Évaluation d'agents dans ATOM

M. Gaciarz, P. Mathieu

Résumé

Evaluer des comportements de trading est une tache difficile. On peut souhaiter réaliser diverses expériences afin de réaliser cette évaluation. Sur ATOM, un package permettant de réaliser facilement différents types d'évaluations est proposé. L'une de ses fonctionnalités notable est de proposer de créer, ou d'utiliser facilement, des compétitions écologiques, un des outil utile à l'évaluation. Nous décrivons ici le fonctionnement et l'utilisation de ce package.

1 Compétition écologique

La compétition écologique est une méthode de sélection inspirée de la biologie et du phénomène de sélection naturelle ([**?**],[**?**]). Plusieurs familles co-évoluent dans un même environnement, comme des espèces animales ou végétales partageant un même milieu. Comme dans la nature, leurs populations varient en fonction du temps, de manière à ce que les meilleures familles voient leurs populations augmenter, et les moins bonnes voient leurs populations diminuer. Cette propriété est intéressante car, sur les marchés financiers comme ailleurs, les plus mauvaises stratégies vont avoir tendance à disparaître, contrairement aux meilleures.

Dans le cadre d'ATOM, une compétition est une succession de simulations, et une famille est un ensemble d'agents du même type, parametrés de la même manière. La population totale de la compétition est constante. Après chaque simulation, appelée "génération", la population de chaque famille est réévaluée en fonction du gain de celle-ci. Le gain d'une famille est le gain total de ses représentants durant la simulation.

1.1 Gain d'un agent

Il nous faut donc définir le gain d'un agent. Deux critères sont possibles : soit la liquidité (*cash*) possédée par un agent, soit la valeur estimée de son portefeuille (wealth), somme de sa liquidité et de la valeur estimée de son portefeuille.

$$wealth = cash + \sum_{i=1}^{i \leq titres} prix_i \times nbTitres_i$$

Ce choix est discutable car si l'agent décidait de se retirer du marché, la somme gagnée par la revente des titres ne correspondrait probablement pas à leur valeur estimée. Cependant cette mesure présente l'avantage de prendre en compte l'ensemble des actifs de l'agent. C'est donc cette mesure que nous choisissons d'utiliser.

1.2 Ajustement du gain

Pour que la population totale reste constante, il faut appliquer une règle de proportionnalité sur le score de chaque famille. Mais le gain individuel d'un agent pouvant être négatif, le gain total d'une famille peut l'être également, or une règle de proportionnalité nécessite des valeurs positives. Pour résoudre ce problème, nous proposons donc de soustraire au gain de chaque agent a le gain du plus mauvais agent p (on soustrait un gain négatif, il s'agit donc d'une addition).
$gain_a \geq gain_p \Rightarrow gain_a - gain_p \geq 0$
De cette manière le gain modifié de chaque agent est positif, ainsi que le gain total de chaque famille. Il est alors possible d'appliquer une règle de proportionnalité pour le calcul des populations des différentes familles d'agents.

Le gain total d'une famille f est la somme des gains modifiés des agents de cette famille.

$$gainTot(f) = \sum_{a \in f} (gain_a - gain_p)$$

1.3 Calcul des populations

La compétition a une population totale constante. À chaque simulation, la population d'une famille est proportionnelle à son gain total durant la simulation précédente.

$$pop(f) = \frac{gainTot(f)}{\sum_{i \in familles} gainTot(i)} popTot$$

La population d'une famille d'agents à la fin d'une compétition représente l'adaptation de ce type d'agent à un environnement particulier (les autres familles d'agents en compétition), et son efficacité dans cet environnement.

1.4 Agents ne prenant pas part à la compétition

Il peut arriver dans certaines compétitions que tous les agents attendent que d'autres fassent le premier pas, et où rien ne se passe. Pour pallier ce problème, nous permettons d'ajouter dans les simulations des familles d'agents qui ne prennent pas part à la compétition, c'est à dire des familles dont la population est constante, indépendement de leurs gains. Ceci permettent par exemple de faire vivre la simulation avec des agents tenant le rôle de fournisseurs de liquidité.

1.5 Comment utiliser le paquetage pour l'évaluation d'agents

La compétition est un outil utile à l'évaluation d'agents car elle permet de créer un environnement dynamique, et ainsi de tester la robustesse des stratégies utilisées par les agents. Néanmoins une compétition seule ne suffit pas à l'évaluation des agents. Pour bien éprouver leur robustesse, et donc obtenir une bonne évaluation de leurs performances, il faut réaliser des compétitions nombreuses et variées. Nous proposons ici un protocole d'évaluation s'appuyant sur la compétition écologique.

