Documentation en français pour `mcq_solver`

Silzinc

20 août 2023

Table des matières

1 Introduction

`mcq_solver` est une bibliothèque écrite en Rust et compilée, avec `maturin` et `pyo3`, en bibliothèque partagée utilisable avec l'interpréteur Python « CPython » version $\geq$ 3.10 (testée uniquement sur la 3.10.6). Les procédés pour l'installer et la désinstaller sont détaillés dans le fichier `README.md` du dépôt.

Ses fonctions permettent, à partir de copies notées d'un QCM inconnu, de déterminer la correction dudit QCM. L'algorithme se base sur la méthode statistique du « recuit simulé » pour converger vers une correction qui expliquerait au mieux les notes obtenues par les différentes copies.

Un exemple d'utilisation de chaque fonction peut être trouvée dans le dossier `examples` du dépôt.

2 Fonctions disponibles

La bibliothèque `mcq_solver` propose deux fonctions. Les arguments sont donnés dans l'ordre avec leurs noms d'origine.

2.1 `solve`

Description Détermine la correction du QCM à partir de données brutes écrites en Python. La sortie est un couple `(result, success)`. Si `success` vaut `True`, alors `result` approche ou est la liste des réponses du QCM. Sinon, `result` est un message d'erreur à afficher.

Arguments obligatoires (À gauche le nom, à droite la description)

`sheets`	Une liste de fiches réponses, chaque fiche réponse étant une liste de caractères
`_answer_tokens`	Une chaîne ou une liste de caractères distincts correspondants aux différentes réponses
`grades`	Une liste d'entiers ≤ 255 correspondants aux notes des fiches

Arguments facultatifs (De gauche à droite, le nom, la valeur par défaut, la description)

`starting_beta`	0,1	La valeur initiale de β dans le recuit
`max_beta`	0,5	La valeur de β pour laquelle l'algorithme s'arrête
`lambda_inv`	1,01	La valeur de $1/\lambda$ dans le recuit

2.2 `solve_from_files`

Description Détermine la correction du QCM à partir de données inscrites sur des fichiers. La sortie est la même que celle de `solve`. Cette fonction peut aider à transporter facilement des centaines de copies avec plus de cent réponses par copie.

Arguments obligatoires (À gauche le nom, à droite la description)

`sheets_path`	Une liste de chemins vers des fichiers réponse
`_answer_tokens`	Identique à `solve`
`grades_path`	Un chemin vers un fichiers contenant les notes, des entiers ≤ 255
`grades_separator`	Un caractère censé séparer les notes dans le fichier au bout de `grades_path` (typiquement une virgule `','` ou un saut de ligne `'\n'`)
`number_of_questions`	Le nombre de questions par fiche réponse

Arguments facultatifs Les mêmes que ceux de `solve`

3 L'algorithme utilisé

3.1 Potentiel à minimiser

Supposons que l'on ait un problème constitué de $q \in \mathbb{N}^*$ questions et de $N \in \mathbb{N}^*$ fiches réponses, notées $F_1, ..., F_N$. Chaque fiche F_n a une note $g_n \in [\![0, q]\!]$ et une liste de réponses $a_{n,1}...a_{n,q}$. On suppose en outre qu'il n'existe qu'une seule correction de QCM $a = a_1...a_q$ engendrant ces résultats (dans le cas contraire, on ne peut pas décider).

Ainsi,

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

L'objectif est de déterminer une liste de réponse vérifiant cette condition qui caractérise ici la correction. Pour cela, on représente une correction candidate par une liste $c = c_1...c_q$ de réponses. On définit le *désaccord* entre c et a sur F_n par

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

Et on introduit la fonction suivante que l'on appellera *potentiel* de c :

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

Il est alors clair que, dans les hypothèses choisies, $a = c \Leftrightarrow \phi(c) = 0$. On va donc construire une suite (c_i) de corrections candidates qui doit faire converger ϕ vers son minimum. La méthode exacte pour ce faire est l'objet de la sous-section suivante.

