UNIVERSITÉ TOULOUSE III PAUL SABATIER

Université de Toulouse

Rapport de stage

Optimisation énergétique de datacenters avec un algorithme de bandits

Institut de Recherche
en Informatique de Toulouse
CNRS - INP - UT3 - UT1 - UT2J

Alexandre GARY
Master 1 MAPI3
2021/2022

Maîtres de stage :
Mme Emmanuelle CLAEYS
M. Georges DA COSTA

16 Avril 2022 - 29 Juillet 2022

Table des matières

Introduction

Ce stage a été effectué à l'Institut de Recherche en Informatique de Toulouse (IRIT), du 19 Avril 2022 au 29 Juillet 2022. L'IRIT est un institut de recherche basé principalement sur le campus de l'Université Toulouse III, fondé en 1990 suite à la fusion de deux unités de recherche du CNRS.
Il compte actuellement plus de 600 membres, permanents et non-permanents, divisés en 7 départements de recherche regroupant 24 équipes différentes.

Tout au long de mon stage, j'ai été encadré par Emmanuelle Claeys et Georges Da Costa, membres respectivement de l'équipe ADRIA (Argumentation, Décision, Raisonnement, Incertitude et Apprentissage : Département Intelligence Artificielle) et de l'équipe SEPIA (Système d'Exploitation, systèmes Répartis, de l'Intergiciel à l'Architecture : Département Architecture, Systèmes, Réseaux).

Mon travail s'inscrit dans le cadre du projet ANR Energumen, dont le but est d'optimiser l'énergie des plates-formes de calcul à large échelle. Le projet fait intervenir 3 instituts de recherche au niveau national l'IRIT, le LIG (Laboratoire d'Informatique de Grenoble) et LIP6 (Laboratoire d'Informatique de Paris 6).

J'aurai pour ma part exploité les données de plusieurs expériences menées sur des serveurs, au cours desquelles différentes applications ont été exécutées à différentes fréquences, le but de mon travail étant de proposer un algorithme choisissant la fréquence optimale pour chaque application, à savoir celle minimisant la consommation énergétique.
Les langages de programmation utilisés ont été : Python pour les algorithmes de bandits directement, et R pour toute la partie tests statistiques, analyse exploratoire et découverte des données. De plus, mon travail est basé sur celui effectué en thèse par Emmanuelle Claeys, qui a mis au point une librairie sur R contenant des algorithmes de bandits. Les librairies python utilisées ont été : numpy, pandas, matplotlib, seaborn et scikit-learn.

Je tiens à remercier toutes les personnes que j'ai pu croiser lors de mon stage, avec qui j'ai pu échanger. Tous ces échanges auront été intéressants et m'auront fourni un point de vue et un angle de réflexion différent à chaque fois. Je remercie en particulier les doctorants de l'équipe Adria, avec qui j'ai partagé le bureau pendant la durée de mon stage. Je voudrais également remercier les membres de l'équipe Sepia, avec lesquels j'ai pu discuter de mon sujet de stage lors des réunions d'équipe.
Enfin, je remercie tout particulièrement mes encadrants : Emmanuelle CLAEYS et Georges Da COSTA, pour leurs conseils précieux tout au long de mon stage, et pour leur investissement.

I – Cadre théorique : Algorithmes de bandits

I.1 Algorithmes de bandits : définition

Introduits dans les années 1960, les algorithmes de bandits font partie de la famille des algorithmes d'apprentissage par renforcement. Ces algorithmes tiennent leur nom des machines à sous des casinos, chaque machine étant actionnée via son bras et donnant une certaine récompense. Le but étant ici de repérer le meilleur bras (la meilleure machine), c'est-à-dire celle donnant les meilleures récompenses.

Un exemple classique pouvant faire appel à un algorithme de bandits est le suivant :

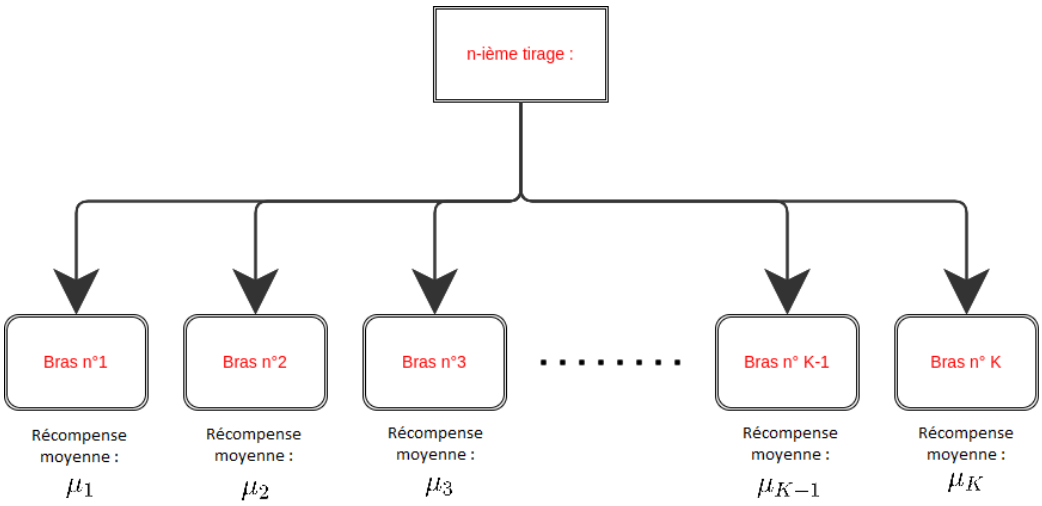
Une équipe de chercheurs en médecine a à sa disposition, pour une maladie donnée, un nombre de K de traitements expérimentaux, et un échantillon de N patients tests. Chaque patient recevra, l'un après l'autre, un traitement, aussi appelé dans ce cadre un "bras". Le but des chercheurs est de trouver en un nombre minimal de "tirages" le traitement le plus efficace. Typiquement, la stratégie de l'algorithme sera une certaine alternance entre exploitation (tirage du traitement le plus efficace) et exploration (tirage d'un traitement choisi au hasard). La phase d'exploration a un rôle important, elle permet de corriger une éventuelle erreur sur le choix du bras optimal.

On peut représenter cette situation en posant un ensemble de variables aléatoires $X_{i,n}$ avec $i \in \{1,...,K\}$ le bras (traitement) choisi et $n \in \{1,...,N\}$ l'itération (le numéro du patient). Ces variables sont toutes supposées indépendantes, et les variables issues d'un même traitement i : $X_{i,1}$, X_i,2, ... sont identiquement distribuées, de moyenne μ_i (ici le taux de guérison du médicament i). Ces moyennes sont inconnues de l'équipe de chercheurs, et les $X_{i,n}$ sont dans cet exemple des variables de Bernoulli, qui valent 1 si le patient n a été guéri avec le traitement i, et 0 dans le cas contraire.

L'objectif est de choisir une stratégie, basée sur les résutats précédents et sur les récompenses obtenues, qui maximise le nombre de patients qui seront rétablis. La performance d'un algorithme sera mesurée au travers d'une quantité appelée le regret, qui représente la différence entre la récompense obtenue en tirant le meilleur bras et la récompense obtenue à chaque tirage. Après un nombre n d'essais, le regret vaut :

$$R_n = n\mu^* - \sum_{t=1}^{n} \mu_{I_t}$$

Avec μ^* la récompense moyenne du meilleur traitement, et μ_{I_t} la récompense obtenue avec la stratégie choisie au tirage t.



La comparaison entre deux algorithmes est effectuée en comparant les courbes de leur regret respectif en fonction des itérations.

Il existe une multitude d'algorithmes différents, chacun avec une stratégie associée. La présentation des algorithmes que j'ai utilisé est l'objet de la partie suivante.

I.2 Présentation des algorithmes utilisés

Dans cette partie, nous considérons un problème de bandits à K bras, et à N tirages au total.
Pour les tests sur Python, nous utiliserons un jeu de données généré aléatoirement. Deux dataframes seront générés, un pour les récompenses, et un autre qui servira de contexte pour l'algorithme linucb.
Les données de contexte seront tirées via une loi uniforme sur l'intervalle $[0,10]$, et ce pour chaque tirage.
Les données représentant les récompenses seront issues d'un tirage de loi binomiale, avec des probabilités de succès différentes selon l'itération.

Image tableaux données

I.2.1 Algorithme d'allocation uniforme

Cette sous-partie est consacrée à un premier algorithme, dit d'allocation uniforme, qui est le plus "naïf". Il consiste, à chaque tirage, à choisir un bras tiré uniformément pari les K bras :

Algorithme 1 Bandits : Allocation uniforme

Entrée: Tableau contenant les récompenses pour chaque bras à chaque itération, et $K \in \mathbb{N}^*$: Nombre de bras
Sortie: Nombre de choix de chaque bras, et estimation des récompenses moyennes
C : Liste contenant les choix à chaque itération
R : Liste contenant les récompenses moyennes obtenues pour chaque bras
Tirer successivement une fois chaque bras
Incrémenter la liste C avec chaque choix
Incrémenter la liste R avec la récompense obtenue pour chaque bras
pour $n \in \{K+1,...,N\}$ **faire**
  Tirer une variable aléatoire uniforme a sur $\{1,...,K\}$
  Sélectionner le bras a
  Stocker le choix a dans la liste C
  Mettre à jour les récompenses moyennes dans la liste R
fin pour

Cet algorithme suit la stratégie la plus simple possible, et nous verrons par la suite lors de la comparaison des regrets que c'est le moins efficace. Il est cependant utile à conserver et à tester à titre de comparaison.

