

Antoine Viton G7
Baptiste Foucras G7

Rapport Prédiction score de football féminin

Modélisation - IA

Sommaire

Introduction.....	2
Modifications apportées au DataFrame.....	3
Mise en place de la colonne 'Win'.....	3
Mise en place de la colonne 'id_home_team' et 'id_away_team'.....	4
Mise en place de la colonne 'etat_match'.....	5
Mise en place de la colonne 'id_tournament'.....	6
Explication des colonnes du dataSet.....	6
Matrice de corrélation.....	7
Séparation des jeux de tests et des jeux d'entraînements.....	8
Utilisation des modèles et comparaison.....	9
Bibliographie.....	11

Introduction

Dans le cadre du cours de Modélisation-IA de 2ème année de DUT Informatique, nous avons été amenés à effectuer un notebook mettant en place un machine learning. Nous avons choisi d'essayer de prédire les résultats d'un match de football. Pour ce faire, nous avons exploité un DataSet, retrouvable à l'adresse suivante :

<https://www.kaggle.com/martj42/womens-international-football-results>

Voici une visualisation des données, qui contiennent tous les matchs internationaux féminins depuis 1969 :

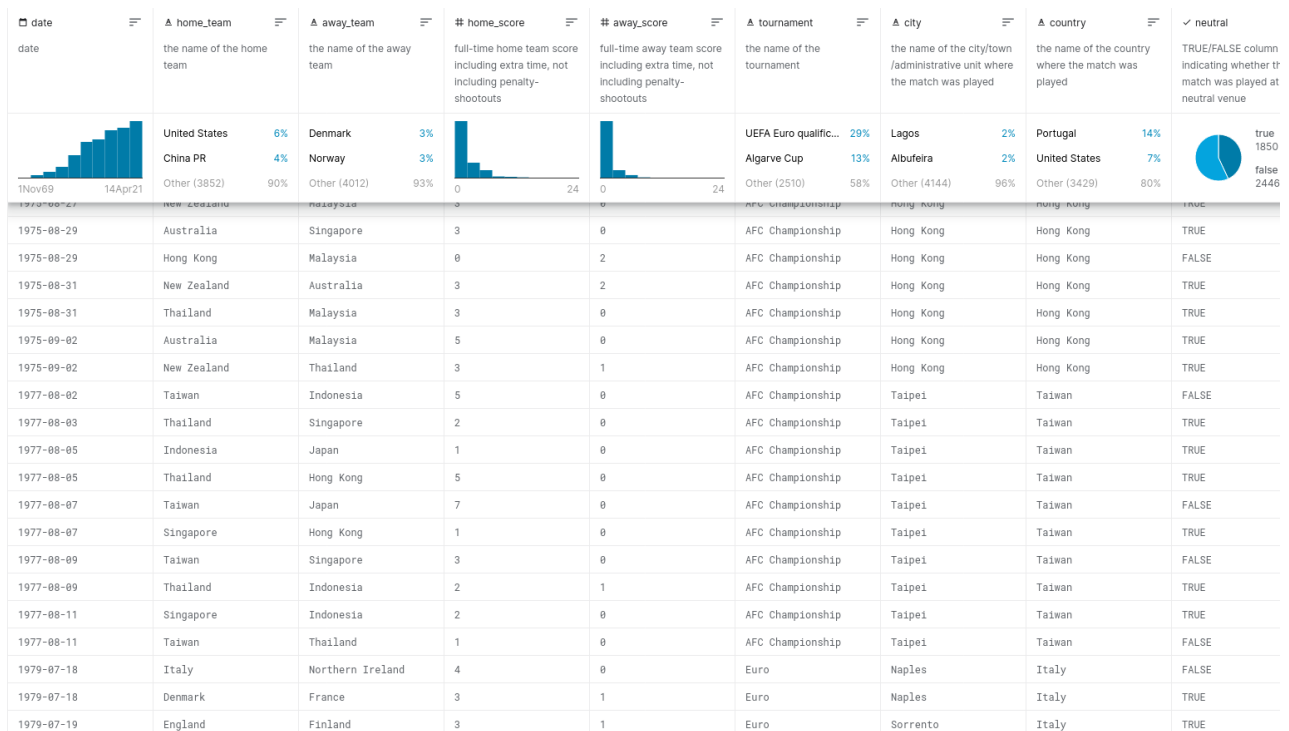


Fig 1 : Organisation des données du DataSet utilisé

Le but de ce projet est donc de classer chaque tuple de données en trois grands groupes (victoire de la home_team, victoire de la away_team et égalité), en fonction de la home_team, de la away_team et d'autres critères.