Soit S l'ensemble des types d'agents à évaluer et E l'ensemble des types d'agents à notre disposition (il est facile d'en concevoir des centaines à partir

des outils fournis par le package `chartists`). A chaque compétition c, on choisit un sous ensemble $E_c \subseteq E$. On crée une famille de population nulle pour chaque type d'agent de l'ensemble $U_c = S \cup E_c$. On incrémente *popTot* fois la population d'une famille choisie au hasard.

On obtient ainsi une compétition de population *popTot*, dont la population est répartie aléatoirement entre différentes familles. Parmi ces familles figurent tous les types issus de S et divers types issus de E. Ceci représente un grand nombre de situations différentes, qui peuvent être initialement déséquilibrées en faveurs de certains types mais qui, en moyenne, n'en avantagent aucun. On prend donc bien en compte la robustesse des agents à la variété des environnements.

On peut donc obtenir une mesure de l'efficacité de chaque famille avec sa population moyenne en fin de compétition sur l'ensemble des compétitions testées. Dans les expériences que nous avons réalisées, des compétitions de 50 générations sont souvent suffisantes pour que la population de chaque famille se stabilise. La famille ayant la plus grande part moyenne de population est la plus efficace.

1.5.1 Exemple

On cherche uniquement à évaluer un type d'agent à apprentissage, qu'on appelle Ag, on a donc $S = Ag$. On cherche à l'évaluer pour des compétitions impliquant un faible nombre d'agents, par exemple 5.

On lancera un grand nombre de compétitions, par exemple 1000, précédées des opérations suivantes. On tire donc au hasard 4 types d'agents parmi une large bibliothèque d'agents. On obtient par exemple : $E = Rsi1, Variation1, Zit1, Zit2$. On créé une compétition avec 5 familles de population nulle à partir de nos 5 types d'agents. Pour une population de 100 agents aléatoirement répartie, on incréménte $popTot = 100$ fois la population d'une famille tirée au hasard parmi les 5 de la compétition. On obtient par exemple la répartition suivante : 15 Ag, 23 $Rsi1$, 23 $Variation1$, 17 $Zit1$, 22 $Zit2$. On lance alors la compétition (50 générations) et on enregistre la part de la population de chaque famille à la fin de celle-ci.

La part moyenne de population de Ag sur les 1000 compétitions est un bon indicateur de la qualité de ce type d'agent.

2 Utilisation du paquetage `compet`

2.1 Création d'une compétition

```
public CompetitionEcologique();
```

2.2 Ajout d'une famille

2.2.1 Paramètres

- une chaîne *name*, nom de la famille
- un booléen *takesPart*, indiquant si la famille prend part à la compétition (*true*) ou pas (*false*).
- un entier *nbAgents*, indiquant le nombre d'agents de cette famille lors de la première génération.

– un agent *ag*, qui sert de modèle pour la création des agents de cette famille. C'est cet agent qui sera cloné pour pour produire l'ensemble de la population de cette famille.

2.2.2 Usage

```
public void addAgentType(String name, boolean takesPart, int nbAgents,
                         Agent ag);
```

2.2.3 Fonctionnement

Ajoute une famille de nom *name* à la compétition. Si *takesPart*, la famille prend part à la compétition et sa population variera avec les générations. Sinon, sa population reste constante. Lors de la première génération, cette famille aura une population de *nbAgents* agents. La population est générée par duplication [1] à partir de l'agent de base *ag*. La méthode *init*() est appellée après chaque duplication. Ainsi on peut, en la redéfinissant, avoir une famille dont les agents sont initialisés différement (par exemple aléatoirement).

2.3 Résultats générés

La compétition produit un CSV (qui peut être écrit dans un fichier ou affiché dans la console) de la forme suivante :

```
gen A B C
0 10 10 10
1 9 12 9
2 5 21 4
3 2 27 1
4 0 30 0
```

Ce fichier représente la population de chaque famille (A, B ou C) à chaque génération (4 générations dans cet exemple en plus de la répartition initiale). Ce csv peut être simplement affiché (par défaut) ou enregistré dans un fichier. Il suffit pour celà d'appeler :

```
maCompet.setOutputFile("mon_fichier.csv");
```

2.4 Logs

Il peut être intéressant de conserver les *logs* d'une ou plusieurs générations. Pour cela, la classe `CompetitionEcologique` possède un attribut :

```
public HashMap<Integer, String> logs;
```

Pour créer un fichier de *log* à la génération i, il suffit d'appeler :

```
maCompet.logs.put(i,"monFichier.log");
```

1. Cette duplication repose sur la sérialisation, c'est pourquoi il est crucial que les agents, signaux, stratégies et politiques soient sérialisables.