I.2.2 Algorithme ϵ - greedy

Le second algorithme présenté ici est l'algorithme appelé "ϵ-greedy". La stratégie de cet algorithme est la suivante :

A chaque tour, l'algorithme joue le meilleur bras (celui avec la récompense la plus élevée) avec probabilité $1 - \epsilon$ (exploitation), et joue un autre bras tiré uniformément avce probabilité ϵ (exploration). A chaque itération, les récompenses moyennes obtenues sont mises à jour.

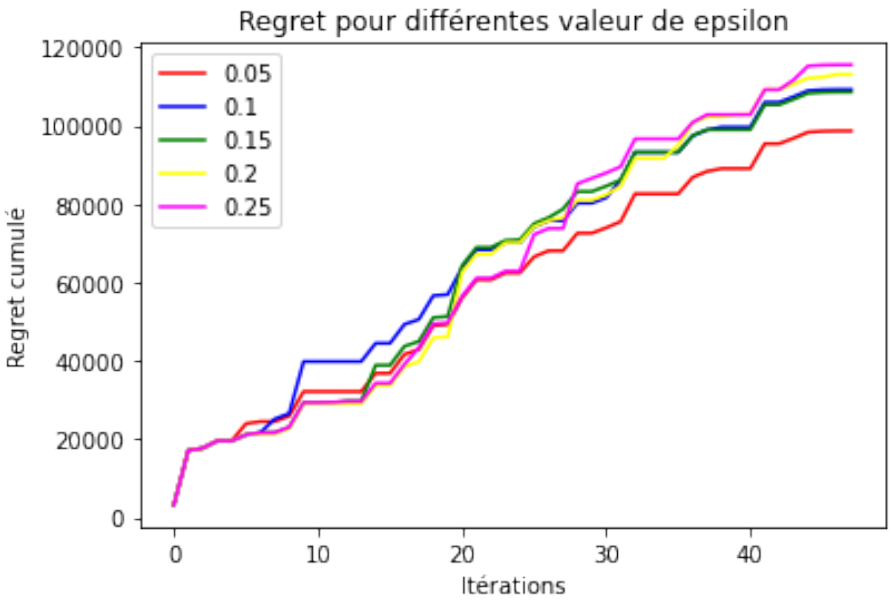
Voici son déroulé :

Algorithme 2 Bandits : ϵ - greedy

Entrée: Tableau contenant les récompenses pour chaque bras à chaque itération, $K \in \mathbb{N}^*$: Nombre de bras et $\epsilon \in [0,1]$: probabilité d'exploration.
Sortie: Nombre de choix de chaque bras, et estimation des récompenses moyennes

```
C : Liste contenant les choix à chaque itération
R : Liste contenant les récompenses moyennes obtenues pour chaque bras
Tirer successivement une fois chaque bras
Incrémenter la liste C avec chaque choix
Incrémenter la liste R avec la récompense obtenue pour chaque bras
pour n ∈ {K + 1, ..., N} faire
    Tirer une variable de Bernoulli X de paramètre 1 − ε
    si X = 1 alors
        Choisir le bras ayant eu les meilleurs résultats jusque-là
        Stocker le choix dans la liste C
        Mettre à jour la liste des récompenses moyennes R
    sinon
        Choisir uniformément un bras parmi les K − 1 autres
        Stocker le choix dans la liste C
        Mettre à jour la liste des récompenses moyennes R
    fin si
fin pour
```

Lors de l'utilisation de cet algorithme, le choix du paramètre ϵ est primordial : une valeur trop élevée entrainera une exploration trop importante, le cas extrême $\epsilon = 1$ correspondant à l'algorithme d'allocation uniforme présenté précédement. Dans le cas contraire, si ϵ est trop proche de 0, l'algorithme aura tendance à trop privilégier l'exploitation du bras ayant obtenu les meilleurs résultats, au risque de faire fausse route et de s'obstiner à tirer un bras qui ne serait pas le meilleur.



$\epsilon = 0.1$ mène à de bons résultats en terme de regret, comme en atteste la figure précédente.
A noter qu'il existe une variante de l'algorithme ϵ - greedy, où la valeur de ϵ est mise à jour à chaque itération, de la manière suivant :

$$\epsilon = \frac{1}{n}$$

avec $n \in \{K+1, ..., N\}$ l'itération dans l'algorithme ci-dessus.
Cette variante n'aura pas été explorée, car ces deux versions sont supplantées par les algorithmes décris dans la section suivante, et n'auraient pas été retenus pour le travail sur les données des expériences.

I.2.3 Algorithmes UCB et LINUCB

Cette dernière section permettra de décrire deux algorithmes, basés sur la création (et la mise à jour à chaque itération) d'un intervalle de confiance de la récompense moyenne pour chaque bras. Le bras joué sera alors celui ayant la borne supérieure de son intervalle de confiance la plus élevée, d'où le nom de Upper Confidence Bound (UCB). Au vu de leur méthode de sélection du bras à chaque tour, ces deux algorithmes sont dits "optimistes".

I.2.3.1 Algorithme UCB

Le premier algorithme est le plus simple des deux, voici son pseudo-code :

Algorithme 3 Bandits : UCB

Entrée: Tableau contenant les récompenses pour chaque bras à chaque itération, et $K \in \mathbb{N}^*$: Nombre de bras
Sortie: Nombre de choix de chaque bras, et estimation des récompenses moyennes
C : Liste contenant les choix à chaque itération
I : Liste contenant les intervalles de confiances obtenus pour la récompense moyenne associée à chaque bras
Incrémenter la liste C avec chaque choix
Incrémenter la liste I avec l'intervalle de confiance obtenu pour chaque bras
pour $n \in \{K+1, ..., N\}$ **faire**
  Choisir le bras a ayant la borne supérieure de l'intervalle de confiance la plus élevée
  Stocker le choix dans la liste C
  Mettre à jour la liste des intervalles de confiance I
fin pour
Récupérer les récompenses moyennes pour chaque bras à partir de la liste des intervalles de confiance I

A chaque itération n, l'intervalle de confiance IC pour un bras b est calculé de la manière suivante :

$$IC_{b,n} = \left] m_b \pm \alpha \sqrt{\frac{2log(n)}{e_b}} \right[$$

Avec :
m_b la récompense moyenne pour le bras b obtenue jusqu'à l'itération n ;
α le niveau de confiance de l'intervalle ;
et e_b le nombre d'essais pour chaque bras jusqu'à l'itération n.

L'algorithme UCB a de meilleures performances que les algorithmes ϵ-greedy et d'allocation uniforme en terme de regret.

I.2.3.2 Algorithme LINUCB

Ce dernier algorithme est un algorithme de bandits contextuels. A chaque itération, avant de sélectionner un bras à jouer, les algorithmes de bandits contextuels créent un contexte pour se donner des indications supplémentaires sur le meilleurs bras à jouer. L'algorithme décrit ici utilise une régression linéaire pour créer ce contexte (d'où le nom de LINUCB). Il existe cependant d'autres méthodes de contextualisation du choix du bras, basées par exemple sur du clustering (Ctree UCB) ou sur une regression à noyau (Kernel UCB), entre autre...

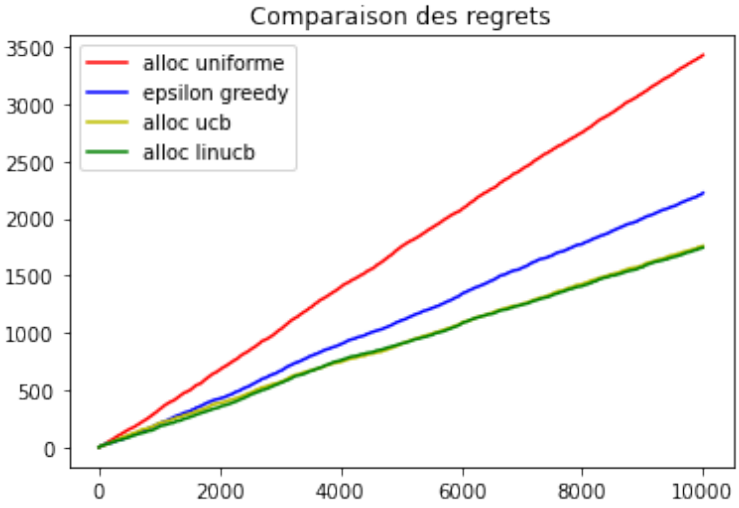
Voici le pseudo-code de l'algorithme étudié ici :

Algorithme 4 Bandits : LINUCB

Entrée: Tableau contenant les récompenses pour chaque bras à chaque itération, et $K \in \mathbb{N}^*$: Nombre de bras
Sortie: Nombre de choix de chaque bras, et estimation des récompenses moyennes
C : Liste contenant les choix à chaque itération
I : Liste contenant les intervalles de confiances obtenus pour la récompense moyenne associée à chaque bras
Incrémenter la liste C avec chaque choix
Incrémenter la liste I avec l'intervalle de confiance obtenu pour chaque bras
pour $n \in \{K+1, ..., N\}$ **faire**
  Création du contexte en utilisant une régression linéaire
  Choisir le bras a ayant les meilleures résultats potentiels en fonction de la regression effectuée
  Stocker le choix dans la liste C
  Mettre à jour la liste des intervalles de confiance I
fin pour
Récupérer les récompenses moyennes pour chaque bras à partir de la liste des intervalles de confiance I

Les intervalles de confiances sont construits à partir de la même formule que pour l'algorithme UCB. La différence majeure est la création d'un contexte avant chaque choix, ce qui en fait l'algorithme ayant le regret optimal, comme nous allons le voir dans la partie suivante.

