One of the key components of machine learning is the ability to model experimental or real world data. We will focus on a simple setup. Assume we are given a set of n data points $\{(x_i, y_i) : x_i, y_i \in \mathbb{N}, \quad x_1 < x_2 < \cdots < x_n, \quad 1 \leq i \leq n\}$ and that we think of these data points of samples of some 'ground truth' model, i.e., we imagine there exists some function $F : \mathbb{R} \to \mathbb{R}$ such that $F(x_i) = y_i$.

The goal is to try to estimate this 'ground truth' based on the samples we are provided with. It is important to point out that if we are talking about real world measurements, the values y_i might not be exact, similar to how reading a length measurement off a ruler might be slightly inaccurate based on how we position our eyes relative to the ruler. This means that we are not necessarily interested in finding the best function that interpolates the exact pairs (x_i, y_i), but something that encapsulates the overall pattern of the data points and is robust to 'measurement error'.

On a more formal level, consider the fact that we could exactly interpolate the pairs (x_i, y_i) either with a piece-wise linear map or with splines or some other piece-wise regular function. This will obviously match the measurements exactly, but it will not 'generalize' well, in the sense that outside of the interval $[x_1, x_n]$, we would have very low confidence about the predictive power of our interpolating function. In machine learning lingo, this is because we 'overfit' the training samples instead of 'extracting the relevant information/model' from them.

Concretely, let us consider the following arrays:

```
x = np.array([0, 1, 2, 3])
y = np.array([-1, 0.2, 0.9, 2.1])
```

We can plot them to see how they would look in a plane. For this, we will import the standard plotting package in Python to the preamble of our code, similar to how we import **numpy**.

```
import matplotlib.pyplot as plt
```

Now we can plot the 4 points in the plane as blue marbles by executing

```
plt.plot(x, y, 'o', label='Original data', markersize=10)
plt.legend()
plt.show()
```

To explain the syntax, we used the *plot* function and we called it on the arrays x and y (in order). The other inputs are optional and serve as plot formatting instructions. The *'o'* command means that the pairs (x_i, y_i) will be presented as marbles of size 10 (the *markersize=10* command) in the default color (*blue*) and the *label* instruction will be used for the legend of the figure. The command 'plt.legend()' constructs the legend and the 'plt.show()' command displays the figure. Note that while the figure is being displayed, Python is still running. Manually closing the figure will allow Python to continue through the code after 'plt.show()', in this case terminating.

Inspecting the figure, we can see that the points are 'almost' colinear. So it makes sense to imagine that the 'ground truth' is a linear function. The

question now becomes how to estimate the 'best' linear fit to the given data. In reality, the answer to this question is not unique and depends on what we mean by 'best'.

The most standard interpretation of 'best' in this context can be thought of as follows. Consider an arbitrary line in the plane. Such a line can be uniquely identified by two parameters, its slope m and its offset or y-intercept c. Using these, we can think of the line as the function $f_{m,c} : \mathbb{R} \to \mathbb{R}, \quad x \mapsto m \cdot x + c$.

With this in mind, we can think of the point-wise error in using $f_{m,c}$ instead of the 'ground truth' F as the quantity $y_i - \hat{y_i} := y_i - (mx_i + c)$. The most natural interpretation of 'best fit' in this context is to imagine the two vectors $(y_1, y_2, \ldots, y_n)$ and $(\hat{y_1}, \hat{y_2}, \ldots, \hat{y_n})$ as points in the n-dimensional space $\mathbb{R}^n$ and to minimize the Euclidean distance between them. This is called the least-squares-error (LSE) solution to the problem. The fact that this is in some sense the most natural interpretation of 'best fit' in this context will become apparent later in Linear Algebra when you discuss Moore-Penrose pseudoinverses of matrices.

So now that we have a notion of 'best fit', we can solve the problem. In other words we can find m and c that minimize the squared point-wise error, in other words, that minimize the function $\phi : \mathbb{R}^2 \to \mathbb{R}$ given by $\phi(m, c) = \sum_{i=1}^{n}(y_i - (mx_i + c))^2$.

