SCIENCES SORBONNE UNIVERSITÉ

PLDAC - Football & Apprentissage

Rapport du projet

Santhos Arichandra, Almehdi Krisni
Master 1 DAC - 2021/2022

Encadrants : Vincent Guigue, Nicolas Baskiotis

Table des matières

1. Introduction

Il ne se passe plus un jour sans que l'intelligence artificielle ne devienne encore plus présente dans nos vies quotidiennes. Tous les secteurs, qu'il s'agisse du secteur automobile, bancaire ou même éducationnel sont affectés et de nouvelles idées en rapport avec l'intelligence artificielle y naissent sans arrêt. L'Homme a aujourd'hui un compagnon, bien qu'il n'existe pas physiquement, dont la puissance et l'utilité n'a de limite que la créativité et l'ingéniosité de son créateur.

Un secteur ayant été particulièrement affecté par l'essor de l'intelligence artificielle, plus communément appelée IA, est celui des jeux et plus précisément celui des jeux vidéo. Du jeu d'échecs au dernier jeu développé par un studio de prestige, on dispose d'une infinité de situations à explorer dans lesquelles on peut mesurer les performances de différents modèles reposant sur l'IA, qu'il s'agisse des sous-branches du Machine Learning ou du Reinforcement Learning.
L'IA permet alors d'adapter l'environnement afin qu'il soit plus cohérent et réaliste en utilisant notamment des architectures à complexité faible, comme les structures d'IA symbolique comme les arbres de comportement. Il y aurait donc un intérêt à implémenter des systèmes plus complexes pouvant reproduire les actions d'un joueur réel, à partir d'informations concernant son comportement.

Il existe aujourd'hui deux grands jeux vidéo de simulation footballistique, FIFA et Football Manager. Le premier est un jeu en temps réel, où les actions des joueurs ont des conséquences immédiates sur le match. Le second est également en temps réel mais où l'on contrôle un entraîneur, on ne peut donc que donner des ordres aux joueurs et leur indiquer quelles actions entreprendre pour chaque situation. Notre projet se rapproche donc plus de la vision Football Manager, puisque que l'on cherche à générer des ordres pendant les rencontres puis appliquer des modèles afin de permettre aux joueurs d'agir en fonction de leur contexte de jeu.

L'objectif de notre projet est donc de développer un jeu du genre entraîneur de football, sauf que les seules commandes à notre disposition sont la création des joueurs et indiquer à ces derniers que faire dans des situations précises. Les principales étapes de réalisation du projet consistent au développement d'un moteur de jeu, de la mise en place d'une interface Web depuis laquelle les utilisateurs vont pouvoir simuler des parties et indiquer les ordres associés à chaque situation puis à l'implémentation de modèles de Machine Learning.

2. Interface Web

L'objectif du projet est de mettre en place dans un premier temps une interface de jeu, sur laquelle les utilisateurs pourront simuler des parties et forger un ensemble de données d'ordres.
Une interface personnelle à chaque utilisateur, c'est-à-dire sa propre installation du jeu sur sa machine, n'étant pas une solution pérenne en raison du nombre de transferts de fichiers générés allant être nécessaire. Une des directives du projet est donc de créer une interface Web depuis laquelle les utilisateurs pourront directement simuler des parties, sans avoir à installer la moindre application ou librairie, en dehors d'un navigateur.

Nous avons utilisé PixiJS, un module Javascript permettant la création d'environnement en 2D. L'objectif principal était initialement de réimplémenter le moteur de jeu en JS (JS = JavaScript), nous en avons rapidement conclu qu'il est plus pratique d'utiliser JS uniquement pour la partie visualisation, à travers une interface web, et Python pour la création et la gestion des parties, ainsi que l'application du Machine Learning une fois le projet bien avancé.
De plus, nous disposons d'un moteur de jeu en Python prêt à être utilisé grâce à un projet réalisé par un de nos encadrants mais nous rentrerons dans les détails lors de la prochaine partie du rapport. Suite à l'implémentation de la partie visuelle du moteur de jeu, nous avons créé une interface pour les utilisateurs afin qu'il puissent simuler leurs matchs dans le meilleur des conforts.

Nous avons alors continué à utiliser JS, pour la création du site et la mise en place des différentes fonctionnalités utilisateur. Une section Compte a donc été implémentée, où les utilisateurs peuvent créer un compte pour accéder au jeu et simuler des parties, permettant à chaque joueur d'avoir un historique de ses parties jouées et de sauvegarder les données qu'il va générer au fur et à mesure qu'il joue des parties. Cette partie compte a été codée via PassportJS. La section a été mise en place grâce au middleware Passport.JS (**middleware** : type de logiciel informatique fournissant des services aux applications logicielles au-delà de ceux offerts par le système d'exploitation).

