FIT1045 Algorithms and programming in Python, S2-2020
Programming Assignment

Assessment value: 22% (10% for Part 1 + 12% for Part 2)

Due: Week 6 (Part 1), Week 11 (Part 2)

Objectives

The **objectives of this assignment** are:

- To demonstrate the ability to implement algorithms using basic data structures and operations on them.
- To gain experience in designing an algorithm for a given problem description and implementing that algorithm in Python.
- To explain the computational problems and the challenges they pose as well as your chosen strategies and programming techniques to address them.

Submission Procedure

You are going to create a single Python module file `regression.py` based on the provided template. Submit a first version of this file at the due date of Part 1, Friday midnight of Week 6. Submit a second version of this file at the due date of Part 2, Friday midnight of Week 11.

For each of the two versions:

1. Save the current version of your `regression.py` into a zip file called `yourStudentID_yourFirstName_yourLastName.zip`.
2. Submit this zip file to Moodle.
3. Your assignment will not be accepted unless it is a readable zip file containing a file called `products.py`.

Important Note: Please ensure that you have read and understood the university's policies on plagiarism and collusion available at `http://www.monash.edu.au/students/policies/academic-integrity.html`. You will be required to agree to these policies when you submit your assignment.
A common mistake students make is to use Google to find solutions to the questions. Once you have seen a solution it is often difficult to come up with your own version. The best way to avoid making this mistake is to avoid using Google. You have been given all the tools you need in workshops. If you find you are stuck, feel free to ask for assistance on Moodle, ensuring that you do not post your code.

Marks and Module Documentation

This assignment will be marked both by the correctness of your module and your analysis of it, which has to be provided as part of your function documentations. Each of the assessed function of your module must contain a docstring that contains in this order:

1. A one line short summary of what the function is computing
2. A formal input/output specification of the function
3. Some usage examples of the function (in doctest style); feel free to augment and alter the examples for the assessed functions that are already provided in the template if you feel they are incomplete or otherwise insufficient (provide some explanations in this case)

4. A paragraph explaining the challenges of the problem that the function solves and your overall approach to address these challenges
5. A paragraph explaining the specific choices of programming techniques that you have used
6. [only for Part 2] A paragraph analysing the computational complexity of the function; this does *not* imply that a specific computational complexity such as $O(n \log n)$ is required just that the given complexity is understood and analysed

For example, if `scaled(row, alpha)` from Lecture 6 was an assessed function (in Part 2 and you are a FIT1045 student) its documentation in the submitted module could look as follows:

```
def scaled(row, alpha):
    """ Provides a scaled version of a list with numeric elements.

    Input : list with numeric entries (row), scaling factor (alpha)
    Output: new list (res) of same length with res[i]==row[i]*alpha

    For example:
    >>> scaled([1, 4, -1], 2.5)
    [2.5, 10.0, -2.5]
    >>> scaled([], -23)
    []

    This is a list processing problem where an arithmetic operation has to be carried out
    for each  element of an input list of unknown size.  This means that a loop is required
    to iterate over all elements and compute their scaled version. Importantly, according
    the specification, the function has to provide a new list instead of mutating the input
    list. Thus, the scaled elements have to be accumulated in a new list.

    In my implementation, I chose to iterate over the input list with a for-loop and to
    accumulate the computed scaled entries with an augmented assignment statement
    (+=) in order to avoid the re-creation of each intermediate list. This solution allows a
    concise and problem-related code with no need to explicitly handle list indices.

    The computational complexity of the function is O(n) for an input list of length n. This
    is because the for loop is executed n times, and in each iteration there is constant
    time operation of computing the scaled element and a constant time operation of
    appending it to the result list (due to the usage of the augmented assignment
    statement instead of a regular assignment statement with list concatenation).
    """
    res = []
    for x in row:
        res += [alpha*x]
    return res
```

Beyond the assessed functions, you are highly encouraged to add additional functions to the module that you use as part of your implementation of the assessed functions. In fact, such a decomposition is essential for a readable implementation of the assessed function, which in turn forms the basis of a coherent and readable analysis in the documentation. All these additional functions also must contain a minimal docstring with the items 1-3 from the list above. Additionally, you can also use their docstrings to provide additional information, which you can refer to from the analysis paragraphs of the assessed functions.