This is essentially a degree 2 equation in either m or c, so one could use school techniques to figure out the roots and then find the average of the roots to determine the minimizer. We will show a method that comes from Analysis. If a continuously differentiable function is minimized inside its domain of definition, then the gradient at the minimum has to be 0. In other words, if we differentiate the function ϕ with respect to m (thinking of c as a constant), we want to find the solution to the equation $\frac{\partial \phi}{\partial m}(m, c) = 0$ and similarly for swapping the roles of m and c, we want to find the solution to $\frac{\partial \phi}{\partial c}(m, c) = 0$. Applying definitions, this translates to the linear system of equations

$$0 = -2\sum_{i=1}^{n} x_i(y_i - (mx_i + c))$$
$$0 = -2\sum_{i=1}^{n}(y_i - (mx_i + c)).$$

Simplifying, this can be rewritten as the 2-equation 2-unknown linear system

$$\sum_{i=1}^{n} x_i y_i - m\sum_{i=1}^{n} x_i^2 - c\sum_{i=1}^{n} x_i = 0$$
$$nc + m\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} y_i = 0.$$

We can now find the solutions via, for example, substitution.

$$m = \frac{n\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)\cdot(\sum_{i=1}^{n} y_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n})^2}$$
$$c = \frac{\sum_{i=1}^{n} y_i - m\sum_{i=1}^{n} x_i}{n}$$

The line $f_{m,c}$ with the slope and offset computed as above will be the line that minimizes the squared point-wise error.

We can also implement this computation in Python, for example as a function that takes as inputs the x and y arrays and outputs the best fit slope and offset.

```
def get_slope_and_offset_ls(x, y):
    number_points = len(x)
    if len(y) != number_points:
        raise Exception('bad inputs')
    sum_x = sum(x)
    sum_xsq = sum(x**2)
    sum_y = sum(y)
    sum_xy = sum(x*y)
    slope = (number_points*sum_xy-sum_x*sum_y)/(number_points*sum_xsq-sum_x**2)
    offset = (sum_y-slope*sum_x)/number_points
    return slope, offset
```

With this, we can expand our previous plotting to visually depict our computed line.

```
m, c = get_slope_and_offset_ls(x, y)
plt.plot(x, y, 'o', label='Original data', markersize=10)
plt.plot(x, m*x + c, 'r', label='Fitted line')
plt.legend()
plt.show()
```

Note that for the y values we used the functional equation $x \mapsto mx + c$ applied to the x values and we used *'r'* to tell Python that the fitted line should be plotted as a simple line in red. Also, note that we can stack multiple 'plt.plot' commands before calling 'plt.show()'. This will have the effect of overlapping the different plotting instructions into a single figure. As we can see, the computed line is very 'close' to the data points.

There are some built in methods in Python for doing the least squares error fit. One of them is specifically for linear models and uses some linear algebra trick to format the input vector x that will more sense a bit later in the Linear Algebra course.

```
A = np.vstack([x, np.ones(len(x))]).T #transforms the row vector x into
# a column vector followed by a column of 1s
m, c = np.linalg.lstsq(A, y, rcond=None)[0]
```

This uses a linear algebra numerical method (based on the pseudoinverse) to calculate the slope and offset. Note that the values are not exactly identical, but very close.

Another far more general option comes from the **scipy** package. This will fit a prescribed class of curves to the data points by minimizing the squared pointwise error. One needs to prescribe the so called 'hypothesis' class of curves that the numerical solver will optimize within. In this case we can define a general linear function by using 'lambda' definitions or directly by

```
def linear_function(x, a, b):
    return a * x + b
```

Now we call call the *curve fit* method from the **scipy** package.

```
from scipy.optimize import curve_fit
m, c = curve_fit(linear_function, x, y)[0]
```

Consider now a different data point set.

```
x = np.array([3, 4, 5, 6, 7, 8, 9, 10, 11,
              12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22])
y = np.array([3, 4, 5, 7, 9, 13, 15, 19, 23,
              24, 29, 38, 40, 50, 56, 59, 70, 89, 104, 130])
plt.scatter(x, y)
```

We can visualize the data by calling the simpler command 'plt.scatter' which will have a very similar effect to the blue marble picture we had of the first data set.

Note that these data points do not seem to come from a line, so choosing linear functions as the 'hypothesis class' in which to look for something close to the 'ground truth' makes little sense. However, the points seem to be arrayed on something resembling a parabola or, more generally, a power law $f_{exp,ct} = ct \cdot x^{exp}$.