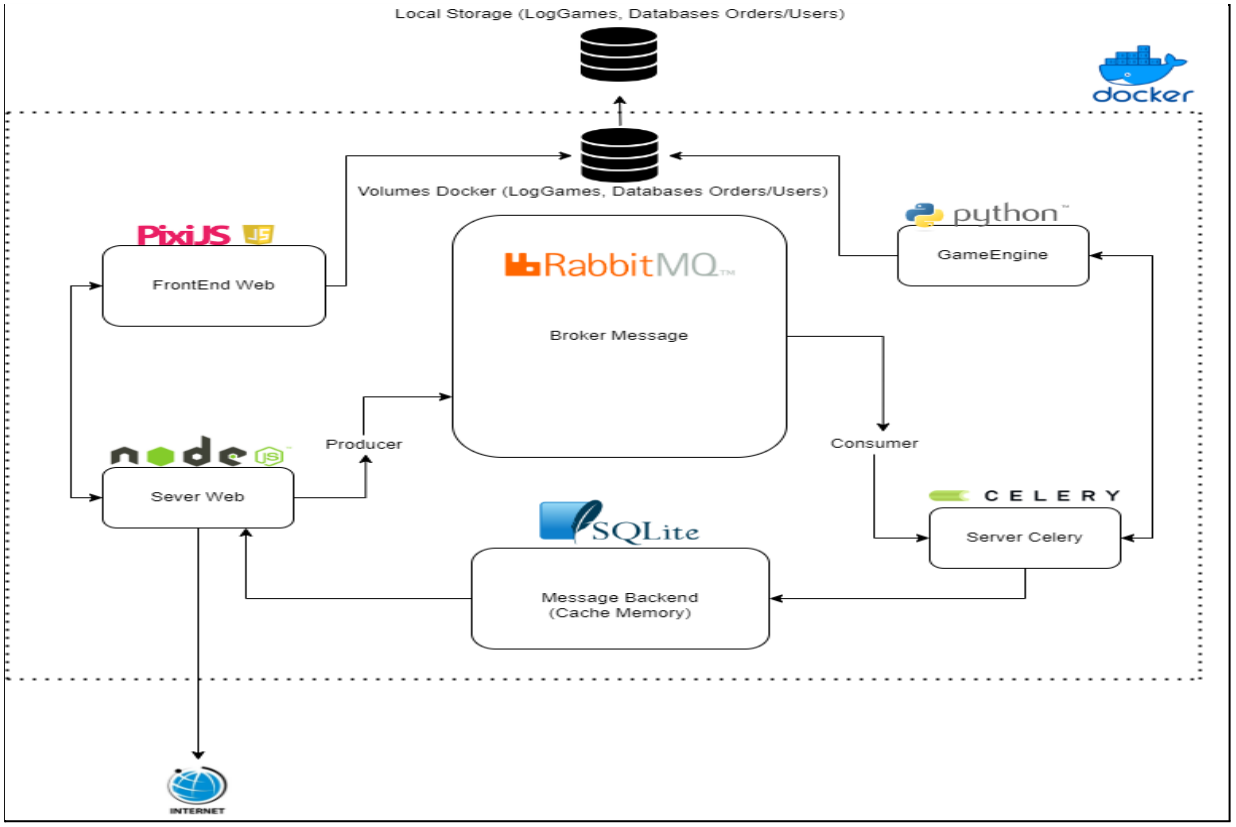
Il nous reste alors à mettre en place la relation entre JS, soit la partie Web du projet, et le lancement des parties, dans la partie Python. Pour cela, nous avons utilisé NodeJS pour contrôler le domaine Web et Celery afin de créer le serveur jouant les matchs en fonction des requêtes.
NodeJS est un environnement open-source permettant d'exécuter du JS en dehors d'un navigateur tandis que Celery est un ordonnanceur de tâches open-source concentré sur les opérations en temps réel.
Afin de lier les deux, nous utilisons RabbitMQ, un message-broker open-source, allant réaliser le transfert de messages, soit de tâches (**message-broker** : module intermédiaire traduisant un message du protocole de l'expéditeur au protocole du récepteur). Celery crée alors des fonctions asynchrones qui seront par la suite envoyées à RabbitMQ.

Afin de relier tous les éléments sans contrainte de machine, nous avons utilisé Docker, allant créer un container dans lequel toutes les dépendances sont mises en place au préalable. Un fichier docker-compose permet alors de démarrer la structure (voir la figure ci-dessous).

