Homework 3

CS 559 - Neural Networks - Fall 2019

Matteo Corain 650088272

October 14, 2019

1 Question 1

1.1 Gradient and Hessian of f

The gradient of the function $f(x,y) = -\log(1-x-y) - \log x - \log y$ may be computed as follows:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x}(x,y) \\ \frac{\partial f}{\partial y}(x,y) \end{bmatrix}$$

$$\frac{\partial f}{\partial x}(x,y) = -\frac{1}{1-x-y}\frac{\partial}{\partial x}(1-x-y) - \frac{1}{x} = \frac{1}{1-x-y} - \frac{1}{x}$$

$$\frac{\partial f}{\partial y}(x,y) = -\frac{1}{1-x-y}\frac{\partial}{\partial y}(1-x-y) - \frac{1}{y} = \frac{1}{1-x-y} - \frac{1}{y}$$

$$\nabla f(x,y) = \begin{bmatrix} \frac{1}{1-x-y} - \frac{1}{x} \\ \frac{1}{1-x-y} - \frac{1}{y} \end{bmatrix}$$

Starting from those result, the Hessian matrix of $f(x,y)$ may be computed as follows:

$$Hf(x,y) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2}(x,y) & \frac{\partial^2 f}{\partial x \partial y}(x,y) \\ \frac{\partial^2 f}{\partial y \partial x}(x,y) & \frac{\partial^2 f}{\partial y^2}(x,y) \end{bmatrix}$$

$$\frac{\partial^2 f}{\partial x^2}(x,y) = \frac{\partial}{\partial x}\frac{\partial f}{\partial x} = -\frac{1}{(1-x-y)^2}\frac{\partial}{\partial x}(1-x-y) + \frac{1}{x^2} = \frac{1}{(1-x-y)^2} + \frac{1}{x^2}$$

$$\frac{\partial^2 f}{\partial x \partial y}(x,y) = \frac{\partial}{\partial x}\frac{\partial f}{\partial y} = -\frac{1}{(1-x-y)^2}\frac{\partial}{\partial x}(1-x-y) = \frac{1}{(1-x-y)^2}$$

$$\frac{\partial^2 f}{\partial y \partial x}(x,y) = \frac{\partial}{\partial y}\frac{\partial f}{\partial x} = -\frac{1}{(1-x-y)^2}\frac{\partial}{\partial y}(1-x-y) = \frac{1}{(1-x-y)^2}$$

$$\frac{\partial^2 f}{\partial y^2}(x,y) = \frac{\partial}{\partial y}\frac{\partial f}{\partial y} = -\frac{1}{(1-x-y)^2}\frac{\partial}{\partial y}(1-x-y) + \frac{1}{y^2} = \frac{1}{(1-x-y)^2} + \frac{1}{y^2}$$

$$Hf(x,y) = \begin{bmatrix} \frac{1}{(1-x-y)^2} + \frac{1}{x^2} & \frac{1}{(1-x-y)^2} \\ \frac{1}{(1-x-y)^2} & \frac{1}{(1-x-y)^2} + \frac{1}{y^2} \end{bmatrix}$$

Using the obtained results, it is possible to analytically compute where the minimum of the given function is located; in fact, this correspond to the point (x, y) for which we have $\nabla f(x, y) = 0$. This condition is equivalent to the solution of the following system of equalities:

$$\nabla f(x,y) = 0 \Rightarrow \begin{cases} \frac{\partial f}{\partial x}(x,y) = \frac{1}{1-x-y} - \frac{1}{x} = 0 \\ \frac{\partial f}{\partial y}(x,y) = \frac{1}{1-x-y} - \frac{1}{y} = 0 \end{cases} \Rightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \end{bmatrix}$$

We expect therefore that both the gradient descent and the Newton's methods will be able to converge to points in the neighborhood of $\left(\frac{1}{3}, \frac{1}{3}\right)$.

1.2 Gradient descent method

The gradient descent method is based on the update rule:

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} \leftarrow \begin{bmatrix} x_n \\ y_n \end{bmatrix} - \eta \nabla f(x_n, y_n)$$

This method was implemented in code using the `gdesc()` function, which takes as arguments the initial point (x_0, y_0), the value of η and the value of ϵ, which defines the minimum entity of the update before stopping the iterations.