Notre but est ici multiple. On cherche d'abord à trouver comment optimiser la prédiction en fonction des données, mais aussi déterminer si certaines informations (terrain neutre ou non, tournoi dans lequel se place le match) ont une importance dans la classification.

Modifications apportées au DataFrame

Notre but est de pouvoir exploiter les colonnes `home_team`, `away_team`, `neutral`, `tournoiement` ; étant donné qu'elles peuvent toutes jouer un rôle dans la détermination du gagnant d'un match.

Il a donc fallu modifier ou créer diverses colonnes, étant donné que les prédictions ne prennent que des valeurs numériques (boolean, int, float...) en compte.

Mise en place de la colonne 'Win'

Le but est de tout d'abord classer chaque tuple en égalité, victoire de l'équipe home ou de l'équipe away.

Pour l'instant, l'information est donnée par `away_score` et `home_score`.

Comme nous ne cherchons pas à prédire les scores, mais seulement qui va gagner, nous allons créer une nouvelle colonne indiquant qui a gagné, et qui servira de résultante des classifications de nos modèles.

Cette colonne contiendra un entier pouvant prendre trois valeurs :

- 0 si égalité
- 1 si victoire de l'équipe domicile
- 2 si victoire de l'équipe extérieure

La création de cette colonne est donc basée sur les colonnes de scores.

```
""" On ajoute d'abord une colonne win pour savoir quelle équipe a gagné.
0 égalité
1 victoire home
2 victoire away """

for index in data_result_match.index :
    if(data_result_match.loc[index, 'home_score'] == data_result_match.loc[index, 'away_score']) :
        data_result_match.loc[index, 'win'] = 0
    elif(data_result_match.loc[index, 'home_score'] > data_result_match.loc[index, 'away_score']) :
        data_result_match.loc[index, 'win'] = 1
    else :
        data_result_match.loc[index, 'win'] = 2

data_result_match['win'] = data_result_match['win'].astype(int)
```

Fig 2 : Création de la colonne 'Win'

Nous obtenons donc le dataframe suivant :

data_result_match.head(10)										
	date	home_team	away_team	home_score	away_score	tournoiement	city	country	neutral	win
0	1969-11-01	Italy	France	1	0	Euro	Novara	Italy	False	1
1	1969-11-01	Denmark	England	4	3	Euro	Aosta	Italy	True	1
2	1969-11-02	England	France	2	0	Euro	Turin	Italy	True	1
3	1969-11-02	Italy	Denmark	3	1	Euro	Turin	Italy	False	1
4	1975-08-25	Thailand	Australia	3	2	AFC Championship	Hong Kong	Hong Kong	True	1
5	1975-08-25	Hong Kong	New Zealand	0	2	AFC Championship	Hong Kong	Hong Kong	False	2

Fig 3 : DataFrame avec la colonne Win ajoutée

Mise en place de la colonne 'id_home_team' et 'id_away_team'

Les équipes jouant le match sont un élément essentiel pour prédire qui va gagner. Notre modèle s'appuiera donc sur ces noms d'équipes. Cependant, comme souligné plus haut, les modèles ne peuvent prendre que des valeurs numériques, et non des chaînes de caractères.

Il faut donc générer un id unique pour chacune des équipes.

```
""" On ajoute une colonne id_away_team et id_home_team pour connaître les équipes ayant joué, sous forme numérique pour traitement """
listePays = sorted(list(set(np.concatenate([data_result_match['home_team'], data_result_match['away_team']]))))
"""
listePays contient ici un ensemble de tout les noms de home_teams et away_teams, de manière unique. On le stocke dans un set,
pour garantir l'unicité de chaque équipe dans tout le match
"""

for index in data_result_match.index :
    data_result_match.loc[index, 'id_home_team'] = listePays.index(data_result_match.loc[index, 'home_team'])
    data_result_match.loc[index, 'id_away_team'] = listePays.index(data_result_match.loc[index, 'away_team'])

data_result_match['id_home_team'] = data_result_match['id_home_team'].astype(int) # pour avoir des entiers, par défaut c'étaient des floats
data_result_match['id_away_team'] = data_result_match['id_away_team'].astype(int) # pour avoir des entiers, par défaut c'étaient des floats
```

Fig 4 : Création des colonnes 'id_home_team' et 'id_away_team'

Nous avons donc créé une liste contenant une fois chaque pays présent dans home_team ou away_team, puis chaque id_team est basé sur l'index dans cette liste de son nom correspondant.