This assignment has a total of 22 marks and contributes to 22% of your final mark. For each day an assignment is late, the maximum achievable mark is reduced by 10% of the total. For example, if the assignment is late by 3 days (including weekends), the highest achievable mark is 70% of 15, which is 10.5. Assignments submitted 7 days after the due date will normally not be accepted.

Mark breakdowns for each of the assessed functions can be found in parentheses after each task section below. Marks are subtracted for inappropriate or incomplete discussion of the function in their docstring. **Again, readable code with appropriate decomposition and variable names is the basis of a suitable analysis in the documentation.**

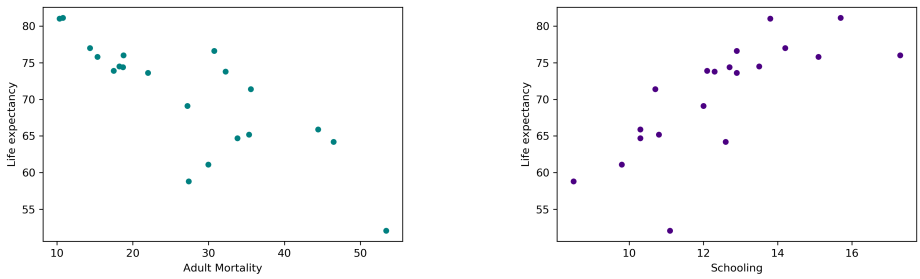
Overview

In this assignment we create a Python module to perform some basic data science tasks. While the instructions contain some mathematics, the main focus is on implementing the corresponding algorithms and finding a good decomposition into subproblems and functions that solve these subproblems. In particular, we want to implement a method that is able to estimate relationships in observational data and make predictions based on these relationships. For example, consider a dataset that measures the average life expectancy for various countries together with other factors that potentially have an influence on the life expectancy, such as the average body mass index (BMI), the level of schooling in that country, or the gross domestic product (GDP) (see Table 1 for an example). Can we predict the life expectancy of Rwanda? And can we predict how the life expectancy of Liberia will change if it improved its schooling to the level of Austria?

Country	Adult Mortality	Infant Deaths	BMI	GDP	Schooling	Life expectancy
Austria	10.77	4.32	25.4	4.55e+11	15.7	81.1
Georgia	18.23	15.17	27.1	1.76e+10	13.5	74.5
Liberia	29.95	74.52	24.0	3.264e+09	9.8	61.1
Mexico	30.73	17.29	28.1	1.22e+12	12.9	76.6
Rwanda	35.31	64.04	22.1	9.509e+09	10.8	?

Table 1: Example dataset on Life Expectancy.

To answer these questions, we have to find the relationships between the target variable—in case of this example the life expectancy—and the explanatory variables—in this example "Adult Mortality", "Infant Deaths", etc. (see Figure 1). For this we want to develop a method that finds linear relationships between the explanatory variables and the target variable and use it to understand those relationship and make predictions on the target variable. This method is called a *linear regression*. The main idea of linear regression is to use data to infer a prediction function that 'explains' a target variable through linear effects of one or more explanatory variables. Finding the relationship between one particular explanatory variable and the target variable (e.g., the relationship between schooling and life expectancy) is called a *univariate regression*. Finding the joint relationship of all explanatory variables with the target variable is called a *multivariate regression*.



(a) Adult mortality vs. life expectancy.

(b) Schooling vs. life expectancy.

Figure 1: Visualization of the relationship between explanatory variables and the target variable.

In this assignment, you will develop functions that perform univariate regression (Part I) and multivariate regression (Part 2) and use them to answer questions on a dataset on life expectancy, similar to the example above.

To help you to visually check and understand your implementation, a module for plotting data and linear prediction functions is provided.

