Projet CNAM

NFE 212 - 2014

Résolution de problèmes de classification par algorithmes évolutionnaires grâce au logiciel DEAP

Stéphane Legrand < stephleg@free.fr >

17 octobre 2014

Résumé

Nous verrons tout d'abord ce que sont les algorithmes évolutionnaires en les resituant dans leur contexte et en rappelant leurs principales caractéristiques. Puis nous détaillerons en quoi consiste les problèmes de classification de données. Le logiciel DEAP (Distributed Evolutionary Algorithms in Python) sera ensuite décrit et nous poursuivrons par son test pratique sur deux exemples de problèmes de classification. Le premier cas porte sur un problème de classification supervisée et une résolution par algorithme génétique. Le deuxième cas est un problème de classification non supervisée avec une tentative de résolution grâce à un programme génétique. Nous conclurons sur les difficultés rencontrées lors de cette mise en pratique et sur les apports du logiciel DEAP.

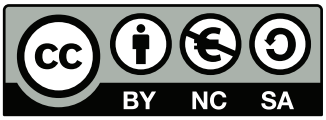

Table des matières

Table des figures

CHAPITRE 1

Introduction

Parmi les algorithmes qui existent pour résoudre les problèmes d'optimisation, on dispose notamment d'algorithmes qui s'inspirent de mécanismes naturels. C'est le cas des algorithmes dits évolutionnaires parmi lesquels on trouve les algorithmes génétiques et les programmes génétiques.

Même si cette famille d'algorithmes a l'inconvénient de ne pas garantir la découverte de la meilleure solution, elle offre toutefois dans la pratique une excellente alternative pour la résolution d'un problème lorsqu'il n'existe aucun algorithme déterministe de complexité raisonnable.

De nombreuses bibliothèques sont disponibles pour développer de tels programmes évolutionnaires (voir le tableau 1.1).

On a choisi ici de s'intéresser à la librairie DEAP. L'objectif est de mesurer

Nom	Langage	Site web
EO	C++	`http://eodev.sourceforge.net/`
ParadisEO	C++	`http://paradiseo.gforge.inria.fr/`
ECJ	Java	`http://cs.gmu.edu/~eclab/projects/ecj/`
JGAP	Java	`http://sourceforge.net/projects/jgap/`
MOEA	Java	`http://www.moeaframework.org/`
Opt4J	Java	`http://opt4j.sourceforge.net/`
DEAP	Python	`https://code.google.com/p/deap/`
Pyevolve	Python	`http://pyevolve.sourceforge.net/`

FIGURE 1.1 – Bibliothèques dédiées au développement de programmes évolutionnaires.

les apports de cette librairie dans le cas du développement d'un algorithme génétique et dans le cas du développement d'un programme génétique. Pour cela, on va réaliser deux tests pratiques qui portent sur la résolution de problèmes de classification.

CHAPITRE 2

Les algorithmes évolutionnaires

Egalement appelés algorithmes évolutionnistes, ils font partie de la famille des métaheuristiques qui sont des algorithmes d'optimisation stochastique [1]. Ce type d'algorithme utilise une part d'aléatoire afin de trouver des solutions optimales (ou tout du moins proches de l'optimal) à des problèmes difficiles pour lesquels aucun algorithme déterministe efficace n'est connu.

Leur principe trouve son inspiration dans deux mécanismes naturels majeurs. D'une part, la reproduction des êtres vivants avec transmission du patrimoine génétique. Les enfants héritent d'une combinaison des gènes de leurs deux parents. D'autre part, la théorie de l'évolution décrite par Charles Darwin au milieu du 19ème siècle. Seuls les individus les mieux adaptés au milieu survivent et ont par conséquent davantage d'opportunités de se reproduire.

2.1 Historique

Quelques dates clefs afin de resituer le contexte historique du développement des algorithmes évolutionnaires [1] :

1954 Premières simulations du processus d'évolution par Barricelli pour résoudre des problèmes d'optimisation.

1965 I. Rechenberg et H.P. Schwefel conçoivent le premier algorithme utilisant des stratégies d'évolution.

1. `http://fr.wikipedia.org/wiki/Métaheuristique` et `http://fr.wikipedia.org/wiki/Programmation_génétique`. Voir également [2].

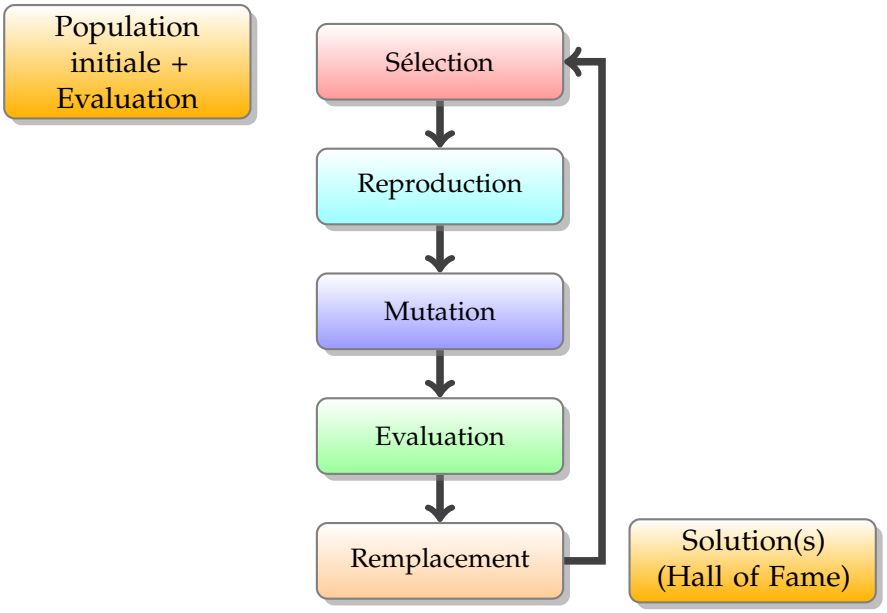

Figure 2.1 – Schéma global d'un algorithme évolutionnaire

1966 Fogel, Owens et Walsh proposent la programmation évolutionnaire.

1975 J. Holland propose les premiers algorithmes génétiques.

1980 Smith utilise la programmation génétique.

1989 Parution de l'ouvrage de D.E. Goldberg sur les algorithmes génétiques.

1992 Koza définit un paradigme standard pour la programmation génétique.

2.2 Schéma global

Les algorithmes évolutionnaires suivent tous un schéma similaire à la figure 2.1 :

1. Une population initiale est générée, par exemple de manière aléatoire, et la performance des individus de cette population est évaluée par rapport au problème posé.
2. Les opérations suivantes sont ensuite réitérées jusqu'à un critère d'arrêt qui peut être un nombre d'itérations maximum ou bien un niveau de performance minimal à atteindre :
   a) Les individus qui se reproduiront pour donner de nouveaux individus enfants sont sélectionnés. Cette sélection des parents tient compte de la performance des individus. Plus un individu est performant, plus il aura de chances de se reproduire.

b) Les parents sélectionnés se reproduisent. Les enfants ainsi générés héritent des gènes de leurs parents.

c) De manière aléatoire, les gènes de certains enfants peuvent connaître des mutations. Ce qui peut leur apporter de nouvelles caractéristiques qui s'avèreront très intéressantes par rapport au problème posé (leur performance s'en verra augmentée).

d) La performance des individus enfants est évaluée par rapport au problème posé.

e) Les individus qui seront éliminés et ne feront plus partie de la génération suivante sont choisis. Ce choix est fonction de la performance des individus, les moins adaptés étant éliminés.

3. A chaque itération, la ou les meilleurs solutions (représentées par un individu ou une population) au problème donné sont conservées dans le « Hall of Fame ». Ce sont ces solutions qui seront proposées par l'algorithme comme réponses au problème.

Nous pouvons constater qu'un algorithme évolutionnaire consiste en un procédé itératif qui implique une population d'individus dont chacun ou la population elle-même représente une solution possible à un problème spécifique parmi un espace de recherche [3]. Cet espace de recherche contient l'ensemble des solutions possibles au problème donné et s'avère beaucoup trop large pour être systématiquement exploré. A chaque étape de l'algorithme, un compromis doit être trouvé entre l'exploration de l'espace de recherche (pour éviter de stagner dans un optimum local) et l'exploitation des résultats déjà obtenus (pour tenter de découvrir de meilleures solutions aux alentours). Les phases de sélection et de reproduction correspondent typiquement à de l'exploitation. Les phases d'initialisation et de mutation correspondent à de l'exploration [2].

2.3 Modélisation informatique

Le schéma global d'un algorithme évolutionnaire décrit précédemment nous permet de dégager les blocs essentiels pour une implémentation informatique. Nous devons tout d'abord déterminer comment modéliser les individus. Il nous faut ensuite réaliser une fonction d'évaluation [2] qui indiquera la performance d'un individu. Enfin, il nous faut quatre opérateurs qui agiront sur la population. Ce sont les opérateurs de sélection, de reproduction, de mutation et de remplacement.

