Université Paris 1, Panthéon Sorbonne

Predict brain deep sleep slow oscillation : Machine Learning Parallélisation

Auteurs :
BRIDA Kevins
VILLE Solal
YAPI Wilfried

Référent :
GOULET Clément

Table des matières

1 Présentation du projet

1.1 ENS Data Challenge : Entreprise Dreem

L'ensemble des éléments relatifs à notre sujet a été mis à disposition par la plateforme de challenge de données *Challengedata.ens*. A l'instar de la plateforme *Kaggle*, cette plateforme fournit un accès facile à des datasets de Machine Learning supervisés, ceci à des fins éducatives. Les ensembles de données étant suffisamment petits pour pouvoir être étudiés sur des machines standards, tout en conservant des caractéristiques intéressantes sur données réelles.

Nous avons choisi de nous intéresser, pour ce projet, à un challenge proposé par l'entreprise Dreem, une startup spécialisée dans les problèmes de sommeil. Leur produit, le *bandeau Dreem*, vise à améliorer la qualité du sommeil des individus, en stimulant certaines parties du cerveau lors des différentes phases de sommeil. L'objet d'étude de notre projet s'effectue sur des enregistrements de 10 secondes, de signaux cérébraux caractéristiques des phases de sommeil profond appelées "oscillation lente". La mesure ainsi que l'enregistrement de ces ondes, symbolisant une forte activité cérébrale, est réalisée par le bandeau dreem sous forme d'EEG (électroencéphalogramme).

Notre objectif est, à partir de la base de données à notre disposition, de prédire si une oscillation lente sera suivie d'une autre lors de la seconde suivant la fin l'enregistrement, en condition *sham* (ie : sans stimulation du bandeau). On se place ici en condition *sham* afin de pouvoir déterminer avec exactitude le moment exact où il est utile de stimuler, ce qui ne sera pas le cas si on prédit une oscillation lente dans la seconde qui va suivre.

1.2 Principes du parallélisme

Le parallélisme est une technique informatique visant à utiliser plusieurs processeurs pour traiter des opérations de manière simultanée : Le but est que la réalisation de toutes ces opérations s'exécute en réduisant le plus possible le temps de traitement.
Cette technique s'écarte de la méthode traditionnelle dite "séquentielle". Les opérations sont traitées de manière successive et le temps d'exécution demeure plus élevé qu'avec du parallélisme.

Le parallélisme peut s'appliquer en utilisant plusieurs processeurs. Le problème en question se retrouve subdivisé en plusieurs parties qui peuvent être résolues de manières concurrentes. Chaque partie se retrouve alors transformée en une série d'instructions étant exécutées par les différents processeurs.

Le parallélisme permet :
- un gain de temps
- un traitement de problèmes plus complexes, dont on aurait du mal à faire face avec une simple approche séquentielle (traitement de grande données)
- de ne pas forcément utiliser des ressources locales

Un classement des différentes machines a été proposé par Flynn en 1966 :
- SISD : Il y a une seule unité de calcul qui a accès à une seule donnée à la fois. C'est un exemple

de l'approche séquentielle et il n'y a aucune parallélisation.
- MISD : Plusieurs processeurs traitent une même donnée mais appliquent des instructions multiples et différentes.
- SIMD : Une même instruction s'opère simultanément, grâce à plusieurs unités de calcul, sur plusieurs données pour avoir des résultats différents.
- MIMD : Plusieurs machines où les processeurs possèdent leur propre mémoire et n'ayant pas accès à celle des autres. Les processeurs sont cependant reliés sur le même réseau. Les différentes machines peuvent coïncider avec le parallélisme sauf la première. S'ajoute à ça le choix du Multiprocessing ou du Multithreading (même si empiriquement les deux sont complémentaires).