Nous obtenons donc le dataframe suivant :

data_result_match.head(10)												
	date	home_team	away_team	home_score	away_score	tournament	city	country	neutral	win	id_home_team	id_away_team
0	1969-11-01	Italy	France	1	0	Euro	Novara	Italy	False	1	90	64
1	1969-11-01	Denmark	England	4	3	Euro	Aosta	Italy	True	1	47	54
2	1969-11-02	England	France	2	0	Euro	Turin	Italy	True	1	54	64
3	1969-11-02	Italy	Denmark	3	1	Euro	Turin	Italy	False	1	90	47
4	1975-08-25	Thailand	Australia	3	2	AFC Championship	Hong Kong	Hong Kong	True	1	170	10
5	1975-08-25	Hong Kong	New Zealand	0	2	AFC Championship	Hong Kong	Hong Kong	False	2	80	125

Fig 5 : DataFrame avec les colonnes id_home_team et id_away_team ajoutées

Mise en place de la colonne 'etat_match'

Se pose un dernier soucis sur la mise en forme des principales données. En effet, actuellement, nous avons trois principales colonnes : id_home_team, id_away_team, neutral. Neutral représente le fait que le match se passe sur terrain neutre (pas chez l'équipe home_team) ou non.

Le soucis est le suivant : soit un match France Italy sur terrain neutre. Actuellement, on peut le représenter dans deux sens :

- France Italy True et Italy France True (home_team, away_team, neutral)
- Pour prédire un résultat France Italy avec terrain neutre, on peut utiliser les deux comme support de prédiction.

Or actuellement, comme nous le verrons dans les comparaisons des modèles, le modèle considère ces deux cas comme différents pour prédire.

On génère donc tous les tuples (i, j, true/false), avec $i \neq j$. Chacun est placé en temps que clé, avec un id unique en valeur. La valeur liée au tuple (i, j, true) est la même que celle du tuple (j, i, true).

```
# on commence par récupérer toute la liste des équipes de notre dataset, pour pouvoir générer tout les tuples possibles, on l'a déjà calculé, c'est listePays
listeTuple = [0] * pow(len(listePays), 2)
i = 0
for tuple1 in listePays:
    for tuple2 in listePays:
        listeTuple[i] = (tuple1, tuple2)
        i += 1
# on a généré tout les tuples possibles, y compris les (i, i), les (i, j) et (j, i). On ne veut pas les (i, i), on va directement les supprimer
listeTuple = [tuple for tuple in listeTuple if tuple[0] != tuple[1]]

etat_match_id = dict()
i = 0
# il reste à les rentrer dans le dictionnaire
for tuple in listeTuple:
    etat_match_id[tuple + (True,)] = i
    i += 1
    etat_match_id[tuple + (False,)] = i
    i += 1

# il reste un soucis, on veut que (i, j, true) et (j, i, true) ait le même id, ce qui n'est pas le cas actuellement. Le but est donc de tout bien remettre en place
for current_tuple in etat_match_id:
    if (current_tuple[2] is True):
        # alors pour (i, j, true), il existe forcément (j, i, true)
        etat_match_id[(current_tuple[1], current_tuple[0], current_tuple[2])] = etat_match_id[current_tuple]

# notre dictionnaire est prêt, reste à générer la nouvelle colonne, nommé etat_match
for index in data_result_match.index:
    data_result_match.loc[index, 'etat_match'] = etat_match_id[(data_result_match.loc[index, 'home_team'], data_result_match.loc[index, 'away_team'], data_result_match.loc[index, 'neutral'])]
data_result_match['etat_match'] = data_result_match['etat_match'].astype(int)
```

Fig 6 : Création de la colonne 'etat_match'

On affecte ensuite chaque id pour tout les tuples du dataSet en fonction de leur home_team, away_team et neutral.