At each iteration, the function updates the estimate of the minimum, appends to the array `xylist` the value of the updated point and to the array `fxy` the value of the function in correspondence of that updated point. At convergence, the algorithm returns the final estimate and the two lists `xylist` and `fxy`.

Results are then plotted using the `plot_traj()` and `plot_fxy()` functions. The first creates a new figure, defines a triangular grid inside the domain $\mathcal{D}$ of the function (keeping distance δ from the domain boundaries and with the given number of points) and plots both the contours of the function at the given levels and the sequence of weights onto that plot. The second function performs a simple plot operation of the given vector.

Obtained results are shown in figures 1 and 2. Those were computed starting from the initial point $(x_0, y_0) = [0.90348 \quad 0.03794]$, decreasing the value of η to 0.01 in order not to make the point jump outside $\mathcal{D}$ and using $\epsilon = 10^{-6}$.

1.3 Newton's method

The Newton's method is based on the weights update rule:

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} \leftarrow \begin{bmatrix} x_n \\ y_n \end{bmatrix} - \eta [Hf(x_n, y_n)]^{-1} \nabla f(x_n, y_n)$$

This method was implemented in code using the `newton()` function, which performs similar action with respect to the one implementing gradient descent. The same functions are used for plotting. The method was made run using the same initial point (x_0, y_0) and minimum update ϵ as before, but using learning parameter $\eta = 1$; results are shown in figures 3 and 4.