Le Multiprocessing consiste à l'ajout de plusieurs processus pour augmenter la vitesse de réponse et de calcul du système. La séparation des processeurs permet une gestion des ressources conjointe plus efficace dans la mesure où un processus, une fois subdivisé, est moins sujet à engorger l'appareil sur lequel il effectue une tâche (Notion de CPU limité). Il existe 2 sous-catégories de Multiprocessing que sont : Le Multiprocessing symétrique et le multiprocessing asymétrique. Le Multiprocessing symétrique est le fait que les processus fonctionnent de manière libre et non contrainte dans le système. Le Multiprocessing Asymétrique modélise une relation de subordination entre les processeurs. Un processeur peut interdire à un autre de réaliser certaines opérations : Il y a donc une répartition des tâches.
Le Multithreading consiste à la multiplication des « threads », tâche, pour un processus. Cette création de thread est plus efficace qu'une création supplémentaire de processus qui épuise les ressources et qui est chronophage. Augmenter le nombre de tâches permet d'augmenter le niveau de réponse du système dans la mesure où une tâche devient plus lente, le processus peut continuer de fonctionner.

Ensuite vient le choix du modèle. Le modèle synchrone ou le modèle asynchrone.
Dans le modèle synchrone il existe une dépendance entre les tâches, de ce fait il faut attendre la fin de réalisation d'une tâche avant de passer à la suivante (First-In-First-Out).

Au contraire, le modèle asynchrone permet de passer à une autre tâche même si la tâche effectuée précédemment reste en cours de réalisation. Chaque machine possède sa propre mémoire et agit de manière indépendante.

1.3 Application sur des algorithmes de Machine Learning

Pour pouvoir effectuer des prédictions sur ce challenge, nous ferons appel à différents algorithmes de Machine Learning. Nous nous plaçons ici dans une problématique de classification, nos labels en sortie pouvant prendre les valeurs :

Labels = 0,1,2
0 = Pas d'oscillation dans la sec qui suit
1 = Une oscillation faible dans la sec qui suit
2 = une oscillation forte dans la seonde qui suit

Les algorithmes utilisés pour des applications de Machine Learning peuvent être "lourds" et avoir une forte empreinte mémoire pour des systèmes informatiques "standard". Encore plus lorsque que l'on est confronté à des données de grande dimension, comme c'est la cas ici.Nos données d'entrainements sont de :

Xtrain = 261,634 (obervations) pour 1261 variables soit 261.634 x 1261
Ytrain = 261,634 (observations) soit 261634 x 1

Les 11 premières variables permettent de caractériser chacun de ces enregistrements (stade de sommeil, amplitude de l'oscillation actuelle, durée de l'oscillation actuelle etc.) . Les 1250 colonnes suivantes correspondent quant à elles aux 10 sec d'enregistrement EEG pour chaque observation (1250 points soit 125 colonnes/sec). Notre Xtrain avant traitement préalable à une empreintre mémoire (à l'importation) de 2.4 go (2,408.14 mo) et de 2.13 mo pour notre Ytrain.

En outre, même si ces algorithmes restent relativement simples à implémenter, la méthode que nous utiliserons ici pour résoudre notre problématique se montrera très gourmande en temps. La cross validation ainsi que le split entre données d'entrainements et données de validation (ou test) vont augmenter considérablement le temps d'éxecution de notre programme. De même, il n'est pas recommandé d'entrainer nos données sur un seul modèle, ceci car si effectivement un algorithme est performant sur un jeu de donnée, rien ne dit qu'il le sera sur des données jamais rencontrées. Nous avons donc entrainé plusieurs modèles et leurs avons appliqué la méthode dîte de "cross validation".

Ici la parallélisation va se révéler déterminante afin d'optimiser la vitesse d'éxecution de notre code. En effet appliquer ces différentes procédures sur nos processeurs nous permet de gagner un temps considérable et de réduire l'empreinte mémoire de notre dataset initiale. Dans la suite de ce document nous montrerons l'importance du gain temps/mémoire réalisé en faisant appel à ces méthodes de multiproccessing et de multhreading.

2 Analyse de code

Nous avons décidé de subdiviser notre travail en deux parties. En effet, ne pouvant pas lancer deux parallélisations distinctes sur un même code via l'utilisation de la méthode *Pool* nous avons pris la décision d'adopter une approche différenciée. Dans notre première fiche de code nous avons importé nos données de manière parallélisée en subdivisant notre base en 8 parties distinctes tout en extrayant par la même occasion des informations utiles/essentielles pour la compréhension de notre challenge (Section 2.1).
Nous avons ensuite enregistré notre transformation de la base Xtrain sous forme de *Pickle* que nous appelons dans notre deuxième fiche de code et sur laquelle nous appliquons des algorithmes de Machine Learning. Cette méthode nous a permis d'effectuer deux parallélisations sur des éléments dictincts et d'accélérer considérablement l'éxecution de nos processus (Section 3.1.1).

