Machine Learning & Symbolic Regression

Hubert Gruniaux Nathan Boyer Nino Boismenu

Novembre 2024

1 Introduction : régression symbolique

La régression symbolique (SR) consiste à trouver la meilleure expression mathématique ajustant des données expérimentales. Une telle solution doit réaliser un compromis entre ajustement fidèle des données et simplicité de l'expression.

Les méthodes les plus utilisés pour effectuer des régressions symboliques sont les *transformers* et les *méthodes génétiques*. Nous avons choisi pour ce projet d'utiliser une méthode génétique, améliorée par des réseaux de neurones.

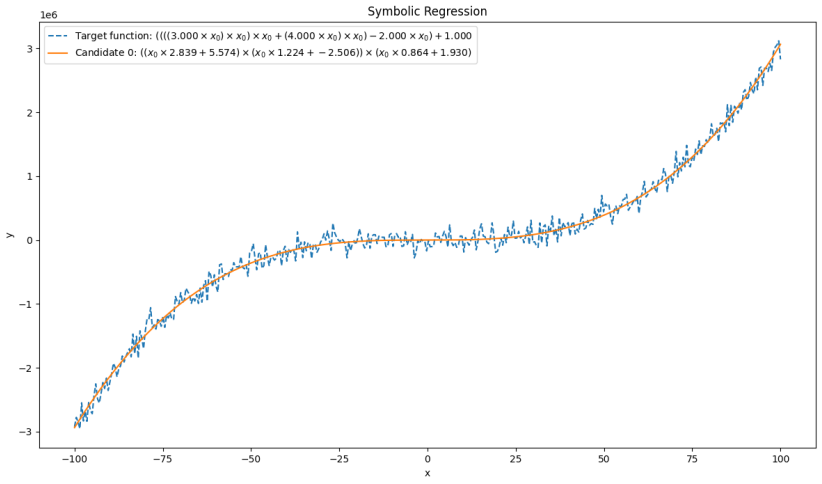


FIGURE 1 – Résultat de la régression de la courbe de fonction $3x^3 + 4x^2 - 2x + 1$ avec du bruit gaussien. Le candidate trouvé par notre programme est $3.002x^3 + 6.454x^2 - 12.632x - 26.959$ (avec une erreur quadratique moyenne de 0.0124) en utilisant seulement un algorithme d'évolution génétique. Les hyperparamètres choisis sont : 10 itérations avec une population de 500 formules.

Une résolution par algorithme génétique est réalisée en trois temps et est construite par analogie à l'évolution du vivant. D'abord, un grand nombre d'expressions mathématiques sont générées aléatoirement, puis chaque expression est évaluée sur les données à ajuster, et enfin des mutations sont réalisées sur la (ou les) meilleurs expressions trouvés. On effectue un grand nombre de cycles mutations-sélection, jusqu'à ce qu'une expression simple et correspondant aux données soit trouvée.

Les données à ajuster peuvent être exprimées comme un ensemble de couples $(x_k, y_k)_{k\in I}$, et on cherche une fonction telle que $f(x_k) = y_k,\ \forall k \in I$.

2 Implémentation de la méthode génétique

2.1 Aperçu général d'un algorithme génétique

Un algorithme génétique repose sur l'idée d'évolution d'une population pour optimiser un objectif. Dans notre cas, la population est un ensemble de fonctions mathématiques, et l'objectif est de minimiser une fonction de perte l entre les fonctions et la courbe expérimentale.

A chaque génération, on génère une nouvelle population de formules par trois moyens distincts :

— en appliquant des mutations aux meilleurs éléments de la génération précédente,
— en combinant différents bons éléments de la génération précédente,
— en introduisant à nouveau des formules aléatoirement générées.

L'algorithme exact est fourni ci-dessous :

$G \leftarrow$ Random population of N formulas
while $\min\{l(f, y), f \in G\} > \tau_{\text{target}}$ **do**
    $C \leftarrow k$ best candidates of G
    $G_m \leftarrow \lambda_m N$ mutations of formulas in C
    $G_c \leftarrow \lambda_c N$ crossover of formulas in C
    $G_r \leftarrow$ Random population of $(1 - \lambda_m - \lambda_c)N$ formulas
    $G \leftarrow G_m \bigcup G_c \bigcup G_r$
end while

La suite de cette section décrit comment sont implémentés la génération aléatoire des formules, les mutations et la combinaison de différentes formules. On mentionne également le problème du choix de la fonction de perte (appelée fonction de *fitness* en génétique).

