First programming report

L. Beltrame*
University of Vienna - Data Science
(Dated: 11 ottobre 2021)

* beltramel99@univie.ac.at

I. NOTE

The programme is highly commented. In the "descriptive part", I wanted to give a general idea about the logical process I undertook.

II. TASK 1

II.1. Subtask 1

The python version I used is: 3.8.10 [MSC v.1916 64 bit (AMD64)]
The Pandas version I used is: 1.3.2
The Numpy version I used is: 1.20.2

II.2. Subtask 2

First, I wrote the init() module. In the init module I made sure that all the inputs of the init were strings.

Next, I checked that the input and output files were of the same type.

Later I checked that the paths exist in the folder I am working in.

Next, I checked that the file was either .txt or .csv.

Next, I checked that the type stated was the same as the corresponding file given (both I/O).

In all the aforementioned cases I raised an error if incompatible data were passed.

After the "control part" I saved ifname, ofname, iftype, oftype in the class.

The first module is "load": depending on the type passed while initializing, it uses the pd.read csv() function and adjusts it either to acquire ".csv" or ".txt" files. The data are stored locally in the class as "data".

The second module is "store": depending on the type passed while initialising, it stores "data" inside the same folder we are working in. Also in this case the data are stored according to the indications given. The return of the store module is a boolean, to control if the data were stored correctly.

The main challenge in the implementation of this DAO was thinking about all the possible exceptions that could occur when passing the input in the init() method.

II.3. Subtask 3

From abc I imported ABC. Then I created the function SubTaskABC, which inherited the abstract class ABC. In SubTaskABC I implemented the abstract method "process" which consisted of a "pass" command.

Subsequently, I created subclass13, which inherited the abstract class SubTaskABC. In this class I created the method process as required in the assignment. If the data were not stored correctly I raised an Exception.

The main challenge I faced was to use the DAO inside the class subclass13, in particular I had to change the DAO's store method since it worked only with pandas DataFrames and not with other objects (e.g. strings).

II.4. Subtask 4

To perform the unit testing I used the "unittest" library.
I checked that the basic operation performed by the two functions (SubTask13 and DAO).
The following tests were performed:

- test type is type: checks that the file and the type of file must correspond reading from the file "test 1.csv" (It checked if an exception was raised).
- test reading: checks the reading from '.csv' by comparing it with was is read by the file 'test 2.csv'.
- test writing: checks the storing method by using the boolean output given if the function stored correctly the file (see DAO class for reference).
- test process1: checks if the class SubTask13 is created from the abstract class and includes a DAO object.
- test process2: checks the process module of SubTask13 from the file 'input13.txt' by comparing the correct string (inserted by hand) and stored one.
- test process3: checks the process module of SubTask13 by acquiring the file 'test 3.txt' and comparing with the correct string (inserted by hand) and stores the string.

I choose to do these tests because they were those required to run the programme without errors.

III. TASK 2

III.1. Subtask 1

The first thing I did when initialising was to create two variables inside my class, one for the number of iterations and one for the DAO given as input in the initialisation.

Next, I loaded my data using the given DAO's module and store them into a 3 x k*3 matrix, where k is the number of individual matrices.

Then, I used my custom function "mat reader" to separate the individual matrices and I store them into a list. To multiply all the matrices I used my custom multiply function. I used a cycle to multiply all the matrices in the aforementioned list.

I acquired, using "timeit.default timer", the time elapsed during the single iteration (by taking the difference of the consecutive timers) and I averaged this value on the number of iterations specified during the initialisation.

I repeated the same procedure using Numpy.

To calculate the speed up I divided the average computational period of the custom function by the Numpy one. The speed up obtained is 2.519. This is due to the fact that the python's compilers are also written in C, and C is a high-performance language. On the other hand, the custom function creates variables in python, thus resulting slower.

Finally, using the DAO, I stored the resulting matrix in a .csv. I attached the speedup to that document directly opening the file.

2, 4, 7, 5, 6, 9
7, 1, 8, 2, 5, 8
6, 0, 2, 1, 1, 4

Tabella I. input21.csv

25,39,78
45,55,103
32,38,62
2.519

Tabella II. output21.csv

III.2. Subtask 2

The first thing I did when initialising was to create two variables inside my class, one for the number of iterations and one for the DAO given as input in the initialisation. Next I loaded, using the DAO, the two columns that I generated randomly, with values from 1 to 100.

To store the results I had I used a dictionary, whose keys were the various statistical parameters we had to compute. Each paramater was reported two times: one for the pandas implementation and one for the "manual" one.