2.1 Extraction et Exploration de données

2.1.1 Objectifs et Méthodes utilisés

L'ensemble de nos données, qu'il s'agisse du Xtrain ou du ytrain sont enregistrées au format «float64 » dans un dossier HDF5. Le format HDF5 (pour Hierarchical Data Format, V5) est un format de type conteneur de fichier. Ce format est donc assimilable à une arborescence de dossiers/fichiers, le tout contenu dans un fichier. Au vu du volume important des données, il était impératif d'affecter à chaque variable un type optimal afin qu'elles occupent une place mémoire la plus petite possible. De manière générale, un ensemble de lignes de code est exécuté par un unique processeur, qu'il s'agisse de tâches différentes ou d'une même tâche répétée sur différentes données. Ce qui peut mettre un temps considérable car elles sont exécutées de manière séquentielle. Il aurait été moins avantageux pour nous de lancer des algorithmes assez complexes de machine Learning et d'optimisation sur de telles données non traitées.

La première étape a donc consisté pour nous, à répartir nos 261634 observations sur nos 8 processeurs,soit 32704 observations/processeur (avec un reste de 2 obs). Ainsi chaque processeur pouvait exécuter un ensemble d'algorithmes d'optimisation et d'exploration statistique sur une base de données plus restreinte. Les processeurs accompliront ces tâches de manière simultanée et se montreront plus rapides,par rapport à l'approche séquentielle sur un seul processeur.

La deuxième partie de la parallélisation a consisté à effectuer différentes tâches sur nos 261,632 observations (Nous avons volontairement retiré 2 observations afin de faire correspondre parfaitement les dimensions de nos données) sur tous nos processeurs.
Pour ce faire nous utilisons essentiellement deux fonctions :
La fonction « Pool » provenant du module « Multiprocessing » permet d'ouvrir un nombre donné de processeurs.
La fonction « Map » contenue dans les fonctions de base de Python, permet d'exécuter une fonction donnée sur des arguments distincts et renvoie le résultat de la fonction pour chaque argument dans une liste.
La combinaison de ces deux fonctions permet à chaque processeur ouvert, d'exécuter la fonction donnée sur un ou plusieurs arguments de manière simultanée. Ces deux méthodes seront donc regroupées dans une fonction qui sera enclenchée par un processeur principal ; et le reste du

code contenu dans le fichier sera exécuté par l'ensemble des processeurs ouverts tout en prenant chacun un ou plusieurs arguments différents.
Cette dernière est elle-même lancée dans une condition : if name= 'main' , cela permet juste d'exécuter le code contenu uniquement dans notre ficher principal et non des scripts importés d'autres fichiers.
Enfin nous stockerons nos données à l'intérieur d'un fichier *Pickle*

2.1.2 Definition et parallélisation des fonctions : *opti / seg*

Viens ensuite l'attribution de format adéquats en fonction des échelles considérées ci-dessous, conditionnellement à la valeur de l'addition des premiers chriffres après la virgule.

```
def opti3(li) :
    sam=0
    for l in li:
        l = str(l)
        a = l.split(".")
        b = int(a[1][0])
        sam = sam + b
```

Listing 2.1 – opti3 partie 1

Si la valeur est nulle : la fonction va prendre la valeur absolue du maximum de la colonne, si elle est inférieure à 127,la colonne est mise en « int8 » avec la fonction « astype() ». Si cette valeur absolue est moins de 32767 la colonne est mise en « int16 », sinon on la met en « int32 ».

```
 if  sam == 0:
        if abs(max(li)) <= 127:
            li= li.astype ("int8")
        elif abs(max(li)) <= 32767:
            li= li.astype ("int16")
        elif abs(max(li)) <= 2147483647:
            li= li.astype ("int32")
```

Listing 2.2 – opti3 partie 2

Cependant si la somme des premiers éléments après la virgule est supérieure à zéro, c'est une colonne de réels. Dans ce cas elle est mise en « float16 » si la valeur absolue du maximum de la colonne est inférieure à 32767 sinon la colonne est mise en « float32 »

```
if sam  > 0:
        if abs(max(li)) <= 32767:
            li= li.astype ("float16")
        elif abs(max(li)) <= 2147483647:
            li= li.astype ("float32")
```