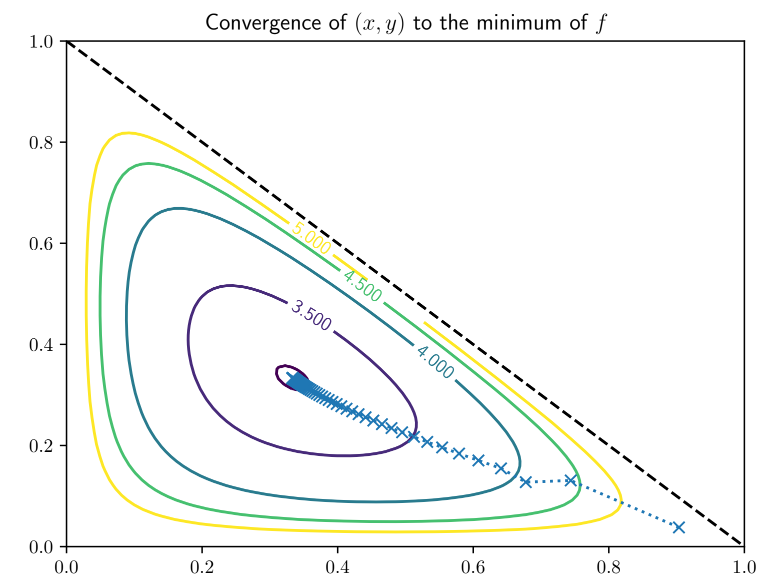

Figure 1: Convergence using gradient descent method

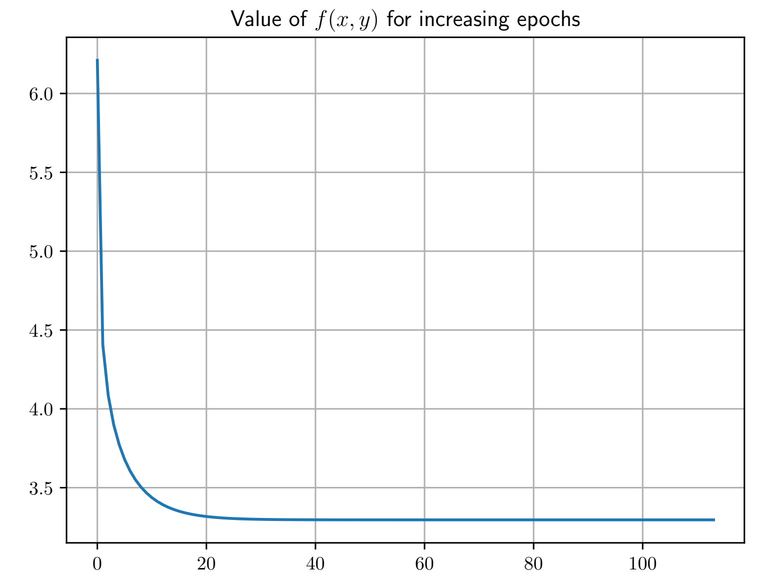

Figure 2: Decrease of f-values using gradient descent method

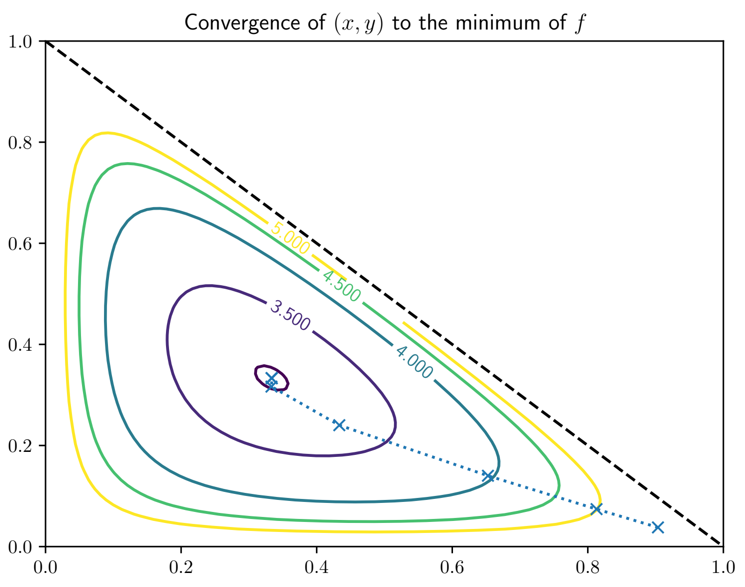

Figure 3: Convergence using Newton's method

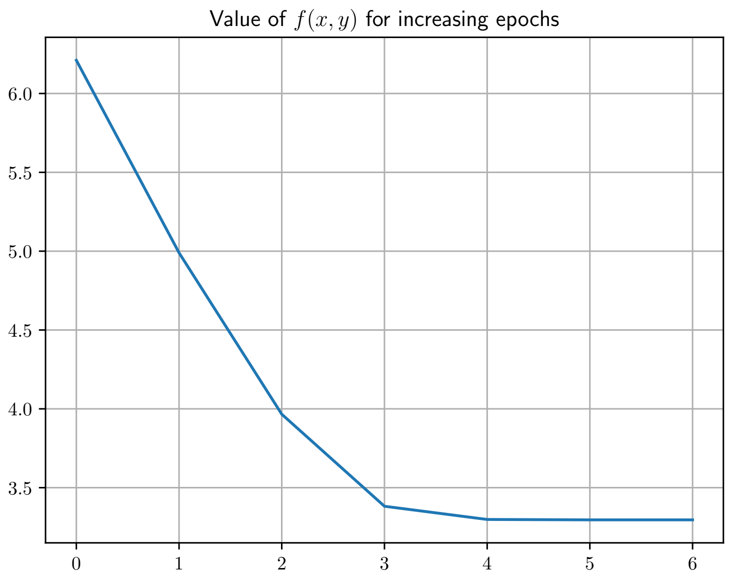

Figure 4: Decrease of f-values using Newton's method

1.4 Comparison

Method	η	Epochs	Convergence value
Gradient descent	0.01	114	[0.33334021 0.33332646]
Newton	1	7	[0.33333333 0.33333333]

Table 1: Summary of the results obtained using gradient descent and Newton's method

As it can be seen from the summary of the results, Newton's method was able to converge in a much lower number of epochs and with a much higher precision to the minimum of $f(x, y)$. The second result was somehow expected because the approximation yielded by the Newton's method, using a second-order term instead of only relying on the linear one, is more precise than the one used for the gradient descent method.

For what the first result is concerned, however, results may vary. In fact, if we use the same value of $\eta < 1$ that is able to guarantee convergence for both for the gradient descent and the Newton's methods, we can observe that the Newton's method convergence is much slower than the gradient descent method's. This was tested using different values for the learning rate and in all cases the gradient descent method outperformed Newton's method in terms of number of epochs needed for convergence, as shown in table 2.