In the first part of the module I created a list containing the two loaded columns. Subsequently, I performed all the required computations of both columns using pandas modules. The result were inserted in the aforementioned dictionary. I repeated the procedure using my custom function (for further information see the file "Function.py", it is commented.)

To conclude the procedure, I created a new list created by concatenating the previous columns. I performed the same analysis mentioned before and saved the results in the same dictionary.

I passed my dictionary to the DAO. I had to upgrade the DAO to store dictionaries (I used the method .from dict included in the class Pandas DataFrame).

In all the previously mentioned operations, I acquired, similarly to the previous subtask, the time needed by the computer to complete all the statistical operation. To get a more consistent result, I averaged this value on the number of iterations specified during the initialisation.

Therefore, I had 4 averaged times, one for each possible combination of columns (two separated columns or a single longer column) and of function used (pandas or custom).

I proceeded to calculate the speedups (custom/pandas) and so I obtained two values, one for each configuration (two separated columns or a single longer column).

For the single column configuration the speedup was: 0.295.

For the double column configuration the speedup was: 0.139.

The pandas implementation is definitely slower than the custom implementation, and this is visible in the speedups.

An interesting trend is that, by enlarging the length of the pandas Series on which I want to perform the analysis, the performances of the pandas increases, remaining nevertheless worse than the manual list implementation.

I suppose that on large dataset pandas become better than a custom implementation, but further analysis is needed to prove that claim.

The results of the analysis are included in the folder submitted.

IV. TASK 3

IV.1. Subtask 1

- The dataset has 8 features and 1 target
- There are 14448 samples in the train dataset
- There are 6192 samples in the train dataset

I split the data into train and test (0.7/0.3) and I shuffled them.

IV.2. Subtask 2

The MSE on the train dataset is: 0.605.
The MSE on the test dataset is: 0.605.

IV.3. Subtask 3

My computed weight vector is:

[5.19e-01, 1.59e-02, -1.91e-01, 9.69e-01, 1.14e-05, -4.22e-03, -6.17e-02, -1.48e-02]^T

The MSE on the train dataset is: 0.605.
The MSE on the test dataset is: 0.605.
The results match the previous ones, as expected by the exact solution the the regression problem.

IV.4. Subtask 4

100000 iterations were too much and did not let appreciate the elbow of the curve, therefore, after some trials, I observed that the curve was sufficiently flat after 5000 iterations. I sampled the results differently, the samplings are:

- 1 + 250 * k for k in range(20) for the non adaptive case
- 1 + 45 * k for k in range(25) for the adaptive case, since it converges faster

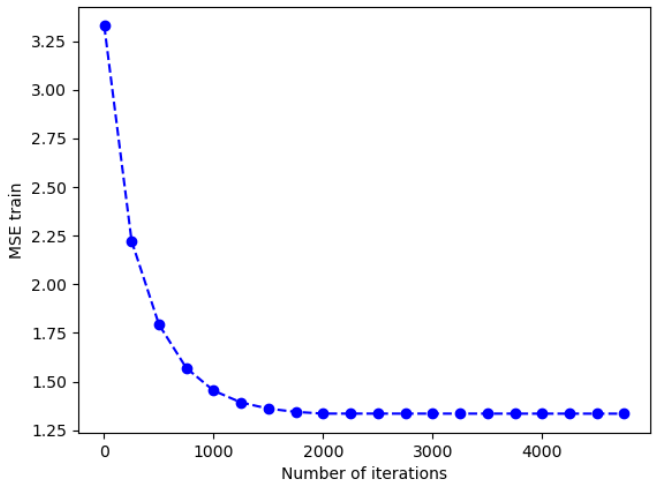

Figura 1. Gradient descent graph for the train dataset. It is clear that around the 2000th iteration we converge to a stable value.

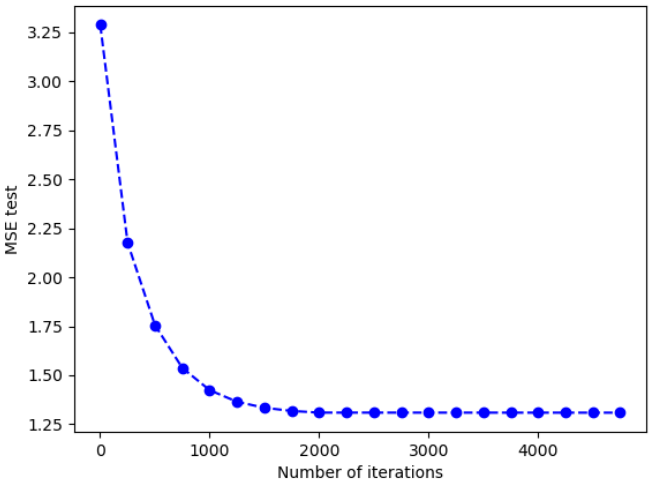

Figura 2. Gradient descent graph for the test dataset. It is clear that around the 2000th iteration we converge to a stable value.