L'algorithme cible de notre premier programme de parallélisation est contenu dans la fonction « seg » qui prend en argument un nombre (nb) et une variable globale qui est une liste de liste vide(dtas). Cette fonction va nous permettre d'extraire les observations par intervalle de 32704 observations selon la formule suivante : [nb*32704 :(nb*32704+32704)]. La formule nous permet d'affecter le périmètre de travail de chaque processeur. La logique est la suivante : Le processeur ayant pour argument 0 téléchargera les observations de 0 à 32704 tant dis que celui ayant pour argument 1 s'occupera des obsersations de 32704 à 65408.... Les données (Xtrain et Ytrain) sont ensuite transformer en DataFrame.

```
dtas = [[], [],[],[],[],[],[],[]]
rest = []
rest2 = []
def seg (nb):
    global dtas
    cont = File("X_train.h5", "r")["features"][nb*32704:(nb*32704+32704)]
    conty =read_csv("y_train_2.csv").as_matrix()[:, 1].squeeze()[nb*32704:(nb
    *32704+32704)]
    conty = DataFrame(conty)
    cont = DataFrame(cont)
```

Afin d'induire du multithreading dans le multiprocessing, nous avons créé 2 fonctions. La première fonction consistera à appliquer "opti3" sur les 11 premières variables. La deuxième fonction aura pour champ d'action les autres variables . Nous avons observé le fait qu'intégrer ces variables dans la fonction opti les transforme en float16 ; de ce fait nous avons choisi d'y appliquer ce format à la main pour un gain de temps. Pour chaque observation, cette fonction va nous permettre d'obtenir les statistiques suivantes : variance, minimum et maximum ; et les regroupe dans une DataFrame.Ces deux fonctions sont lancées de manière asynchrone (= indépendante).

Le Multithreading a son importance dans notre manière de procéder dans la mesure où travailler sur les 11 premières et sur les autres n'est pas la même. La pertinence de cet outil survient car il permet d'avoir un gain de temps significatif. Par la suite, nous allons assembler les trois bases (les Xtrain, les Ytrain et les Statistiques), de sorte à ce qu'on ait le Xtrain et le Ytrain qui soient affichés avec leur statistiques. Enfin vient l'exploiration des données qui nous donne la heatmap de corrélation entre les variables cibles et leur histogrammes.

```
def op1():
    global rest
    rest = cont.iloc[:,0:11].apply(opti,axis=0)
def op2():
    global rest2
    rest2 = cont.iloc[:,11:].astype("float16")
    mini =rest2.T.min()
    maxi =rest2.T.max()
    varr =rest2.T.var()
    rest2 = concat([mini,maxi,varr], axis=1).reindex(rest2.index)

    th1 = Thread(target=op1)
    th2 = Thread(target=op2)

    th1.start()
    th2.start()
    th1.join()
    th2.join()

    dt = concat([conty,rest2,rest],axis=1)

    X_train_cor = dtas.iloc[:,0:11].corr()
    top_corr_features = X_train_cor.index
    figure(figsize=(20,20))
    heatmap(dtas[top_corr_features].corr(), annot=True, cmap='RdYlGn')
    dtas.hist(bins=50, figsize=(20,15)) #cumulative="TRUE" for CDF

    return dtas
```

Ensuite nous allons appliquer la fonction "Seg" sur numbers (qui est défini dans If Main), qui affectera un argument par processeur : le résultat affiché sera une liste de DataFrames. Nous allons donc les concaténer pour obtenir une Dataframe. Cependant cette Dataframe ne sera pas complète car nous avions défini un champ d'action pour chaque processeur. C'est pour cela nous

redéfinissons l'index pour qu'il puisse aller de 0 jusqu'à 261632. Cette DataFrame sera enregistrée dans le fichier Pickle (pour les raisons énoncées plus haut). Avec résultat nous utilisons close pour arrêter l'activité des différents processeurs à la fin de leur travail et join pour cesser l'activité du processeur principal : Cela permet à tous les processeurs de finir leur travail et d'afficher leur temps d'action global.

```
def seg_mp(numbers):
    start_time = time()
    p= Pool()
    result = p.map(seg, numbers)
    result = concat(result)
    result=result.set_index([Index(range(261632))])
    result.to_pickle("D:\\SORBONNE\\M1\\S1\\AML\\xtrain")
    print(result)
    p.close()
    p.join()
    end_time = time() - start_time
    print(f"Processing {len(numbers)} numbers took {end_time} time using
    multiprocessing.")

if _name=='main_':
    nbr = range(8)
    seg_mp(nbr)
```