η	Epochs (gradient descent)	Epochs (Newton's method)
0.01	114	988
0.015	77	685
0.02	57	527
0.025	44	430
0.03	34	364
0.035	25	316
0.04	26	280
0.045	24	241
0.05	-	228

Table 2: Number of epochs for convergence using different η values

This result may be motivated by analyzing the step size of both methods. In fact, if we compare the update term $-\nabla f$ of the gradient descent method with the update term $-[Hf]^{-1}\nabla f$ of the Newton's method, we can observe that, for this particular function, the entity of the first is much bigger than the second (in terms of its norm). Let us consider for example what happens in correspondence of the initial point $(x_0, y_0) = [0.90348 \quad 0.03794]$:

$$\nabla f(x_0, y_0) = \begin{bmatrix} 15.96427362 \\ -9.28681822 \end{bmatrix}$$

$$[Hf(x_0, y_0)]^{-1}\nabla f(x_0, y_0) = \begin{bmatrix} 0.09058607 \\ -0.03618637 \end{bmatrix}$$

Therefore, if we use a value of η that is able to make the gradient descent method converge, we cannot expect the Newton's method to converge very fast, since its update terms are consistently smaller than the gradient descent's ones. In the case of our function, Newton's method shows its advantages in terms of epochs only when the value of η surpasses the critical value for the convergence of gradient descent.

1.5 Complete Python code

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Use TeX for text rendering
plt.rc('text', usetex=True)

# Set the random seed for reproducibility
np.random.seed(2019)

def f(x, y):
    return - np.log(1-x-y) - np.log(x) - np.log(y)

def gradf(x, y):
    return np.array([
        1/(1-x-y) - 1/x,
        1/(1-x-y) - 1/y
    ]).reshape(2, 1)

def hessf(x, y):
    return np.array([
        1/((1-x-y)**2) + 1/(x**2),
        1/((1-x-y)**2),
        1/((1-x-y)**2),
        1/((1-x-y)**2) + 1/(y**2)
    ]).reshape(2, 2)

def gdesc(xy0, eta, eps):
    # Initialize parameters
    xy = np.array(xy0).reshape(2, 1)
    xylist = [np.copy(xy)]
    fxy = [f(xy[0], xy[1])]

    # Run the algorithm until the gradient is nearly zero
    while True:
        # Save old point
        xy_old = np.array(xy)

        # Perform an update
        xy = xy - eta*gradf(xy[0], xy[1])
        xylist.append(np.copy(xy))
        fxy.append(f(xy[0], xy[1]))

        # Return if the norm of the update is small enough
        if np.linalg.norm(xy - xy_old) < eps:
            return xy, xylist, fxy

def newton(xy0, eta, eps):
    # Initialize parameters
    xy = np.array(xy0).reshape(2, 1)
    xylist = [np.copy(xy)]
    fxy = [f(xy[0], xy[1])]
    while True:
        # Save old point
        xy_old = np.array(xy)

        # Perform an update
        xy = xy - eta * np.linalg.inv(hessf(xy[0], xy[1])) @ gradf(xy[0], xy[1])
        xylist.append(np.copy(xy))
        fxy.append(f(xy[0], xy[1]))

        # Return if the norm of the update is small enough
        if np.linalg.norm(xy - xy_old) < eps:
            return xy, xylist, fxy

def plot_traj(xylist, xymin):
    plt.figure()

    # Create a triangular domain
    xv = []
    yv = []
    delta = 0.025
    numpts = 100

    xv = np.linspace(delta, 1-delta, numpts)
    yv = np.linspace(delta, 1-delta, numpts)
    zv = np.ndarray((len(xv), len(yv)))

    for i in range(len(xv)):
        for j in range(len(yv)):
            if (yv[j] < 1 - delta - xv[i]):
                zv[i][j] = f(xv[i], yv[j])
            else:
                zv[i][j] = np.inf

    # Plot the function contours
    contours = plt.contour(xv, yv, zv, levels=[round(f(xymin[0], xymin[1]), 1), 3.5, 4, 4.5, 5])
    plt.clabel(contours)

    # Plot the trajectory
    plt.plot([x[0][0] for x in xylist], [x[1][0] for x in xylist], marker="x", linestyle="dotted")

    # Plot the domain limit
    plt.plot([0, 1], [1, 0], color="black", linestyle="dashed")

    # Set graph characteristics
    plt.axis([0, 1, 0, 1])
    plt.title("Convergence of $(x,y)$ to the minimum of $f$")

def plot_fxy(fxy):
    plt.figure()
    plt.plot(fxy)
    plt.title("Value of $f(x,y)$ for increasing epochs")
    plt.grid()