I.2.4 Comparaison du regret pour chaque algorithme



La figure ci-dessus montre le regret cumulé pour les différents algorithmes présentés dans cette première partie. Le gain généré par l'utilisation des algorithmes basés sur les intervalles de confiance est important par rapport aux deux autres méthodes. De plus, on voit que l'algorithme ayant le regret cumulé le plus faible est le LINUCB.

II – Optimisation de la consommation énergétique

II.1 Présentation des données et propriétés vérifiées

II.1.1 Présentation des données

Mon travail est basé sur des expériences menées sur un datacenter. 8 applications différentes ont été exécutées sur des serveurs, chacune à des fréquences comprises entre 1.2 GHz et 2.4 GHz. J'ai donc travaillé à la fois sur les données des expériences directement, mais également sur un fichier csv récapitulatif.

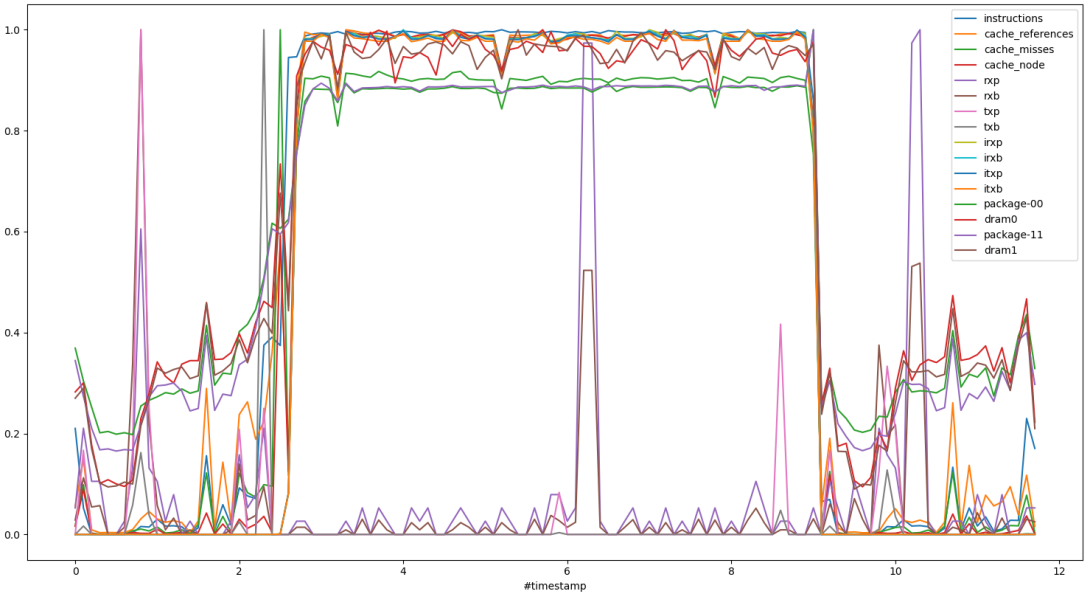
Interessons nous d'abord à ce fichier récapitulatif, nommé "raw data". Il contient une partie données pour chaque expérience, à savoir : le nom de l'application exécutée, les durées d'exécution pour chaque fréquence (de 1.2 à 2.4 GHz, avec un saut de 0.1 GHz à chaque fois), et l'énergie consommée, également pour chaque fréquence. J'ai divisé ce tableau en deux, pour avoir une partie "contexte" (les noms et les durées) et une partie "récompenses" (les consommations énergétiques). Voici l'en tête de ces deux dataframes :

	bt	cg	ep	ft	is	lu	mg	sp	duration
1	0	0	0	0	0	1	0	0	28.63
2	0	0	0	0	1	0	0	0	15.86
3	0	0	0	1	0	0	0	0	11.73
4	0	0	0	0	1	0	0	0	15.86
5	0	0	0	0	0	0	0	1	27.80

	ref_energy	ref_energy.1	ref_energy.2	ref_energy.3	ref_energy.4	ref_energy.5	ref_energy.6
1	-4638.06	-4382.08	-4131.74	-4039.200	-3973.810	-3859.20	-3792.500
2	-2394.86	-2330.20	-2337.40	-2277.600	-2319.840	-2371.07	-2375.290
3	-1865.07	-1773.90	-1754.62	-1674.860	-1648.115	-1579.30	-1572.080
4	-2339.35	-2360.56	-2342.05	-2248.775	-2217.605	-2236.80	-2282.245
5	-5309.80	-5177.16	-5075.00	-4908.980	-4811.100	-4788.72	-4732.200

A noter que dans le dataframe des récompenses, toutes les énergies sont consommées. En effet, les algorithmes de bandits sont fait de manière à maximiser les récompenses obtenues, or nous voulons minimiser la consommation énergétique. Une solution a donc été de passer en négatif toutes les valeurs.

La première partie de mon travail aura donc été sur ce jeu de données. Mais, le but étant de ne plus utiliser les noms, j'ai aussi travaillé avec les données brutes des expériences. Ce sont aussi des fichiers csv, contenant les relevés effectués au cours du temps. Ces données sont représentées dans la figure ci-dessus.



On y retrouve différentes données récoltées, comme par exemple la consommation du processeur (package-00 et -11), celle de la mémoire (dram0 et dram1), ou encore les instructions, entre autre...

II.1.2 Hypothèses vérifiées

Nous allons ici nous intéresser aux hypothèses vérifiées par les données "récompenses" (énergies) du fichier "raw data", en nous posant les deux questions suivantes :
- Les données sont-elles stationnaires ?
- Sont elles normalement distribuées ?

Les données utilisées pour ces tests étant des séries temporelles, il est important de vérifier quelles propriétés elles vérifient.

II.1.2.1 Stationnarité

Nous étudierons tout d'abord la stationnarité des données. pour se faire, nous utiliserons un test statistique : le test augmenté de Dickey-Fuller (ou test ADF). Ce test est basé sur la détection d'une racine unité dans une série temporelle. Une racine unité est une unité de mesure de la stationnarité dans une série temporelle.

Le test augmenté de Dickey-Fuller est une variante du test de Dickey-Fuller, basé sur une régression linéaire. Les hypothèses de ce test sont les suivantes :

- Hypothèse $H0$: présence d'une racine unité : série non stationnaire
- Hypothèse $H1$: absence de racine unité : série non stationnaire

On effectue ce test pour les récompenses de chaque bras, on a donc 13 tests différents à effectuer, sur R, avec la commande $adf.test$ de la librairie $tseries$. Le niveau de confiance est fixé à 95%.

A titre d'exemple, les résultats de ce tests sont affichés ci-dessous.

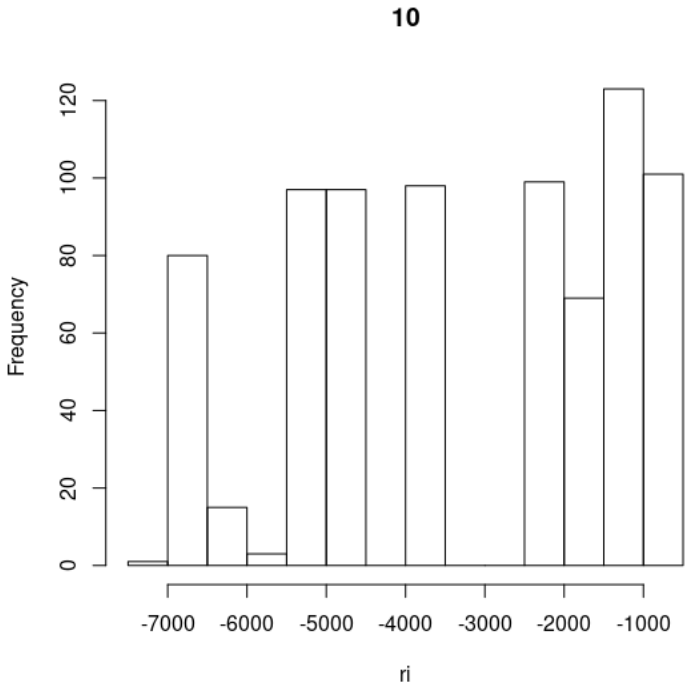
Name	Type	Value
t13	list [6] (S3: htest)	List of length 6
statistic	double [1]	-8.74399
parameter	double [1]	9
alternative	character [1]	'stationary'
p.value	double [1]	0.01
method	character [1]	'Augmented Dickey-Fuller Test'
data.name	character [1]	'r13'

Pour chaque bras, nous allons comparer la p-valeur à 0.05 (car le niveau de confiance est de 95%).
Les résultats sont similaires à ce bras pour tous les autres : La p-valeur est inférieure à 0.05, donc on rejette à chaque fois l'hypothèse H0. De fait, nous pouvons conclure que les récompenses sont toutes stationnaires.