2.2 Traitement des expressions mathématiques

Une fonction mathématique est traitée comme un arbre dans notre programme :

— Les variables et les constantes sont les feuilles.
— Les opérateurs binaires ($+$, $-$, $\times$, $/$, etc.) et unaires (sin, exp, etc.) sont les noeuds internes.

Pour le moment, nous nous sommes limités aux fonctions de $\mathbb{R} \to \mathbb{R}$, donc avec une seule variable.

2.3 Génération aléatoire d'une formule

De part la représentation sous forme d'arbre des expressions, une façon simple de résoudre ce problème est de le décomposer récursivement. A chaque nœud de l'expression, on génère une (resp. deux) sous-formule.s si elle est dans une expression unaire (resp. binaire).

Exemples de quelques fonctions générées aléatoirement par notre système :

— $\arctan(e^{1.60-(x*2.59+2.18)} * 4.19 + -1.20) * 1.84 + 0.81$

— $(x * 3.03 + -4.74) * (x * 0.67 + 1.40)$

— $e^{x*-0.17+-0.59} * (-1.51) + -4.84$

2.4 Choix de la fonction de perte

Pour évaluer une expression et ainsi ne garder que les meilleurs éléments d'une population, nous évaluons l'expression sur le jeu de données et calculons l'écart quadratique moyen.

De plus, afin d'éviter l'apparition de fonctions trop compliquées, nous avons ajouté dans la fonction de coût un terme proportionnel à la complexité de la formule.

Ainsi, la fonction de coût peut être exprimée comme suit :

$$l(f) = \eta \left(\frac{\sum_{k \in I} (f(x_k) - y_k)^2}{\sum_{k \in I} y_k^2} \right) + (1 - \eta)\,\mathrm{D(f)} \tag{1}$$

Avec $D(f)$ la profondeur de l'expression, et $\eta \in [0, 1]$ un hyperparamètre.

2.5 Mutations implémentées

A l'aide de la génération aléatoire d'expression exposée ci-dessus, la première étape d'un algorithme génétique peut être réalisée. Il est ensuite nécessaire d'effectuer des mutations sur les expressions les plus proches des données.

Les mutations que nous avons utilisées sont :

— La variation aléatoire des constantes à l'aide d'une loi normale,

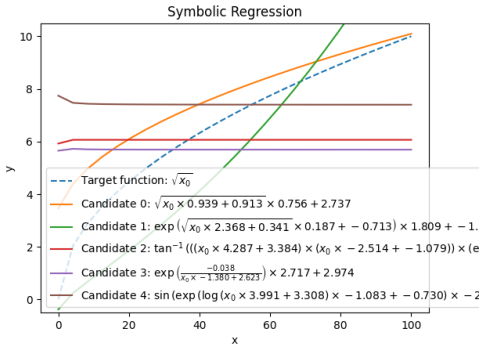
— L'échange de deux termes d'une expression binaire non commutative (-, /),

— Le remplacement d'un opérateur binaire/unaire par une autre,

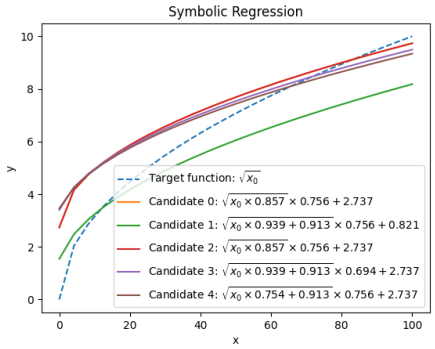
— L'ajout d'un opérateur binaire/unaire,

— La suppression d'opérateurs binaire/unaire.