# Set algorithm parameters
eta = 0.01
eps = 1e-6
xymin = np.array([1/3, 1/3])

# Select random initial point
x0 = np.random.uniform(0, 1)
y0 = np.random.uniform(0, 1-x0)
print("Initial point: ({}, {})".format(x0, y0))
print("GD update term in (x0, y0):\n{}".format(gradf(x0, y0)))
print("NM update term in (x0, y0):\n{}".format(np.linalg.inv(hessf(x0, y0)) @ gradf(x0, y0)))

# Run gradient descent
xy_gd, xylist_gd, fxy_gd = gdesc([x0, y0], eta, eps)
print("Gradient descent iterations: {}".format(len(xylist_gd) - 1))
print("Gradient descent convergence point: {}".format(xy_gd.transpose()))

# Plot results
plot_traj(xylist_gd, xymin)
plot_fxy(fxy_gd)

# Run Newton's method
eta = 1
eps = 1e-6

xy_n, xylist_n, fxy_n = newton([x0, y0], eta, eps)
print("Newton's method iterations: {}".format(len(xylist_n) - 1))
print("Netwon's method convergence point: {}".format(xy_n.transpose()))

# Plot results
plot_traj(xylist_n, xymin)
plot_fxy(fxy_n)

# Show plots
plt.show()
```

2 Question 2

2.1 Linear least squares fit

The values for the parameters w_0 and w_1 of the linear least squares fit for the given data set can be obtained by minimizing the value of:

$$E = \sum_{i=1}^{50} (y_i - (w_0 + w_1 x_i))^2$$

Let us rewrite this expression by introducing the matrices:

$$X = \begin{bmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_{50} \end{bmatrix}, Y = \begin{bmatrix} y_1 & y_2 & \dots & y_{50} \end{bmatrix}$$

$$\Rightarrow E = \sum_{i=1}^{50} (y_i - (w_0 + w_1 x_i))^2 = ||Y - WX||^2$$

This corresponds to a simple linear optimization problem, which can be solved in a closed form using the pseudoinverse method:

$$W = YX^+$$

Where X^+ denotes the Moore-Penrose pseudoinverse of X. Given that this X has linearly independent rows, the pseudoinverse X^+ may be computed as:

$$X^+ = X^T\left(XX^T\right)^{-1} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_{50} \end{bmatrix} \left(\begin{bmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_{50} \end{bmatrix} \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_{50} \end{bmatrix} \right)^{-1}$$

$$= \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_{50} \end{bmatrix} \begin{bmatrix} 50 & \sum_{i=1}^{50} x_i \\ \sum_{i=1}^{50} x_i & \sum_{i=1}^{50} x_i^2 \end{bmatrix}^{-1} = \frac{1}{50\sum_{i=1}^{50} x_i^2 - \left(\sum_{i=1}^{50} x_i\right)^2} \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_{50} \end{bmatrix} \begin{bmatrix} \sum_{i=1}^{50} x_i^2 & -\sum_{i=1}^{50} x_i \\ -\sum_{i=1}^{50} x_i & 50 \end{bmatrix}$$

$$= \frac{1}{50\sum_{i=1}^{50} x_i^2 - \left(\sum_{i=1}^{50} x_i\right)^2} \begin{bmatrix} \sum_{i=1}^{50} x_i^2 - x_1\sum_{i=1}^{50} x_i & -\sum_{i=1}^{50} x_i + 50x_1 \\ \sum_{i=1}^{50} x_i^2 - x_2\sum_{i=1}^{50} x_i & -\sum_{i=1}^{50} x_i + 50x_2 \\ \vdots & \vdots \\ \sum_{i=1}^{50} x_i^2 - x_{50}\sum_{i=1}^{50} x_i & -\sum_{i=1}^{50} x_i + 50x_{50} \end{bmatrix}$$

Therefore, the analytic solution of this optimization problem is:

$$W = YX^+ = \frac{\begin{bmatrix} y_1 & y_2 & \dots & y_{50} \end{bmatrix}}{50\sum_{i=1}^{50} x_i^2 - \left(\sum_{i=1}^{50} x_i\right)^2} \begin{bmatrix} \sum_{i=1}^{50} x_i^2 - x_1\sum_{i=1}^{50} x_i & -\sum_{i=1}^{50} x_i + 50x_1 \\ \sum_{i=1}^{50} x_i^2 - x_2\sum_{i=1}^{50} x_i & -\sum_{i=1}^{50} x_i + 50x_2 \\ \vdots & \vdots \\ \sum_{i=1}^{50} x_i^2 - x_{50}\sum_{i=1}^{50} x_i & -\sum_{i=1}^{50} x_i + 50x_{50} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\sum_{i=1}^{50} x_i^2 \sum_{i=1}^{50} y_i - \sum_{i=1}^{50} x_i \sum_{i=1}^{50} x_i y_i}{50\sum_{i=1}^{50} x_i - \left(\sum_{i=1}^{50} x_i\right)^2} & \frac{-\sum_{i=1}^{50} x_i \sum_{i=1}^{50} y_i + 50\sum_{i=1}^{50} x_i y_i}{50\sum_{i=1}^{50} x_i - \left(\sum_{i=1}^{50} x_i\right)^2} \end{bmatrix}$$

Applying the presented formulas on a randomly generated dataset (the random seed is fixed for reproducibility), we obtain the following values for the linear interpolation coefficients:

$$w_0 = 0.15612$$

$$w_1 = 0.99762$$

Figure 5 shows the approximation line computed using the least squares method.

2.2 Gradient descent method

Let us recall that the gradient descent method is based on the weights update rule:

$$w \leftarrow w - \eta \nabla E$$

In our case, as obtained before:

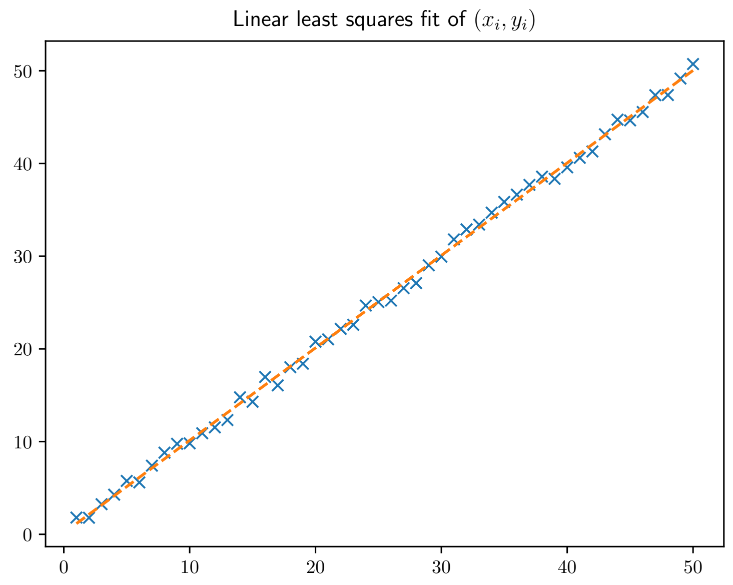

Figure 5: Linear least squares fit of points (x_i, y_i)

$$\nabla E = \begin{bmatrix} -2\sum_{i=1}^{50}(y_i - (w_0 + w_1 x_i)) & -2\sum_{i=1}^{50} x_i(y_i - (w_0 + w_1 x_i)) \end{bmatrix}^T$$

Using as the stopping condition of the algorithm a minimum requirement on the norm for the update term (set to $\epsilon = 10^{-6}$), the gradient descent method converges (using $\eta = 2.25 * 10^{-5}$ in order not to make the result diverge) after 10749 epochs to the following weights:

$$w_0 = 0.15429$$

$$w_1 = 0.99767$$

Those values are actually quite similar to the ones computed when solving the equations in a closed form, and they can possibly be even closer if we decreased furtherly the value of ϵ. In fact, we expect that it would be possible to approximate the regression coefficients as computed using the method of the linear least squares fit with arbitrary precision, if we can afford to make the algorithm run for a sufficient number of epochs.

The described solution may be interpreted as an "offline" learning algorithm, in which we consider the entire data set (due to the presence of the summations) before updating our weights. We may think of implementing also an online version of the algorithm, that considers one data point at a time and updates weights according to the rule:

$$w \leftarrow w - \eta(y_i - w_0 - w_1 x_i) \begin{bmatrix} 1 & x_i \end{bmatrix}$$

For all $(x_i, y_i), i = 1, \ldots, 50$. Using this methodology with the same parameters as before, the weights converge after 10858 epochs to the following values:

$$w_0 = 0.15728$$
$$w_1 = 0.99804$$

These weights are obviously different from the ones obtained using the standard gradient descent method, but similar considerations could be made on the precision of the approximation.

The proposed method to implement gradient descent, based on the function `batch_gdesc()`, allows to use a generic batch size b to perform weights update; for each epoch, the algorithm iterates $\frac{50}{b}$ times, computing the partial sums only on the elements in the i-th batch (from bi-th item to $b(i+1)$-th item) and using the obtained values for weights update.

2.3 Newton's method

Let us recall that the Newton's method is based on the weights update rule:

$$w \leftarrow w - \eta[HE]^{-1}\nabla E$$