The MSE is worse than the one obtained with the exact solution, but it is good. Both graphs show that the highest value of the cost function (MSE in our case) is maximum when w = 0. The mse decreses with the number of iterations, as we would expect.

IV.5. Subtask 5

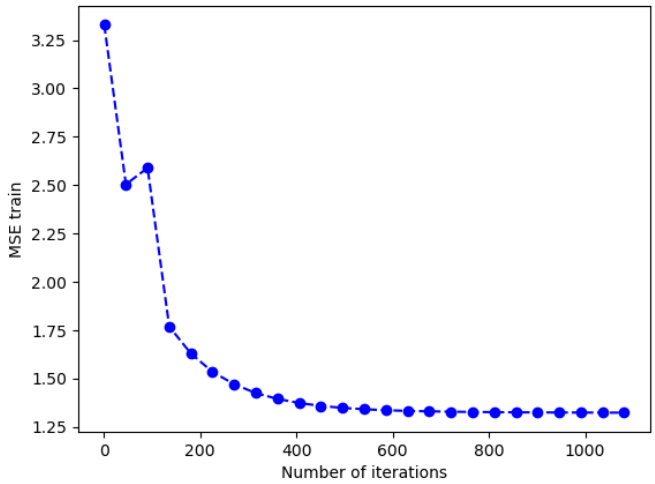

Figura 3. Gradient descent graph for the train dataset. It is clear that around the 700th iteration we converge to a stable value. We used the bold driver heuristic to update the learning rate.

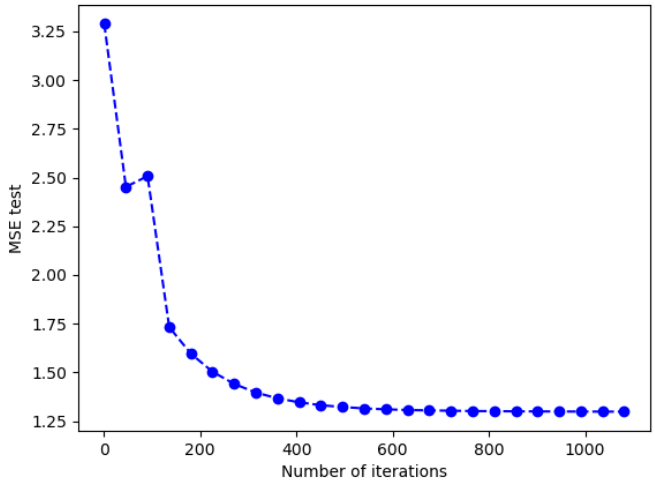

Figura 4. Gradient descent graph for the test dataset. It is clear that around the 700th iteration we converge to a stable value. We used the bold driver heuristic to update the learning rate.

By confronting with the previous graphs, it is auto-evident that the MSE converges faster.

IV.6. Subtask 6

MinMaxScaler transforms the data according to the following:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{1}$$

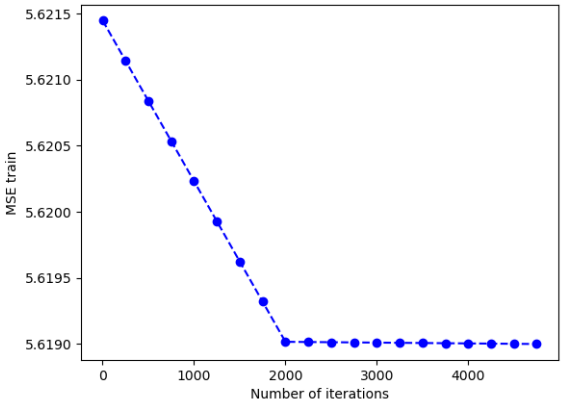

Figura 5. Gradient descent graph for the train dataset. It is clear that around the 2000th iteration we converge to a stable value. We used the MinMaxScaler when preprocessing the input data.

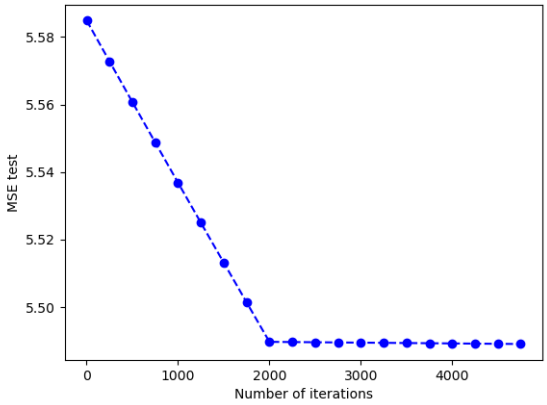

Figura 6. Gradient descent graph for the test dataset. It is clear that around the 2000th iteration we converge to a stable value. We used the MinMaxScaler when preprocessing the input data.