II.1.2.2 Distribution des données

Nous allons maintenant nous intéresser à la distribution des mêmes données de récompenses, et en particulier si elles suivent une loi normale ou non. Cette vérification sera faite de deux manières différentes : dans un premier temps en traçant l'histogramme des récompenses pour chaque bras, puis à l'aide d'un test statistique.

En jettant un oeil à l'histogramme ci-dessus, nous voyons clairement que les récompenses de ce bras ne sont pas normalement distribuées. Et les résultats sont les mêmes pour les autres bras.



En complément, nous pouvons effectuer un test statistique sur ces données pour confirmer cette observation. Le test utilisé ici sera un test de $Shapiro - Wilk$, dont les hypothèses sont les suivantes :
- Hypothèse $H0$: L'échantillon testé est issu d'une distribution normale.
- Hypothèse $H1$: L'échantillon testé n'est pas normalement distribué.

Voici la statistique de ce test :

$$W = \frac{\left(\sum_{i=1}^{n} a_i x_{(i)}\right)^2}{\sum_{i=1}^{n} (x_i - \overline{x})^2}$$

Avec : - $x_{(i)}$: la i-ème statistique d'ordre, c'est-à-dire le i-ème plus petit nombre dans l'échantillon,
- $\overline{x}$: la moyenne de l'échantillon,

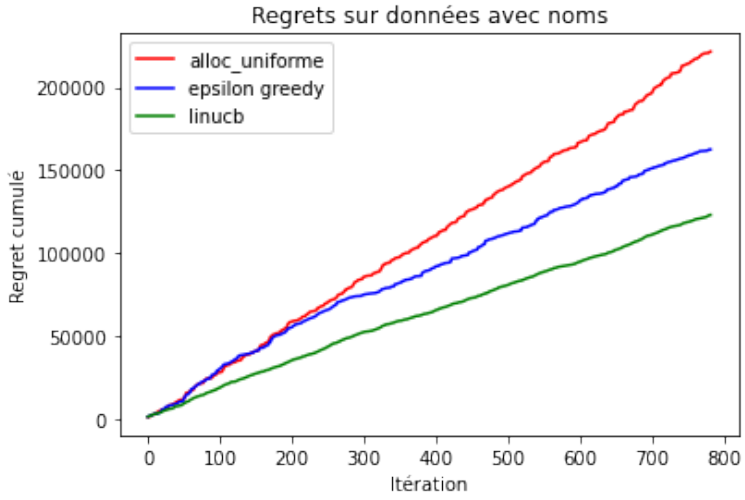
- a_i : la constante donnée par :
$(a_1, ..., a_n) = \frac{m^T V^{-1}}{(m^T V^{-2} m)^{1/2}}$
où $m = (m_1, ..., m_n)^T$ est le vecteur espérance des statistiques d'ordre de n lois normales iid, et V est la matrice de variance-covariance de ces statistiques d'ordre.

Comme pour le test de stationnarité dans la section précédente, un apercu du resultat de ce test, effectué sous R et pour les récompenses du bras n°13 est présenté sur la ci-dessous.

si	list [4] (S3: htest)	List of length 4
statistic	double [1]	0.9133306
p.value	double [1]	9.961321e-21
method	character [1]	'Shapiro-Wilk normality test'
data.name	character [1]	'ri'

Ces résultats nous permettent donc de rejeter l'hypothèse $H0$ au profit de l'hypothèse $H1$, ce qui confirme les observations faites grace à l'histogramme précédent : les récompenses ne sont pas normalement distribuées.

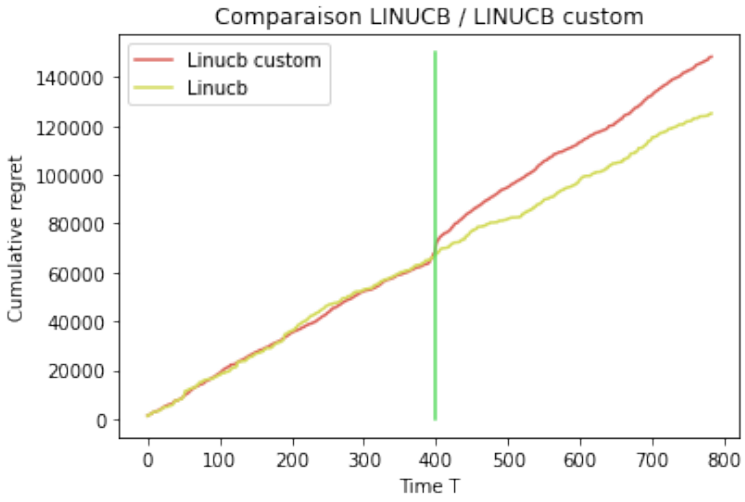
II.2 Travail sur les données avec les noms



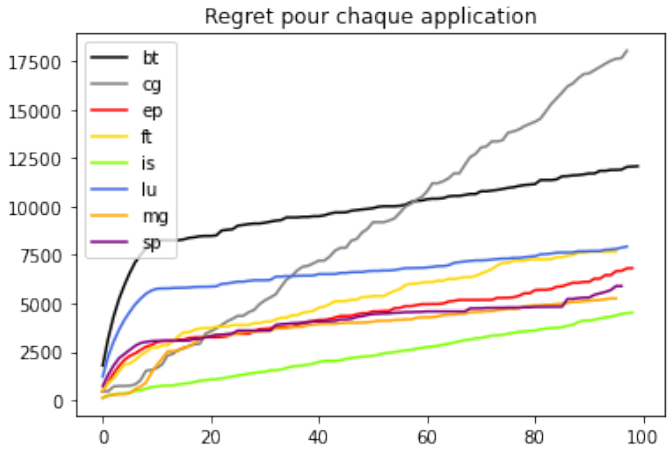
La première partie du travail a été d'exploiter les résultats des expériences en divisant les données selon le nom de l'application, avec les données présentées en première partie. Plusieurs algorithmes de bandits ont été testés : Bandit uniforme, ϵ-greedy et LINUCB. Le tracé du regret pour chaque algorithme, en fonction de l'itération, est présenté en début de page.
Comme attendu, c'est bien l'algorithme LINUCB qui obtient les meilleurs résultats, c'est donc cet algorithme là qui sera utilisé par la suite.

Cet algorithme a également été légèrement modifié, pour tester les effets d'une remise à 0 des variables à un certain temps t fixé en amont. Les comparaisons de regrets entre ce LINUCB dit "custom" et le LINUCB original sont tracées en haut de la page suivante.

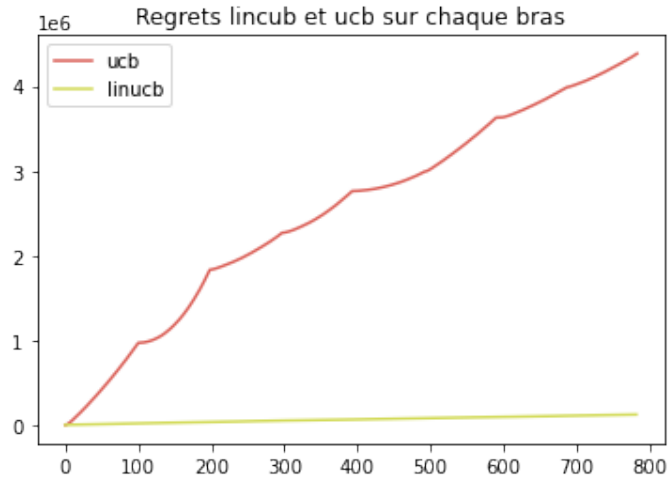
Les deux algorithmes ont un regret sensiblement égal. Cependant, lorsque les données ne remplissent pas certaines hypothèses, il est possible que le LINUCB custom soit plus efficace. En effet, le fait de remettre les coefficients à 0 réinitialise la "préférence" de l'algorithme envers un certain bras, et donc permet de corriger une potentielle erreur dans l'identification du meilleur bras.



Une autre étude interessante à faire est la comparaison entre le LINUCB classique, et un algorithme UCB sur chaque bras. Ces comparaisons sont effectuées par les tracés des regrets correspondant sur cette page.



L'algorithme LINUCB obtient des résultats largement meilleurs que l'algorithme UCB, ce qui est attendu étant donné que le LINUCB créé un contexte à partir des données pour essayer de prédire le bras attribuant la meilleure récompense à chaque tirage.



Après réalisation de tous ces tests, les résultats de l'algorithmes LINUCB ont été sélectionnés pour poursuivre le travail, et attribuer une fréquence optimale pour chaque application.

La fonction python LINUCB, disponible en annexe (p. 33 - 35), ne renvoie pas directement le bras correspondant à une consommation électrique optimale (bien que l'on pourrait avoir cette information directement en regardant quel bras a été le plus joué). La fonction renvoie, entre autre, un tableau de coefficients pour chaque bras et chaque fréquence, avec en plus un coefficient associé à la durée d'exécution de chaque application (figure en bas de page).
Chaque colonne correspond ici à une application, et chaque ligne à une fréquence de processeur : "ref-energy" correspondant à une fréquence de 1.2GHz, "ref-energy.1" à une fréquence de 1.3GHz, et ainsi de suite...