Listing 2.3 – seg MP

2.1.3 Difficultés rencontrées

Les difficultés majeures de la première partie de la parallélisation peuvent se regrouper en deux parties. Tout d'abord, il a été très difficile de regrouper de manière ordonnée les résultats de chaque processeur dans une même liste. A cela est venu s'ajouter la difficulté des « variables globales » : l'utilisation d'une « variable globale » au sein d'une fonction qui est exécutée sur plusieurs arguments, entraine une modification successive du contenu de cette variable et cela peut renvoyer des résultats incohérants.

2.2 Apprentissage supervisé : *Logit, Random Forest, KNN*

2.2.1 Objectifs et Méthodes utilisés

Dans cette seconde partie nous aborderons l'aspect pratique de ce projet. Précedemment nous avons appliqué une parallélisation sur l'importation de données, nettoyé notre base et réalisé des traitements statistiques préalables à l'application d'algorithmes de Machine Learning. Désormais il nous faut prédire les sorties ypred et les comparer aux valeurs effectivement observées. Pour ce faire nous avons choisis d'utiliser trois méthodes que sont : La Regression logistique *(Logit)*, le Random Forest *(RDF)* et la méthode des K plus proches voisins *(KNN)*. Afin de pouvoir continuer à réduire le temps d'exécution de notre programme nous avons utilisé la méthode *Pool* du package *Mutlprocessing*.

Pour réaliser ces classifications nous avons fait appel à la bibliothèque de Machine Learning *Scikit-Learn*, notamment des packages *modelSelection, StandardScaler* pour les étapes de prépocessing et les packages *KNeighborsClassifier, RandomForestClassifier,LogisticRegression* pour les classifications elles-mêmes.

Pour obtenir des résulats optimaux nous avons procédé par étape. Dans un premier temps nous avons extrait le ytrain du dataset Xtrain. Une fois extrait grâce à la fonction unpickler, nous avons défini une liste de tuples*models*, qui regroupe nos trois classificateurs et leurs noms sous formes d'abbréviations. On a défini par la suite une fonction predict qui va appeler chaque éléments de la liste models un par un. Pour les 3 algorithmes ont utilisera la méthode *TrainTestSplit* qui nous a permi de fractionner notre matrice Xtrain en sous-ensembles aléatoires. Une fois les Xtrain, Xtest, Ytrain, Ytest déterminés nous avons appliqué une méthode de *scaling* sur nos Xtrain, Xtest lorsque nous les avons utilisés avec les algorithmes *Logit* et *RDF*. Pour l'algotihme *KNN* nous avons gardé notre Xtrain en l'état, l'application du scaling complexifiant de manière considérable la classification (Section 2.2.3). Nous avons ensuite appliqué la méthode *Kfold* qui permet de déterminer le nombre de fois ou l'on souhaite subdivisé puis entrainer nos différents algorithmes. On a entrainé ensuite ces 10 randoms split avec la méthode *CrossValScore* en fixant les sorties que l'on souhaitait obtenir sur *"accuracy"* comme le veulent les règles fixées pour le challenge.

2.2.2 Définition et parallélisation de la fonction : *predict*

Une fois les données extraites, traitées et enregistrées sur le pickle il s'agit désormais d'en sortir une prédiction. Dans un premier temps nous allons séparer la partie du Ytrain que l'on a sauvegardé avec le *Pickle*. Pour ce faire on sélectionne la colonne qui correspond à Ytrain grace à la commande *iloc*, on passe ensuite ce vecteur colonne en array de type *int8*. On choisi de passer notre Ytrain en array plutôt qu'en Dataframe car nos algorithme ont besoin d'un argument qui soit sous cette forme, le choix du int8 se fait quant à lui car il est préférable que notre Ytrain occupe le moins d'espace possible. Pour le Xtrain, il s'agira de l'ensemble des colonnes restantes du *Pickle* que nous avons gardé en mémoire à la fin de la phase 1 de notre code.