In this case, we can do a simple trick before employing our known linear regression method. Indeed, let us assume that the data points (x_i, y_i) are sampled from a power law, i.e., that $y_i = ct \cdot x_i^{exp}$. Taking the natural logarithm on both sides implies $\log y_i = \log ct + exp \cdot \log x_i$. This means that looking at the 'log-log' plot of the data, we are back to performing linear regression to determine exp as the slop of the log-log data and $\log ct$ as the offset. Once we obtain these, we can immediately find the 'best' interpolator within our 'hypothesis class' as $x \mapsto e^{\log ct} \cdot x^{exp}$.

```
xlog = np.log(x)
ylog = np.log(y)
plt.scatter(xlog, ylog, color='purple')
plt.xlabel('Log(x)')
plt.ylabel('Log(y)')
plt.title('Log-Log Plot')
plt.show()
```

Now we can recover our 'best' interpolating function using or custom slope and offset finder method or any of the other options.

```
exponent, constant = get_slope_and_offset_ls(xlog, ylog)
print(exponent, constant)
plt.plot(x, y, 'o')
plt.plot(x, x**exponent*np.exp(constant), 'r')
plt.show()
```

As we can see, this is not a perfect interpolation (especially for the larger x values), but still quite 'close'.

Finally, let us illustrate another numerical technique, namely that of approximating roots of a (continuous) function. We don't need to go into Analysis details here, but we are interested in functions that have the *intermediate value property*. Say we have a real valued function f and some real numbers a and b. If for every height h between $f(a)$ and $f(b)$ there exists some intermediate input $x \in [a, b]$ such that $f(x) = h$, then f is said to have the *intermediate value property*. Note that if f is continuous, i.e., if we can draw its graph without raising our pencil off the paper, then f will have this property.

So let us now consider such a function f and assume that we have some real numbers $a < b$ such that the signs of $f(a)$ and $f(b)$ are different (e.g. $f(b) < 0 < f(a)$). Then, by the *intermediate value property*, we know there must be some $c \in [a, b]$ such that $f(c) = 0$. In other words, we know there is a root of f in the interval $[a, b]$. The simplest technique to approximate this root is called the *bisection method*. The idea behind it is simple. Take the midpoint $m = \frac{a+b}{2}$. Then either $f(m) = 0$ (and then we explicitly found the root), or $f(m)$ will have the same sign as either $f(a)$ or $f(b)$. If the sign is shared with $f(a)$, we can bisect again, but using the half-interval $[m, b]$ instead of $[a, b]$. Otherwise we bisect again using the half-interval $[a, m]$. At each step the size of the interval containing the root will be halved, so after n steps, we will have found the root with precision at worse $\frac{b-a}{2^n}$.

Here is a simple recursive implementation of the *bisection method* in Python.

```
def bisect_for_root(f, a, b, tol):
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception('cannot bisect, endpoints have same sign')
    m = (a+b)/2
    if np.abs(f(m)) < tol:
        return m
    elif np.sign(f(m)) == np.sign(f(a)):
        return bisect_for_root(f, m, b, tol)
    elif np.sign(f(m)) == np.sign(f(b)):
        return bisect_for_root(f, a, m, tol)
```

Note that since we cannot always expect to find the exact root after a finite number of steps, we have to instead specify a tolerance margin ('tol'). The output will be a number which is at most 'tol' away from the real root.

We can use this method to approximate the square root of 2. Consider the 'function handle' defined via 'lambda' notation:

```
f = lambda x: x**2 -2
```

This simplify defines the polynomial $x \mapsto x^2 - 2$, the positive root of which is clearly $\sqrt{2}$.

Calling

```
r = bisect_for_root(f, 0, 2, 10**(-12))
print(r)
```

will produce an approximation of $\sqrt{2}$ accurate to the first 12 decimal places by iteratively bisecting the interval $[0, 2]$. We can compare this to the 'actual' (the native approximation of Python) that is produced by

```
print(np.sqrt(2))
```

Finally, there is a very general solver in the **scipy** package that can solve for roots of a function or solutions to systems of equations using a variety of well-optimized standard algorithms. This is the 'fsolve()' method. The way this method operates is syntactically different from our *bisection method*. It does not look for a root in a given interval, but around an initial guess. Note that the closer the initial guess is to the initial value, the faster and the more accurate the output of *fsolve* will be. We will use it to estimate $\sqrt{2}$ with an initial guess of 1 for the root of $x \mapsto x^2 - 2$.

```
from scipy.optimize import fsolve
r = fsolve(f, 1)[0]
print(r)
```