Part 1: Univariate Regression (10%, due in Week 6)

The first task is to model a linear relationship between one explanatory variable and the target variable.

The data in this assignment is always represented as a table containing m rows and n columns and a `list` of length m containing the corresponding target variables. The table is represented as a `list` with m elements, each being a `list` of n values, one for each explanatory variable.

An example dataset with one explanatory variable x and a target variable y would be

```
>>> x = [1,2,3]
>>> y = [1,4,6]
```

and an example dataset with two explanatory variables $x^{(1)}, x^{(2)}$ would be

```
>>> data = [[1,1],[2,1],[1,3]]
>>> y = [3,4,7]
```

Task A: Optimal Slope (2 Marks)

Let us now start by modelling the linear relationship between one explanatory variable x and the target variable y based on the data of m observations $(x_1, y_1), \ldots, (x_m, y_m)$. For that, let's start out simple by modelling the relation of x and y as

$$y = ax \ , \tag{1}$$

i.e., a straight line through the origin. For this model we want to find an optimal **slope** parameter a that fits the given data as good as possible. A central concept to solve this problem is the **residual vector** defined as

$$r = \begin{pmatrix} y_1 - ax_1 \\ y_2 - ax_2 \\ \cdots \\ y_m - ax_m \end{pmatrix} \ ,$$

i.e., the m-component vector that contains for each data point the difference of the target value and the corresponding predicted value. Intuitively, for a good fit, all components of the residual vector should have a small magnitude (being all 0 for a perfect fit). A popular approach is to determine the optimal parameter value a as the one that minimises the sum of squared components of the residual vector, i.e., the quantity

$$r \cdot r = r_1^2 + r_2^2 + \cdots + r_m^2 \ ,$$

which we also refer to as the **sum of squared residuals**. With some math (that is outside the scope of this unit) we can show that for the slope parameter a that minimises the sum of squared residuals it holds that the explanatory data is orthogonal to the residual vector, i.e.,

$$x \cdot r = 0 \ . \tag{2}$$

Here, $\cdot$ refers to the **dot product** of two vectors, which in this case is

$$x \cdot r = x_1 r_1 + x_2 r_2 + \cdots + x_m r_m \ .$$

Pluging in the definition of r into Equation 2 yields

$$(x \cdot x)\, a = x \cdot y \tag{3}$$

from which we can easily find the optimal slope a.

Instructions

Based on Equation 3, write a function `slope(x, y)` that computes the optimal slope for the simple regression model in Equation 1.

Input: Two lists of numbers (`x` and `y`) of equal length representing data of an explanatory and a target variable.

Output: The optimal least squares slope (`a`) with respect to the given data.

For example:

```
def slope(x, y):
"""
Computes the slope of the least squares regression line
(without intercept) for explaining y through x.
>>> slope([0, 1, 2], [0, 2, 4])
2.0
>>> slope([0, 2, 4], [0, 1, 2])
0.5
>>> slope([0, 1, 2], [1, 1, 2])
1.0
>>> slope([0, 1, 2], [1, 1.2, 2])
1.04
"""
```

If you want to visualize the data and predictor, you can use the plotting functions provided in `plotting.py`. For example

```
>>> X = [0, 1, 2]
>>> Y = [1, 1, 2]
>>> a = slope(X, Y)
>>> b = 0
>>> linear(a,b, x_min=0, x_max=2)
>>> plot2DData(X, Y)
>>> plt.show()
```

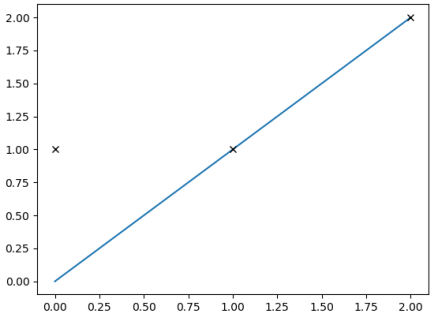


Figure 2: Example plot generated with the functions provided in `plotting.py`.

will produce the plot given in Figure 2.