Par la suite on a défini une liste vide *models* à laquelle on ajoute 3 tuples qui correspondent chacun :
i) A l'abbréviation de l'algotihme de classification en premier argument
ii) Au modèle lui correspondant en deuxième argument
De même on a défini deux listes vides, la première *results* contiendra les résultats pour chaque modèle, la deuxième *names* contiendra les abbréviations des différents modèles.

```
y_train = data.iloc[:,:1].to_numpy().astype("int8")
test = data.iloc[:,1:]

models = []
models.append(('LR', LogisticRegression(solver='lbfgs', multi_class='ovr')))
models.append(('RDF', RandomForestClassifier(n_estimators=10)))
models.append(('KNN', KNeighborsClassifier()))
results=[]
names=[]
```

Désormais on a défini la fonction predict avec l'argument *mod*. On a établi un chronomètre qui nous permet de définir le temps d'éxecution de notre fonction. On fixe les variables, *test, Ytrain, results* et *names* comme des variables globales. Sans cela on aboutira à des résulats incohérents (on veut utiliser ces variables globales sans modifier leur valeur).
On a défini ensuite les variables N et M. N nous sert à sélectionner le premier élément du tuple, c'est à dire l'abbréviation de chaque modèle, M sert quant à lui à sélectionner le deuxième élément de chaque tuple, c'est à dire le classificateur.

Par la suite on introduit un *if-else* statement qui appliquera une méthode de *scaling* ou pas selon le modèle. Pour le classifier KNN nous n'appliquerons pas cette méthode de *features scaling* sur le datasat, le contraire accroitrait de façon exponentielle le temps d'éxecution de notre classification. Si le *if* statement n'est pas respecté alors on applique la méthode de features scaling ce qui accélerera considérablement la vitesse de classification ainsi que la précision, en particulier pour notre *Logit*. Après on utilise la méthode *Train-Test-Split* pour subdivisé nos dataset en plusieurs morceaux, le Xtrain étant la partie qui sera entrainé et le Xtest la partie qui permettra d'effectuer le test de performance sur données inconnues. Nous avons fixé la taille de Xtest, ytest à 0.2 de façon arbitraire ainsi que le *random-state*(=7) qui n'est autre que l'initialisation du générateur de nombres pseudo-aléatoires.

```

def predict(mod):
    start_time = time()
    global test, y_train , results, names
#for name,model in models:
    N = mod[0]
    M = mod[1]
    if N == 'KNN' :
        tests, X_test, y_train, y_test = model_selection.train_test_split(test.
    to_numpy(), y_train, test_size=0.20, random_state=7)

    else :
        sc = StandardScaler()
        X_train_scaled = sc.fit_transform(test)
        X_train_scaled =X_train_scaled.astype("float16")
        tests, X_test, y_train, y_test = model_selection.train_test_split(
    X_train_scaled, y_train, test_size=0.20, random_state=7)
```

Listing 2.4 – Predict partie 1

On introduit ensuite la fonction *simplefilter*, cette fonction nous permet de ne pas avoir de message "Future Warning" qui s'affiche pour l'affichage de nos résultats. Si de tels warning peuvent se révéler utiles afin de mettre en exergue des problèmes au niveau des algorithmes ici il n'en est rien, le message indiquant simplement un changement dans la notation au niveau des hyperparamètres par défaut des classificateurs. On s'en débarasse donc, tout en gardant en tete qu'ils existent. On applique ensuite la méthode *crossValScore* qui permet d'entrainer nos modèles sur des jeux d'entrainements et de les tester sur nos Xtest. Parmi les arguments

de la fonction *crossValScore* on retrouve la variable Kfold. Cette variable joue un rôle essentiel dans cette fonction. En effet plutôt que d'entrainer seulement un Xtrain et un Xtest on spécifie le nombre de fold sur lequel on souhaite effectuer nos cross-validation. Cette opération nous permet de sortir via la variable *results* dix scores pour chacun de nos trois algorithmes. On notera que *results* est sous la forme d'une liste de trois numpy array, chacun des éléments de cette liste représentant les dix scores, mesuré par l'*accuracy*, de nos trois algorithmes. On calcule ensuite la moyenne et la variance de ces dix scores pour nos trois modèles.

```
simplefilter(action='ignore', category=FutureWarning)
kfold=model_selection.KFold(n_splits=10,random_state=7)
cv_results= model_selection.cross_val_score(M,tests,y_train,cv=kfold,scoring='
    accuracy')
results.append(cv_results)
names.append(N)
msg = "%s: %f (%f)" % (N, cv_results.mean(), cv_results.std())
print(msg)
print("le temps mis est de : ",time()-start_time)
```