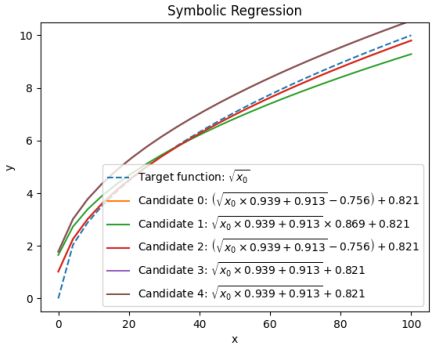
3 Résultats de cette méthode



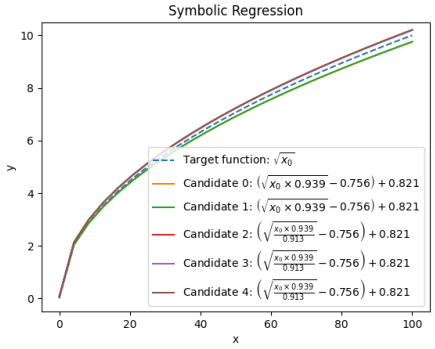
(a) Itération 1



(b) Itération 2



(c) Itération 3



(d) Itération 4

FIGURE 2 – Exemple d'exécution pour trouver la fonction $\sqrt{x}$ sur l'interval $[0, 50]$ avec seulement 25 échantillons et du bruit gaussien. On trouve un candidat viable au bout de 4 itérations.

La méthode génétique parvient à retrouver certaines expressions simples en un temps raisonnable (figure 2). Néanmoins, le programme est parfois bloqué dans des minimas locaux et on obtient souvent des formules trop compliqués (figure 3), ce qui demande un ajustement des hyperparamètres au cas par cas.

4 Perspectives

4.1 Amélioration des mutations initiales grâce à un réseau de neurones

Afin d'améliorer la rapidité du réseau de neurones, nous avons imaginé construire un réseau de neurones qui prend en entrée un ensemble de points $(x, f(x))$ pour une certaine fonction f et donne en sortie une prédiction de la probabilité d'utilisation dans la fonction

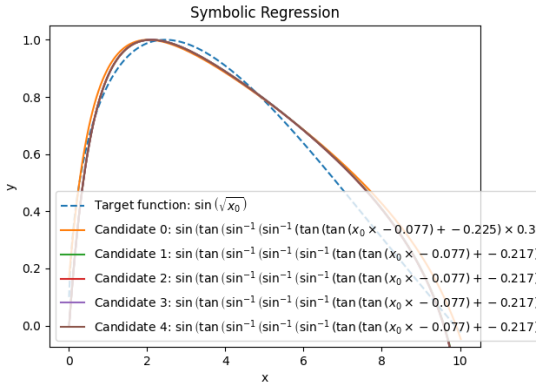


FIGURE 3 – Exemple d'*overfitting*. Les candidats trouvés sont beaucoup trop complexes. Un η plus important aurait dû être choisi pour essayer de limiter le problème.

f de chaque opérateur unaire.

Les opérateurs unaires auxquels on s'intéresse sont : exp, sin, tan, atan, arcsin, sqrt et log. (On a pris soin de ne détecter que des opérations qui ne sont pas *sensibles* aux transformations affines)

Le réseau neuronal imaginé est constitué de 2 couches de *conv1d* avec *pooling*, puis 2 couches linéaires (à noter que toute couche intermédiaire a une activation Relu).

Le réseau neuronal est entraîné et testé sur des formules de profondeur 3 et il obtient, pour l'instant en ne l'entraînant que sur 15 époques, 87% d'*accuracy*, ce qui est assez correct étant donné qu'il n'est pas rare que certaines opérations soient totalement cachées par d'autres (e.g. $20\sin(x) - 0.01\sqrt{x}$, $\sqrt{x}$ est négligeable) ou bien que certains opérateurs soient équivalents (e.g. $\sqrt{x} = \exp(\frac{1}{2}\log(x))$).

Nous n'en sommes qu'au début de cette amélioration, donc nous ne pouvons pas encore évaluer précisément son efficacité.

4.2 Amélioration des mutations

Une autre idée serait d'intégrer un deuxième réseau de neurones consacré aux processus de mutations pour pouvoir les guider. Le plus simple pourrait être d'utiliser le même réseau que précédemment, qui donne seulement quelles opérations unaires sont les plus probables d'êtres présentes dans la formule, et donc de générer plus ou moins de mutations sur ces opérations. Mais beaucoup d'autres pistes peuvent êtres explorées.