On rentre alors dans les détails de l'exécution. Au moment où un utilisateur veut créer un nouveau match, soit via NodeJS, le serveur Node implémenté crée alors un message demandant la création du match. RabbitMQ transfère alors ce message dans la file d'attente et c'est alors que Celery intervient en récupérant le premier message de la file d'attente, soit le plus ancien. Il y a alors lecture du message et l'action correspondante est effectuée, comme par exemple, créer un match. Le serveur Celery est en attente de messages de RabbitMQ. On pourrait traiter plusieurs messages de manière parallèle mais il faut pour celà avoir plusieurs serveurs Celery actifs (**Celery = single thread**).

On arrive alors à la réponse de Celery, qui est de créer une partie sauvegardée au sein d'une base de données propre à Docker l'ensemble des logs de matchs et des ordres enregistrés par les utilisateurs et également de communiquer au serveur Node l'identifiant du match fraîchement créé. La réponse est gérée par une base de données SQL incluse dans le cache appelée Message Backend. Disposer des identifiants des matchs nous permet de lire les logs correspondants et de visionner la partie sans problème.Donc dans cette approche les matchs sont dans un premier temps créer en Python en fond et ensuite quand ils sont terminées mis à disposition de l'utilisateur pour les visionner.

Architecture globale



3. Gestions des données

Les données sont le cœur de tout apprentissage lié au Machine Learning. Il est donc primordial de créer les structures adéquates afin de les sauvegarder et les entretenir.
On conserve alors trois catégories de données, dans deux formats distincts. La première concerne le déroulement des matchs et la seconde les ordres donnés par les utilisateurs lors du visionnement des parties, qui seront utilisés dans le cadre de l'apprentissage. La troisième concerne les informations utilisateurs afin de gérer les connexions aux comptes personnels sur le site.
Le module NeDB, présent sur JS, permet la création de petites base de données de manière simple et efficace et assure la transition vers MongoDB, dans le cas où l'on disposerait d'un grand nombre de données ce pour quoi NeDB n'est pas destiné, du fait que les architectures sont compatibles.

A la création d'un compte sur le site, on sauvegarde un identifiant, une adresse mail ainsi qu'un mot de passe. Les mots de passe sont cryptés afin de protéger les comptes utilisateurs.
Ces informations sont conservées dans un fichier au format **.db** où l'on dispose pour utilisateur de :

- son nom
- son email
- son mot de passe crypté
- les identifiants des matchs qu'il a simulé (pour conserver un historique des parties)

Pendant la visualisation des matchs, les utilisateurs peuvent donner un ordre à un joueur particulier, ce qui influencera sa décision à l'avenir.
On conserve également ces données un fichier .db où l'on dispose pour chaque ordre donné :

- coordonnées du joueur receveur d'ordre
- coordonnées de la balle et de chaque joueur (qu'il soit allié ou adversaire)
- score actuel
- ordre, qui correspond donc aux labels
- ID utilisateur (pratique pour la création de modèles propres à chaque utilisateurs)

Les données des matchs, elles sont stockées après le déroulement des parties dans des fichiers au format **.txt** où chaque ligne correspond à un temps de jeu.
A chacun de ces temps, on dispose de :

- l'itération de jeu
- la position de la balle
- la position et le vecteur vitesse de chacun des joueurs
- le score

4. Premiers mouvements

Reprise du moteur de jeu

Comme mentionné précédemment, nous disposons d'un moteur codé en Python, mis à notre disposition par l'un de nos encadrants, M. Nikola Baskiotis, afin de nous permettre de progresser plus rapidement dans notre projet.
Il s'agit d'un moteur de jeu 2D basé sur les accélérations et les interactions terrain-joueur-ballon afin de faire progresser le match. Les vitesses et directions des joueurs sont engendrées par des additions et soustractions sur le vecteur accélération de chaque joueur. Cela permet alors d'obtenir une simulation très réaliste au niveau des déplacements des joueurs ou de la balle, puisqu'il est impossible de s'arrêter brusquement ou d'effectuer des virages impossibles physiques (retournement à 180 degrés instantané par exemple).

En dehors de quelques paramètres de jeu, comme la vitesse maximale de déplacement ou la force de frappe maximale, avec lesquels nous avons expérimenté, nous n'avons pas eu à effectuer de modifications au niveau de la structure du code fourni. On peut extraire les informations du jeu à chaque itération puisque l'on dispose de variables permettant d'accéder aux états de la balle et de chaque joueur. Nous disposons désormais d'un moteur de jeu opérationnel, il est maintenant temps de passer à la simulation de comportements et la création de stratégies basiques afin de fonder notre base d'ordres.