Listing 2.5 – Predict partie 2

Enfin nous allons appliquer la fonction *predictMP* sur numbers (qui est défini dans If Main), qui affectera un argument par processeur : Chaque élément de notre liste *models*. Ici on limite le recours à pool pour seulement 3 processeurs, l'équivalent du nombre de modèle à entrainer. Là encore la logique reste la même que dans la première phase de notre travail, nous utiliserons close pour arrêter l'activité des différents processeurs à la fin de leur travail et join pour cesser l'activité du processeur principal. On lance ensuite notre code sur l'invite de commmande.

```
def predictmp(numbers):
    start_time = time()
    p= Pool(3)
    result = p.map(predict, numbers)
    p.close()
    p.join()
    end_time = time() - start_time
    print(f"Processing {len(numbers)} numbers took {end_time} time using
    multiprocessing.")

if __name__=='__main__':
    predictmp(models)
```

Listing 2.6 – Predict mp

2.2.3 Difficultés rencontrées

Une des premières difficultés pour cette seconde parallélisation a été de choisir les bons algorithmes de classification pour cette présentation. L'objectif étant de trouver un score qui s'approche de ceux obtenus à partir du Benchmark, tout en automatisant la procédure au maximum. Rapidement certains algrothimes ont été écartés, c'est la cas notamment du SVM *(Support Vector Machine)*. Une autre difficulté a été au niveau des algorithmes sélectionnés eux mêmes. En effet, si on a pu constaté que nos algorithmes Logit et RDF avait une vitesse d'éxecution qui s'améliorait après avoir leur avoir appliqué la fonction scale, ce n'est pas le cas de l'algorithme KNN. Il s'agissait donc d'appliquer au deux premiers le dataset Xtrain scaled et le non scale au second. Relativemment simple à appliquer cette étape nous aura pri un temps considérable, notamment pour identifier la source de cette sous-performance. Une fois le problème décelé nous avons pu le résoudre, gagnant au passage un nombre de seconde non négligeable et améliorant notre accuracy sur ypred. (voir section 3.1.2)

3 Résultats

3.1 Gain issus de la parallélisation

3.1.1 Temps

i) Extraction et Exploration de données (CODE 1)

La première phase de notre projet est divisée entre la phase d'importation et d'extraction des données du dossier HDF5, suivie par une analyse statistique de l'ensemble de nos variables.

```
   ...:
   ...: data = trans(xtrain,ytrain)
   ...: print("le temps pour l'extraction et l'optimisation des données est de",time()-time1)
   ...:
   ...: time2 = time()
le temps pour l'extraction et l'optimisation des données est de 45.0361270904541
```

Avec notre code non parallélisé l'extraction ainsi que l'optimisation des données présentées dans (section 2.1) nous prend 45.036 secondes.

```
[261632 rows x 15 columns]
Processing 8 numbers took 18.377301454544067 time using multiprocessing.
```

Comparativement à cela on obtient avec notre code parallélisé, pour l'extraction, l'optimisation et les sorties graphiques, un temps de 18.37 secondes. On constate que le gain est important ici (26,6 secondes) soit 59.2% en terme de vitesse d'éxecution. Lorsqu'on procède par segmentation, la méthode séquentielle est sous-optimale.

ii) Classification supervisée (CODE 2)

En plus du gain issu de l'extraction on peut distinguer un second gain de temps en ce qui concerne les classifications. Plutôt que de les exécuter les unes à la suites des autres on les executes simultanément.

```
KNN: 0.452918 (0.001977)
le temps mis est de :  23.332470893859863
LR: 0.499529 (0.003175)
le temps mis est de :  11.162120580673218
RDF: 0.471461 (0.004126)
le temps mis est de :  49.54448580741882
le temps mis par les 3 modeles est :  84.48987102508545
```