Task B: Optimal Slope and Intercept (3 Marks)

Consider the example regression problem in Figure 3. Even with an optimal choice of a the regression model $y = ax$ of Equation 1 does not fit the data well. In particular the point $(0, 1)$ is problematic because the model given in Equation 1 is forced to run through the origin and thus has no degree of freedom to fit a data point with $x = 0$. To improve this, we have to consider regression lines that are not forced to run through the origin by extending the model equation to

$$y = ax + b \tag{4}$$

where in addition to the slope a we also have an additive term b called the **intercept**. Now we have two parameters a and b to optimise, but it turns out that a simple modification of Equation 3 lets us solve this more complicated problem.

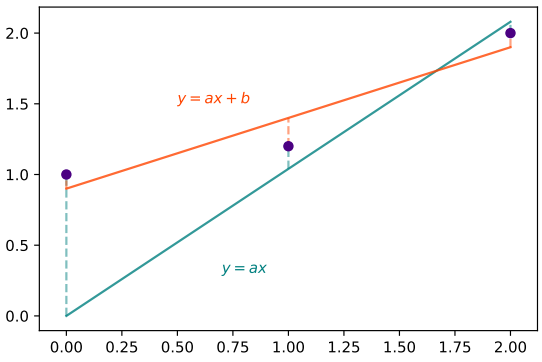


Figure 3: Linear regression with and without an intercept for three points $(0, 1), (1, 1.2)$, and $(2, 2)$.

Instead of the residual being orthogonal to the explanatory data as is, we now require orthogonality to the *centred* version of the data. That is, $\overline{x} \cdot r = 0$ where $\overline{x}$ denotes the centred data vector defined as

$$\overline{x} = \begin{pmatrix} x_1 - \mu \\ x_2 - \mu \\ \cdots \\ x_m - \mu \end{pmatrix}$$

with $\mu = (x_1 + x_2 + \cdots + x_m)/m$ being the mean value of the explanatory data. Again, we can rewrite the

above condition as a simple linear equation involving two dot-products

$$(\overline{x} \cdot \overline{x})\, a = \overline{x} \cdot y \tag{5}$$

which again allows to easily determine the slope a. However, now we also need to find the optimal intercept b corresponding to this value of a. For this we can simply find the average target value minus the average fitted value when only using the slope (or in other words the average residual):

$$b = ((y_1 - ax_1) + \cdots + (y_m - ax_m))/m \ . \tag{6}$$

Instructions

Using Equations (5) and (6) above, write a function `line(x, y)` that computes the optimal slope and intercept for the regression model in Equation 4

Input: Two lists of numbers (`x` and `y`) of equal length representing data of an explanatory and a target variable.

Output: A tuple `(a,b)` where `a` is the optimal least squares slope and `b` the optimal intercept with respect to the given data.

For example

```
>>> line([0, 1, 2], [1, 1, 2])
(0.5, 0.8333333333333333)
```

Task C: Best Single Variable Predictor (4 Marks)

We are now able to determine a regression model that represents the linear relationship between a target variable and a single explanatory variable. However, in usual settings like the one given in the introduction, we observe not one but many explanatory variables (e.g., in the example 'GDP', 'Schooling', etc.). As an abstract description of such a setting we consider n variables $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ such that for each j with $0 \leq j < n$ we have measured m observations $x_1^{(j)}, \ldots, x_m^{(j)}$. These conceptually correspond to the columns of a given data table. The individual rows of our data table then become n-dimensional data points that we again denote by $x_1, \ldots x_m$, but now x_i represent not a single number but a vector

$$x_i = (x_i^{(1)}, \ldots, x_i^{(n)}) \ .$$

A general, i.e., multi-dimensional, linear predictor is then given by an n-dimensional weight vector a and an intercept b that together describe the target variable as:

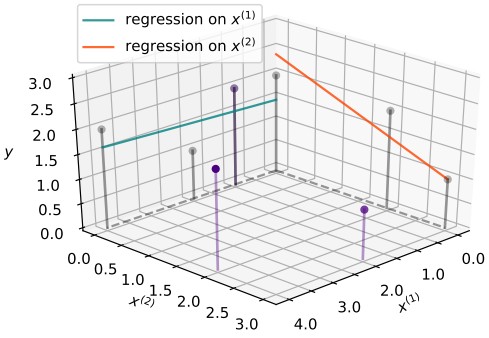


Figure 4: Example with two explanatory variables $x^{(1)}$, $x^{(2)}$ and target variable y with $m = 3$ observations given in Table 2. We can see that the model $y = -0,29x^{(2)} + 2.14$ is more accurate than the model $y = 0.07x^{(1)} + 1.5$.

$$y = (a \cdot x) + b \ , \tag{7}$$

i.e., we generalise Equation 4 by turning the slope a into a n-component linear **weight vector** and replace simple multiplication by the dot product (the intercept b is still a single number). Part 2 of the assignment will be about finding such general linear predictors. In this task, however, we will start out simply by finding the best univariate predictor and then represent it using a multivariate weight-vector a. Thus, we need to answer two questions:

1. How to find the best univariate predictor (i.e., the one with the smallest sum of squared residuals)?
2. How to represent it as a multivariate predictor (i.e., in the form of Equation (7))?

$x^{(1)}$	$x^{(2)}$	y
1	0	2
2	3	1
4	2	2

Table 2: Example dataset with $n = 2$ explanatory variables $x^{(1)}$ and $x^{(2)}$, and a target variable y with $m = 3$ observations.

Consider the simple example from Figure 4 with two explanatory variables (the data is shown in Table 2). Here, the univariate model

$$y = -0.29x^{(2)} + 2.14 \ ,$$

i.e., $a^{(2)} = -0.29$, $b^{(2)} = 2.14$, is better than

$$y = 0.07x^{(1)} + 1.5 \ ,$$

i.e., $a^{(1)} = 0.07$, $b^{(1)} = 1.5$. Hence, our multivariate model in the form of Equation (7) is

$$y = ((0, -0.29) \cdot x) + 2.14 \ .$$

Of course, for a general dataset with n explanatory variables $x^{(1)}, \ldots, x^{(n)}$ and a target variable y we have to compare n different univariate predictors and embed the best one into an n-dimensional weight vector.

Instructions

Based on the functions for fitting univariate regression lines, write a function `best_single_predictor(data, y)` that computes a general linear predictor (i.e., one that is defined for multi-dimensional data points) by picking the best univariate prediction model from all available explanatory variables.

Input: Table `data` with `m > 0` rows and `n > 0` columns representing data of `n` explanatory variables and a list `y` of length `m` containing the values of the target variable corresponding to the rows in the table.

Output: A pair `(a, b)` of a list `a` of length `n` and a number `b` representing a linear predictor with weights `a` and intercept `b` with the following properties:

a) There is only one non-zero element of `a`, i.e., there is one `i in range(n)` such that `a[j]==0` for all indices `j!=i`.

b) The predictor represented by `(a, b)` has the smallest possible squared error among all predictors that satisfy property a).

For example (see Figure 4):

```
>>> data = [[1, 0],
...         [2, 3],
...         [4, 2]]
>>> y = [2, 1, 2]
>>> weights, b = best_single_predictor(data, y)
>>> weights, b
([0.0, -0.2857142857142858], 2.1428571428571432)
```

Task D: Regression Analysis (1 Marks)

You have now developed the tools to carry out a regression analysis. In this task, you will perform a regression analysis on the life-expectancy dataset an excerpt of which was used as an example in the overview. The dataset provided in the file `/data/life_expectancy.csv`.

Instructions

Find the best single predictor for the life expectancy dataset and use this predictor to answer the following two questions:

1. What life expectancy does your model predict for Rwanda?
2. What life expectancy does your model predict for Liberia, if Liberia improved its "schooling" to the level of Austria?

You can perform the analysis in a function `regression_analysis` or dynamically in the console. Provide the results of your analysis in the documentation.