Ces résultats sont une sorte de coefficients de régression. Par exemple, si l'on prend le cas de l'application "is", qui a été testée par l'algorithme pour les fréquences 1.3, 1.5, 2.0, 2.2 et 2.3 GHz, pour associer une consommation à chaque fréquence, il faut ajouter au coefficient de cette colonne le coefficient correspondant dans la colonne duration, tout en le multipliant par la durée d'exécution récupérée dans les données initiales.
La fonction "calcul-freq" fait ce calcul (disponible en annexe p. 38 - 40), et renvoie un tableau récapitulatif associant à chaque application sa fréquence optimale, avec la consommation électrique correspondante et le gain énergétique moyen par rapport à la consommation à d'autres fréquences. Ce tableau est affiché sur la page suivante.

	bt	cg	ep	ft	is	lu	mg	sp	duration
ref_energy	0.000000	0.000000	0.000000	0.000000	0.000000	578.747727	-577.952808	0.000000	-202.429460
ref_energy.1	0.000000	0.000000	0.000000	0.000000	507.490282	0.000000	-295.386867	0.000000	-210.919329
ref_energy.2	0.000000	180.472471	0.000000	322.622631	0.000000	0.000000	-480.951296	306.096751	-204.592094
ref_energy.3	0.000000	169.975323	0.000000	0.000000	577.556272	0.000000	-595.847021	243.287447	-200.515186
ref_energy.4	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	-445.751978	454.239042	-205.740219
ref_energy.5	0.000000	242.589958	374.489256	377.881212	0.000000	0.000000	-595.459162	288.452650	-200.496425
ref_energy.6	0.000000	171.584259	0.000000	0.000000	0.000000	0.000000	-53.677946	0.000000	-220.696090
ref_energy.7	0.000000	0.000000	0.000000	0.000000	0.000000	861.728324	-844.975700	0.000000	-192.998833
ref_energy.8	867.558983	-140.193945	498.488031	238.442759	130.089318	723.829788	-1960.904060	-448.317297	-154.386319
ref_energy.9	0.000000	236.413450	0.000000	0.000000	0.000000	0.000000	-518.352867	470.848627	-206.755312
ref_energy.10	0.000000	68.285907	0.000000	0.000000	504.643810	0.000000	-737.636131	454.190242	-196.370667
ref_energy.11	0.000000	-34.943656	0.000000	0.000000	229.933156	1119.991090	-1236.602122	0.000000	-178.795480
ref_energy.12	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	-459.281658	490.393106	-207.764633

	Frequence optimale	Consommation optimale	Gain moyen
bt	2.0	3331.671713	2897.604678
cg	2.1	1026.840410	204.548272
ep	2.0	190.948250	843.054201
ft	2.0	1074.902362	753.132021
is	2.0	1897.548867	764.851028
lu	2.3	1907.508985	2265.119612
mg	2.4	5852.418687	181.052242
sp	2.2	3411.396701	927.945845

II.3 Travail sur les données sans les noms

Cette dernière partie parcourra le travail effecuté sur les données détaillées des expériences, sans utiliser les noms. Le but ici est de proposer un clustering de ces données, en ne prenant à aucun moment en compte le nom des applications, et de faire tourner un algorithme de bandits sur chaque cluster pour y associer une consommation optimale. J'ai choisi de faire un clustering Kmeans sur les données.

Pour se faire, j'ai tout d'abord exploité les données des fichiers récapitulatifs de chaque session d'expérience, dont la figure ce-dessous donne un aperçu. La colonne "fmax" correspond à la fréquence du processeur lors de l'exécution de l'application pendant l'expérience, et la colonne 'startTime" correspond au temps initial de lancement de l'application, et c'est cette valeur qui fait figure d'identification pour récupérer les résultats détaillés de chaque expérience dans le fichier correspondant.

Pour que l'algorithme de bandits soit efficace, il faut que toutes les données prises en contexte soient à la même "échelle", c'est-à-dire que dans notre cas il nous faut trier les expériences pour ne garder que celle à une fréquence "fmax" donnée. Toutes les fréquences "fmax" auront été testées pour comparer les résultats, comme présenté plus bas.

Une autre problématique du projet, lorsque les clusters auront été créés avec des fréquences associées, est d'arriver à associer un cluster à toute nouvelle application inconnue qui arrivera sur le serveur d'exécution, et ce le plus rapidement possible. Il aura fallu tenir comtpe de cela lors du clustering, en le construisant à partir des données de début d'exécution de chaque application. Cependant, les 4 premières secondes correspondent, de manière générale, au chargement de l'application (figure p. 12), ces données ne représentent donc pas réellement son comportement en exécution. C'est pourquoi j'ai choisi d'omettre cette période de temps sur chaque expérience. De même que plusieurs "fmax" auront été testées, j'ai aussi selectionné plusieurs "délais" pour récolter les données détaillées.

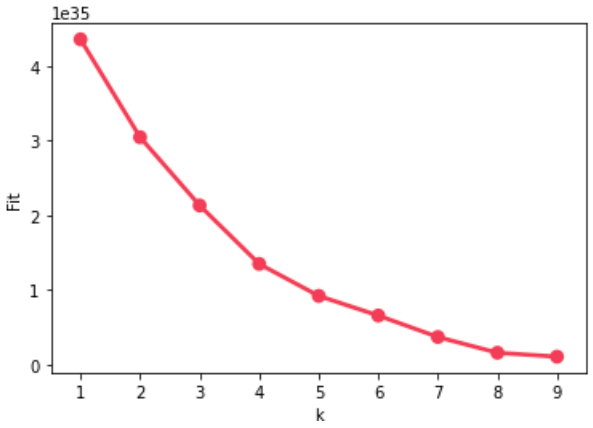
	hostname	fullname	nproc	duration	startTime	endTime	fmin	fmax	hostlist	basename	e
0	graoully-1.nancy.grid5000.fr	bt-C-64	64	40.57	1637601498	1637601609	1200000	1200000	graoully-1.nancy.grid5000.fr;graoully-16.nancy...	npb_graoully-1.nancy.grid5000.fr_1637601437	30254.2
1	graoully-1.nancy.grid5000.fr	bt-C-64	64	37.76	1637601626	1637601734	1200000	1300000	graoully-1.nancy.grid5000.fr;graoully-16.nancy...	npb_graoully-1.nancy.grid5000.fr_1637601437	28471.8
2	graoully-1.nancy.grid5000.fr	bt-C-64	64	35.64	1637601750	1637601855	1200000	1400000	graoully-1.nancy.grid5000.fr;graoully-16.nancy...	npb_graoully-1.nancy.grid5000.fr_1637601437	27519.5
3	graoully-1.nancy.grid5000.fr	bt-C-64	64	33.67	1637601870	1637601974	1200000	1500000	graoully-1.nancy.grid5000.fr;graoully-16.nancy...	npb_graoully-1.nancy.grid5000.fr_1637601437	27879.1
4	graoully-1.nancy.grid5000.fr	bt-C-64	64	32.12	1637601990	1637602092	1200000	1600000	graoully-1.nancy.grid5000.fr;graoully-16.nancy...	npb_graoully-1.nancy.grid5000.fr_1637601437	25741.7

Mais avant de comparer les résultats pour différentes fréquences et différents pas de temps, nous pouvons étudier les résultats pour une fréquence et un délai donnée. Ici, j'ai sélectionné les expériences exécutées à 2.4 GHz, et avec un délai de 10 secondes. Les données détaillées des expériences sont présentées en début de page suivante.

instructions	cache_references	cache_misses	cache_node	rxp	rxb	txp	txb	irxp	irxb	itxp	
7.126050e+09	2.063024e+07	1.068034e+07	6.790390e+06	5.897119	477.176955	0.506173	264.473251	18262.230453	8.380612e+06	18262.152263	8.38059
7.120380e+09	2.039575e+07	1.062264e+07	6.761612e+06	5.599174	429.239669	0.148760	119.648760	18166.024793	8.336804e+06	18166.004132	8.33680
7.100371e+09	2.039909e+07	1.061428e+07	6.753175e+06	5.565574	426.286885	0.184426	122.409836	18175.151639	8.341428e+06	18174.954918	8.34105
7.141135e+09	2.043908e+07	1.062542e+07	6.751956e+06	5.698347	448.190083	0.322314	138.764463	18165.913223	8.336802e+06	18165.979339	8.33681
4.928587e+09	3.244595e+07	5.439335e+06	4.103259e+06	1.838235	259.367647	1.382353	569.926471	76533.529412	3.552697e+07	76524.588235	3.55222
...	...	...	...	...	...	...	...	...	...	...	
4.708899e+09	7.206684e+07	1.421589e+07	1.065412e+07	1.811594	141.192029	0.434783	233.840580	6306.420290	3.035007e+06	6306.108696	3.03478
3.635065e+09	2.254729e+07	1.014367e+07	6.673391e+06	1.364000	786.856000	0.636000	176.648000	35175.864000	1.811652e+07	34690.348000	1.78653
3.709421e+09	2.253363e+07	1.013092e+07	6.660942e+06	1.365462	778.650602	0.598394	264.718876	34950.465863	1.799941e+07	34954.321285	1.80012
3.632964e+09	2.242635e+07	1.032884e+07	6.890166e+06	1.318725	759.254980	0.581673	261.529880	34477.760956	1.775560e+07	34738.932271	1.78908
3.683452e+09	2.248119e+07	1.034567e+07	6.844611e+06	1.286853	179.031873	0.490040	257.422311	34510.035857	1.777199e+07	34728.621514	1.78851

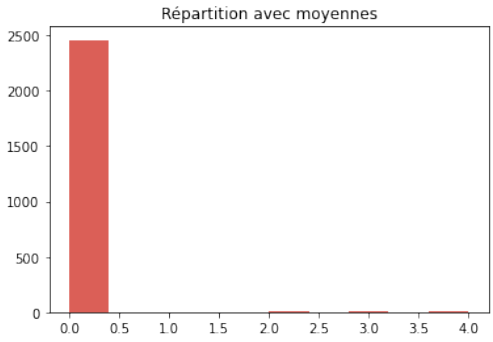
Ces données ont été condensées de la manière suivante : pour chaque expérience, j'ai conservé les données du pas de temps allant de la 4ème seconde à la 14ème. J'ai ensuite fait la moyenne pour chaque covariable (pour le 1er test) ou la médiane pour chaque covariable (pour le 2nd test). Chaque expérience correspond donc à une ligne de la figure ci-dessus.