The results are generally worse than the previous case, it might be due the initial learning rate chosen since it was manually chosen to get a nice result on the non-scaled data.

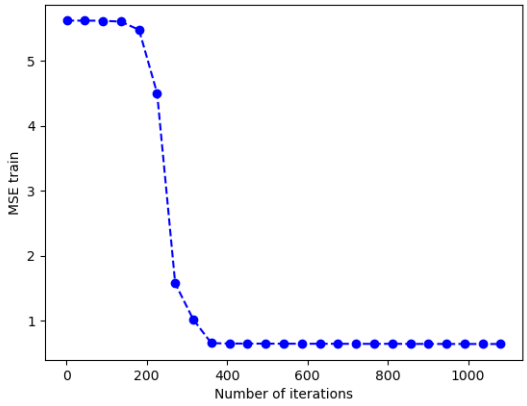

Figura 7. Gradient descent graph for the train dataset. It is clear that around the 400th iteration we converge to a stable value. We used the bold driver heuristic to update the learning rate. We used the MinMaxScaler when preprocessing the input data.

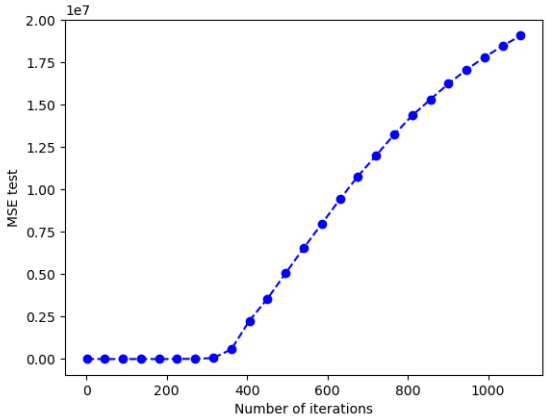

Figura 8. Gradient descent graph for the test dataset. We used the bold driver heuristic to update the learning rate. We used the MinMaxScaler when preprocessing the input data.

The last graph is exploding since I used only 5000 iteration. Probably if I would have given the algorithm more iterations we would have a result similar to the penultimate graph.

This intuition proved to be right, since with 50000 iterations the graph shows a really slow decrease. But more iterations might be needed to obtain a convergence.

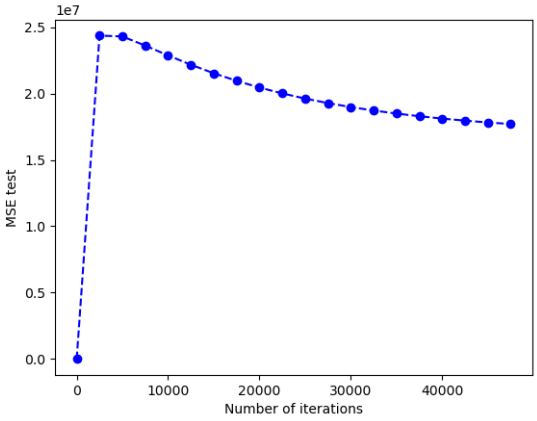

Figura 9. Gradient descent graph for the test dataset. We used the bold driver heuristic to update the learning rate. We used the MinMaxScaler when preprocessing the input data.

The penultimate graph is indeed really good, in fact the mse at the 5005th iteration was: 0.622, which is really close the to the minimizer of the optimization problem.

IV.7. Subtask 7

First, I applied the MinMaxScaler to my data and then I used the Polynomial Features "fit transform" to the scaled data.

I used scikit linear regression to perform the regression.

I had the following results:

```
The MSE (TRAIN) of the lin. reg. with poly of order  2  is:  0.4236009471571964
The MSE (TEST) of the lin. reg. with poly of order  2  is:  0.4549723374856868
The MSE (TRAIN) of the lin. reg. with poly of order  3  is:  0.34348443747552854
The MSE (TEST) of the lin. reg. with poly of order  3  is:  19.315927622566825
The MSE (TRAIN) of the lin. reg. with poly of order  4  is:  0.281704580493528
The MSE (TEST) of the lin. reg. with poly of order  4  is:  13812.427158026308
```

Figura 10. Resulting MSE for the various Polynomial Features.

As the order of the polynomial increases, the training MSE constantly decrease. On the other hand, the testing MSE increases. This means that the model is overfitting.