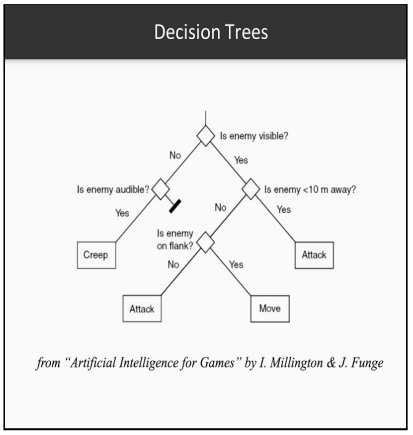
Création de stratégies simples

Initialement, nous ne disposons d'aucune donnée utilisateur puisqu'il est impossible de simuler alors que nous ne disposons d'aucun modèle. On entreprend alors la création de stratégies statiques, c'est-à-dire de stratégies ne dépendant que de conditions fixes telles que la position du joueur, le score actuel du match ou le déplacement des joueurs adverses.
Afin de mettre en place une telle stratégie, il faut admettre l'hypothèse que les joueurs sont omniscients et peuvent accéder à n'importe quelle information, qu'il s'agisse d'informations à propos de leur équipe, de l'équipe adverse, de la balle ou du terrain.

Une implémentation simple et efficace de stratégie statique est celle de l'arbre de décision. Ces structures sont notamment utilisées dans les jeux vidéo pour la création d'histoires où les choix du joueur influencent l'histoire (ex. The Witcher) mais également pour gérer les actions de PNJ, les Personnages Non Jouables (ex. gardes dans Skyrim).

Coïncidence, les joueurs ne sont pas contrôlables pendant le match puisque leurs actions devraient dépendre au final des ordres donnés par les utilisateurs. On peut donc appliquer une telle structure pour une première collecte de données.

Un arbre de décision est composé de briques effectuant en fonction d'une condition de l'environnement ou du joueur, soit une redirection vers une brique spécialisée, soit une action.
On commence alors par mettre en place deux arbres de décision, un pour le profil Attaquant et un pour le profil Défenseur. Il a été pendant un moment envisagé d'implémenter un arbre pour le profil Milieu, mais en raison d'un manque de complexité de l'arbre prototype (actions retournées beaucoup trop similaires au profil Attaquant), nous avons décidé d'abandonner cette idée.



Les profils sont attribués à chaque joueur lors de la création du match, et renvoie à chaque instant de jeu quelle action le joueur doit entreprendre en fonction du match. De nouvelles variables sont alors créées et partagées entre les joueurs afin de représenter le caractère omniscient de chaque joueur.

Pour chaque profil, il existe un certain nombre d'hyper-paramètres permettant de contrôler plus ou moins le fait qu'un joueur soit plus ou moins entreprenant dans ses décisions offensives avec la possibilité de frapper de loin ou de chercher à se démarquer. On contrôle également le caractère craintif des défenseurs lors de phases de jeu défensives, comme par exemple le fait que si un joueur adverse en possession de la balle est trop proche des cages, on cherche à venir lui faire barrage au plus vite.

5. Machine Learning

Dans cette partie, on se concentre sur la prédiction des actions que les joueurs doivent exécuter à partir de la base d'ordres. Il est dans un premier temps nécessaire de réaliser une extraction de features depuis les données brutes afin de sélectionner quelles informations seront utilisées lors du Machine Learning.

Données

Nous avons alors réfléchi aux meilleures features à extraire depuis les données dont on dispose dans la base de données des ordres. On ne peut pas donner toutes les features possibles et imaginables puisque cela reviendrait à surcharger les exemples et il est possible que certaines dimensions n'aient aucune valeur.
La liste des features retenues sont donc :

- position X, Y du joueur
- distances joueur - balle
- distance joueur - cages
- distance joueur - cages adverses
- distances joueur - coéquipier le plus proche
- distance joueur - adversaire le plus proche
- distance joueur - coéquipier le plus proche devant lui
- distance joueur - adversaire le plus proche devant lui
- distance joueur - coéquipier le plus proche derrière lui
- distance joueur - adversaire le plus proche derrière lui
- nombre de coéquipiers dans la moitié de terrain de son équipe
- nombre d'adversaires dans la moitié de terrain de son équipe
- nombre de coéquipiers dans la moitié de terrain adverse
- nombre d'adversaires dans la moitié de terrain adverse

Intéressons nous désormais sur la récupération des données, plus précisément des exemples et des labels de classe. La base regroupant toutes ces informations au format **.db** et le séparateur étant la virgule, assez embêtant pour une lecture directe avec Python, on convertit dans un premier temps les données au sein d'un fichier au format **.txt**, plus simple à manipuler. On a donc les fichiers **orderRecup.js** et **fileExtract.py** du dossier **extractData** contenant les fonctions nécessaires pour effectuer toutes les transformations et obtenir les données dont nous avons besoin.
En sortie, les modèles renvoient l'action à effectuer prédite en fonction de la situation de jeu utilisée pour la prédiction.