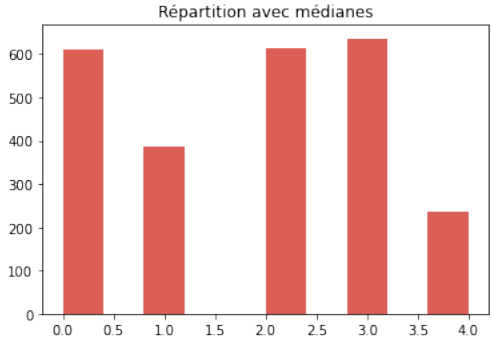
Pour paramétrer l'algorithme Kmeans avec un nombre de clusters adéquat, j'ai utilisé la méthode du coude, tracée sur la figure suivante.



Suite à l'analyse de ce graphique, j'ai opté pour un clustering avec 5 clusters différents. Jetons maintenant un oeil à la répartition par cluster, lorsqu'on utilise la valeur moyenne ou bien la valeur médiane (figures en début de page 22).

Les résultats sont bien meilleurs avec la médiane qu'avec la moyenne. Sans viser une répartition totalement uniforme, un clustering avec presque toutes les expériences dans le même cluster n'est pas acceptable. Un tel déséquilibre est probablement engendré par des valeurs extrêmes pour certaines covariables, qui "faussent" en quelques sortes la valeur moyenne. La médiane est elle peu impactée par ces valeurs extrêmes, ce qui explique la différence de résultats.



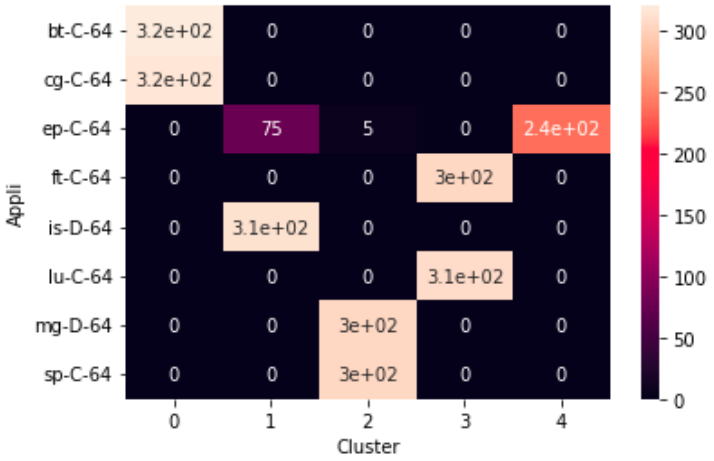


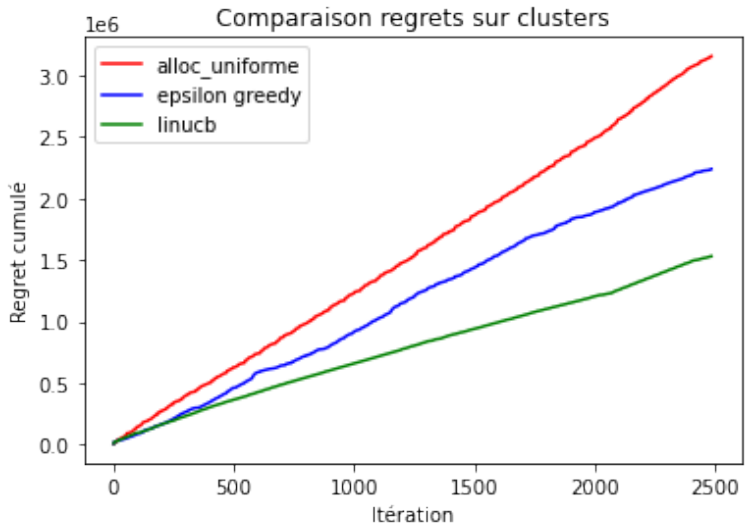
C'est donc les médianes de chaque covariable qui seront conservées pour la suite.

Un aure indicateur interessant à afficher est la matrice de confusion associée au clustering, mettant en relation les indices des clusters attribués à chaque expérience, avec le nom de l'application exécutée lors de la même expérience, visible en bas de page.

On y voit une corrélation très importante entre le cluster attribué et le nom de l'application, bien que ce dernier n'ait pas été utilisé pour le clustering. L'algorithme Kmeans a donc bien analysé et réparti les applications en fonction de leur comportement. La seule application à ne pas être dans un unique cluster est l'application "ep", bien qu'elle soit en majorité dans le cluster n°4.

Maintenant que les clusters ont été créés, il est temps d'exécuter les algorithmes de bandits dessus, tout en ayant au préalable récupéré les consommations energétiques et les durées d'éxécution, pour créer des dataframes de contexte et de récompenses similaires aux figures page 11.





Comme présenté sur la figure ce-dessus, et comme dans les autres situations, c'est l'algorithme LINUCB qui a le regret le plus faible.

Maintenant que nous avons le nombre de cluster optimal (5), la méthode de construction du dataframe des données (valeurs médianes) et la méthode de bandits (LINUCB), nous pouvons nous concentrer sur le choix de la meilleure combinaison délai / fréquence de référence. Pour cela, j'ai exécuté la fonction "clustering-bandits" du notebook en annexe (p. 41) pour différentes combinaisons temps / fréquence de référence, avant de récupérer la consommation optimale totale pour chaque cas. Les résultats sont affichés ici :

	5	6	7	8	9	10	11
1200000	55403.693235	55403.693235	55403.693235	55403.693235	55403.693235	55403.693235	55403.693235
1300000	59688.159415	59680.623980	59680.623980	59680.623980	59680.623980	59680.623980	59680.623980
1400000	61269.826567	61269.826567	61269.826567	61269.826567	61269.826567	61269.826567	61269.826567
1500000	61667.661291	61667.661291	61667.661291	61667.661291	61667.661291	61667.661291	61667.661291
1600000	64323.482138	64323.482138	64313.521728	64323.482138	64323.482138	64323.482138	64323.482138
1700000	67467.783922	67467.783922	67467.783922	67467.783922	67467.783922	67467.783922	67467.783922
1900000	70598.530944	70598.530944	70598.530944	70598.530944	70598.530944	70598.530944	70598.530944
2000000	71630.195813	71630.195813	71630.195813	71630.195813	71630.195813	71630.195813	71630.195813
2100000	71922.669051	71922.669051	71922.669051	71922.669051	71922.669051	71922.669051	71922.669051
2200000	71304.237197	71304.237197	71304.237197	71304.237197	71304.237197	71304.237197	71304.237197
2300000	71975.639338	71955.352746	71975.639338	71975.639338	71975.639338	71975.639338	71975.639338
2400000	78420.049768	78420.049768	78420.049768	78420.049768	78420.049768	78420.049768	78420.049768

Ces résultats nous montre que, peu importe le délai sur lequel on garde les mesures, une fréquence de référence de 1.2 GHz permet de consommer moins que dans les autres cas.

Voici par ailleurs le tableau de consommation de chaque cluster, avec la fréquence optimale

associée, pour une fréquence de référence de 1.2 GHz et un temps fixé à 7 secondes (figure en début de page suivante).

	Frequence optimale	Consommation optimale	Gain moyen
0	2.3	14565.016882	9556.939688
1	1.9	17810.923460	3715.426509
2	2.3	9945.570908	8623.261680
3	2.3	2089.162050	4583.371741
4	2.0	10993.019935	5118.458999

La dernière étape est d'attribuer un cluster à toute nouvelle application qui arrive sur le serveur. Pour cela, la fonction "attribue-cluster" du notebook en annexe p. 41 récupère les centroides du clustering effectué précédement, et compare la distance euclidienne entre l'application et chaque centroide. Le cluster attribué est celui correspondant à la plus petite distance.

Conclusion

L'objectif de ce stage était de mettre au point un algorithme permettant d'attribuer une fréquence de processeur aux applications dont les données ont été recueillies au préalable, puis d'attribuer une fréquence optimale à chaque nouvelle application arrivant sur le serveur, tout en utilisant un algorithme de bandits. Ces deux objectifs ont été rempli, mais il reste encore plusieurs pistes à explorer.