Nous obtenons donc le dataframe suivant :

data_result_match.head(10)													
	date	home_team	away_team	home_score	away_score	tournament	city	country	neutral	Win	id_home_team	id_away_team	etat_match
0	1969-11-01	Italy	France	1	0	Euro	Novara	Italy	False	1	90	64	34509
1	1969-11-01	Denmark	England	4	3	Euro	Aosta	Italy	True	1	47	54	18060
2	1969-11-02	England	France	2	0	Euro	Turin	Italy	True	1	54	64	20754
3	1969-11-02	Italy	Denmark	3	1	Euro	Turin	Italy	False	1	90	47	34475
4	1975-08-25	Thailand	Australia	3	2	AFC Championship	Hong Kong	Hong Kong	True	1	170	10	4158
5	1975-08-25	Hong Kong	New Zealand	0	2	AFC Championship	Hong Kong	Hong Kong	False	2	80	125	30809

Fig 7 : DataFrame avec la colonne etat_match ajoutée

Mise en place de la colonne 'id_tournament'

Par la suite, nous voudrions étudier si le tournoi dans lequel a lieu le match a une influence pour l'issue du match.

```
""" On ajoute une colonne id_tournois pour connaitre les tournois, sous forme numérique pour traitement """  
  
liste_tournois = sorted(list(pd.unique(data_result_match['tournament'])))  
for index in data_result_match.index :  
    data_result_match.loc[index, 'id_tournois'] = liste_tournois.index(data_result_match.loc[index, 'tournament'])  
data_result_match['id_tournois'] = data_result_match['id_tournois'].astype(int)
```

Fig 8 : Création de la colonne 'id_tournament'

De la même façon que pour les id de teams, on génère une liste de tous les tournois présents dans nos données. Par la suite chaque id de tournoi correspond à son index dans la liste.

Au final, on obtient le DataFrame suivant :

data_result_match.head(10)														
	date	home_team	away_team	home_score	away_score	tournament	city	country	neutral	Win	id_home_team	id_away_team	etat_match	id_tournoi
0	1969-11-01	Italy	France	1	0	Euro	Novara	Italy	False	1	90	64	34509	13
1	1969-11-01	Denmark	England	4	3	Euro	Aosta	Italy	True	1	47	54	18060	13
2	1969-11-02	England	France	2	0	Euro	Turin	Italy	True	1	54	64	20754	13
3	1969-11-02	Italy	Denmark	3	1	Euro	Turin	Italy	False	1	90	47	34475	13
4	1975-08-25	Thailand	Australia	3	2	AFC Championship	Hong Kong	Hong Kong	True	1	170	10	4158	2
5	1975-08-25	Hong Kong	New Zealand	0	2	AFC Championship	Hong Kong	Hong Kong	False	2	80	125	30809	2

Fig 9 : DataFrame avec la colonne id_tournoi ajoutée

Explication des colonnes du dataSet

Nous avons en figure 8 le DataFrame final, prêt pour les modèles de SkLearn.

Nous avons la date du match (YYYY-MM-DD), la première équipe (équipe à domicile si neutral est Faux), et la seconde équipe se confrontant.

Nous avons ensuite le score effectué par la première puis la seconde équipe lors du match.

Nous avons le nom du tournoi, la cité et le pays dans lequel se déroule le tournoi. Neutre représente si le match se déroule sur terrain neutre ou non. La colonne Win vaut 0 pour une égalité, 1 pour une victoire de la première équipe et 2 sinon. Elle se base sur les deux colonnes de score.

Se trouve ensuite les id des équipes, l'état du match (qui se base sur les deux équipes et le fait que le match se passe sur terrain neutre). On trouve ensuite l'id du tournoi.

neutral, id_home_team, id_away_team, etat_match et id_tournoi seront les colonnes exploitées par nos modèles.

Matrice de corrélation

Nous avons commencé par étudier la corrélation entre nos colonnes. Pour ce faire, nous avons utilisé `pandas.DataFrame.corr()`, calculant les corrélations entre toutes les colonnes. Cette méthode propose plusieurs méthodes pour calculer ces corrélations. Il existe la méthode de Pearson, de Spearman et de Kendall.