Sans Parallélisation on trouve un résultat de 84.49 secondes contre 60.83 avec multiprocssing soit un gain de 28% en terme de vitesse d'execution.

```
LR: 0.499807 (0.004032)
le temps mis est de :  13.964880228042603
KNN: 0.454089 (0.003047)
le temps mis est de :  35.60889768600464
RDF: 0.473104 (0.003996)
le temps mis est de :  59.13483643531799
Processing 3 numbers took 60.83103632926941 time using multiprocessing.
```

3.1.2 mémoire

Pris isolément l'ensemble de notre dataset est sous la forme d'un *narray* de format 261,634 x 1261. L'ensemble de nos variables sont enregistrées au format *float64*. Ce dataset de base pèse 2.5 GB, lorsqu'il n'est pas optimisé.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 261634 entries, 0 to 261633
Columns: 1261 entries, 0 to 1260
dtypes: float64(1261)
memory usage: 2.5 GB
```

Le gain sur notre dataset est conséquent puisqu'il nous permet de convertir les variables dans le type qui leur correspond et qui minimise l'empreinte mémoire. On obtient donc un dataset de 629.5 MB, après optimisation. C'est à dire un gain de 74,82% par rapport à notre base initiale.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 261634 entries, 0 to 261633
Columns: 1261 entries, 0 to 1260
dtypes: float16(1253), int16(6), int32(1), int8(1)
memory usage: 629.5 MB
```

3.2 Accuracy de nos modèles

On a pu constaté que les résultats de nos classifications n'étaient pas impactés par la méthode utilisée, qu'elle soit séquentielle ou parallélisée. Cela va de soi, les opérations étant les mêmes, c'est seulement la vitesse d'éxécution de notre algorithme qui est impactée.

Toutefois, il est intéressant de regarder les éléments qui pourraient nous permettre d'accroître nos résulats et éventuellement la vitesse à laquelle nous les obtenons. A l'heure actuelle le 1er du classement de notre challenge détient un score de 0.5354, ce qui vient nous conforter dans l'idée que le choix des variables est déterminant afin d'optimiser le score de nos algotihmes.

L'apport d'observations supplémentaires, à partir de 100,000 observations, n'a ici qu'un impact limité sur l'amélioration de nos modèles (table 3.1). On pourra donc prendre la décision d'entrainer nos modèles sur les 100,000 premières obserations, cela afin de pouvoir accélérer encore un peu plus la vitesse d'éxécution de notre code.

Classificateur	Nb observations	accuracy	temps
Logit	100	0.587 (0.159)	0.149
	1000	0.417 (0.052)	0.164
	10,000	0.491 (0.009)	0.488
	100,000	0.499 (0.004)	5.068
KNN	100	0.575 (0.114)	0.027
	1000	0.405 (0.028)	0.065
	10,000	0.443 (0.018)	0.629
	100,000	0.452 (0.007)	11.059
RDF	100	0.625 (0.168)	0.134
	1000	0.422 (0.049)	0.158
	10,000	0.451 (0.014)	1.843
	100,000	0.469 (0.006)	23.403

TABLE 3.1 – Analyse des résultats

On constate également que c'est l'algorithme le "plus simple" qui obtient ici le meilleur résultat avec nos 261,632 obeservations. Dans leur Benchmark, l'entreprise Dreem utilise un algorithme de classification *RandomForest* avec cent arbres. On peut clairement affirmer au regard de nos résultats que l'utilisation d'un tel algorithme est ici inefficient et coûteux en temps, les améliorations à faire étant plutôt à chercher du coté des variables utilisées.

3.3 Améliorations

Pour améliorer nos résultats, nous aurions pu utiliser une méthode nous permettant d'optimiser le réglage de nos hyperparamètres à travers la méthode Grid Search. En effet cette dernière teste différentes combinaisons de valeurs associées aux hyperparamètres et les optimise. Il est logique qu'il ne suffise pas d'optimiser chaque hyperparamètre et d'y assembler les résultats. L'avantage de cette méthode est qu'elle est parallélisable, ses traitements pouvant se faire de manière indépendante.

De plus nous aurions pu diviser la partie de cross validation en plusieurs étapes que nous aurions lancé en thread plutôt qu'en séquentielle, cela aurait toutefois necessité la programmation d'un algorithme à part entière plutôt que le recours à une *built-in* fonction du package *Skicit-Learn.*