Tout d'abord, concernant les algorithmes de bandits, il en existe plusieurs autres qu'il aurait pu être intéressant de tester, comme par exemple l'algorithme Thompson Sampling, dont le fonctionnement est basé sur des statistiques Bayesiennes. Dans le même ordre d'idée, le clustering effectué dans la partie II.3 repose sur un algorithme Kmeans, choisi de manière arbitraire. J'aurais également pu tester d'autres méthodes de clustering, avec par exemple une classification ascendante hiérarchique (CAH).

Par ailleurs, mon travail pourrait être combiné à celui d'un autre stage effectué à l'IRIT, par Melissa Mekbel, qui a travaillé sur les correspondances entre séries temporelles (DTW). On pourrait imaginer un clustering basé sur cette méthode.

Enfin, à titre plus personnel, mon temps passé au sein de l'IRIT m'aura permis de me faire une première idée du monde de la recherche en laboratoire. Bien que le débouché principal de la filière MAPI3 ne soit pas la recherche, cette expérience aura été intéressante et enrichissante.

Annexes

Ces annexes regroupent les principales fonctions pythons créées pendant la durée de mon stage. L'ordre de ces fonction suit celui du déroulé du rapport : les fonctions des algorithmes de bandits apparaissent en premier, suivi de celles qui m'ont permis de calculer les fréquences optimales.

Annexe 1 : Algorithmes de Bandits

```
########################################################################
                        #### Fonction generales ####
########################################################################




def PlayArm(iter,arm,S,visitor_reward):
    #mean
    arm = int(arm)
    iter = int(iter)
    S[0,arm] = (S[0,arm]*S[1,arm] + visitor_reward[iter,arm]) / (S[1,arm]+1)
    #play
    S[1,arm] += 1
    return S

def ControlDataMissing(visitor_reward):
    t = pd.DataFrame.isnull(visitor_reward)
    c = sum(pd.DataFrame.sum(t))

    if c > 0 :
        raise ValueError('Missing data in arm results database')

def DataControlK(visitor_reward, K = None):
    if K == None :
        K = np.shape(visitor_reward)[1]

    if K < 2 :
        raise ValueError('Arm must be superior or equal to 2')

    if np.shape(visitor_reward)[1] != K :
        raise ValueError('Each arm need a result')

def BanditRewardControl(visitor_reward, K=None):
    DataControlK(visitor_reward,K)
    ControlDataMissing(visitor_reward)

def DataControlContextReward(dt, visitor_reward):

    # Match size control
    if(np.shape(dt)[0] != np.shape(visitor_reward)[0]):
        raise ValueError(
            "Number of rows in contextual data and reward data are not␣
 ↪equal")
```

```
#########################################################################
                        #### Calcul regret ####
#########################################################################



def SimpleRegret(choice,visitor_reward):
    visitor_reward
    n = visitor_reward.shape[0]
    regret = np.zeros(n)
    for i in range(n):
        regret[i] = RegretValue(choice[i],np.array(visitor_reward)[i,:])
    return regret

def RegretValue(arm, vec_visitor_reward):
    return max(vec_visitor_reward) - vec_visitor_reward[int(arm)]

def cumulativeRegret(choice,visitor_reward):
    regret = SimpleRegret(choice,visitor_reward)
    return np.cumsum(regret)
```

```
#########################################################################
                        #### Bandit uniforme ####
#########################################################################



def UniformBandit(visitor_reward, K = None):

    if K == None :
        K = visitor_reward.shape[1]

    # control :
    BanditRewardControl(visitor_reward,K)

    # data formating :
    visitor_reward = np.array(visitor_reward)

    # keep list of choices :
    n = np.shape(visitor_reward)[0]
    choice = np.zeros(n)
    S = np.zeros((2,K))

    tic = time.time()

    if K > n :
```

```
        print("Warning : more arm than visitor")

        for j in range(n):
            S = PlayArm(j,j,S)
        choice = np.arange(1,n+1,1)
        toc = tme.time()
        return [S,choice,toc-tic]

    else :

        for i in range(n):
            choice[i] = ((i+1)%K)
            S = PlayArm(i,i%K, S, visitor_reward)
        toc = time.time()

        return [S,choice,toc-tic]
```

```
########################################################################
                        #### Epsilon greedy ####
########################################################################



def ConditionForEpsilonGreedy(S, epsilon=0.25,K=None):
    if K == None :
        K = np.shape(S)[1]

    # choose the best with 1-epsilon proba. 0 : best arm, 1  : other arm
    u = np.random.binomial(1,1-epsilon)

    # the best one have been choose
    if u == 1 :
        return int(np.argmax(S[0,:]))

    # randomly select another arm :
    else :
        m = np.argmax(S[0,:])
        l = np.arange(0,K,1)
        l = np.delete(l,m)
        return np.random.choice(l)

def EpsilonGreedy(visitor_reward0, K = None, epsilon = 0.25):
    if K == None :
        K = np.shape(visitor_reward0)[1]
    # Control :
    BanditRewardControl(visitor_reward = visitor_reward0, K = K)

    # data formating :
    visitor_reward = np.array(visitor_reward0)

    # keep list of choice :
```

```
    choice = np.zeros(np.shape(visitor_reward)[0])
    S = np.zeros((2,K))
    tic = time.time()

    if K >= np.shape(visitor_reward)[0] :
        print(" Warning : More arm than visitors !")

        for j in range(np.shape(visitor_reward)[0]):
            S = PlayArm(j,j,S,visitor_reward)

        choice[0:np.shape(visitor_reward)[0]] = np.arange(0,np.
↪shape(visitor_reward)[0]+1,1)

        if K > np.shape(visitor_reward)[0] :
            S[:,j:K] = 0

        return [S, choice]

    else :

        # Initialisation
        for j in range(K):
            S = PlayArm(j, j, S, visitor_reward)
            choice[1:K] = np.arange(1,K,1)

        for i in range(K,np.shape(visitor_reward)[0]):
            choice[i] = ConditionForEpsilonGreedy(S, epsilon)
            S = PlayArm(i,choice[i],S,visitor_reward)


        toc = time.time()

        # coef estimate :
        theta_hat = S[0,:]

        # real coef :
        theta = pd.DataFrame.mean(visitor_reward0)

        return [S, choice, toc-tic, theta_hat, theta]
```

```
#########################################################################
                        #### UCB ####
#########################################################################



def ProbaMaxForUCB(S,iter,alpha,K):
    choice = np.zeros(K)
    for j in range(K):
        choice[j] = S[0,j] + alpha*sqrt(2*log(iter+1)/S[1,j])
```

```
    return choice

def ConditionForUCB(S,iter,alpha,K):
    return np.argmax(ProbaMaxForUCB(S,iter,alpha,K))

def UCB(visitor_reward0, K = None, alpha = None):

    if K == None :
        K = visitor_reward0.shape[1]
    if alpha == None :
        alpha = 1

    BanditRewardControl(visitor_reward0, K)

    # data formating
    visitor_reward = np.array(visitor_reward0)
    n = np.shape(visitor_reward)[0]

    # keep list of choices
    choice = np.zeros(n)
    proba = np.zeros(n)
    S = np.zeros((2,K))
    tic = time.time()

    if K >= n :
        print("Warning : More arms than visitors")

        for j in range(n):
            proba[j] = max(ProbaMaxForUCB(S,j,alpha,K))
            S = PlayArm(j,j,S,visitor_reward)
        choice = np.arange(0,n+1,1)

        if K > n :
            S[:,j:K] = 0
        return [S,choice]
    else :

        # initialisation

        for j in range(K):
            proba[j] = max(ProbaMaxForUCB(S,j,alpha,K))
            S = PlayArm(j,j,S,visitor_reward)
            choice[j] = j

        for i in range(K,n):
            choice[i] = ConditionForUCB(S,i,alpha,K)
            proba[i] = max(ProbaMaxForUCB(S,i,alpha,K))
            S = PlayArm(i,choice[i],S,visitor_reward)

        toc = time.time()
```

```
        # coef estimate
        theta_hat = S[0,:]

        # real coef
        theta = pd.DataFrame.mean(visitor_reward0)

        return [S,choice,proba,toc-tic,theta_hat,theta]
```

```
##########################################################################
                        #### LINUCB ####
##########################################################################



def ReturnRealTheta(dt, visitor_reward, option = 'linear'):

    K = np.shape(visitor_reward)[1]
    n_f = np.shape(dt)[1]
    theta = []

    temp = pd.DataFrame.copy(dt)
    if option == 'linear' :
        for i in range(K):
            pred = visitor_reward[:,i]
            reg = sm.OLS(endog = pred,exog = temp).fit()
            # Resutats differents par rapport a la regression en R
            theta.append(reg.params)


    # A faire : regression log

    else :
        raise ValueError("Regression non lineaire")

    return theta

def LINUCB(dt, visitor_reward0, alpha = None, K = None,
           IsRewardAreBoolean = False):

    if alpha == None :
        alpha = 1
    if K == None :
        K = np.shape(visitor_reward0)[1]
    if IsRewardAreBoolean == None :
        IsRewardAreBoolean = False
    # control data :
    DataControlK(visitor_reward0,K)
    DataControlContextReward(dt, visitor_reward0)

    # Data Formating :
```

```
visitor_reward = np.array(visitor_reward0)

# ContextMatrix :
D = np.array(dt)
n = np.shape(dt)[0]
n_f = np.shape(D)[1]

# Keep the past choice for regression :
choices = np.zeros(n)
rewards = np.zeros(n)
proba = np.zeros(n)

# Parameters to modelize :
theta_hat = np.zeros((K,n_f))
theta_hat_colnames = dt.columns
theta_hat_index = visitor_reward0.columns

# Regression variable :
b = np.zeros((K,n_f))
A = np.zeros((K,n_f,n_f))

# Tempory variable :
p = np.zeros(K)

# Time keeper :
tic = time.time()

#Initialization :
for j in range(K):
    A[j,:,:] = np.eye(n_f)

for i in range(K):
    x_i = D[i,:]
    x_i0 = np.reshape(x_i,(n_f,1))
    for j in range(K):
        A_inv = np.linalg.inv(A[j,:,:])
        theta_hat[j,:] = A_inv.dot(b[j,:])
        ta = np.transpose(x_i0).dot(A_inv).dot(x_i0)
        a_upper_ci = alpha * sqrt(ta)
        a_mean = theta_hat[j,:].dot(x_i)
        p[j] = a_mean + a_upper_ci

    # Choose the highest :
    choices[i] = i

    # save probability :
    proba[i] = p[i]

    # see what kind of result we get :
    rewards[i] = visitor_reward[i,int(choices[i])]
```

```
        # update the input vector :
        x_it = np.transpose(x_i0)
        A[int(choices[i]),:,:] += x_i0.dot(x_it)
        b[int(choices[i])] += x_i*rewards[i]


    for i in range(K,n):
        x_i = np.array(D[i,:])
        x_i0 = np.reshape(x_i,(n_f,1))
        for j in range(K):
            A_inv = np.linalg.inv(A[j,:,:])
            theta_hat[j,:] = A_inv.dot(b[j,:])
            ta = np.transpose(x_i0).dot(A_inv).dot(x_i)
            a_upper_ci = alpha * sqrt(ta)
            a_mean = theta_hat[j,:].dot(x_i)
            p[j] = a_mean + a_upper_ci

        # choose the highest :
        choices[i] = np.argmax(p)

        # save probability :
        proba[i] = max(p)

        # see what kind of result we get :
        rewards[i] = visitor_reward[i,int(choices[i])]

        # update the input vector :
        x_it = np.transpose(x_i0)
        A[int(choices[i]),:,:] += x_i0.dot(x_it)
        b[int(choices[i])] += x_i*rewards[i]


    tac = time.time()

    # return real theta from a rigid regression :
    if IsRewardAreBoolean ==  False :
        theta = ReturnRealTheta(
            dt=dt,visitor_reward=visitor_reward, option = 'linear')

    # return real theta from logit regression :
    if IsRewardAreBoolean ==  True :
        theta = ReturnRealTheta(
            dt=dt,visitor_reward=visitor_reward, option = 'linear')

    theta_hat = pd.DataFrame(theta_hat)
    theta_hat.columns = theta_hat_colnames
    theta_hat.index = theta_hat_index

    theta = pd.DataFrame(theta)
    theta.columns = theta_hat_colnames
    theta.index = theta_hat_index
```

```
    # return data, models, groups and results :
    return [proba, theta_hat, theta, choices, tac - tic]
```