Quel modèle choisir ?

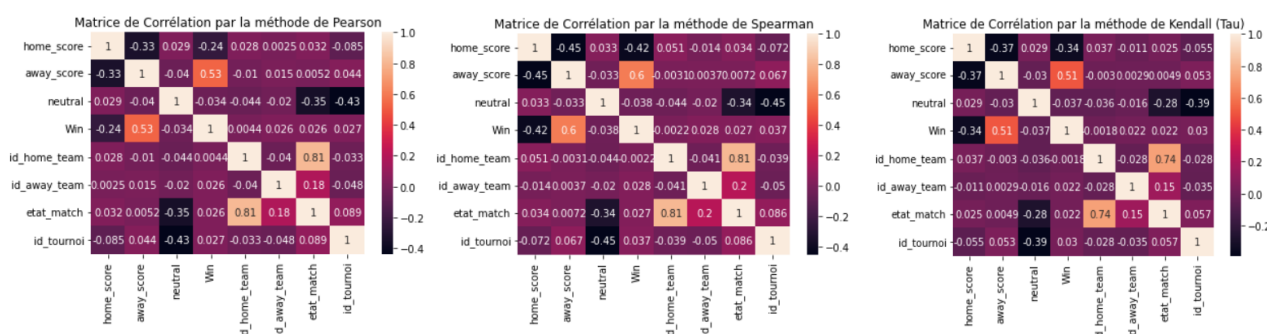


Fig 10 : Matrice de corrélation selon les trois méthodes de calculs

Le modèle de Pearson possède pour hypothèse que les variables à corrélérer soient continues ou ordinales appariées.

Le soucis du modèle de Pearson est qu'il est délaissé sur de petits échantillons. Cependant, nous en possédons un important, de plus de 4000 données.

Nos variables suivent en revanche bien une loi normale, comme nous avons pu le voir lors du cours de Probabilité&Statistiques, où toutes variables aléatoires répétées à l'infini tends vers une loi normale.

Le modèle de Kendall s'appliquent aux variables continues ou ordinales appariées. Nos variables ne sont pas continues, ni naturellement ordonnée (ordinales). De plus, nos variables sont normales, ce qui favorisent encore plus le modèle de Pearson. Le modèle de Kendall est le plus pessimiste des trois, cela se voit bien dans notre matrice.

Le modèle de Spearman est un cas particulier du modèle de Pearson. Il a pour spécificité de détecter en plus les relations monotones, qui ne possèdent donc pas de variations d'évolutions de X en fonction de Y, avec X et Y deux variables aléatoires. Cependant, les variables doivent être continues ou ordinales appariées

Nous allons ici préférer le premier modèle :

- Nous pouvons observer une corrélation positive (comprise entre 0.7 et 1) entre etat_match et id_home_team.
- On observe un taux à 0.5 entre Win et away_score, probablement car la seconde équipe gagne souvent, et que le reste se partage entre victoire de l'équipe 1, et égalité.

Séparation des jeux de tests et des jeux d'entraînements

Pour réaliser notre classification il est nécessaire de séparer nos deux jeux données en deux jeux :

- un jeu d'entraînement (x_train et y_train) pour entraîner les modèles
- un jeu de test (x_test, y_test) pour tester le modèle

```
""" Ici nous choisissons d'utiliser 75% du dataset pour entraîner le modèle et 25% pour effectuer nos tests."""  
x_train, x_test, y_train, y_test = train_test_split(data_result_match[['id_home_team', 'id_away_team', 'neutral']],  
                                                  data_result_match['Win'], test_size = 0.25, random_state = 0)
```

Fig 12 : Utilisation de la fonction train_test_split

Dans le premier paramètre de la fonction train_test_split on précise quels critères vont être utilisés pour la classification. Dans le second paramètre de la fonction train_test_split on précise quels sont les critères (les colonnes) qui doivent être utilisés pour classer les rencontres de football, dans notre cas ce sera toujours la colonne 'Win'. On met la test_size à 0,25 c'est à dire que 25 % des données seront utilisées dans le jeu de test et 75 % seront utilisées dans le jeu d'entraînement. Le paramètre random_state est laissé à 0 pour qu'à chaque exécution la répartition des données dans les deux jeux (test et train) soient identiques.