On distingue alors deux grandes familles d'actions. Les familles sont relativement petites puisque l'on favorise des actions simples et concrètes plutôt que de créer des actions complexes et spécifiques à certaines situations. On présente désormais les familles et les ordres qui leurs sont propres :

Attaque

- **aller vers les cages adverses** : on se déplacer uniquement vers les cages adverses
- **suivre la balle et fonce** : on se dirige vers la balle, équivalent du mode fonceur
- **frapper la balle** : on applique une force sur le ballon si il est à proximité dans la direction des cages adverses, sinon on se dirige vers la balle
-

Défense

- **aller vers ses cages** : on se déplace vers ses cages
- **aller vers le centre de son camp** : on se dirige au point situé entre la balle et ses cages
- **dégager la balle** : on applique une force au ballon si il est à proximité dans une direction contraire à celle des cages de notre équipe, sinon on se dirige vers la balle

Modèles

Les différents modèles utilisés dans le cadre du projet sont le K-NN (K-Nearest Neighbors), le SVM (Support Vector Machine), le Random Forest ainsi qu'un modèle de Boosting.
Chacun de ces modèles possède des propriétés et objectifs différents, comme par exemple le KNN allant chercher les exemples les plus similaires à un exemple donné ou le SVM allant pouvoir classifier chaque ordre de manière linéaire. Dans notre cas, le SVM est donc multi-classe. Le Random Forest quant à lui crée un arbre de décision d'une profondeur donnée afin de séparer les exemples en fonction des attributs. Tous les modèles sont implémentés grâce à **scikit learn**.
Chacun de ces modèles possède des hyper-paramètres spécifiques. On les présente dans un premier temps puis on cherche à les optimiser dans la prochaine section du rapport.

KNN

- **algorithm** : permet de choisir la méthode de sélection des plus proches voisins ("brute force", "ball_tree", "kd_tree")
- **n_neighbors** : correspond au nombre de voisins allant être étudié, soit la valeur K. En raison du nombre de données relativement faible, on prédit devoir utiliser un K faible
- **weights** : définit la manière dont on attribue les poids de chacun des plus proches voisins. Si le paramètre est instancié à "uniform", tous les voisins ont un poids similaire. Si il est instancié à "distance", alors plus le voisin est proche, plus son influence dans la décision est important

SVM

- **C** : contrôle de la marge du SVM et donc de la tolérance aux erreurs
- **kernel** : noyau de transformation des données utilisé. Il en existe un grand nombre comme “linear”, “poly”, “sigmoid” et quelques autres

RandomForest / DecisionTree

- **class_weight** : attribution d’un poids ou non aux différentes classes. Si l’on décide d’utiliser des poids, alors il existe différentes sélections des valeurs de poids. Si on choisit le mode “balanced”, alors le poids est égal à la fréquence d’apparition dans l’ensemble du jeu de données. Si on choisit le mode "balanced_subsample", alors la valeur des poids est égale à la fréquence d’apparition de la classe dans le sous-arbre étudié. On peut également utiliser un dictionnaire de la forme **{class_label : weight}** pour attribuer les poids.
- **criterion** : méthode de calcul du gain d’information lié à un attribut sélectionnée. On peut utiliser les critères “gini” pour utiliser la formule de l’impureté de Gini, “entropy” et “log-loss” pour utiliser l'entropie de Shannon.
- **max_depth** : profondeur maximale de l’arbre. Si il n’est pas instancié, alors l’arbre se développe jusqu’à obtenir des feuilles pures ou jusqu’à ce que toutes les feuilles ne contiennent plus assez d’exemples pour former un sous-arbre (**min_samples_leaf**).

Boosting

- **loss** : fonction de perte utilisée pour l’optimisation. On peut choisir la “log_loss”, “deviance” ou “exponential”
- **max_depth** : profondeur maximale des arbres. Il n’est pas forcément nécessaire d’utiliser des valeurs excessivements élevées
- **n_estimators** : nombre d’arbres allant être utilisés. Le Boosting est particulièrement résistant au sur-apprentissage donc utiliser un grand nombre d’arbres retourne généralement de meilleurs résultats

6. Résultats et comparaison

Afin d'obtenir les meilleures performances, il est nécessaire d'utiliser toutes les combinaisons d'hyper-paramètres possibles, soit les paramètres sélectionnés par l'humain, afin d'utiliser nos algorithmes dans les meilleures conditions.
Essayer toutes les combinaisons d'hyper-paramètres à la main est une tâche longue et fastidieuse et une action humaine répétée pourrait être source d'erreurs. On se tourne donc vers le Grid Search afin de chercher les meilleures combinaisons. Le Grid Search retourne après son exécution les modèles, la précision et les paramètres utilisés pour l'obtenir.