2. Appelée également fonction de « fitness ».

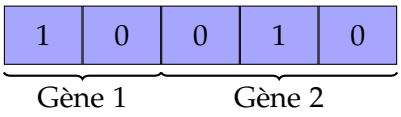

Figure 2.2 – Exemple de modélisation d'un individu dans un algorithme génétique.

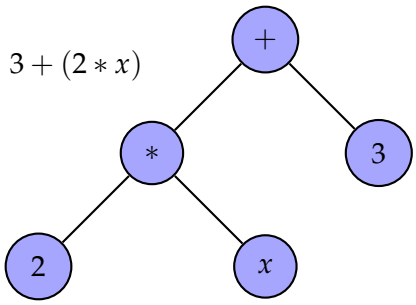

Figure 2.3 – Exemple de modélisation d'un individu dans un programme génétique.

Représentation de la population

Il est donc tout d'abord nécessaire de trouver une représentation informatique des individus. Cette représentation est spécifique au problème donné puisqu'elle en représente une solution. Comme il a été choisi de mettre en pratique la résolution de problèmes de classification via un algorithme génétique et un programme génétique, nous allons détailler les représentations informatiques typiques pour ces deux techniques.

Un individu dans le cas d'un algorithme génétique

Un individu est ici représenté par un champ de bits découpé en gènes (voir figure 2.2). Chaque gène représente un attribut de la solution au problème. Autrement dit chaque gène représente une dimension de l'espace de recherche.

Un individu dans le cas d'un programme génétique

Dans le cas d'un programme génétique, les individus manipulés représentent des programmes informatiques. Une solution au problème posé est ainsi donnée par le résultat de l'exécution d'un individu/programme. La représentation typiquement utilisée est celle d'un arbre (voir figure 2.3). Avec les noeuds (les non terminaux) qui représentent les fonctions et opérateurs et les feuilles (les terminaux) qui représentent les variables et les constantes [4].

$$f(I) = V$$

FIGURE 2.4 – Fonction d'évaluation.

Individu	$f(x_i)$
1	30
2	60
3	10

L'individu 2 a 6 fois plus de chance d'être sélectionné que l'individu 3

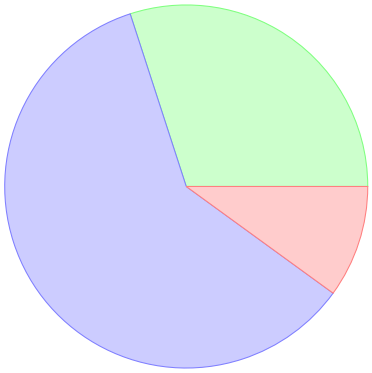

FIGURE 2.5 – Exemple de sélection par roulette artificielle.

Il est à noter que d'autres modèles de représentation existent. Comme par exemple sous la forme d'une suite séquentielle d'instructions. On parle alors de programmation génétique linéaire [3].

Fonction d'évaluation

De manière à être en mesure de classer les individus selon leur performance par rapport au problème posé (afin de pouvoir sélectionner les meilleurs), il est nécessaire de disposer d'une fonction d'évaluation (voir figure 2.4). Cette fonction prend en paramètre un individu I et calcule une valeur V qui représente son niveau de performance.

Cette fonction d'évaluation est entièrement spécifique au problème. C'est par ailleurs très souvent le calcul de cette évaluation qui est le plus coûteux en temps d'exécution dans un algorithme évolutionnaire. Cette fonction peut en effet être extrêmement complexe selon le problème posé.

Opérateur de sélection

Le procédé de sélection des individus destinés à se reproduirent favorise toujours ceux qui sont jugés les plus performants grâce à la fonction d'évaluation. Un exemple de méthode de sélection est celui de la roulette artificielle (voir figure 2.5). Avec cette méthode, la probabilité pour un individu d'être choisi est directement proportionnelle à sa performance.

3. « Linear Genetic Programming » (LGP).

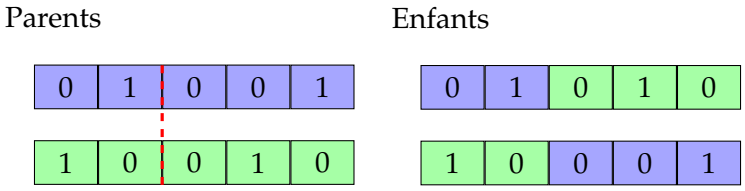

Figure 2.6 – Exemple de méthode de reproduction dans un algorithme génétique.

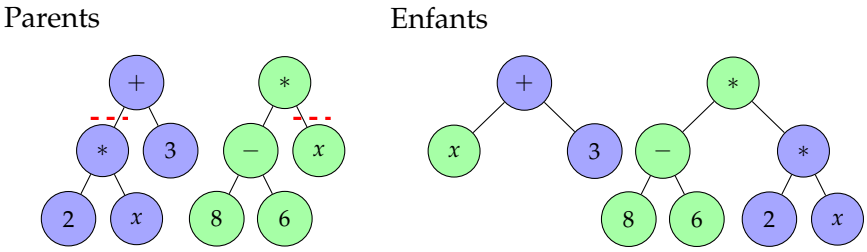

Figure 2.7 – Exemple de méthode de reproduction dans un programme génétique.

Il semble toutefois que cette méthode n'est plus vraiment utilisée aujourd'hui [2]. D'autres méthodes la remplace avec notamment la sélection par le rang ou la sélection par tournoi.

Opérateur de reproduction

Les individus sélectionnés pour être parents se reproduisent deux à deux pour donner naissance à deux nouveaux individus. Ces enfants héritent d'un croisement du patrimoine génétique de leurs parents.

Reproduction dans le cas d'un algorithme génétique

Plusieurs méthodes existent dont notamment celle du croisement à point unique. Un point de coupe est choisi chez les parents et on procède à l'échange des parts ainsi définies pour produire les enfants (voir figure 2.6).

Reproduction dans le cas d'un programme génétique

Nous retrouvons la même technique que pour l'algorithme génétique mis à part que le point de coupe s'applique à une branche de l'arbre qui représente l'individu (voir figure 2.7). De manière à préserver la cohérence sémantique des enfants/programmes ainsi générés, il est possible d'ajouter des contraintes de typage sur les branches et les terminaux de l'arbre [3].

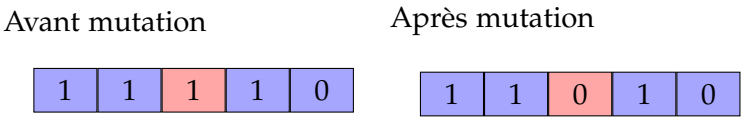

FIGURE 2.8 – Exemple de méthode de mutation dans un algorithme génétique.

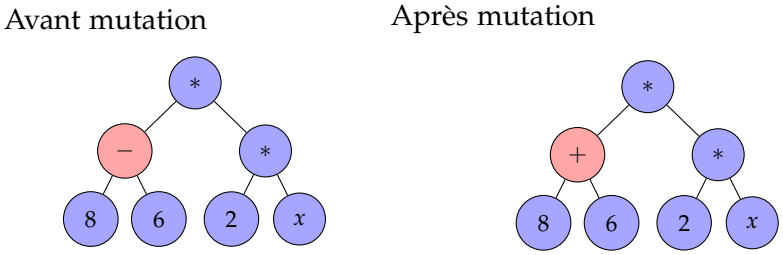

FIGURE 2.9 – Exemple de méthode de mutation dans un programme génétique.

Seuls les types compatibles sont autorisés à interagir. On parle alors de programmation génétique fortement typée [4].

Opérateur de mutation

Un ou plusieurs individus choisis aléatoirement peuvent voir un de leurs gènes muter là aussi de manière aléatoire. Cette mutation peut fortement affecter les caractéristiques de l'individu. Et par là-même ses performances (en moins ou en plus) par rapport au problème donné.

Mutation dans le cas d'un algorithme génétique

Dans le cas où l'individu est représenté sous la forme d'un champ de bits, on choisit de manière aléatoire de modifier l'un de ces bits (voir figure 2.8).

Mutation dans le cas d'un programme génétique

Une méthode consiste à modifier un noeud ou un terminal de l'arbre qui représente l'individu/programme (voir figure 2.9). Il est également possible de couper une branche choisie aléatoirement et développer là aussi de manière aléatoire un sous-arbre à l'endroit de ce point de coupe afin de remplacer la branche supprimée [3].

4. « Strongly Typed Genetic Programming » (STGP).

