each case. For example, in the first sentence, the model can pay more attention to the words *catch* and *bus*, while in the second sentence, it can focus on *Paul* and *president*.

10.3.1 Processing

{subsubsec:proc

The input data to a transformer is a collection of vectors $\{\vec{x}^{(i)}\}$ in $\mathbb{R}^m$ where $i = 1, \ldots, N$. As it is usual in these notes, each element of the i-th vector $\vec{x}_j^{(i)}$ is called a *feature* and the data vectors are called *tokens*. These tokens may correspond to words withing a corpus of text, to a patch of pixels within an image, or to any other type of data sensible to be represented (embedded) as a vector. We will associate a matrix $X \in \mathbb{R}^{N\times m}$ to the collection of vectors $\{\vec{x}^{(i)}\}$ as follows:

$$X = \begin{bmatrix} --(\vec{x}^{(1)})^T -- \\ --(\vec{x}^{(2)})^T -- \\ \vdots \\ --(\vec{x}^{(N)})^T -- \end{bmatrix} \tag{10.3}$$

The fundamental block of the Transformer will take the matrix X as input and create a new matrix, $\widehat{X}$ of the same size:

$$\widetilde{X} = \text{TransformerLayer}(X)$$

The idea is to create a new matrix $\widetilde{X}$ that contains the same information as X, but with the features of each token enhanced by the attention mechanism. We can of course stack (compose) several of these layers in sequence to create a deeper model capable of learning more complex relationships between the tokens. This single transformer layer has two stages: the one acting on columns (features) corresponding to the attention mechanism, and the one acting on rows (tokens) which corresponds to the effect of transforming the features within each token.

Attention

Let us denote by $\vec{y}^{(1)}, ..., \vec{y}^{(N)}$ the rows (tokens) of the matrix $\widetilde{X}$. Each of these tokens should live in an embedding space with a richer semantic structure than the tokens of X. Since each token in X correspond to some data type (say words) and we want to capture some semantic relation between them, each vector $\vec{y}^{(i)}$ should depend on all the tokens from X, *i. e.*, $\vec{x}^{(1)}, \vec{x}^{(2)}, ..., \vec{x}^{(N)}$. The simplest thing to do is to assume that each $\vec{y}^{(i)}$ depends linearly, or it is linear combination of the tokens in X:

$$\vec{y}^{(i)} = \sum_{j=1}^{N} a_{ij}\vec{x}^{(j)}$$

where α_{ij} are the coefficients of the linear combination. These coefficients will be called *attention weights*. We expect these coefficients to be close to zero whenever the input tokens are not relevant to the output token. For instance, in our previous example: *"I need to run to catch the bus"*, the attention weights for the output token associated with the word *catch* should be high for the words *catch*, *to* (second), *bus* and *run* for we need to focus on the object of the action (the bus) and the preceding action (run) to understand the meaning of *catch*. On the other hand, the attention weights should be low for the words *I* and *need* and the first *to*.

In the following table we present some linguistic intuition about how the weights should be distributed. The first column contains the words of the sentence, the second and third columns contain notation for the input and output tokens, respectively. The fourth column contains the words that should receive low attention weights, while the fifth column contains the words that should receive high attention weights.

add specific ref label to the table

We will then impose the following constraints on the attention weights:

- The attention weights are non-negative: $a_{ij} \geq 0$, as we want to avoid situations in which one coefficient can become large and positive while another one compensated by being large and negative. This is not desirable in our case, as we want to focus on the most relevant tokens.

{tab:attention_ir

Word	Input Token	Output Token	Low Attention	High Attention
I	x^1	y^1	the, bus, catch	I, need, run
need	x^2	y^2	the, bus	need, I, to (first), run
to	x^3	y^3	the, bus, I, catch	to (first), need, run
run	x^4	y^4	I, need	run, to (first), to (second), catch, bus
to	x^5	y^5	I, need, to (first)	to (second), run, catch, bus
catch	x^6	y^6	I, need, to (first)	catch, to (second), run, the, bus
the	x^7	y^7	I, need, to (first), run	the, bus, catch
bus	x^8	y^8	I, need	bus, the, catch, run

- The attention weights sum to one: $\sum_{j=1}^{N} a_{ij} = 1$. This is a normalization condition that ensures that if an output token pays more attention to one input token, it pays less attention to the others.

Remark 10.4. Notice that if we assume that the outputs are instead linear combinations of basis functions of the input tokens $\phi_1, ..., \phi_N : \mathbb{R}^m \to \mathbb{R}$, the two conditions above ensure that the basis functions form a partition of unity.