Tableau des résultats obtenus par Grid Search

	score	best_parameters
KNN	0.710366	{'algorithm': 'auto', 'n_neighbors': 3, 'weights': 'distance'}
SVM	0.667683	{'C': 1, 'kernel': 'linear'}
RandomForest	0.765244	{'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 15}
Boosting	0.710366	{'loss': 'deviance', 'max_depth': 2, 'n_estimators': 50}

Le tableau ci-dessus contient donc les meilleurs scores de précision de chaque algorithme ainsi que les meilleurs paramètres trouvés. On remarque une précision moyenne de 71%, avec le Random Forest légèrement au-dessus de la moyenne et le SVM légèrement en dessous.
On peut expliquer le fait que le SVM retourne un résultat inférieur à moyenne puisque étant donné le fait que le SVM réalise une séparation linéaire des données.

Pour tester nos algorithmes et donc nos stratégies, nous avions au début tester des features prenant peu d'informations ce qui découlait à des exemples très peu différents entre les différentes classes.
Nous avons donc dans un deuxième ajouter les features permettant de quantifier les joueurs dans les différentes parties du terrain. Ça a permis aux modèles d'avoir des résultats un peu meilleurs mais pas au niveau de ce que nous avons. Ce sont surtout les dimensions du plus proche devant et derrière, qui nous ont permis d'avoir de meilleurs résultats. Ceci est normal car via ses features nous avons une très grande compréhension du jeu sur un horizon local du joueur, ce qui permet d'avoir des résultats un peu meilleurs. Toutefois nous avons essayé de faire de la visualisation mais celle-ci n'a pas été très concluante (certains groupes mais avec dans chacun d'entre eux une mixité de classes différentes) dû au fait que certains ordres peuvent être appliqué à différentes positions du jeu.
Ce qui explique encore une fois les difficultés du SVM a avoir de bon score.

Parmi les valeurs d'hyper-paramètres retournés, nous retrouvons certaines valeurs que nous avions pu prédire en raison du contexte d'apprentissage. Pour chaque modèle :

- KNN : **weights** = **distance** permet de pénaliser les points éloignés, soit les contextes d'actions fortement différents et également **n_neighbors** ayant une valeur faible, puisque dans une petite base de données, il y a possiblement moins de voisins pertinents.
- Random Forest : **class_weight** = **balanced** sert à ne pas défavoriser ou favoriser une classe, soit une action, parmi les autres

Nous disposons désormais des meilleurs modèles. Il est alors temps de les confronter les uns aux autres afin de déterminer les différences d'un point de vue actions des joueurs en fonction des modèles. Dans le code source récupéré, la fonctionnalité de créer et simuler des tournois au sein du fichier **battleAI.py**.
La fonctionnalité Tournoi réalise des matchs aller-retour afin de limiter au maximum l'influence de l'aléatoire sur les résultats..

Les différentes équipes allant participer au tournoi sont donc :

- l'équipe KNN-3 (algorithm = auto, n_neighbors = 3, weights = distance)
- l'équipe SVM (C = 1, kernel = linear)
- l'équipe RandomForest (class_weight = balanced, criterion = entropy, max_depth = 6)
- l'équipe Bagging (loss = deviance, max_depth = 2, n-estimators = 50

Les matchs se déroulent de la manière suivante : à chaque instance de jeu, on réalise une prédiction d'action pour chaque équipe en utilisant le modèle de l'équipe.
Étant donné le fait que l'on réalise ces matchs dans le but de déterminer quel modèle a l'avantage sur les autres pendant les confrontations, on utilise l'ensemble des données de la base d'ordres plutôt que d'utiliser des données propres à un utilisateur.
Il est fortement dommage que nous ne disposons pas de plus de données par utilisateur puisqu'il aurait été intéressant d'étudier le comportement des modèles pour quelques utilisateurs tirés aléatoirement.

Afin d'obtenir des moyennes de score et pas des scores fixes sur un seul tournoi, on simule 5 tournois afin de mieux comparer les modèles entre eux. On conserve le nombre de points moyen, le nombre de victoires, nuls et défaites et le nombre de buts marqués et encaissés.