Opérateur de remplacement

L'opérateur de remplacement détermine la composition finale de la génération suivante. On peut distinguer deux principaux types de méthodes. Le premier type, que l'on peut appeler remplacement stationnaire, consiste à conserver une taille de population constante. A chaque génération, les enfants remplacent soit l'intégralité des parents soit une partie seulement. Dans ce dernier cas, les meilleurs parents sont conservés pour la génération suivante de manière à garder la même taille de population. Le second type, que l'on peut appeler remplacement élitiste, consiste à avoir une taille de population croissante. On conserve toujours le meilleur individu. Un enfant est inclu dans la génération suivante uniquement s'il est au moins meilleur que le moins performant des individus de la génération parente. On peut ensuite imaginer toute une palette de variations entre ces deux principaux procédés de remplacement.

CHAPITRE 3

Les problèmes de classification

Les problèmes de classification consistent à classer des données selon des classes. On distingue la classification supervisée et la classification non supervisée.

3.1 Classification supervisée

Les différentes catégories dans lesquelles classer les données sont connues à l'avance. Dans ce cas de figure, le problème consiste à découvrir les règles de classement qui permettent de ranger une donnée dans telle ou telle catégorie. Il sera ensuite possible d'utiliser ces règles pour classer des données futures non encore disponibles.

Les problèmes suivants sont des exemples de classification supervisée [5] :

— La reconnaissance de formes dans une image (reconnaissance et identification de caractères manuscrits par exemple) ou une vidéo.
— Reconnaissance de parole.
— Reconnaissance de l'auteur ou de la thématique d'un texte.
— Détection des spams.
— Aide au diagnostic médical.
— Détection de gènes dans une séquence ADN.
— Attribution ou non d'un prêt à un client d'une banque.
— La classification de produits boursiers pour tenter de savoir s'ils sont ou non sous-évalués.

Parmi les méthodes de résolution utilisées en dehors des algorithmes évolutionnaires, il existe :

— La méthode des k plus proches voisins.

— Le classifieur Bayésien naïf.
— Les arbres de décision (ID3, C4.5, CART...).
— Les réseaux de neurones.
— Les machines à vecteurs de support (« Support Vector Machine »).

3.2 Classification non supervisée

Cette fois les catégories sont inconnues. Le problème consiste ici à classer les données dans une partition de sous-ensembles qui ne sont pas connus à l'avance. Il est ainsi possible de parvenir à constituer des groupes à la fois homogènes et distincts. Si toutes les données d'un groupe sont considérées comme similaires, on a un groupe homogène. Si toutes les données d'un groupe sont considérées comme différentes des autres, on a un groupe distinct.

Les applications possibles sont notamment [6] :

— En marketing, pour découvrir des groupes de clients aux comportements similaires.
— En biologie, pour la classification des plantes et des animaux.
— Dans le domaine des assurances, pour identifier les assurés les moins rentables ou pour identifier les cas possibles de fraudes.
— Dans la gestion des villes, pour identifier des groupes d'habitations.
— Dans l'étude des tremblements de terre, pour identifier les zones dangereuses.
— Pour la classification des documents disponibles sur le web.

Parmi les méthodes de résolution utilisées en dehors des algorithmes évolutionnaires, il existe :

— L'algorithme des k-moyennes.
— Les modèles de mélanges.
— Le regroupement hiérarchique.
— Le modèle de Markov caché.
— Les cartes auto adaptatives.
— La théorie adaptive de résonance.

CHAPITRE 4

Le logiciel DEAP

DEAP [7] est une librairie développée en langage Python depuis 2010. Sa vocation est de permettre un prototypage rapide de programmes basés sur des algorithmes évolutionnaires. Cette librairie est librement disponible [1] sous licence GNU Lesser General Public License (GNU LGPL) version 3.

Parmi ses principales fonctionnalités, il existe notamment :

— Les algorithmes génétiques peuvent utiliser toutes sortes de représentations pour les individus : listes, tableaux, ensembles, dictionnaires, arbres...
— Possibilité d'utiliser la programmation génétique fortement typée.
— Inclue les algorithmes évolutionnaires $(\mu + \lambda)$ et (μ, λ), la Stratégie d'Evolution avec Adaptation de la Matrice de Covariance [2] et l'optimisation multi-objectifs.
— Inclue la technique de co-évolution (coopérative et compétitive) de populations multiples.
— Gestion d'un « Hall of Fame » pour conserver les meilleurs individus au fil des générations successives.
— Possibilité de définir des sauvegardes régulières du système évolutionnaire.
— Module d'indicateurs de performance (« benchmarks »).

4.1 Prérequis et installation

DEAP nécessite au minimum la version 2.6 du langage Python mais la version 2.7 est recommandée afin de disposer de l'ensemble des fonction-

1. `https://pypi.python.org/pypi/deap/`
2. CMA-ES pour « Covariance Matrix Adaptation Evolution Strategy »

nalités. DEAP est par ailleurs compatible avec la version 3 de Python. Il est également recommandé d'installer les librairies « Numpy » [3] (requis pour CMA-ES) ainsi que « matplotlib » [4] (pour une visualisation des résultats).

La procédure d'installation recommandée pour DEAP est d'utiliser « easy_install » ou bien « pip ». Les commandes à utiliser sont d'une simplicité enfantine :

```
easy_install deap
```

ou bien

```
pip install deap
```

Dans le cadre du développement des scripts utilisés pour ce projet, l'installation de DEAP via l'utilitaire « pip » sur un système FreeBSD 9.2 s'est déroulée sans aucun problème.

3. http://www.numpy.org/
4. http://matplotlib.org/

CHAPITRE 5

Mise en pratique

5.1 Classification supervisée

Définition du problème

L'objectif de l'algorithme génétique est de découvrir des règles de classification pour un ensemble de données. Ces données proviennent du fichier « Zoo » disponible sur le site de l'Université de Californie [1]. Il s'agit de différents animaux classés en familles. Les caractéristiques de ces données sont les suivantes :

— Nombre d'éléments : 101.
— Nombre de classes : 7.
— Nombre d'attributs : 16.

En plus du nom de l'animal et de la famille à laquelle il appartient, on dispose de 15 attributs booléens (poils, plumes, ovipare, produit du lait, animal volant, animal aquatique, prédateur, possède des dents, vertébré, possède des poumons, vénimeux, nageoires, possède une queue, domestique, de la taille d'un chat) et d'un attribut numérique (nombre de pattes) qui peut prendre les valeurs 0, 2, 4, 5, 6 ou 8.

Algorithme

La technique utilisée pour la résolution est un algorithme génétique qui permet de découvrir un ensemble de règles SI-ALORS [8] qui conduisent à une classe donnée.

1. http ://archive.ics.uci.edu/ml/datasets/Zoo

Représentation d'un individu

Le chromosome d'un individu est divisé en autant de gènes qu'il existe d'attributs. Chaque gène correspond à une condition SI et l'individu correspond à l'ensemble des conditions SI d'une règle. La partie ALORS de la règle n'a pas besoin d'être codée au niveau du chromosome car l'algorithme recherche les règles pour une seule classe à la fois. Ainsi, pour le problème présenté ici, il sera nécessaire d'exécuter au moins 7 fois l'algorithme si l'on souhaite découvrir les règles pour l'ensemble des classes.

Chaque gène est sous-divisé en trois champs : un poids, un opérateur et une valeur. Le champ poids est un nombre réel compris entre 0 et 1. Il indique si l'attribut correspondant est présent dans la règle. Si le poids est inférieur ou égal à une valeur fixée par l'utilisateur, la condition est supprimée de la règle. Le champ opérateur indique le type d'opérateur de comparaison à utiliser (égal, différent). Le champ valeur contient une des valeurs autorisées pour l'attribut. Cette valeur est codée sous la forme d'une liste de 0 et de 1. Le nombre de bits utilisés est proportionnel au nombre de valeurs possibles pour l'attribut.

Pour le problème décrit ici, ces trois champs sont codés ainsi (voir également l'exemple en figure 5.1) :

— La valeur limite pour le champ poids est fixé à 0.3. Par conséquent si la valeur du champ est inférieure ou égale à cette valeur, la condition ne sera pas active dans la règle.
— Si le champ opérateur a la valeur 0, il s'agit de l'opérateur de non égalité. Si la valeur est 1, il s'agit de l'opérateur d'égalité.
— Pour le champ valeur, deux cas sont possibles. Dans le cas où l'attribut est de type booléen, si le champ est [0, 1] alors la valeur du booléen est « Faux ». Si le champ est [1, 0], la valeur du booléen est « Vrai ». Dans le cas où l'attribut est celui qui indique le nombre de pattes, six valeurs sont possibles : 0, 2, 4, 5, 6 ou 8. La position du bit égal à 1 dans le champ détermine alors laquelle de ces valeurs est à prendre en compte. Par exemple, si le champ est [0, 0, 0, 1, 0, 0], le nombre de pattes est égal est 5.

Fonction d'évaluation

A chaque exécution de l'algorithme, on recherche les règles uniquement pour une classe donnée. Chaque exécution va donc chercher à résoudre un problème à deux classes. Soit il s'agit de la classe donnée, soit ce n'est pas la classe donnée.