2.5 Exécution d'une compétition

2.5.1 Paramètres

Pour exécuter une compétition écologique, il est nécessaire de préciser les paramètres suivants :

- un entier $maxGenerations$, nombre de générations après laquelle la competition est arrêtée
- un Day day, forme que prend une journée
- un entier $nbJours$, nombre de journées dans une génération
- un entier $nbOrderBooks$, nombre de carnets d'ordres utilisés dans la simulation
- un entier $defaultCash$, cash possédé par chaque agent au début de chaque génération

2.5.2 Usage

```
public void run(int maxGenerations, Day day, int nbJours,
                int nbOrderBooks, long defaultCash);
```

2.5.3 Fonctionnement

Comme expliqué précédement, une compétition est une succession de simulations à la fin de laquelles la population est réévaluée. Chaque simulation est une succession de $nbJours$ jours de la forme day, avec $nborderBooks$. La simulation s'arrête :

- après $maxGenerations$ générations
- lorsqu'il reste une seule famille vivante (c'est-à-dire de population non-nulle)
- lorsque le *wealth* d'aucun agent n'a varié entre le début et la fin de la simulation (plus rien ne se passe)

3 Fabrication de simulations grâce à diverses expériences

Nous recommandons la lecture de l'article "évaluer des stratégies dans un environnement complexe" avant de lire cette section, afin de mieux cerner les enjeux de l'évaluation des agents, et de mieux connaître les différentes expériences que nous évoquons dans cette section. Nous expliquons ici comment, à partir de l'expérience dite des sous-ensembles, créer toute une variété d'expériences différentes.

3.1 Expérience de coévolution

Une expérience très simple consiste à plonger différents agents dans une même simulation et observer le gain de chacun dans cette simulation. Le package permet de réaliser très simplement cette expérience. Il est possible de réaliser l'expérience à l'échelle de l'individu ou à l'échelle de familles d'individus du même type. Il est également possible de répéter l'expérience un grand nombre de fois et d'obtenir un résultat moyen.

3.1.1 Paramètres

Pour réaliser cette expérience, il est nécessaire de préciser les paramètres suivants :

- une liste d'agents *agents*, qui contient les différents agents participant à la simulation
- un entier *nbIter* qui est le nombre de fois qu'on répètera l'expérience
- un entier *popParFamille*, qui est le nombre d'individus par lequel sera représenté chaque type d'agents

3.1.2 Usage

```
public static void coEvolution(ArrayList<Agent> agents, int nbIter,
                               int popParFamille);
```

3.1.3 Lecture des résultats

Voici un exemple de résultats pour cette expérience. La partie classement contient, pour chaque famille, la part d'arrivée à chaque position du classement en termes de wealth. Dans cet exemple, la famille testF a obtenu le meilleur wealth moyen dans 40% des simulations, le deuxième meilleur wealth dans 30% des simulations, le troisième dans 20% des simulations, etc.

La partie wealth moyen contient le wealth moyen de chaque famille. Dans cet exemple, les agents de la famille TestA ont obtenu un wealth moyen de -1292905.5\$, les agents de la famille TestB ont obtenu un wealth moyen de -314579.9\$, etc.

```
---------classement---------
TestA 0.0 0.0 0.0 0.0 0.0 1.0
TestB 0.0 0.0 0.0 0.0 1.0 0.0
TestC 0.1 0.3 0.3 0.3 0.0 0.0
TestD 0.3 0.1 0.3 0.3 0.0 0.0
TestE 0.2 0.3 0.2 0.3 0.0 0.0
TestF 0.4 0.3 0.2 0.1 0.0 0.0

---------wealth moyen---------
-1292905.5 -314579.9 344741.7 454626.8 426652.2 381464.7
```

3.2 Expérience des sous-ensembles

L'expérience des sous-ensembles est une expérience plus élaborée permettant d'obtenir des résultats encore plus significatifs dans le cadre de l'évaluation des agents. Celle-ci consiste à comparer n agents, qui forment un ensemble S, en ne les confrontant pas directement entre eux, mais en les confrontant à une base d'agents E.

Pour chaque agent $s \in S$ on répète *iter* fois l'expérience décrite ci-après. On choisit aléatoirement un sous-ensemble $E_i \in E$ de manière à ce que $|\{s\} \cup E_i| = taille_i$. $taille_i$ peut être constant ou défini aléatoirement pour chaque itération. On crée alors une simulation peuplée différentes familles d'agents, chaque famille étant une population d'agents identiques correspondant à un des

agents de $\{s\} \cup E_i$. La population de chaque famille peut être égale entre les différentes familles ou aléatoire.