Tableaux des scores

	Points	**V / N / D**	**Marqué / Encaissé**
RandomForest	16	3.8 / 6.5 / 1.1	35.4 / 2.4
SVM	14.1	2.6 / 7.1 / 1.2	27.2 / 3.5
Boosting	13.8	2.6 / 5.8 / 2.1	23.1 / 6.8
K-NN 3	10.8	2.5 / 5.1 / 2.7	16.7 / 39.8
K-NN 5	10.8	2.1 / 5.2 / 2.8	15.2 / 36.5
K-NN 10	8.7	1.7 / 4.3 / 3.8	13.7 / 44.6

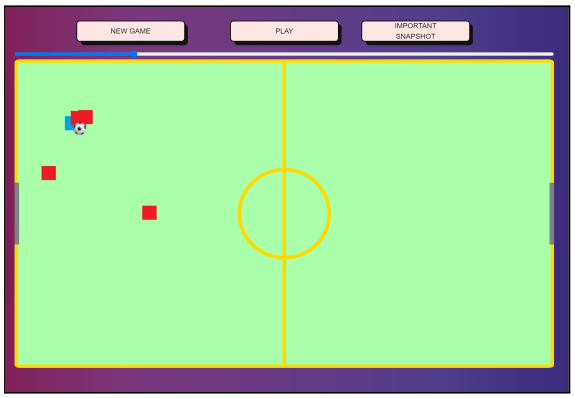
On constate que le RandomForest permet d'avoir les meilleurs résultats, ce qui est en lien avec la précision des modèles. De manière générale, on constate que les arbres sont plus performants que les autres modèles et cela est sûrement dû aux features qui sont explicites et permettant donc d'avoir des choix plus simples à effectuer au niveau des branchements. Plus précisément, étant donné que nous utilisons essentiellement les distances et surtout les distances Front et Behind, qui permettent d'avoir une compréhension la plus claire de l'état actuel. Le SVM fait également partie des meilleurs modèles, ce qui peut paraître étonnant au vu des scores de prédiction que nous avons obtenu suite à l'exécution du Grid Search.

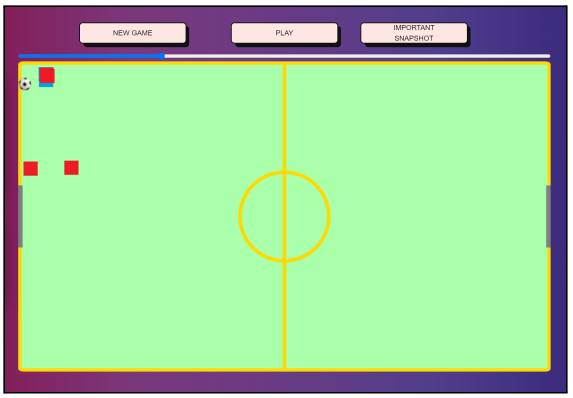
Cette différence peut s'expliquer via le fait que plusieurs ordres peuvent être applicables à plusieurs situations. De plus, le SVM va via son noyau ajouter de nouvelle dimension permettant donc de pouvoir distinguer les familles d'ordres qui sont complètement opposées (d'où de bon résultat en match) mais n'arrivera pas à distinguer les ordres qui sont similaires dans une situation donnée. Finalement les KNN se situent en dernière position, surement à cause du point que nous avons relevé pour le SVM et malheureusement le KNN ne fait pas d'augmentation de dimensions et donc très compliqué pour l'algorithme de faire un choix meilleur que les autres modèles.

7. Apparition de comportements

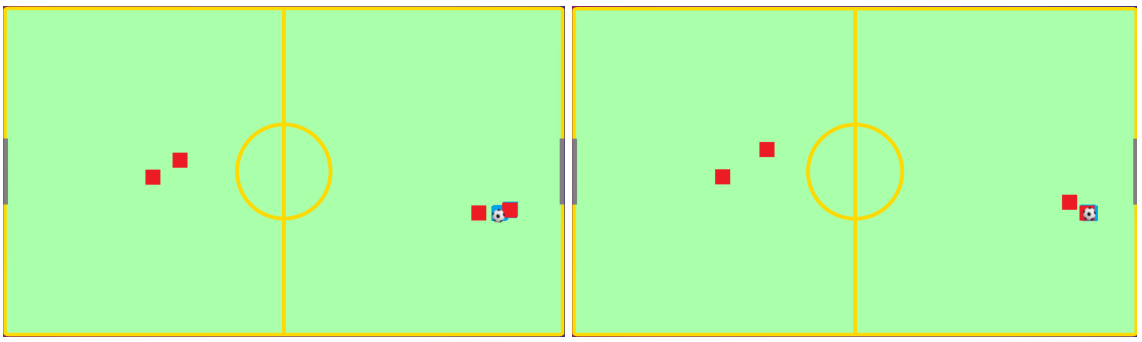
On effectue une première analyse des parties afin de mieux comprendre le fonctionnement des modèles tout au long des matchs.
On constate que les joueurs placés initialement en position défensive, soit devant les cages de leur équipe, restent en position défensive et ne cherchent pas à se déplacer vers la balle. Ce comportement était attendu puisque la majorité des ordres enregistrés en position défensive est de défendre les cages de son équipe.