The Hessian matrix of E can be computed as follows:

$$HE = \begin{bmatrix} \frac{\partial^2 E}{\partial w_0^2} & \frac{\partial^2 E}{\partial w_0 \partial w_1} \\ \frac{\partial^2 E}{\partial w_1 \partial w_0} & \frac{\partial^2 E}{\partial w_1^2} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial w_0}\frac{\partial E}{\partial w_0} & \frac{\partial}{\partial w_0}\frac{\partial E}{\partial w_1} \\ \frac{\partial}{\partial w_1}\frac{\partial E}{\partial w_0} & \frac{\partial}{\partial w_1}\frac{\partial E}{\partial w_1} \end{bmatrix} = \begin{bmatrix} 100 & 2\sum_{i=1}^{50} x_i \\ 2\sum_{i=1}^{50} x_i & 2\sum_{i=1}^{50} x_i^2 \end{bmatrix}$$

It can be verified that, after a single epoch using $\eta = 1$, this method makes the weights w converge to the global minimum of E:

$$w_0 = 0.15612$$
$$w_1 = 0.99762$$

This result was expected since function E is quadratic: this means that its local quadratic approximation corresponds, in any point, to the function itself. Therefore, since each iteration of the Newton's method moves the weights to the minimum of the function's quadratic approximation around w, a single iteration is needed to converge to the global minimum of E, independently on the initial selection of weights.

It is also possible to derive those results analytically by writing the expression of the weights after the update; in the following, w_0 and w_1 denote the initial weights (before the update), while w_0' and w_1' denote the weights after a single update using Newton's method:

$$\begin{bmatrix} w_0' \\ w_1' \end{bmatrix} = w - \eta[HE]^{-1}\nabla E =$$

$$\begin{bmatrix} w_0 \\ w_1 \end{bmatrix} - \frac{1}{200\sum_{i=1}^{50} x_i^2 - 4\left[\sum_{i=1}^{50} x_i\right]^2} \begin{bmatrix} 2\sum_{i=1}^{50} x_i^2 & -2\sum_{i=1}^{50} x_i \\ -2\sum_{i=1}^{50} x_i & 100 \end{bmatrix} \begin{bmatrix} -2\sum_{i=1}^{50}(y_i - (w_0 + w_1 x_i)) \\ -2\sum_{i=1}^{50} x_i(y_i - (w_0 + w_1 x_i)) \end{bmatrix}$$

$$
\begin{aligned}
w_0' = w_0 - &\frac{-4\sum_{i=1}^{50} x_i^2 \sum_{i=1}^{50}(y_i - w_0 - w_1 x_i) + 4\sum_{i=1}^{50} x_i \sum_{i=1}^{50} x_i(y_i - w_0 - w_1 x_i)}{200\sum_{i=1}^{50} x_i^2 - 4\left[\sum_{i=1}^{50} x_i\right]^2} \\
&= \frac{4\sum_{i=1}^{50} x_i^2 \sum_{i=1}^{50} y_i - 4\sum_{i=1}^{50} x_i \sum_{i=1}^{50} x_i y_i}{200\sum_{i=1}^{50} x_i^2 - 4\left[\sum_{i=1}^{50} x_i\right]^2} = \frac{\sum_{i=1}^{50} x_i^2 \sum_{i=1}^{50} y_i - \sum_{i=1}^{50} x_i \sum_{i=1}^{50} x_i y_i}{50\sum_{i=1}^{50} x_i^2 - \left[\sum_{i=1}^{50} x_i\right]^2}
\end{aligned}
$$

$$
\begin{aligned}
w_1' = w_1 - &\frac{4\sum_{i=1}^{50} x_i \sum_{i=1}^{50}(y_i - w_0 - w_1 x_i) - 200\sum_{i=1}^{50} x_i(y_i - w_0 - w_1 x_i)}{200\sum_{i=1}^{50} x_i^2 - 4\left[\sum_{i=1}^{50} x_i\right]^2} \\
&= \frac{-4\sum_{i=1}^{50} x_i \sum_{i=1}^{50} y_i + 200\sum_{i=1}^{50} x_i y_i}{200\sum_{i=1}^{50} x_i^2 - 4\left[\sum_{i=1}^{50} x_i\right]^2} = \frac{50\sum_{i=1}^{50} x_i y_i - \sum_{i=1}^{50} x_i \sum_{i=1}^{50} y_i}{50\sum_{i=1}^{50} x_i^2 - \left[\sum_{i=1}^{50} x_i\right]^2}
\end{aligned}
$$

As it can be seen, the expression of the weights after a single update using Newton's method does not depend on the initial value of w_0 and w_1 and, most importantly, is analogous to the one derived for solving the linear least squares problem.

2.4 Complete Python code