3.2.1 Paramètres

Pour réaliser cette expérience, il est nécessaire de préciser les paramètres suivants :

- une liste d'agents s, contenant les agents que l'on cherche à évaluer
- une liste d'agents e, contenant les agents servant à peupler les simulations
- un entier $sizeCompetMin$, nombre minimum d'agents différents intervenant dans chaque itération
- un entier $sizeCompetMax$, nombre maximum d'agents différents intervenant dans chaque itération
- un entier $nbIter$, nombre d'itérations à exécuter de la simulation pour chaque agent de s
- un booléen $verbose$, permettant d'avoir accès à un mode plus verbeux (debug)
- un entier $totPop$, population d'agents dans chaque simulation

3.2.2 Usage

```
public static void subSetClassic(ArrayList<Agent> s, ArrayList<Agent> e,
                                 int sizeCompetMin, int sizeCompetMax,
                                 int nbIter, boolean verbose,int totPop);
```

3.2.3 Lecture des résultats

Voici un exemple de résultats pour cette expérience. La partie classement contient, pour chaque famille, la part d'arrivée à chaque position du classement. Les différents agents des S ayant été évalués dans des simulations séparées, c'est pourquoi la somme de chaque position du classement ne fait pas 1.

La partie wealth moyen contient le wealth moyen de chaque famille, classés du meilleur au moins bon.

```
---------classement---------
TestA 0.0 0.1 0.1 0.0 0.2 0.1 0.3 0.1 0.1

---------classement---------
TestB 0.2 0.0 0.0 0.0 0.2 0.4 0.1 0.1 0.0

---------wealth moyen---------
1 : TestB (-329721.2916666666)
2 : TestA (-2.7009036060454544E10)
```

3.3 Expérience des sous-ensembles avec compétition écologique

On peut améliorer l'expérience des sous-ensembles en prenant en compte l'adaptativité de l'environnement, c'est à dire en réalisant cette expérience, non pas sur de simples simulations mais sur des compétitions écologiques. L'expérience s'utilisera de la même manière, mais aura bien sûr des résultats différents.

3.3.1 Paramètres

Pour réaliser cette expérience, il est nécessaire de préciser les paramètres suivants :

- une liste d'agents s, contenant les agents que l'on cherche à évaluer
- une liste d'agents e, contenant les agents servant à peupler les simulations
- un entier $sizeCompetMin$, nombre minimum d'agents différents intervenant dans chaque itération
- un entier $sizeCompetMax$, nombre maximum d'agents différents intervenant dans chaque itération
- un entier $nbIter$, nombre d'itérations à exécuter de la simulation pour chaque agent de s
- un booléen $verbose$, permettant d'avoir accès à un mode plus verbeux (debug)
- un entier $totPop$, population d'agents dans chaque simulation

3.3.2 Usage

```
public static void subSetEcolo(ArrayList<Agent> s, ArrayList<Agent> e,
                               int sizeCompetMin, int sizeCompetMax,
                               int nbIter, boolean verbose,int totPop);
```

3.3.3 Lecture des résultats

Voici un exemple de résultats pour cette expérience. La partie classement contient, pour chaque famille, la part d'arrivée à chaque position du classement. Cependant, le classement ne se fait pas sur le wealth des agents, mais sur la population finale moyenne de chaque famille d'agents, qui est elle-même liée aux gains des agents durant les simulations successives qui constituent les compétitions.

La partie "populations moyennes" contient les populations finales moyennes de chaque famille, classées de la meilleure à la moins bonne. Dans l'exemple ci-dessous, la famille testB finissait les compétitions écologiques avec en moyenne 23.4 agents.

```
---------classement---------
TestA 0.2 0.0 0.0 0.0 0.2 0.6


---------classement---------
TestB 0.2 0.2 0.0 0.0 0.0 0.6

---------populations moyennes---------
1 : TestB (23.4)
2 : TestA (6.4)
```

3.3.4 Fichier de population

Afin de mieux représenter les résultats obtenus il peut être utile une courbe de la population moyenne d'une famille d'agents à chaque génération. Ce fichier

est généré pour chaque famille d'agents sous le nom "data/mafamille.csv". Celui-ci prend la forme suivante :

```
gen TestB
0 0.182
1 0.276
2 0.298
3 0.286
4 0.258
5 0.256
(...)
```

On peut voir dans cet exemple que la famille Test B a eu une part de population moyenne de 0.182 à l'initialisation, puis de 0.276 après la première génération... Attention, il ne s'agit plus de population en nombre d'agents, mais en part de population. Ainsi 0.182 signifie qu'en moyenne, 18.2% de la population totale de la compétition est de la famille TestB. Cela permet de faciliter les comparaisons entre plusieurs expériences potentiellement différentes.

4 Conclusion

Le package `eval` permet de créer facilement des évaluations de différent types, incluant des compétitions écologiques, ce qui permet d'évaluer la robustesse de la stratégie utilisée par un agent dans un environnement dynamique où une sélection naturelle a lieu.

Références