Part 2: Multivariate Regression (12%, due in Week 11)

In part 1 we have developed a method to find a univariate linear regression model (i.e., one that models the relationship between a single explanatory variable and the target variable), as well as a method that picks the best univariate regression model when multiple explanatory variables are available. In this part, we develop a multivariate regression method that models the joint linear relationship between all explanatory variables and the target variable.

Task A: Greedy Residual Fitting (6 Marks)

We start using a greedy approach to multivariate regression. Assume a dataset with m data points $x_1, \ldots, x_m$ where each data point x has n explanatory variables $x^{(1)}, \ldots, x^{(n)}$, and corresponding target variables $y_1, \ldots, y_m$. The goal is to find the slopes for all explanatory variables that help predicting the target variable. The strategy we use greedily picks the best predictor and adds its slope to the list of used predictors. When all slopes are computed, it finds the best intercept.

For that, recall that a greedy algorithm iteratively extends a partial solution by a small augmentation that optimises some selection criterion. In our setting, those augmentation options are the inclusion of a currently unused explanatory variable (i.e., one that currently still has a zero coefficient). As selection criterion, it makes sense to look at how much a previously unused explanatory variable can improve the data fit of the current predictor. For that, it should be useful to look at the current residual vector

$$r = \begin{pmatrix} y_1 - a \cdot x_1 \\ y_2 - a \cdot x_2 \\ \ldots \\ y_m - a \cdot x_m \end{pmatrix} ,$$

because it specifies the part of the target variable that is still not well explained. Note that a the slope of a predictor that predicts this residual well is a good option for augmenting the current solution. Also, recall that an augmentation is used only if it improves the selection criterion. In this case, a reasonable selection criterion is again the sum of squared residuals.

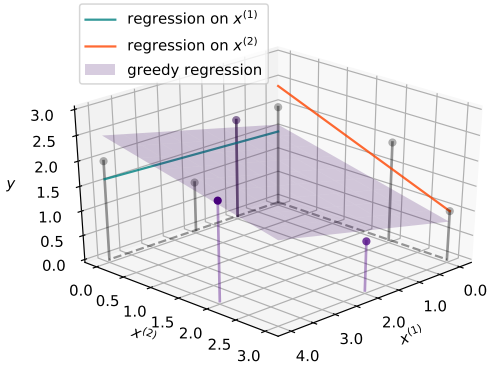


Figure 5: Continuation of the example given in Figure 4. The plane depicts the multivariate regression obtain from greedy residual fitting. The error of this model is lower than the individual regressions on $x^{(1)}$ and $x^{(2)}$

What is left to do is compute the intercept for the multivariate predictor. This can be done similar to Equation 6 as

$$b = \left((y_1 - a \cdot x_1) + \cdots + (y_m - a \cdot x_m) \right) / m \ .$$

The resulting multivariate predictor can then be written as in Equation 7. An example multivariate regression is shown in Figure 5.

Instructions

Write a function `greedy_predictor` that computes a multivariate predictor using the greedy strategy similar to the one described above.

Input: A data table `data` of explanatory variables with `m` rows and `n` columns and a list of corresponding target variables `y`.

Output: A tuple `(a,b)` where `a` is the weight vector and `b` the intercept obtained by the greedy strategy.

For example (the first example is visualized in Figure 5)

```
>>> data = [[1, 0],
...         [2, 3],
...         [4, 2]]
>>> y = [2, 1, 2]
>>> weights, intercept = greedy_predictor(data, y)
>>> weights, intercept
```

```
[0.21428571428571436, -0.2857142857142858] 1.642857142857143

>>> data = [[0, 0],
...         [1, 0],
...         [0, -1]]
>>> y = [1, 0, 0]
>>> weights, intercept = greedy_predictor(data, y)
>>> weights, intercept
[-0.49999999999999994, 0.7499999999999999] 0.7499999999999999
```

Task B: Optimal Least Squares Regression (6 Marks)