Afin de déterminer la performance d'un individu, chaque élément (chaque animal) du jeu de données lui est présenté pour établir un classement. Quatre

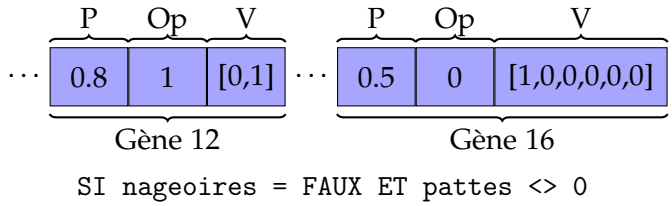

FIGURE 5.1 – Exemple d'un individu dans le cas du problème « Zoo » (les gènes non représentés ont un poids <= 0.3).

	classe réelle = classe donnée	classe réelle <> classe donnée
animal compatible avec règle	vp	fp
animal non compatible avec règle	fn	vn

FIGURE 5.2 – Types de résultats pour le classement d'un élément.

types de résultats sont alors possibles :

- Vrai positif (vp) : les caractéristiques de l'animal respectent la règle et la classe réelle correspond à la classe donnée pour l'exécution de l'algorithme.
- Faux positif (fp) : les caractéristiques de l'animal respectent la règle et la classe réelle ne correspond pas à la classe donnée pour l'exécution de l'algorithme.
- Vrai négatif (vn) : les caractéristiques de l'animal ne respectent pas la règle et la classe réelle ne correspond pas à la classe donnée pour l'exécution de l'algorithme.
- Faux négatif (fn) : les caractéristiques de l'animal ne respectent pas la règle et la classe réelle correspond à la classe donnée pour l'exécution de l'algorithme.

Ce que l'on peut résumer par la figure 5.2.

Le calcul de la fonction d'évaluation se base sur le nombre de fois où ces différents types de résultats se produisent pour l'individu évalué. On va ainsi combiner les indicateurs dits de sensibilité (équation 5.1) et de spécificité (équation 5.2) pour obtenir la mesure d'évaluation (équation 5.3).

$$Se = vp/(vp + fn) \tag{5.1}$$

$$Sp = vn/(vn + fp) \tag{5.2}$$

$$Evaluation = Se * Sp \tag{5.3}$$

Opérateurs

Les opérateurs de reproduction et de sélection utilisent des mécanismes classiquement présents dans les algorithmes génétiques. La reproduction s'effectue par croisement à deux points de coupe. La sélection s'effectue par tournois à trois individus.

La mutation est par contre adaptée au génotype des individus. Il y a ainsi trois opérateurs de mutation, un pour chaque champ des gènes. Chacun de ces opérateurs a 30 % de chance de s'appliquer à un individu. Le mécanisme de mutation pour le champ poids consiste à additionner ou à soustraire une valeur aléatoire à la valeur courante dans la limite du domaine autorisé (entre 0 et 1). Le mécanisme de mutation pour le champ opérateur consiste à inverser l'opérateur courant puisque seules deux valeurs sont possibles (égalité ou non égalité). Le mécanisme de mutation pour le champ valeur consiste dans le cas d'un attribut booléen à inverser la valeur courante. Dans le cas de l'attribut qui indique le nombre de pattes, il consiste à choisir aléatoirement une autre valeur parmi toutes celles possibles.

Résultats

Le code source complet de l'algorithme génétique est présenté en annexe A. La population est fixée à 300 individus. L'exécution se déroule sur 50 générations. L'essai porte ici à titre d'exemple sur la classe des batraciens (classe n° 5). La figure 5.3 montre les trois meilleurs individus obtenus. On voit que ces règles correspondent effectivement à des caractéristiques discriminantes pour les animaux batraciens.

Le graphique de la figure 5.4 montre l'évolution de la moyenne des évaluations pour la population ainsi que l'évolution du maximum des évaluations pour la population. On constate que la performance moyenne des individus progresse peu jusqu'à la dixième génération pour ensuite connaître une forte progression entre la dixième et la vingtième génération. Par ailleurs, il suffit de moins de vingt générations pour obtenir au moins un individu avec une évaluation de valeur maximale.

```
Individu numéro 1 (fitness = 1.0)
   plumes    = faux   ET ovipare     = vrai
ET lait      = faux   ET aquatique   = vrai
ET vertébré  = vrai   ET poumons     = vrai
ET nageoires = faux

Individu numéro 2 (fitness = 1.0)
   ovipare   = vrai   ET aquatique   = vrai
ET dents     = vrai   ET vertébré    = vrai
ET nageoires = faux   ET taille chat = faux
ET pattes   <> 0

Individu numéro 3 (fitness = 1.0)
   ovipare   = vrai   ET aquatique   = vrai
ET dents     = vrai   ET vertébré    = vrai
ET nageoires = faux   ET taille chat = faux
ET pattes   <> 8
```

FIGURE 5.3 – Règles obtenues après exécution de l'algorithme.

5.2 Classification non supervisée

Définition du problème

L'objectif du programme génétique est de découvrir des sous-ensembles de données parmi le même fichier « Zoo » utilisé dans le cadre du problème de classification supervisée (voir la section 5.1 page 19). On supprime simplement de ce fichier la colonne qui indique la classe à laquelle appartient l'animal.

Algorithme

La technique utilisée pour la résolution est un programme génétique qui permet de découvrir des groupes de données plus ou moins homogènes et plus ou moins distincts [9].

Représentation d'un individu

Un individu représente une solution au problème sous la forme d'une arborescence de formules logiques. Chaque formule logique est constituée d'un nombre variable de prédicats.

Pour ce problème, on définit ainsi :

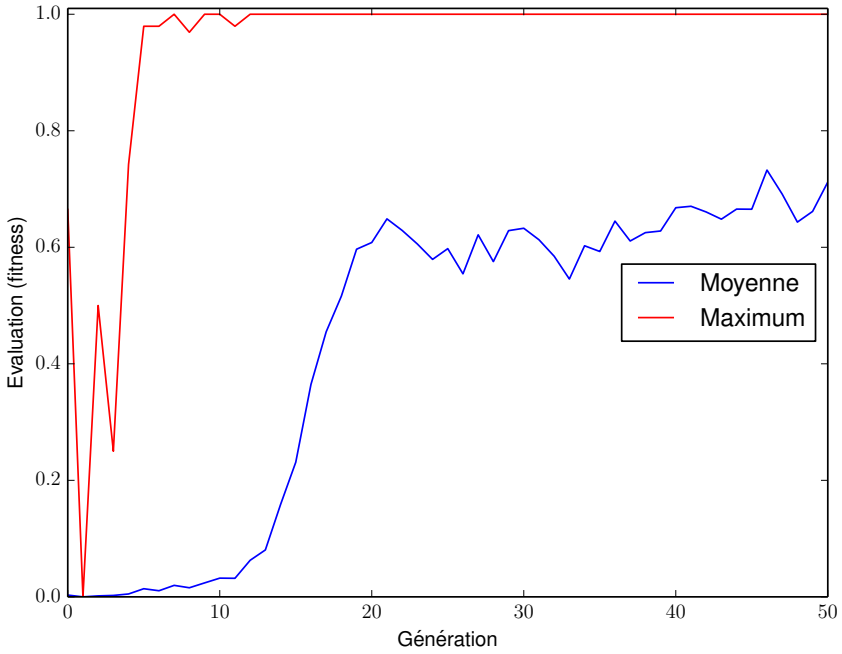

Figure 5.4 – Performance des individus au cours des générations.

— Deux fonctions de comparaison (égalité, inégalité) entre un attribut de type booléen et une valeur booléenne.
— Deux fonctions de comparaison (égalité, inégalité) entre un attribut de type entier et une valeur entière.
— Un opérateur ET logique.
— Un opérateur de sélection SI . . .ALORS . . .SINON.
— Des constantes et des terminaux aléatoires utilisés pour les valeurs de comparaison.