3.2 Algorithme de recuit simulé

Le recuit simulé (*simulated annealing* en anglais) est une méthode statistique générale pour chercher l'extremum d'une fonction difficile à analyser. Voici comment elle est appliquée ici :

— **Mise en place**

1. Choisir une valeur initiale pour la *température* $T_0 > 1$.
2. Choisir une valeur d'arrêt pour la température $T_0 > T^* > 1$. L'algorithme continuera d'itérer tant que $T > T^*$.
3. Choisir une valeur constante pour un coefficient $0 < \lambda < 1$. On choisit généralement $\lambda = 0,99$ mais on prendra plutôt $1/\lambda = 1,01$ ici.
4. Choisir une première correction candidate c_0 comme la copie ayant eu la meilleure note (ou une au hasard s'il y a conflit).

— **Boucle principale (tant que $T > T^*$ et $\phi(c) \neq 0$)**

1. Choisir une question $k \in [\![0, q]\!]$ au hasard et changer au hasard la valeur de c pour obtenir une autre candidate c'.
2. Noter la variation de potentiel $\Delta\phi = \phi(c') - \phi(c)$.
3. Si $\Delta\phi < 0$, remplacer c par c'.
4. Sinon, remplacer c par c' avec une probabilité $\exp\left(-\frac{\Delta\phi}{T}\right)$ et, en cas de changement, remplacer T par $\lambda T < T$.
5. Garder en mémoire la candidate qui avait le plus bas potentiel au cours de la boucle.

— **Sortie : candidate au plus bas potentiel**

3.3 Optimisations machine

La première optimisation est de remplacer T par le *coefficient de Boltzmann* $\beta = \frac{1}{T}$ (la constante de Boltzmann est prise égale à 1 dans cette représentation) et λ par $1/\lambda$ pour que la machine ne fasse pas de divisions, plus coûteuses que les multiplications.

Une seconde optimisation, pour calculer rapidement une valeur de $d_n(c, a)$, est de vectoriser la boucle principale en utiliser les *SIMD*. Cette optimisation a été retirée du code car le compilateur de Rust fonctionne avec *LLVM* qui applique une optimisation similaire automatiquement.

Enfin, initialement, toutes les structures de données étaient initialement allouées sur le tas, ce qui permettait *in fine* de calculer une valeur de $d_n(c, a)$ en environ 20 nanosecondes. Pour que l'utilisateur puisse choisir facilement le nombre de questions et de fiches réponses, les allocations sont maintenant faites dynamiquement sur le tas, ce qui a un peu réduit cette performance.

4 Performances

Plusieurs dizaines de milliers de QCMs ont été générés aléatoirement [1] avec des centaines de copies aléatoires corrigées pour chacun. Le nombre de réponses possibles allait de 2 à 8 et le nombre de questions restait à 120.

4.1 Correction de l'algorithme

Lorsque 2 choix sont possibles par questions, 150 copies suffisaient à résoudre le problème très souvent. Pour 8 réponses possibles, il en fallait le double, voire le triple. On observe généralement que, sur 5000 QCMs générés aléatoirement, l'algorithme échoue à en résoudre entre 1 et 5. La note de la candidate était tout de même bien plus élevée que la note maximale des copies.

4.2 Vitesse de l'algorithme

Dans les mêmes conditions, l'algorithme prenait tout au plus quelques millisecondes pour résoudre un QCM. 5000 QCMs se trouvaient résolus en un total de 2,5 secondes généralement.

1. Toutes les entrées aléatoires suivaient une loi uniforme

5 Organisation du dossier `src`

Ce dossier contient le code source en Rust de la bibliothèque. Les fonctions qui y sont définies sont légèrement modifiées et ré-exportées par le script `__init__.py` dans le dossier `python/mcq_solver`.

- Le fichier `lib.rs` réunit les autres modules et exporte les fonctions Rust `_solve` et `_solve_from_files` appelées par la partie Python de la bibliothèque.
- `vec_util.rs` définit une fonction qui n'est plus utilisée pour créer un tableau dynamique de Rust à partir d'une fonction.
- `sheet.rs` définit la structure `Sheet` pour les fiches réponses et le type `Answer` pour les réponses.
- `parameters.rs` définit la structure `AnnealingParameters` encapsulant les paramètres du recuit.
- `annealing.rs` définit notamment la structure `AnnealingSolver` et la fonction `solve_mcq` permettant d'utiliser l'algorithme de recuit pour corriger le QCM.
- `parsing.rs` définit l'arrière-fond de `solve_from_files`.
- `basic.rs` définit l'arrière-fond de `solve`.