- x_train contient les colonnes 'id_home_team', 'id_away_team' et 'neutral' pour les données d'entraînements.
- x_test contient les colonnes 'id_home_team', 'id_away_team' et 'neutral' pour les données de tests.
- y_train contient la colonne 'Win' pour les données d'entraînements.
- y_test contient la colonne 'Win' pour les données de tests.

Cette répartition a été effectuée deux fois supplémentaires dans notre projet car nous avons voulu tester nos modèles en utilisant différentes colonnes pour effectuer la classification.

```
""" Ici nous choisissons d'utiliser 75% du dataset pour entraîner le modèle et 25% pour effectuer nos tests."""  
x_train, x_test, y_train, y_test = train_test_split(data_result_match[['id_home_team', 'id_away_team', 'etat_match']],  
                                                  data_result_match['Win'], test_size = 0.25, random_state = 0)
```

Fig 13 : Utilisation de la fonction train_test_split avec d'autres colonnes

```
""" Ici nous choisissons d'utiliser 75% du dataset pour entraîner le modèle et 25% pour effectuer nos tests."""  
x_train, x_test, y_train, y_test = train_test_split(data_result_match[['id_home_team', 'id_away_team', 'etat_match', 'id_tournoi']],  
                                                  data_result_match['Win'], test_size = 0.25, random_state = 0)
```

Fig 14 : Utilisation de la fonction train_test_split avec les colonnes qui fournissent la meilleure classification

Utilisation des modèles et comparaison

Nous avons utilisé trois modèles qui permettent de réaliser notre classification :

- DecisionTreeClassifier (arbre de classification)

```
[ ] """ Nous entraînons le modèle en utilisant la méthode de classification avec l'arbre de décision """

arbre_decision = DecisionTreeClassifier(random_state = 0, max_depth = 20)
clf = arbre_decision.fit(x_train, y_train)

[ ] """ Nous effectuons notre classification sur notre jeu de test en utilisant le modèle entraîné avec
la méthode de classification avec l'arbre de décision """

sPredict = clf.predict(x_test)

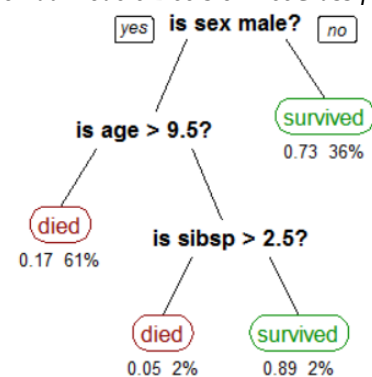
scoreArbre3 = accuracy_score(y_test, sPredict)

print(scoreArbre3)

0.5819366852886406
```

Fig 15 : Utilisation du modèle DecisionTreeClassifier*

L'arbre de décision fonctionne avec un arbre. Chaque nœud représente un booléen, pour déterminer si un élément précis de l'entrée est vrai ou faux. Quand on donne une entrée en prédiction, il descend l'arbre pour trouver les sorties correspondant aux entrées données.



Le paramètre `random_state` est une sorte de graine pour le random, c'est à dire qu'à chaque exécution nous aurons les mêmes valeurs pour la classification. Le paramètre `max_depth` définit la profondeur de l'arbre de décision utilisé. Nous utilisons une profondeur de 20 car une profondeur trop grande concentrerait la classification sur le jeu d'entraînement en prenant moins en compte le test. Une profondeur trop petite ferait tendre la classification vers le hasard car le jeu d'entraînement serait peu utilisé pour classer les matchs. Dans les deux cas, la précision de la classification serait impactée négativement (l'`accuracy_score` serait plus faible).

- Kneighbors (le plus proche voisin)

```
[ ] """ Nous entraînons le modèle en utilisant la méthode de classification avec le plus proche voisin """

KNN = KNeighborsClassifier()
clf = KNN.fit(x_train, y_train)

[ ] """ Nous effectuons notre classification sur notre jeu de test en utilisant le modèle entraîné avec
la méthode de classification avec le plus proche de voisin """

sPredict = clf.predict(x_test)

scoreKNN3 = accuracy_score(y_test, sPredict);
print(scoreKNN3)

0.5400372439478585
```

Fig 16 : Utilisation du modèle Kneighbors

Le modèle de Kneighbors prends k (par défaut 5) entrées, dont les valeurs sont les plus proches de celle de l'entrée à prédire. A partir des sorties (ici Win) de ces entrées, KNN prédit la sortie.