Fonction d'évaluation

La fonction d'évaluation (équation 5.4) se base sur une mesure d'homogénéité $\mathcal{H}$ et une mesure de séparabilité $\mathcal{S}$ des sous-ensembles de données. Le coefficient μ permet de faire varier le poids relatif des deux mesures.

$$Evaluation = \mathcal{H} + \mu \cdot \mathcal{S} \tag{5.4}$$

Pour calculer les valeurs de $\mathcal{H}$ et de $\mathcal{S}$, il est d'abord nécessaire de définir la notion de distance entre deux éléments du jeu de données. Dans le cas présent, les attributs représentent soit des booléens soit une catégorie (pour le

nombre de pattes). On peut par conséquent utiliser une distance de Hamming pour tous les attributs [2]. Avec $\delta(x_j, y_j)$ défini comme la distance de Hamming sur l'attribut j de x et y qui sont deux éléments du jeu de données et l défini comme étant le nombre d'attributs, on utilise l'équation 5.5 pour calculer la distance entre deux éléments.

$$d(X, Y) = \frac{1}{l} \cdot \sum_{j=1}^{l} (\delta(x_j, y_j)) \tag{5.5}$$

Avec w_k défini comme la cardinalité d'un sous-ensemble C_k et P_k défini comme le centroïd du sous-ensemble, l'homogénéité de ce sous-ensemble peut être calculé avec l'équation 5.6.

$$\mathcal{H}(C_k) = -\frac{\sum_{X \in C_k} [d(X, P_k)]}{w_k} \tag{5.6}$$

Dans le calcul de la fonction d'évaluation, la composante $\mathcal{H}$ correspond à la moyenne pondérée de l'homogénéité des sous-ensembles (équation 5.7). La composante $\mathcal{S}$ correspond à la moyenne pondérée des distances entre les centroïds des sous-ensembles (équation 5.8).

$$\mathcal{H} = \frac{\sum_{i=1}^{m} w_i \cdot \mathcal{H}(C_i)}{\sum_{i=1}^{m} w_i} \tag{5.7}$$

$$\mathcal{S} = \frac{\sum_{i=1}^{m-1} \sum_{j=i+1}^{m} w_i \cdot w_j \cdot d(C_i, C_j)}{\sum_{i=1}^{m-1} \sum_{j=i+1}^{m} w_i \cdot w_j} \tag{5.8}$$

Opérateurs

Les opérateurs de reproduction et de sélection utilisent des mécanismes classiquement présents dans les programmes génétiques. La reproduction s'effectue par croisement à un point de coupe. La sélection s'effectue par tournois à dix individus.

La mutation utilise également un opérateur standard. On sélectionne aléatoirement un noeud de l'arbre. Ce noeud est remplacé par un sous-arbre généré de manière aléatoire.

Résultats

Une version fonctionnelle mais non aboutie du code source complet du programme génétique est présenté en annexe B. A titre indicatif, le résul-

2. Pour l'attribut portant sur le nombre de pattes, on considère que la distance est la même quelle que soit la valeur de différence entre ce nombre. Ainsi, pour cet attribut, la distance entre un animal avec 0 patte et un animal avec 8 pattes et la distance entre un animal avec 2 pattes et un animal avec 4 pattes est strictement identique.

```
Classe 1000 : abeille(6), mouche(6), mite(6), guêpe(6)
Classe 1001 : grenouille(5), scorpion(7), serpent_marin(3),
              méduse(7)
Classe 1002 : dauphin(1), vison(1), ornithorynque(1),
              marsouin(1), phoque(1), otarie(1)
Classe 1003 : palourde(7), crabe(7), écrevisse(7), homard(7),
              poulpe(7), étoile de mer(7)
Classe 1004 : poulet(2), colombe(2), perruche(2), tortue(3)
Classe 1005 : canard(2), mouette(2), manchot(2),
              oiseau_bec_en_ciseaux(2), oiseau_stercoraire(2),
              cygne(2)
Classe 1006 : antilope(1), buffle(1), cerf(1), éléphant(1),
              roussette(1), girafe(1), lièvre(1), oryx(1),
              écureuil(1), chauve_souris_vampire(1),
              campagnol(1), wallaby(1)
Classe 1007 : puce(6), moucheron(6), coccinelle(6), limace(7),
              termite(6), ver(7)
Classe 1008 : grenouille(5), crapaud(5)
Classe 1009 : verrat(1), guépard(1), léopard(1), lion(1),
              lynx(1), taupe(1), mangouste(1), opossum(1),
              putois(1), puma(1), raton_laveur(1), loup(1)
Classe 1010 : perche(4), carpe(4), poisson_chat(4), chevaine(4),
              chien_de_mer(4), églefin(4), hareng(4), triton(5),
              brochet(4), piranha(4), hippocampe(4), orvet(3),
              sole(4), reptile_sphénodon(3), thon(4)
Classe 1011 : veau(1), cochon_inde(1), chèvre(1), hamster(1),
              poney(1), chat(1), renne(1)
Classe 1012 : vipère(3), raie(4)
Classe 1013 : oryctérope(1), ours(1), gorille(1)
Classe 1014 : corneille(2), flamant(2), faucon(2), kiwi(2),
              alouette(2), autruche(2), faisan(2), nandou(2),
              moineau(2), vautour(2), roitelet(2)
Classe 1015 : fille(1)
```

Figure 5.5 – Les 16 sous-ensembles obtenus après exécution du programme génétique. Le nombre entre parenthèses indique la classe effectivement indiquée dans le jeu de données original. Par exemple, l'animal « abeille » appartient à la classe n° 6 dans le fichier d'origine.

tat obtenu avec ce programme (population de 1600 individus et exécution pendant 200 générations) est illustré sur la figure 5.5. Bien que la fonction d'évaluation utilise les équations décrites dans [9] (voir section 5.2 page 24), le programme ne respecte pas dans son intégralité l'algorithme. En particulier, le nombre total des sous-ensembles recherchés est fixé au départ de l'exécution via la plage de valeurs pour la constante « randF » (ligne 92 du script principal de l'annexe B). Alors que l'algorithme original permet d'avoir un nombre dynamique de sous-ensembles.

CHAPITRE 6

Conclusion

Pour la mise en pratique de la résolution du problème de classification supervisée avec un algorithme génétique, DEAP s'est avéré parfaitement adapté. Il a été simple de capitaliser sur les fonctions standards qu'il intègre et de spécialiser certaines parties de l'algorithme lorsqu'il était nécessaire d'avoir un comportement spécifique. La simplicité d'utilisation du langage Python et son côté dynamique sont par ailleurs dans ce contexte des atouts. J'ai pu ainsi développer un programme fonctionnel relativement rapidement.

Pour la tentative de résolution du problème de classification non supervisée avec un programme génétique, l'essai s'est avéré infructueux. D'une part, l'algorithme m'est apparu plus difficile à appréhender. Et par conséquent plus difficile à traduire en programme informatique. D'autre part, l'utilisation de DEAP dans ce cadre s'est avéré beaucoup plus délicat. L'absence de typage statique du langage Python m'est apparu ici comme une difficulté supplémentaire. Par ailleurs, j'ai trouvé que DEAP manque de documentations et d'exemples pour la partie programmation génétique. Il serait certainement intéressant de tester une autre librairie pour la réalisation de ce programme génétique. En particulier une bibliothèque disponible pour un langage typé comme Java par exemple. On pourrait alors mieux distinguer dans quelle mesure l'adéquation de la librairie, la nature du langage et la complexité de l'algorithme peuvent jouer sur la facilité ou la difficulté du développement.