Maybe we have to add some more details here. So far it is not that important, just intuition we get from this

Self-attention

Let us discuss how to determine the attention weights a_{ij}. The idea first is to use an approach similar to the one used in problems related with information retrieval. The following image shows the basic idea used to find the attention weights:

add image here

The main idea is to see each of the input vectors $\vec{x}^{(i)}$ as a value vector that will be used to create the output tokens. We will also use $\vec{x}^{(i)}$ as the key vector for the ith input token. Finally we consider each $\vec{x}^{(j)}$ as query vector for the output $\vec{y}^{(j)}$. To achieve the constraints on the attention weights, we will use a softmax function (with no probabilistic interpretation) so that:

$$a_{ij} = \frac{\exp((\vec{x}^{(i)})^T \cdot \vec{x}^{(j)})}{\sum_{k=1}^{N} \exp((\vec{x}^{(i)})^T \cdot \vec{x}^{(k)})}$$

By grouping all the output tokens in a single $N \times m$ matrix Y we obtain a nice matrix formula for the output tokens:

$$Y = \text{Softmax}(XX^{\top})X \qquad (10.5) \quad \text{\{\{Eq.subsec.self\_attention.1\}\}}$$

where the softmax function applied on a $N \times N$ matrix $C = [c_{ij}]$ is a new matrix whose entries given by:

$$\text{Softmax}(C)_{ij} = \frac{\exp(c_{ij})}{\sum_{k=1}^{N} \exp(c_{ik})}$$

This process is called *self-attention* because the same input tokens are used as queries, keys and values. The attention weights are computed by taking the dot product of the query vector with the key vectors, and then applying the softmax function. We will see some variations of this later on.

Network parameters

So far, the transformation we have described to find the output tokens is fixed in the sense that there are no adjustable parameters and therefore this has no learning capacity from data. We would like to build a network that has some flexibility to choose features to focus on when determining the output tokens. For this, we can start by defining modified feature vectors via a linear transformation to the input tokens through a matrix U of learnable parameters:

$$\widehat{X} = XU$$

where U is an $m \times m$ matrix of learnable weight parameters. Notice that this is analogous to a linear layer in a neural network. By replacing X by $\widehat{X}$ in (10.5) we obtain the following expression for the output tokens:

(10.6) {{Eq.subsec.self_attention.2}}
$$\begin{aligned} Y &= \mathrm{Softmax}(\widehat{X}\widehat{X}^\top)\widehat{X} \\ &= \mathrm{Softmax}(XUU^\top X^\top)XU \end{aligned}$$

Remark 10.7. This approach has one remarkable characteristic. The matrix $\widehat{X}\widehat{X}^\top$ is symmetric which will in turn imply a symmetric behavior in the attention mechanism. We need much more flexibility, for instance, many tasks in NLP require tokens (words) to be strongly associated with other tokens but not in the opposite sense: The word *hardware* is strongly associated with the word *computer* but the latter may be associated with many other words and its association with the word *hardware* may not be that strong.

To overcome this limitation we still use the same idea but with independent learnable weight matrices for the query, key and value vectors. We will denote these matrices by W_q, W_k and W_v respectively. We then consider:

$$\begin{aligned} Q &= XW_q & \qquad \dim W_q &= m \times m_k \\ K &= XW_k & \qquad \dim W_k &= m \times m_k \\ V &= XW_v & \qquad \dim W_v &= m \times m_v \end{aligned}$$

where m_k and m_v are the dimensions of the key and value vectors respectively. The dimensions are chosen so that we can perform dot products between the query and key vectors (a typycal choice is to take $m_k = m$). Additionally, m_v will govern the dimension of the output tokens. Finally we obtain an expression for the output tokens as follows:

(10.8) {{Eq.subsec.self_attention.3}}
$$Y = \mathrm{Softmax}(QK^\top)V$$

Remark 10.9. Recall that gradients of the softmax function become exponentially small for high magnitude inputs. To avoid these numerical issues, we can use rescale the product of the query and key vectors before applying softmax. This is done by dividing the dot product $QK^\top$ by $\sqrt{m_k}$. The *scaled self-attention* then takes the form:

(10.10) {{Eq.subsec.self_attention.4}}
$$Y = \mathrm{Softmax}\left(\frac{QK^\top}{\sqrt{m_k}}\right)V$$

The structure of the scaled self-attention layer is summarized in the following diagram:

10.4 Transformer models

{subsec:transfor

References

[BCB16] D. Bahdanau, K. Cho, and Y. Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. May 19, 2016. doi: 10.48550/arXiv.1409.0473. arXiv: 1409.0473 [cs]. url: http://arxiv.org/abs/1409.0473 (visited on 04/20/2025). Pre-published (cit. on p. 16).

[Mik+13] T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. arXiv.org. Jan. 16, 2013. url: https://arxiv.org/abs/1301.3781v3 (visited on 05/21/2025) (cit. on p. 15).

[Vas+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. ukasz Kaiser, and I. Polosukhin. *Attention Is All You Need*. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017 (cit. on p. 16).