English documentation for `mcq_solver`

Silzinc

August 20, 2023

Contents

1 Introduction

`mcq_solver` is a library written in Rust and compiled, with `maturin` and `pyo3`, into a shared library accessible by a Python "CPython" interpreter, version $\geq$ 3.10 (tested on 3.10.6). The installation and uninstallation processes are explained in the repository's `README.md` file.

Its functions allow, from the answer sheets of a MCQ and their grades, to determine the correction of said MCQ. The algorithm is based on the "simulated annealing" statistical method in order to converge to a correction that would explain the best way the grades observed.

An example for each function can be found in the repository's `examples` directory.

2 Available functions

The library `mcq_solver` proposes two functions. Their arguments are given below in order with their original names.

2.1 `solve`

Description Determines the correction of a MCQ from raw Python data. The output is a couple `(result, success)`. If `success` is `True`, then `result` approaches or is the MCQ's correction. Otherwise, `result` is an error message to print out.

Mandatory arguments (On the left lies the name, on the right lies description)

`sheets`	A list of answer sheets, each sheet being a list of answer tokens
`_answer_tokens`	A string or a list of distincts tokens (characters) corresponding to the different possible answers
`grades`	A list of integers ≤ 255 corresponding the the grades of the sheets

Optional arguments (From left to right, the name, the default value, the description)

`starting_beta`	0.1	β initial value in the annealing algorithm
`max_beta`	0.5	The value of β for which the algorithm stops
`lambda_inv`	1.01	The value of $1/\lambda$ in the annealing algorithm

2.2 `solve_from_files`

Description Determines the correction of an MCQ based on data written in text files. The output is the same as for `solve`. This function can help shipping and using the data of hundreds of sheets, each containing more than a hundred answers.

Mandatory arguments (On the left lies the name, on the right lies description)

`sheets_path`	A list of paths to the answer sheets
`_answer_tokens`	Identical to `solve`'s
`grades_path`	A path leading to a file containing the grades, integers ≤ 255
`grades_separator`	A token meant to separate the grades in the file at the end of `grades_path` (typically comma `','` or a newline `'\n'`)
`number_of_questions`	The number of questions of the MCQ

Optional arguments The same as `solve`'s

3 The algorithm used

3.1 Potential to minimize

Suppose we have a problem consisting in $q \in \mathbb{N}^*$ questions and $N \in \mathbb{N}^*$ answer sheets, denoted $F_1, ..., F_N$. Each sheet F_n has a grade $g_n \in [\![0, q]\!]$ and a list of answers $a_{n,1}...a_{n,q}$. We also suppose that there is only one correction to the MCQ $a = a_1...a_q$ that can give the resulting grades (otherwise, the problem is unsolvable).

Therefore,

$$\forall n, g_n = \#\{k \in [\![0, q]\!] \,|\, a_{n,k} = a_k\}$$

The goal is to determine the list of answers satisfying this condition that is a characteristic of the correction. To this end, we represent a candidate correction by a list $c = c_1...c_q$ of answers. We then define the *disagreement* between c and a over F_n by

$$d_n(a, c) = \#\{k \in [\![0, q]\!] \,|\, a_{n,k} = a_k\} - \#\{k \in [\![0, q]\!] \,|\, a_{n,k} = c_k\}$$

And we introduce the following function, that we will call c's *potential*:

$$\phi(c) = \sum_{n=1}^{N} d_n(a, c)^2$$

It is therefore clear that, under the chosen conditions, $a = c \Leftrightarrow \phi(c) = 0$. And so we will construct a sequence of candidates (c_i) that musts make ϕ converge to its minimum. The exact method is described in the next subsection.

3.2 Simulated annealing algorithm

The simulated annealing is a general statistical method to search for the extremum of a function that is hard to analyze. Here is how we use it in our case:

- **Setup**
  1. Choose an initial value for the *temperature* $T_0 > 1$.
  2. Choose a stopping value for the temperature $T_0 > T^* > 1$. The algorithm will iterate as long as $T > T^*$.
  3. Choose a constant value for a coefficient $0 < \lambda < 1$. The usual value picked is $\lambda = 0.99$ but here we will rather take $1/\lambda = 1.01$.
  4. Choose the first candidate c_0 to be the answer sheet with the best grade (or randomly picked between the best sheets if there are several)
- **Main loop (while $T > T^*$ and $\phi(c) \neq 0$)**
  1. Randomly choose a question $k \in [\![0, q]\!]$ and change randomly c's answer to this question to construct another candidate c'.
  2. Note the change in potential $\Delta\phi = \phi(c') - \phi(c)$.
  3. If $\Delta\phi < 0$, replace c by c'.
  4. Otherwise, replace c by c' with a probability $\exp\left(-\frac{\Delta\phi}{T}\right)$ and, if the change happens, replace T by $\lambda T < T$.
  5. Memoize the candidate with the lowest potential along the execution.
- **Output: lowest potential candidate**

3.3 Optimizations

The first optimization lies in remplacing T by the *Boltzmann coefficient* $\beta = \frac{1}{T}$ (the Boltzmann constant is taken equal to 1 in this representation) and λ by $1/\lambda$ so that there is no division to make in the algorithm, and these are more expansive than multiplications.

A second one, to compute fast a value of $d_n(c, a)$, if to vectorize the main loop of the calculation (the loop that counts matching answers) by using the *SIMD*s. This optimization was removed from the code as Rust's compiler is powered by the *LLVM* toolchain which applies a similar optimization automatically.

Finally, initially, every structure was statically allocated on the stack, which allowed *in fine* to compute a value of $d_n(c, a)$ in about 20 nanoseconds. In order to allow the user to choose easily the number of questions per answer sheets and other parameters, the structures are now dynamically allocated on the heap, which reduced this performance a bit.

4 Performances

Several dozens of thousands of randomly[1] generated MCQs with hundreds of answer sheets for each where given to the algorithm. The number of possible answers per question ranged from 2 to 8 and the number of questions was 120.

4.1 Correction

When 2 choices per question are given, 150 copies were often enough to solve the problem. For 8 choices, this number had to be two or threefolded. What was generally observed is that, out of 5000 MCQs, the algorithm failed solving between 1 and 5 of them. The grade of the candidate was still much higher than the best grade of the input sheets.

4.2 Speed

In the same conditions, it took the algorithm at most a few milliseconds to solve an MCQ. 5000 of them were solved in 2.5 seconds generally.

[1]Every random input followed a uniform law

5 Organization of the `src` directory

This folder contains the Rust source code of the library. The functions defined here are slightly modified and re-exported by the script `__init__.py` in the directory `python/mcq_solver`.

- The file `lib.rs` declares the other modules and exports the Rust functions `_solve` and `_solve_from_files` called by the library's Python layer.
- `vec_util.rs` defines a function that is not used anymore to initialize a Rust dynamic array from a generator function.
- `sheet.rs` defines the structure `Sheet` for the answer sheets and the type `Answer` for the answers.
- `parameters.rs` defines thestructure `AnnealingParameters` encapsulating the parameters of the annealing algorithm.
- `annealing.rs` defines in particular the structure `AnnealingSolver` and the function `solve_mcq` allowing to use the annealing algorithm to correct the MCQ.
- `parsing.rs` defines the backend of `solve_from_files`.
- `basic.rs` defines the backend of `solve`.