ANNEXE A

Code pour l'algorithme génétique

A.1 Script principal

```
1  # -*- coding: utf-8 -*-
2
3  import random
4  import array
5  import numpy
6  from collections import OrderedDict
7
8  # Import des fonctions DEAP
9  from deap import base
10 from deap import creator
11 from deap import tools
12
13 # Import des fonctions spécifiques pour ce script
14 import algoFunctions
15
16 # On cherche à maximiser la fonction d'évaluation
17 creator.create("FitnessMax", base.Fitness, weights=(1.0,))
18 creator.create("Individual", list, fitness=creator.FitnessMax)
19
20 toolbox = base.Toolbox()
21
22 # Définition d'un individu
23 toolbox.register("individual", algoFunctions.generate)
24
25 # Définition de la population
```

```
26 toolbox.register("population", tools.initRepeat, list,
27         toolbox.individual)
28
29 # Définition des opérateurs à appliquer
30 toolbox.register("evaluate", algoFunctions.fitness)
31 # Crossover sur 2 points de coupe
32 toolbox.register("mate", tools.cxTwoPoint)
33 # Mutation spécifique
34 toolbox.register("mutate", algoFunctions.mutate)
35 # Sélection des individus par tournois à 3 participants
36 toolbox.register("select", tools.selTournament, tournsize=3)
37
38 # Statistiques sur le déroulement de l'algorithme
39 stats = tools.Statistics(key=lambda ind: ind.fitness.values)
40 stats.register("avg", numpy.mean)
41 stats.register("std", numpy.std)
42 stats.register("min", numpy.min)
43 stats.register("max", numpy.max)
44
45 # "Hall of Fame", on conserve les 30 meilleurs individus
46 hof = tools.HallOfFame(30)
47
48 def main():
49         # Compteur de générations
50         countGen = 0
51
52         # Probalité de crossover 100%
53         CXPB = 1
54         # Probabilité de mutation 100%
55         # (car la probabilité est gérée dans algoFunctions.mutate)
56         MUTPB = 1
57         # Nombre de générations produites
58         NGEN = 50
59         # Nombre d'individus dans une population
60         NPOP = 300
61
62         # Population initiale
63         pop = toolbox.population(n=NPOP)
64         # Evaluation de la population initiale
65         fitnesses = map(toolbox.evaluate, pop)
66         for ind, fit in zip(pop, fitnesses):
67                 ind.fitness.values = fit
68
69         # Mise à jour des statistiques et du hall of fame
70         record = stats.compile(pop)
71         hof.update(pop)
72
```

```
73          # Pour la génération du graphique
74          lX = []
75          lY1 = []
76          lY2 = []
77          lX.append(countGen)
78          lY1.append(record['avg'])
79          lY2.append(record['max'])
80
81          # Boucle de production des générations
82          for g in range(NGEN):
83                  countGen += 1
84
85          # Sélection des nouveaux individus
86                  offspring = toolbox.select(pop, len(pop))
87          # Clonage de ces individus
88          # (qui seront modifiés par crossover/mutation)
89                  offspring = map(toolbox.clone, offspring)
90
91          # Application du crossover sur les nouveaux
92          # individus clonés précédemment
93                  for child1, child2 in zip(offspring[::2], offspring[1::2]):
94                          if random.random() < CXPB:
95                                  toolbox.mate(child1, child2)
96                                  del child1.fitness.values
97                                  del child2.fitness.values
98
99          # Application de la mutation sur les nouveaux
100         # individus clonés précédemment
101                 for mutant in offspring:
102                         if random.random() < MUTPB:
103                                 toolbox.mutate(mutant)
104                                 del mutant.fitness.values
105
106         # Evaluation des nouveaux individus
107                 invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
108                 fitnesses = map(toolbox.evaluate, invalid_ind)
109                 for ind, fit in zip(invalid_ind, fitnesses):
110                         ind.fitness.values = fit
111
112         # On remplace l'ancienne génération par la nouvelle
113                 pop[:] = offspring
114
115         # Mise à jour des statistiques et du Hall of fame
116                 record = stats.compile(pop)
117                 hof.update(pop)
118
119         # Ajoute points X, Y pour graphique
```

```
120                 lX.append(countGen)
121                 lY1.append(record['avg'])
122                 lY2.append(record['max'])
123 
124         return pop, lX, lY1, lY2
125 
126 # Programme principal
127 if __name__ == "__main__":
128         pop, lX, lY1, lY2 = main()
129 
130     # On affiche les 3 meilleurs individus du Hall of fame
131         l = []
132         for e in hof:
133                 s = algoFunctions.stringRulesFitness(e)
134                 if (s not in l):
135                         l.append(s)
136 
137         length = len(l)
138         if (length > 3):
139                 length = 3
140 
141         for i in range(length):
142                 print (l[i])
143 
144         # On affiche le graphique
145         import matplotlib.pyplot as plt
146         import matplotlib.backends.backend_pdf as pdf
147         plt.rcParams['text.usetex'] = True
148         plt.rcParams['text.latex.unicode']=True
149         fig, ax1 = plt.subplots()
150         line1 = ax1.plot(lX, lY1, "b-", label="Moyenne")
151         line2 = ax1.plot(lX, lY2, "r-", label="Maximum")
152         ax1.set_xlabel('Génération')
153         ax1.set_ylabel("Evaluation (fitness)")
154         lns = line1 + line2
155         labs = [l.get_label() for l in lns]
156         ax1.legend(lns, labs, loc="center right")
157         plt.ylim([0,1.01])
158         ax1.set_autoscaley_on(False)
159         plt.plot()
160         plt.savefig('graph_fitness.pdf', format='pdf')
```

A.2 Fonctions auxiliaires (algoFunctions)

```
1 # -*- coding: utf-8 -*-
2 
```

```
3  import random
4  import csv
5
6  # Import des fonctions DEAP
7  from deap import creator
8
9  # Renvoie le nombre de pattes pour un individu
10 def nbLegs(v):
11         for i in range(6):
12                 if v[i] == 1:
13                         if (i <= 2):
14                                 return (i * 2)
15                         else:
16                                 if (i == 3):
17                                         return (5)
18                                 if (i == 4):
19                                         return (6)
20                                 if (i == 5):
21                                         return (8)
22
23 # Représentation d'un individu en clair
24 def stringIndividual(ind):
25         attributes = ["hair", "feathers", "eggs", "milk", "airborne",
26                 "aquatic", "predator", "toothed", "backbone", "breathes",
27                 "venomous", "fins", "tail", "domestic", "catsize"]
28         s = ""
29         def trueFalse(v1, v2):
30                 if (v1 == 1 and v2 == 0):
31                         return "true"
32                 else:
33                         return "false"
34         def operator(v):
35                 if (v == 0):
36                         return "<>"
37                 else:
38                         return "="
39         for i in range(15):
40                 s += "weight: " + str(ind[i*4]) + "\n"
41                 s += "operator: " + operator(ind[(i*4) + 1]) + "\n"
42                 s += attributes[i] + ": "
43                 s += trueFalse(ind[(i*4) + 2], ind[(i*4) + 3]) + "\n"
44         s += "weight: " + str(ind[-8]) + "\n"
45         s += "operator: " + operator(ind[-7]) + "\n"
46         s += "legs: " + str(nbLegs(ind[-6:])) + "\n"
47         return s
48
49 # Représentation règles et fitness d'un individu en clair
```

```
50 def stringRulesFitness(ind):
51         threshold = 0.3
52         attributes = ["poils", "plumes", "ovipare", "lait",
53                 "volant", "aquatique", "prédateur", "dents",
54                 "vertébré", "poumons", "vénimeux", "nageoires",
55                 "queue", "domestique", "taille chat"]
56         rules = []
57         def trueFalse(v1, v2):
58                 if (v1 == 1 and v2 == 0):
59                         return "vrai"
60                 else:
61                         return "faux"
62         def operator(v):
63                 if (v == 0):
64                         return "<>"
65                 else:
66                         return "="
67         for i in range(15):
68                 weight = ind[i*4]
69                 op = operator(ind[(i*4) + 1])
70                 val = trueFalse(ind[(i*4) + 2], ind[(i*4) + 3])
71                 # si poids <= threshold
72                 # on ne tient pas compte de la condition
73                 if (weight > threshold):
74                         s = op + " " + val
75                         if (op == "<>"):
76                                 if (val == "vrai"):
77                                         s = "= faux"
78                                 else:
79                                         s = "= vrai"
80                         rules.append(attributes[i] + " " + s)
81         weight = ind[-8]
82         op = operator(ind[-7])
83         val = nbLegs(ind[-6:])
84         # si poids <= threshold
85         # on ne tient pas compte de la condition
86         if (weight > threshold):
87                 rules.append("pattes " + op + " " + str(val))
88         s = "Individu : \n"
89         s += "  Evaluation (fitness) : "
90                 + str(ind.fitness.values[0]) + "\n"
91         s += "  Règle : \n" + "\nET ".join(rules) + "\n"
92         return s
93
94 # Génération d'un individu
95 def generate():
96         l = []
```

```
97          # Attributs vrai/faux
98          for i in range(15):
99                  gene = []
100                 # Poids
101                 rand = random.uniform(0, 1)
102                 gene.append(rand)
103                 # Opérateur (0 : <>, 1 : =)
104                 rand = random.randint(0, 1)
105                 gene.append(rand)
106                 # Valeurs
107                 rand = random.randint(0, 1)
108                 if rand == 1: # Vrai
109                         gene.extend([1, 0])
110                 else: # Faux
111                         gene.extend([0, 1])
112                 l.extend(gene)
113         # Attribut 6 valeurs
114         gene = []
115         # Poids
116         rand = random.uniform(0, 1)
117         gene.append(rand)
118         # Opérateur (0 : <>, 1 : =)
119         rand = random.randint(0, 1)
120         gene.append(rand)
121         # Valeur
122         rand = random.randint(0, 5)
123         tmp = [0, 0, 0, 0, 0, 0]
124         tmp[rand] = 1
125         gene.extend(tmp)
126         l.extend(gene)
127         ind = creator.Individual(l)
128         return ind
129
130 # Opérateur de mutation sur un individu
131 def mutate(ind):
132         mut = False
133         mutProb = 0.3 # 30% pour chaque type de mutation
134
135         # Poids
136         if random.random() < mutProb :
137                 mut = True
138                 iGene = random.randint(0, 15)
139                 v = random.uniform(0, 1)
140                 op = random.randint(0, 1)
141                 if (op == 0): # soustraction
142                         nv = ind[iGene * 4] - v
143                         if (nv < 0):
```

```
144                                 nv = 0
145                 else: # addition
146                         nv = ind[iGene * 4] + v
147                         if (nv > 1):
148                                 nv = 1
149                 ind[iGene * 4] = nv
150         # Opérateur
151         if random.random() < mutProb:
152                 mut = True
153                 iGene = random.randint(0, 15)
154                 if (ind[(iGene * 4) + 1] == 0):
155                         ind[(iGene * 4) + 1] = 1
156                 else:
157                         ind[(iGene * 4) + 1] = 0
158         # Valeur
159         if random.random() < mutProb:
160                 mut = True
161                 iGene = random.randint(0, 15)
162                 if (iGene < 15):
163                         v1 = ind[(iGene * 4) + 2]
164                         v2 = ind[(iGene * 4) + 3]
165                         if (v1 == 0):
166                                 ind[(iGene * 4) + 2] = 1
167                         else:
168                                 ind[(iGene * 4) + 2] = 0
169                         if (v2 == 0):
170                                 ind[(iGene * 4) + 3] = 1
171                         else:
172                                 ind[(iGene * 4) + 3] = 0
173                 else: # gène "legs"
174                         pos1 = 0
175                         for i in range(-6, 0):
176                                 if ind[i] == 1:
177                                         pos1 = i
178                         ind[pos1] = 0
179                         r = None
180                         while True:
181                                 r = random.randint(1, 6)
182                                 if r != -pos1:
183                                         break
184                         ind[-r] = 1
185         return(ind)
186
187 # Lecture du fichier CSV des données à classer
188 def readcsv(path):
189         l = []
190         with open(path, 'rb') as f:
```

```
191                 h = csv.reader(f, delimiter=',')
192                 for d in h:
193                         l.append(d)
194         return l
195
196 # Règle de comparaison sur l'attribut "nombre de pattes"
197 def compareRuleLegs(ind, e):
198         result = True
199         # si poids <= threshold on ne tient pas compte de la condition
200         threshold = 0.3
201         weight = ind[-8]
202         operator = ind[-7]
203         legs_ind = nbLegs(ind[-6:])
204         legs_e = int(e)
205         if (weight > threshold):
206                 if (operator == 1): # Opérateur égalité
207                         if (legs_ind == legs_e):
208                                 result = True
209                         else:
210                                 result = False
211                 else: # Opérateur inégalité
212                         if (legs_ind == legs_e):
213                                 result = False
214                         else:
215                                 result = True
216         else:
217                 result = True
218         return (result)
219
220 # Règle de comparaison pour les attributs de type booléens
221 def compareRuleBools(ind, e, bools):
222         result = True
223         # si poids <= threshold
224         # on ne tient pas compte de la condition
225         threshold = 0.3
226         def trueFalse(v1, v2):
227                 if (v1 == 1 and v2 == 0):
228                         return True
229                 else:
230                         return False
231         weight = ind[bools * 4]
232         operator = ind[(bools * 4) + 1]
233         val_ind = trueFalse(ind[(bools * 4) + 2],
234                 ind[(bools * 4) + 3])
235         val_e = False
236         if (int(e) == 1):
237                 val_e = True
```

```
238         if (weight > threshold):
239                 if (operator == 1): # Opérateur égalité
240                         if (val_ind == val_e):
241                                 result = True
242                         else:
243                                 result = False
244                 else: # Opérateur inégalité
245                         if (val_ind == val_e):
246                                 result = False
247                         else:
248                                 result = True
249         else:
250                 result = True
251         return (result)
252
253 # Fonction d'évaluation d'un individu
254 def fitness(ind):
255         tp = 0 # Vrai positif
256         fp = 0 # Faux positif
257         tn = 0 # Vrai négatif
258         fn = 0 # Faux négatif
259         l = readcsv("zoo.data")
260         lines = 0
261         # Classe pour laquelle l'algorithme
262         # va rechercher les règles de classement
263         ## searchedClass = 1 # Mammifères
264         ## searchedClass = 4 # Poissons
265         searchedClass = 5 # Batraciens
266         ## searchedClass = 6 # Insectes
267         for d in l:
268                 lines += 1
269                 fields = 0
270                 bools = 0
271                 classIsPredicted = True
272                 classInLine = int(d[17])
273                 for e in d:
274                         fields += 1
275                         if (fields >= 2 and fields <= 17):
276                                 if (fields == 14): # nb pattes
277                                         # on compare avec la règle "legs"
278                                         if (not compareRuleLegs(ind, e)):
279                                                 classIsPredicted = False
280                                                 break
281                                 else:
282                                         # on compare avec la règle correspondante
283                                         if (not compareRuleBools(ind, e, bools)):
284                                                 classIsPredicted = False
```

```
285                                                 break
286                                         bools += 1
287                 if (classIsPredicted and classInLine == searchedClass):
288                         tp += 1
289                 if (classIsPredicted and classInLine != searchedClass):
290                         fp += 1
291                 if ((not classIsPredicted) and classInLine != searchedClass):
292                         tn += 1
293                 if ((not classIsPredicted) and classInLine == searchedClass):
294                         fn += 1
295         se = tp
296         if (tp + fn > 0):
297                 se = float(tp) / float(tp + fn) # Sensibilité
298         sp = tn
299         if (tn + fp > 0):
300                 sp = float(tn) / float(tn + fp) # Spécificité
301         fitness = se * sp
302
303         return fitness, # /!\ : il faut retourner un tuple !
```