Une deuxième action défensive que l'on constate est que lorsque l'adversaire s'approche des cages avec la balle, nous avons 1 ou 2 joueurs allant courir vers la balle afin d'empêcher la progression de l'adversaire. Il s'agit de la stratégie optimale à notre avis dans ce cas, puisque les passes entre joueurs n'étant pas implémentées dans le jeu, faire barrage à l'adversaire lorsqu'il s'approche dangereusement des cages permet d'entraver sa progression.
De plus, un autre comportement que nous constatons, c'est que durant ces situations nous avons certains joueurs qui au contraire vont se retirer la zone chaude pour aller partir en défense vers les cages ou vers le milieu du camp. Ca permet donc de former un second rideau, ce qui est un comportement très intéressant mais également très efficace.





Ce que l'on constate également est que lorsque des joueurs, qu'ils soient initialement en position défensive ou offensive, sont à proximité de la balle et un peu éloignés de leurs cages, restent chasser la balle jusqu'à ce qu'un but soit marqué. C'est probablement dû au fait que la majorité des ordres enregistrés lorsque le joueur est proche sont de la classe **frapper la balle** ou **courir vers la balle**.



De plus, un comportement offensif que nous apercevons est que lorsque les joueurs se dirige vers les cages adverses pour marquer un but (figure de gauche), nous constatons qu'il y a un des joueurs qui va s'écarter de l'action pour se diriger vers les cages adverses et non vers la balle (figure de droite). Ce comportement permet donc d'augmenter les chances de marquer un but, ce qui en fait donc un très bon comportement en phase offensive.

Les phases de jeu lorsque la balle se trouve à proximité d'un point de corner sont également problématiques puisque les joueurs y restent souvent bloqués indéfiniment. La balle ne pouvant sortir du terrain en raison des règles de jeu, on a donc une succession de rebonds de la balle liés aux frappes des joueurs sur les bordures du terrain. Cela rend donc certains matchs complètement inintéressants du point de vue divertissement pour l'utilisateur mais également d'un point de vue Machine Learning puisqu'il semblerait qu'aucune action implémentée ne permette de résoudre le problème. Pour résoudre ce problème, il faudrait probablement créer une nouvelle action de jeu à ajouter à la liste actuelle des ordres.

8. Possibles améliorations

Concernant les améliorations à réaliser, tout d'abord il faudrait pouvoir tester les modèles avec un plus grand nombre de données. De plus ce qui serait intéressant, c'est de pouvoir implémenter dans les features le score ce que nous n'avons pas réellement pu faire car il faut donc encore plus de données pour pouvoir voir cette feature vraiment être utile dans les modèles.
Une approche intéressante à pousser est celle du Reinforcement Learning, qui nous permettrait alors d'implémenter les notions de décision et de récompense afin de possiblement améliorer nos récompenses. Une idée d'architecture est la suivante :

- **États** - positions propre, des joueurs (coéquipiers et adversaires) et de la balle
- **Actions** - ordre que nous avons explicité dans la partie Ordre
- **Récompenses** - combinaison linéaire entre la distance de la balle aux cages adverses, de la différence de score (avec un poids important) et de la distance moyenne des adversaires à la balle (poids négatif)

9. Conclusion

Au final, on constate qu'avec un nombre d'ordre assez réduits nous pouvons déjà observer des comportements intéressants, ce qui est un point positif. Toutefois, les résultats sont encore bien loin d'être totalement satisfaisants et cela est sûrement dû à un manque de données.

Malheureusement, nous avons rencontré un problème de cross-domain sur la mise en production du site Web sur le serveur, ce qui nous a empêché de pouvoir récolter un maximum de données avec l'apport des utilisateurs.
De plus, même si avec le peu d'ordre dont on dispose on arrive à observer des comportements pertinents, la présence de comportements bloquants implique qu'il sera sûrement nécessaire d'avoir des ordres plus spécifiques. La prochaine étape serait donc de récupérer un maximum de données pour pouvoir tester les modèles plus efficacement avec des données plus équiprobables. Au-delà de l'aspect donnée, il serait également intéressant de pousser nos modèles sur des modèles de Deep Learning mais aussi du Reinforcement Learning.

A l'avenir, si le prochain est mis à disposition de futurs étudiants