

MACHINE LEARNING FROM SCRATCH

SANTIAGO TORO

April 12, 2025

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur 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

I Preliminaries	3
1 Introduction	3
2 Multivariate Calculus	3
3 Probability Theory	3
3.1 Probabilities	3
3.2 Probability Distributions	3
II Supervised Learning	4
4 Regression	4
4.1 Linear Regression	4
4.1.1 Non linear models	6
4.1.2 Maximum likelihood	6
4.1.3 Gradient Descent and LMS algorithm	8
4.1.4 The Normal Equations	9
4.2 Python Implementation	9
4.2.1 Cost, Gradient and Gradient Descent	9
4.2.2 House price prediction	12

5	Classification and Logistic Regression	13
5.1	Logistic Regression	13
5.1.1	The perceptron algortihm	13
5.2	Python Implementation	13
6	Generalized Linear Models	13
III	Deep Learning	14
7	Neural Networks	14

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

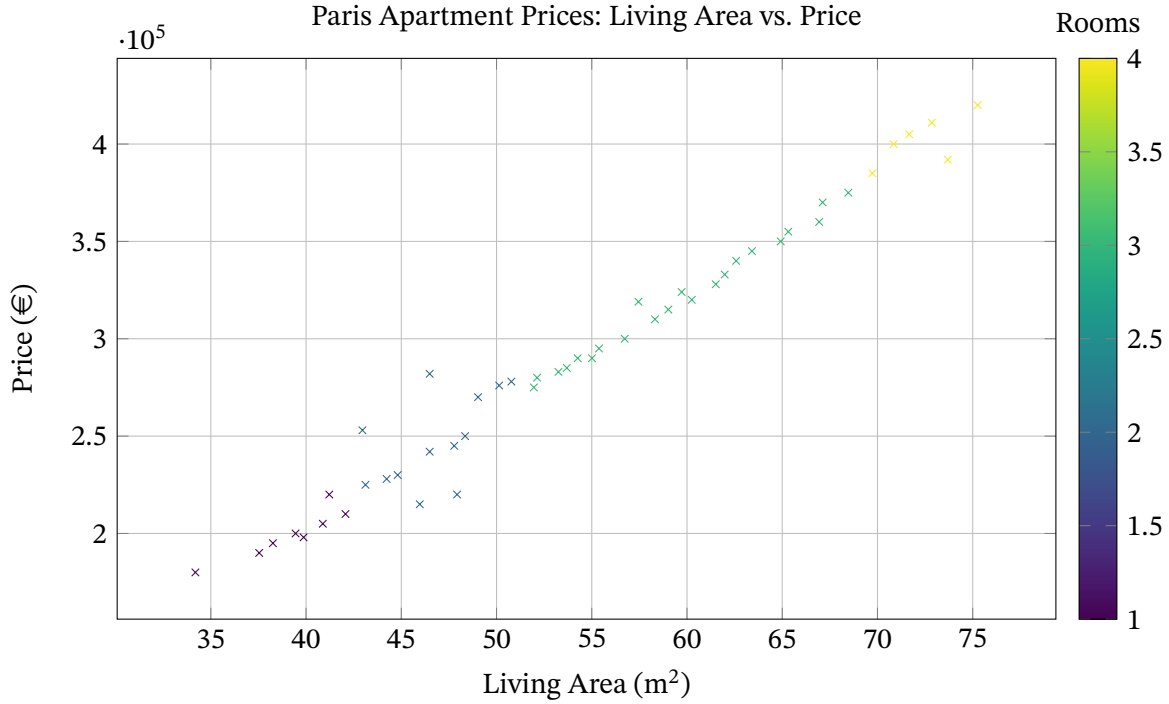
4.1 Linear Regression

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 $\{\vec{x}_n\}_{n=1,2,\dots,N}$ 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:

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

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:

$$(4.2) \quad \{\{\text{Eq. Sec. Lin. Reg. 1}\}\} \quad y(\vec{x}, \vec{w}) = w_0 + w_1x_1 + \dots + w_mx_m$$

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, \dots, w_m and also of the input variables x_1, \dots, 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 $\varphi_0, \dots, \varphi_m : \mathbb{R}^m \rightarrow \mathbb{R}$ are a collection of *basis* functions. Then, we can consider models of the form:

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

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

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

where $\vec{w} = (w_0, \dots, w_m)^T$ and $\varphi = (\varphi_0, \dots, \varphi_m)^T$.

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 classical machine learning to use some form of fixed 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 powerful set of basis functions 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. This turns out to be very useful for problems about image recognition.

In the following table we provide a list of some common useful choices of the basis functions (assuming a single feature x):

Basis functions $\varphi_j(x)$	Model equation $y(x, \vec{w})$	Model nature
x^j	$w_0 + w_1 x + w_2 x^2 + \dots + w_m x^m$	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 $\{(\vec{x}^{(i)}, y^{(i)}) : i = 1, 2, \dots, N\}$. 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

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

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 $\varepsilon \simeq \mathcal{N}(0, \sigma^2)$. Therefore, the probability density of ε is given by

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

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,

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

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.,

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

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

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 $\varepsilon^{(i)}$ 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:

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

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

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

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

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* $J(\vec{w})$:

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

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, $\vec{w} = (w_0, w_1, \dots, w_m)$ and then iteratively update the parameters using the following rule:

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

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 :

$$(4.14) \quad \{\{\text{Eq. Sec. Lin. Reg. 6}\}\} \quad \nabla J(\vec{w}) = \begin{bmatrix} \frac{\partial J}{\partial w_0} \\ \frac{\partial J}{\partial w_1} \\ \vdots \\ \frac{\partial J}{\partial w_m} \end{bmatrix} = - \begin{bmatrix} \sum_{i=1}^N (y^{(i)} - \vec{w}^T \varphi(\vec{x}^{(i)})) \varphi_0(\vec{x}^{(i)}) \\ \sum_{i=1}^N (y^{(i)} - \vec{w}^T \varphi(\vec{x}^{(i)})) \varphi_1(\vec{x}^{(i)}) \\ \vdots \\ \sum_{i=1}^N (y^{(i)} - \vec{w}^T \varphi(\vec{x}^{(i)})) \varphi_m(\vec{x}^{(i)}) \end{bmatrix}$$

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

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

This rule is called the *Least Mean Squares update rule*, sometimes also referred 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 η

Require: Feature functions $\varphi_j, j = 0, 1, \dots, m$

- 1: Initialize $\vec{w} \leftarrow \vec{0}$ or random values
 - 2: **repeat**
 - 3: $\vec{w} \leftarrow \vec{w} + \eta \sum_{i=1}^N (y^{(i)} - \vec{w}^T \varphi(\vec{x}^{(i)})) \varphi(\vec{x}^{(i)})$
 - 4: **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 $(y^{(i)} - \vec{w}^T \varphi(\vec{x}^{(i)}))\varphi(\vec{x}^{(i)})$ 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*:

4.2 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:

$$(4.17) \quad \{\{\text{Eq. Sec. Lin. Reg. 7}\}\} \quad J(\vec{w}) = \frac{1}{2} \sum_{i=1}^N (y^{(i)} - \vec{w}^T \varphi(\vec{x}^{(i)}))^2$$

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

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

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

$$(4.18) \quad \{\{\text{Eq. Sec. Lin. Reg. 8}\}\} \quad J(\vec{w}) = \frac{1}{2} \sum_{i=1}^N (y^{(i)} - \vec{w}^T \vec{x}^{(i)})^2$$

where $x_0^{(i)} = 1$ for all $i = 1, \dots, N$. The code to compute the cost function for a single training example \vec{x} and target value y is then given by:

{code:compute

```
1 def compute_cost(X, y, w):
2     """
3     Compute the cost function for logistic regression.
4
5     Parameters:
6     X : numpy array (N,m). Data matrix where N is the number of samples and m is
7         the number of features.
8     y : numpy array (N,). Vector or target values.
9     w : numpy array (M+1,). Weights vector including the bias term w_0.
10
11     Returns:
12     Cost value : float.
13     """
14     N, m = X.shape
15     X = np.hstack((np.ones((N, 1)), X)) # Add dummy feature 1 term to X
16     w = w.reshape(-1, 1) # Ensure w is a column vector
17     y = y.reshape(-1, 1) # Ensure y is a column vector
18     cost = (1/2)*np.sum((y - X@w)**2)
19     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. Finally 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

```
1 def compute_gradient(X, y, w):
2     """
```

```

3  Compute the gradient for linear regression.
4
5  Parameters:
6  X : numpy array (N,m). Data matrix where N is the number of samples and m is
    the number of features.
7  y : numpy array (N,). Vector or target values.
8  w : numpy array (m+1,). Weights vector including the bias term w_0.
9
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 4.19. 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 $1 \times N$ vector and the latter is a $N \times (m + 1)$ matrix. The result is a $1 \times (m + 1)$ 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.

```

1 def gradient_descent(X, y, w_in, cost_function, gradient_function, eta=0.01,
    num_iter=1000):
2     """
3     Batch gradient descent algorithm for linear regression. Updates the weights
        w by taking num_iters gradients steps with learning rate eta
4
5     Parameters:
6     X (ndarray (N,m)) : Data matrix with N samples and m features.
7     y (ndarray (N,)) : Vector of target values.
8     w_in (ndarray (m+1,)) : Initial weights vector including the bias term w_0.
9     cost_function : Function to compute the cost.
10    gradient_function : Function to compute the gradient.
11    eta (float) : Learning rate.
12    num_iter (int) : Maximum number of iterations.
13    tol (float) : Tolerance for convergence.
14
15    Returns:
16        w (ndarray (m+1,)) : Final weights vector.
17        cost_history (list) : Cost history during training.

```

{code:gradient

```

18  """
19  N,m = X.shape
20  X = np.hstack((np.ones((N, 1)), X)) # Add dummy feature 1 term to X
21  w = copy.deepcopy(w_in).reshape(-1, 1) # Ensure w is a column vector
22  y = y.reshape(-1, 1) # Ensure y is a column vector
23
24  cost_history = [] # To store the cost at each iteration for plotting or
    analysis
25
26  for i in range(num_iter):
27      # Compute gradient using
28      gradient = gradient_function(X,y,w)
29      w -= eta * gradient
30
31      # Save the cost at each iteration
32      cost_history.append(cost_function(X,y,w))
33
34      # Print cost and gradient information every 10 times or as many as
    num_iters if less than 10
35      if i% math.ceil(num_iter / 10) == 0:
36          print(f"Iteration {i:4d}: Cost = {cost_history[-1]:8.4f}, Gradient Norm
    = {np.linalg.norm(gradient):8.4f}")
37
38  return w, cost_history

```

Listing 3: Gradient Descent

Observation 4.20. 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:

$$(4.21) \quad \{\{\text{Eq. Sec. Lin. Reg. 9}\}\} \quad J(\vec{w}) = \frac{1}{2N} \sum_{i=1}^N (y^{(i)} - \vec{w}^T \vec{x}^{(i)})^2$$

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>.

5 Classification and Logistic Regression

5.1 Logistic Regression

5.1.1 The perceptron algorithm

5.2 Python Implementation

6 Generalized Linear Models

Part III

Deep Learning

7 Neural Networks