```
import numpy as np
import matplotlib.pyplot as plt

# Use TeX for text rendering
plt.rc('text', usetex=True)

# Set the random seed for reproducibility
np.random.seed(2019)

def gradf(xi, yi, w, start, end):
   return -2 * np.array([
      sum([yi[i] - w[0] - w[1] * xi[i] for i in range(start, end)]),
      sum([(yi[i] - w[0] - w[1] * xi[i]) * xi[i] for i in range(start, end)])
   ]).reshape(2, 1)

def hessf(xi, yi, w, start, end):
   return 2 * np.array([
      50,
      sum(xi[start:end]),
      sum(xi[start:end]),
      sum([xi[i] ** 2 for i in range(start, end)])
   ]).reshape(2, 2)

def batch_gdesc(w0, eta, eps, bsize):
   # Initialize parameters
   w_gd = np.array(w0)
   epochs = 0

   while True:
      # Increment epochs and save old weights
      epochs = epochs + 1
      w_old = np.array(w_gd)
```

```
        # Update summing according to the batch size
        for i in range(50 // bsize):
            w_gd = w_gd - eta * gradf(xi, yi, w_gd, i*bsize, (i+1)*bsize)

        # Return if the norm of the update is small enough
        if np.linalg.norm(w_gd - w_old) < eps:
            return w_gd, epochs

        # Print message every 1000 epochs
        if epochs % 1000 == 0:
            print("Epoch {}: w = {}".format(epochs, w_gd.transpose()))

# Create vectors
xi = [i + 1 for i in range(50)]
yi = [x + np.random.uniform(-1, 1) for x in xi]

# Compute linear least squares fit
X = np.array([1] * 50 + xi).reshape(2, 50)
Y = np.array(yi).reshape(1, 50)
w_ls = (Y @ np.linalg.pinv(X)).transpose()
print("LS fit: m = {}, q = {}".format(w_ls[1][0], w_ls[0][0]))

# Compute weights using gradient descent
w0 = np.array([np.random.uniform(-1, 1), np.random.uniform(-1, 1)]).reshape(2, 1)
eta = 2.25e-5
eps = 1e-6
bsize = 1

w_gd, epochs = batch_gdesc(w0, eta, eps, bsize)
print("GD fit: m = {}, q = {} (epochs: {})".format(w_gd[1][0], w_gd[0][0], epochs))

# Compute weights after one iteration of Newton's method
w_nm = w0 - np.linalg.inv(hessf(xi, yi, w0, 0, 50)) @ gradf(xi, yi, w0, 0, 50)
print("NM fit: m = {}, q = {}".format(w_nm[1][0], w_nm[0][0]))

# Plot the LLS fit
plt.plot(xi, yi, marker="x", linestyle="none")
plt.plot(xi, [w_ls[1][0]*x + w_ls[0][0] for x in xi], linestyle="dashed")
plt.title("Linear least squares fit of $(x_i, y_i)$")
plt.show()
```