- SVM

```
[ ] """ Nous entraînons le modèle en utilisant la méthode de classification avec SVM """
clf = svm.SVC(gamma = 0.001)
clf.fit(x_train, y_train)

SVC(gamma=0.001)

[ ] """ Nous effectuons notre classification sur notre jeu de test en utilisant le modèle entraîné avec
la méthode de classification avec SVM """

sPredict = clf.predict(x_test)

scoreSVM3 = accuracy_score(y_test, sPredict);
print(scoreSVM3)

0.5921787709497207
```

Fig 17 : Utilisation du modèle SVM

Le modèle SVM essaye de déterminer une frontière de toutes les entrées, qui possèdent l'issue en commun. Ensuite, lorsqu'on lui donne une entrée, il cherchera à la placer dans une partie déterminée par les frontières, qui déterminera sa sortie.

Le modèle SVM prend pour paramètres :

- Le C qui sert à lisser les données, plus le C est petit moins il y aura d'erreurs, et la réciproque est vraie.
- Le gamma est utile lorsqu'on utilise un « kernel gossian RBF » (c'est d'ailleurs ce qu'on utilise), plus il est petit moins la ligne de décision sera courbée.

On aurait pu tester à l'aide de GridSearchCV plusieurs associations C, gamma et kernel pour déterminer avec lequel on a le plus d'accuracy, cependant son utilisation surcharge le notebook, rendant l'exploitation de GridSearchCV impossible.

Ces modèles ont été utilisés trois fois avec à chaque fois des colonnes différentes pour effectuer la classification :

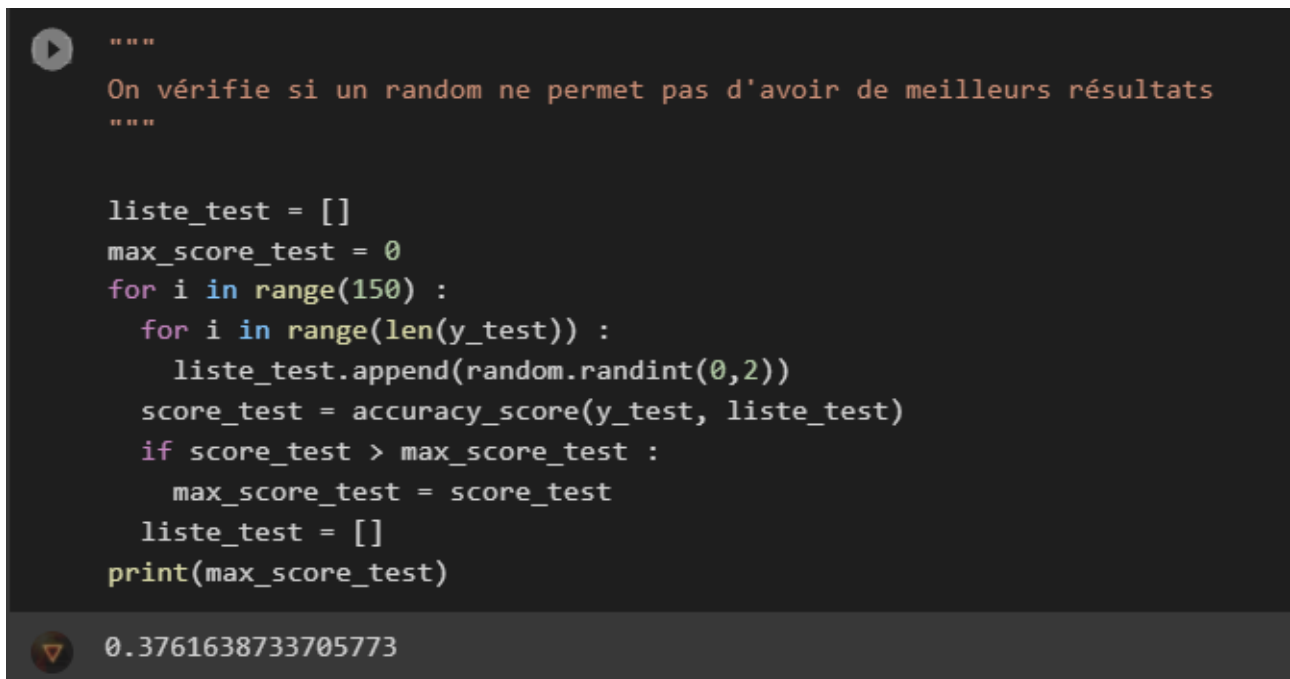
- Jeu numéro 1 : x_test et y_test contiennent les colonnes : 'id_home_team', 'id_away_team' et 'neutral'
- Jeu numéro 2 : x_test et y_test contiennent les colonnes : 'id_home_team', 'id_away_team' et 'etat_match'
- Jeu numéro 3 : x_test et y_test contiennent les colonnes : 'id_home_team', 'id_away_team', 'etat_match' et 'id_tournois'