ANNEXE B

Version alpha du code pour le programme génétique

B.1 Script principal

```
1  # -*- coding: utf-8 -*-
2
3  import random
4  import operator
5  import csv
6  import itertools
7
8  import numpy
9
10 from deap import algorithms
11 from deap import base
12 from deap import creator
13 from deap import tools
14 from deap import gp
15
16 import progFunctions
17
18 # On lit les données à partir du fichier CSV.
19 data = progFunctions.readcsv("zoo_fr.data")
20 # On transforme les données lues :
21 # - suppression de la première (nom animal) et dernière colonne (classe)
22 #   puisque ces données sont ici inutiles.
23 # - la colonne "nb pattes" est déplacée en dernière colonne.
24 # On fixe le typage des données.
25 nData = []
```

```
for line in data:
        nbColumn = 0
        nLine = []
        for e in line:
                nbColumn += 1
                if (nbColumn > 1 and nbColumn < 18 and nbColumn != 14):
                        v = False
                        if (e == "1" or e == 1):
                                v = True
                        nLine.append(v)
                if (nbColumn == 14):
                        nbLegs = int(e) + 100
        nLine.append(nbLegs)
        nData.append(nLine)

# On définit un nouvel ensemble de primitives (GP fortement typée).
# En entrée, les attributs : 15 booléens et 1 entier (nb pattes).
# En sortie, la classe sous la forme d'un flottant.
args = []
args.extend(itertools.repeat(bool, 15))
args.append(int)
pset = gp.PrimitiveSetTyped("MAIN", args, float)

# Opérateurs booléens
pset.addPrimitive(operator.and_, [bool, bool], bool)

# Fonction test égalité 2 booléens
def eq_b(left, right):
        if (left == right):
                return True
        return False

# Fonction test inégalité 2 booléens
def neq_b(left, right):
        if (left != right):
                return True
        return False

# Fonction test égalité 2 entiers
def eq_i(left, right):
        if (left == right):
                return True
        return False

# Fonction test inégalité 2 entiers
def neq_i(left, right):
        if (left != right):
```

```
73                  return True
74          return False
75
76  # Opérateurs de comparaisons
77  pset.addPrimitive(eq_b, [bool, bool], bool)
78  pset.addPrimitive(neq_b, [bool, bool], bool)
79  pset.addPrimitive(eq_i, [int, int], bool)
80  pset.addPrimitive(neq_i, [int, int], bool)
81
82  # Fonction if_then_else avec retour float
83  def if_t_e(ifStmt, thenStmt, elseStmt):
84      if ifStmt: return thenStmt
85      else: return elseStmt
86
87  pset.addPrimitive(if_t_e, [bool, float, float], float)
88
89  # Terminaux
90  pset.addEphemeralConstant("randI",
91          lambda: random.randint(100, 108), int)
92  pset.addEphemeralConstant("randF",
93          lambda: float(random.randint(1000, 1015)), float)
94  pset.addTerminal(False, bool)
95  pset.addTerminal(True, bool)
96
97  # On renomme les attributs pour davantage de clarté
98  pset.renameArguments(ARG0="hair")
99  pset.renameArguments(ARG1="feathers")
100 pset.renameArguments(ARG2="eggs")
101 pset.renameArguments(ARG3="milk")
102 pset.renameArguments(ARG4="airborne")
103 pset.renameArguments(ARG5="aquatic")
104 pset.renameArguments(ARG6="predator")
105 pset.renameArguments(ARG7="toothed")
106 pset.renameArguments(ARG8="backbone")
107 pset.renameArguments(ARG9="breathes")
108 pset.renameArguments(ARG10="venomous")
109 pset.renameArguments(ARG11="fins")
110 pset.renameArguments(ARG12="tail")
111 pset.renameArguments(ARG13="domestic")
112 pset.renameArguments(ARG14="catsize")
113 pset.renameArguments(ARG15="legs")
114
115 creator.create("FitnessMax", base.Fitness, weights=(1.0,))
116 creator.create("Individual", gp.PrimitiveTree,
117         fitness=creator.FitnessMax)
118
119 toolbox = base.Toolbox()
```

```
120 toolbox.register("expr", gp.genHalfAndHalf, pset=pset,
121         type_=pset.ret, min_=1, max_=6)
122 toolbox.register("individual", tools.initIterate,
123         creator.Individual, toolbox.expr)
124 toolbox.register("population", tools.initRepeat,
125         list, toolbox.individual)
126 toolbox.register("compile", gp.compile, pset=pset)
127
128 def evalInd(individual):
129         lr = []
130         func = toolbox.compile(expr=individual)
131         for d in nData:
132                 # print d
133                 result = func(*d[:16])
134                 lr.append(result)
135
136         # On construit une liste des clusters
137         lcl = progFunctions.clusterList(lr)
138
139         # On calcule le centroid de chaque cluster
140         lct = progFunctions.centroidList(nData, lcl)
141
142         # On calcule les HCk
143         lhck = progFunctions.lhck(nData, lcl, lct)
144
145         # On calcule le HCL
146         hcl = progFunctions.hcl(lcl, lhck)
147
148         # On calcule le SCL
149         scl = progFunctions.scl(lcl, lct)
150
151         # On calcule le fitness
152         mu = 0.85
153         f = hcl + (mu * scl)
154
155         return f,
156
157
158 toolbox.register("evaluate", evalInd)
159 toolbox.register("select", tools.selTournament, tournsize=10)
160
161 toolbox.register("mate", gp.cxOnePoint)
162 toolbox.register("expr_mut", gp.genHalfAndHalf, min_=0, max_=4)
163 toolbox.register("mutate", gp.mutUniform,
164         expr=toolbox.expr_mut, pset=pset)
165
166 toolbox.decorate("mate",
```

```
167         gp.staticLimit(operator.attrgetter('height'), 6))
168 toolbox.decorate("mutate",
169         gp.staticLimit(operator.attrgetter('height'), 6))
170
171 def main():
172     random.seed(10)
173     pop = toolbox.population(n=1600)
174     hof = tools.HallOfFame(3)
175     stats = tools.Statistics(lambda ind: ind.fitness.values)
176     stats.register("avg", numpy.mean)
177     stats.register("std", numpy.std)
178     stats.register("min", numpy.min)
179     stats.register("max", numpy.max)
180
181     algorithms.eaSimple(pop, toolbox, 0.9, 0.8, 200,
182                 stats, halloffame=hof, verbose=True)
183
184     return pop, stats, hof
185
186 if __name__ == "__main__":
187         import pygraphviz as pgv
188
189         pop, stats, hof = main()
190
191         print "Solution:"
192         i = 0
193         for e in hof:
194                 print
195                 print "Individu: "
196                 print e
197                 print
198
199                 func = toolbox.compile(expr=e)
200                 lr = []
201                 for d in nData:
202                         result = func(*d[:16])
203                         lr.append(result)
204                 dictR = dict()
205                 for r in lr:
206                         dictR[r] = []
207                 k = 0
208                 for d in data:
209                         name = d[0] + "(" + d[17] + ")"
210                         r = lr[k]
211                         dictR[r].append(name)
212                         k += 1
213                 for k in dictR:
```

```
214                         animals = dictR[k]
215                         print "Class ", str(k), ": ", ", ".join(animals)
216                 print ""
217
218                 nodes, edges, labels = gp.graph(e)
219                 g = pgv.AGraph()
220                 g.add_nodes_from(nodes)
221                 g.add_edges_from(edges)
222                 g.layout(prog="dot")
223                 for j in nodes:
224                         n = g.get_node(j)
225                         n.attr["label"] = labels[j]
226                 g.draw("tree" + str(i) + ".pdf")
227                 i += 1
```