Recall that the central idea for finding the slope of the optimal univariate regression line (with intercept) was Equation 2 that stated that the residual vector has to be orthogonal to the values of the centred explanatory variable. For multivariate regression we have many variables, and it is not surprising that for an optimal linear predictor $a \cdot x + b$, it holds that the residual vector is orthogonal to *each* of the centred explanatory variables (otherwise we could change the predictor vector a bit to increase the fit). That is, instead of a single linear equation, we now end up with n equations, one for each data column $x^{(i)}$,

$$(\overline{x}^{(i)} \cdot \overline{x}^{(1)})\, a_1 + (\overline{x}^{(i)} \cdot \overline{x}^{(2)})\, a_2 + \cdots + (\overline{x}^{(i)} \cdot \overline{x}^{(n)})\, a_n = \overline{x}^{(i)} \cdot y \tag{8}$$

For the weight vector a that satisfies these equations for all $i = 0, \ldots, n-1$, you can again simply find the matching intercept b as the mean residual when using just the weights a for fitting:

$$b = ((y_1 - a \cdot x_1) + \cdots + (y_m - a \cdot x_m))/m \ .$$

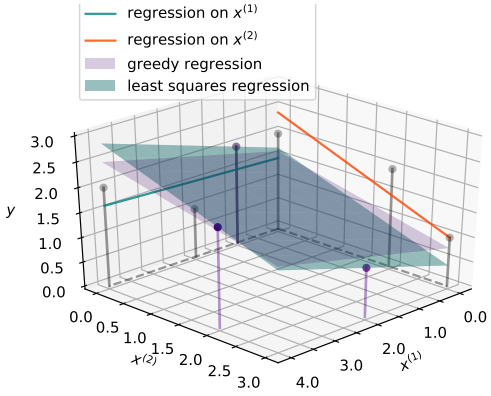


Figure 6: Continuation of the example given in Figure 5. The plot depicts the optimal least squares regression and compares it to the greedy solution (for reference, the best single predictors are shown as well). The least squares regression has the smallest error.

In summary, we know that we can simply transform the problem of finding the least squares predictor to solving a system of linear equation, which we can solve by Gaussian Elimination as covered in the lecture. An illustration of such a least squares predictor is given in Figure 6.

Instructions

1. Write a function `equation(i, data, y)` that produces the coefficients and the right-hand-side of the linear equation for explanatory variable $x^{(i)}$ (as specified in Equation (8)).

   Input: Integer `i` with `0 <= i < n`, data matrix `data` with `m` rows and `n` columns such that `m > 0` and `n > 0`, and list of target values `y` of length `n`.

   Output: Pair `(c, d)` where `c` is a list of coefficients of length `n` and `d` is a float representing the coefficients and right-hand-side of Equation 8 for data column `i`.

   For example (see Figure 6):

   ```
   >>> data = [[1, 0],
   ...         [2, 3],
   ...         [4, 2]]
   >>> target = [2, 1, 2]
   >>> equation(0, data, target)
   ([4.666666666666666, 2.3333333333333326], 0.3333333333333326)
   >>> equation(1, data, target)
   ([2.333333333333333, 4.666666666666666], -1.3333333333333335)
   ```

2. Write a function `least_squares_predictor(data, y)` that finds the optimal least squares predictor for the given data matrix and target vector.

   Input: Data matrix `data` with `m` rows and `n` columns such that `m > 0` and `n > 0`.

Output: Optimal predictor `(a, b)` with weight vector `a` (`len(a)==n`) and intercept `b` such that `a`, `b` minimise the sum of squared residuals.

For Example:

```
>>> data = [[0, 0],
...         [1, 0],
...         [0, -1]]
>>> y = [1, 0, 0]
>>> weights, intercept = least_squares_predictor(data, y)
>>> weights, intercept
([-0.9999999999999997, 0.9999999999999997], 0.9999999999999998)

>>> data = [[1, 0],
...         [2, 3],
...         [4, 2]]
>>> y = [2, 1, 2]
>>> weights, intercept = least_squares_predictor(data, y)
>>> weights, intercept
([0.2857142857142855, -0.4285714285714285], 1.7142857142857149)
```