	Jeu numéro 1	Jeu numéro 2	Jeu numéro 3
DecisionTreeClassifier	0,573	0,563	0,582
Kneighbors	0,556	0,533	0,540
SVM	0,534	0,588	0,592

Fig 18 : Tableau contenant les accuracy_score en fonction des colonnes (servant à la classification) et du modèle

On observe donc des variations de l'accuracy_score en fonction du modèle mais également en fonction des colonnes utilisées pour la classification. La « meilleure classification » est donc obtenue avec le modèle SVM en utilisant le jeu numéro 3. L'arbre de décision semble également assez fiable. On peut par ailleurs noter que le tournoi semble jouer un rôle pour faciliter la prédiction de l'issue du match.

Nous avons aussi testé la classification des matchs en utilisant le hasard.



```
"""
On vérifie si un random ne permet pas d'avoir de meilleurs résultats
"""

liste_test = []
max_score_test = 0
for i in range(150) :
    for i in range(len(y_test)) :
        liste_test.append(random.randint(0,2))
    score_test = accuracy_score(y_test, liste_test)
    if score_test > max_score_test :
        max_score_test = score_test
    liste_test = []
print(max_score_test)
```

0.3761638733705773

Fig 19 : Classification en utilisant le hasard

Le but de cette classification est de vérifier l'intérêt et le fonctionnement de nos modèles. On a un `accuracy_score` de 0,38 au maximum (lors de 150 classifications effectuées au hasard), bien inférieur à ceux de nos modèles (qui naviguent entre 0,5 et 0,6). On peut en déduire que nos modèles sont plus intéressants que de laisser le hasard faire les choses. Nos modèles ont donc un intérêt et permettent de réaliser une classification des matchs plus fiable que le hasard. On aurait également pu utiliser le modèle `DummyClassifier` pour réaliser la classification aléatoire.

Conclusion

Nous parvenons, avec SVM, à pratiquement 0.59 de prédiction réussie. Cela est explicable par le fait que chaque équipe nationale possède un niveau qui fluctue relativement peu sur des durées longues. Ainsi, un match A contre B possédera une issue relativement similaire sur une période de temps assez étendue.

Cela ne serait pas le cas sur des clubs de foot, où les joueurs bougent beaucoup de clubs, et jouent donc un grand rôle dans la compétitivité du club.

Bien entendu, pour avoir une prédiction plus fine, on pourrait prendre en compte des données sur les joueurs de chaque équipe, ce qui permettrait d'augmenter le score de la prédiction de manière significative.

Bibliographie

<https://lemakistatheux.wordpress.com/2013/05/21/le-coefficient-de-correlation-et-le-test-de-pearson/>

<https://lemakistatheux.wordpress.com/2013/05/26/le-coefficient-de-correlation-et-le-test-de-kendall/>

<https://louernos-nature.fr/test-correlation-statistique-langage-r/>

<https://blocnotes.iergo.fr/breve/nominales-ordinales-intervalles-et-ratios/>

<https://zestedesavoir.com/tutoriels/1760/un-peu-de-machine-learning-avec-les-svm/#1-probleme-de-classification-svm-vous-pouvez-repeter-la-question>