B.2 Fonctions auxiliaires (progFunctions)

```
1  # -*- coding: utf-8 -*-
2
3  import csv
4
5  def readcsv(path):
6          l = []
7          with open(path, 'rb') as f:
8                  h = csv.reader(f, delimiter=',')
9                  for d in h:
10                         l.append(d)
11         return l
12
13 def clusterList(lr):
14         l = dict()
15         for e in lr:
16                 l[e] = []
17         i = 0
18         for e in lr:
19                 l[e].append(i)
20                 i += 1
21         return l
22
23 def mostFrequentVal(l):
24         nbCols = len(l[0])
25         mfv = [dict() for x in range(nbCols)]
26         centroid = [0 for x in range(nbCols)]
27         for e in l:
28                 i = 0
29                 for v in e:
```

```
30                          mfv[i][v] = 0
31                          i += 1
32          for j in range(nbCols):
33                  for e in l:
34                          v = e[j]
35                          mfv[j][v] += 1
36          for j in range(nbCols):
37                  nbmax = 0
38                  mostv = 0
39                  for e in l:
40                          v = e[j]
41                          if (mfv[j][v] >= nbmax):
42                                  nbmax = mfv[j][v]
43                                  mostv = v
44                  centroid[j] = mostv
45          return centroid
46
47  def centroidList(data, lcl):
48          l = []
49          for k in lcl:
50                  c = lcl[k]
51                  ldata = []
52                  for i in c:
53                          ldata.append(data[i])
54                  l.append(mostFrequentVal(ldata))
55          return l
56
57  def distance(indA, indB):
58          d = 0
59          l = len(indA)
60          for i in range(l):
61                  vA = indA[i]
62                  vB = indB[i]
63                  if (vA != vB):
64                          d += 1
65          return (float(float(d) / float(l)))
66
67  def lhck(data, lcl, lct):
68          l = []
69          i = 0
70          for k in lcl:
71                  c = lcl[k]
72                  ldata = []
73                  for ind in c:
74                          ldata.append(data[ind])
75                  sumDistance = 0
76                  for pat in ldata:
```

```
77                          centroid = lct[i]
78                          sumDistance += distance(pat, centroid)
79                  i += 1
80                  h = - (float(float(sumDistance) / float(len(ldata))))
81                  l.append(h)
82          return l
83
84  def hcl(lcl, lhck):
85          h = 0
86          i = 0
87          sumL = 0
88          for k in lcl:
89                  c = lcl[k]
90                  l = len(c)
91                  hci = lhck[i]
92                  h += float(float(l) * float(hci))
93                  sumL += l
94                  i += 1
95          h = float(float(h) / float(sumL))
96          return h
97
98  def scl(lcl, lct):
99          s = 0
100         sumL = 0
101         i = 0
102         for ki in lcl:
103                 ci = lcl[ki]
104                 li = len(ci)
105                 cti = lct[i]
106                 j = 0
107                 for kj in lcl:
108                         cj = lcl[kj]
109                         lj = len(cj)
110                         ctj = lct[j]
111                         s += float(float(li) * float(lj)
112                                 * float(distance(cti, ctj)))
113                         sumL += float(float(li) * float(lj))
114                         j += 1
115                 i += 1
116         if (float(sumL) > 0):
117                 s = float(float(s) / float(sumL))
118         return s
```

Références

[1] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Disponible gratuitement en téléchargement : http://cs.gmu.edu/~sean/book/metaheuristics/.

[2] Marc Schoenauer Éric Goubault, Frédéric Nataf. Calcul parallèle. http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/poly/index.html.

[3] Moshe Sipper. *Evolved to Win*. Lulu, 2011. Disponible gratuitement en téléchargement : http://www.lulu.com/shop/moshe-sipper/evolved-to-win/ebook/product-18719826.html.

[4] Pedro G. Espejo, Sebastián Ventura, and Francisco Herrera. A survey on the application of genetic programming to classification. *Trans. Sys. Man Cyber Part C*, 40(2) :121–144, March 2010. http://dx.doi.org/10.1109/TSMCC.2009.2033566.

[5] Sylvain Arlot. Classification supervisée : des algorithmes et leur calibration automatique, 2009. http://www.di.ens.fr/~arlot/enseign/2009Centrale/cours-classif.pdf.gz.

[6] Matteo Matteucci. A tutorial on clustering algorithms. http://home.deib.polimi.it/matteucc/Clustering/tutorial_html/.

[7] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP : Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13 :2171–2175, jul 2012. https://code.google.com/p/deap/.

[8] M. V. Fidelis, H. S. Lopes, A. A. Freitas, and Ponta Grossa. Discovering comprehensible classification rules with a genetic algorithm. In *In Proc. of the 2000 Congress on Evolutionary Computation*, pages 805–810, 2000. http://kar.kent.ac.uk/22013/1/Discovering_comprehensible_classification_rules_with_a_genetic_algorithm.pdf.

[9] I. De Falco, E. Tarantino, A. Della Cioppa, and F. Fontanella. An innovative approach to genetic programming-based clustering. http://www.cs.bham.ac.uk/~wbl/biblio/cache/http___webuser.unicas.it_fontanella_papers_WSC04.pdf.