Annexe 2 : Optimisation de la consommation énergétique

```
def load_data(freq):
    list_freq =␣
 ↪[1200000,1300000,1400000,1500000,1600000,1700000,1800000,1900000,
                2000000,2100000,2200000,2300000,2400000]

    if freq not in list_freq :
        raise ValueError('Incorrect frequency')

    path ='/home/agary/Documents/Data/npb_fast/02_remove_wm/'
    f1 = 'npb_graoully-'
    f2 = ['1','1','1','1','1','2','3','6','10','10','11','11','12','12','13',
          '13','13','13','14','15','16']
    f3 = '.nancy.grid5000.fr_'
    f4 = ['1637601437','1637609938','1637875436','1637899219',
          '1637920836','1637601340','1637525287','1637920832',
          '1637525761','1637601438','1637609677','1637875436',
          '1637898864','1637899935','1636306023','1637525287',
          '1637899037','1637899639','1637920837','1637609637',
          '1637875436']
    f5 = '_cleaned.csv'


    filename = f1+f2[0]+f3+f4[0]+f5
    df_t = pd.read_csv(path + filename, sep=' ')
    df = df_t.loc[df_t['fmax']==freq]

    for i in range(1,len(f2)):
        filename = f1+f2[i]+f3+f4[i]+f5
        temp = pd.read_csv(path + filename, sep=' ')
        temp = temp.loc[temp['fmax']==freq]

        df = pd.concat([df,temp])

    return df
```

```
def clustering(df,t,c):

    if t <= 4 :
        raise ValueError("time t must be >= 4")

    path ='/home/agary/Documents/Data/npb_fast/02_remove_wm/'
    m,n = df.shape
    machines = ['1','2','3','4','5','6','7','8','9',
                '10','11','12','13','14','15','16']
    dataframe0 = pd.DataFrame()
    list_server = []
    list_app = []
    list_duration = []
```

```
    list_basename = []

    for i in range(m):

        basename = df.iat[i,9]
        path_temp = path + basename + '_mojitos/'
        # creation du chemin jusqu'au fichier correspondant
        key = df.iat[i,4]
        app = df.iat[i,1]

        for j in machines :

            name = 'graoully-' + j + '.nancy.grid5000.fr_' + app + '_' +␣
↪str(key)
            filename = path_temp + name

            # test si fichier existe : si oui : manip :

            if os.path.exists(filename) :
                dt = pd.read_csv(filename,sep=' ')
                time = dt.iat[0,0]
                end = dt.iat[-1,0]
                dt_0 = dt[dt['#timestamp']<=time+t]
                dt_partial = dt[dt['#timestamp']>=time+4]
                dt_median = dt_partial.median()

                dataframe0 = dataframe0.append(dt_median,ignore_index=True)
                list_server.append(j)
                list_basename.append(basename)
                list_app.append(app)
                list_duration.append(end-time)

    dataframe0['server'] = list_server
    dataframe0['basename'] = list_basename
    dataframe0['app'] = list_app
    dataframe0['duration'] = list_duration
    dataframe0 = dataframe0.drop(['#timestamp'],axis=1)

    dt_final0 = dataframe0.drop(['server','app','basename'],axis=1)
    #print(dt_final0.isnull().sum().sum())
    kmeans0 = KMeans(n_clusters=c)
    kmeans0.fit(dt_final0);

    dataframe0['kmeans_labels'] = kmeans0.labels_

    return dataframe0,kmeans0
```

```
def calcul_freq(data,res_linucb):

    liste_clusters = [0,1,2,3,4]
```

```
frequences = [1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2.0,2.1,2.2,2.3,2.4]

temps_execution = []
consommations = []
best_consos = []

best_freq = [] # On stocke la frequences optimale
gains = [] # On stocke les gains moyens

for k in range(len(liste_clusters)) :
    clust = liste_clusters[k]

    # Calcul des temps moyens :

    indices_clusters = np.where(data['kmeans_labels'] == clust)[0]
    #On récupère les indices associés à l'application
    n = data.shape[0]
    temps = np.zeros(13)

    for i in range(n):
        if i in indices_clusters :
            for j in range(13):

                temps[j] += data.iat[i,j+1]

    temps /= len(indices_clusters)
    temps_execution.append(temps)

    # Calcul de la consommation prévue

    consos = []
    for i in range(13):
        c = res_linucb.iat[i,k]
        d = res_linucb.iat[i,-1]


        energie = abs(c + temps[i]*d)
        consos.append(energie)

    consommations.append(consos)
    indices_min = np.argmin(consos)
    best_consos.append(min(consos))

    best_freq.append(frequences[indices_min])

    # Construction du tableau recaptulatif :

    consos.remove(best_consos[-1])
    moy = np.mean(consos)

    if moy == 0 :
```

```
            gains.append('NA')
        else :
            gains.append(moy - best_consos[-1])

    a = np.array([best_freq, best_consos, gains])
    a = np.transpose(a)
    recap = pd.DataFrame(data=a, index=liste_clusters,
                         columns=['Frequence optimale',
                                  'Consommation optimale', 'Gain moyen'])
    return [recap, temps_execution, consommations]
```

```
def bandits(dataframe0):

    rewards = pd.DataFrame(dataframe0[['app','basename']])
    m,n = rewards.shape
    path = '/home/agary/Documents/Data/npb_fast/02_remove_wm/'
    energies = np.zeros((13,m))
    durations = np.zeros((13,m))

    for i in range(m):

        filename = rewards.iat[i,1]+'_cleaned.csv'
        data = pd.read_csv(path+filename,sep=' ')
        data = data.loc[data['fullname']==rewards.iat[i,0]]

        if data.shape[0] > 13 :
# lorsqu'une même application soit testée plusieurs fois dans un même fichier
            s = data.shape[0] // 13
            en = list(data['energy'])
            du = list(data['time'])
            for j in range(s):
                en_temp = en[int(j*13):int((j+1)*13)]
                du_temp = du[int(j*13):int((j+1)*13)]
                energies[:,i] = en_temp
                durations[:,i] = du_temp

        else :
            energies[:,i] = list(data['energy'])
            durations[:,i] = list(data['time'])

    energies = np.transpose(energies)
    durations = np.transpose(durations)

    rewards = pd.concat([rewards,pd.DataFrame(energies)],axis=1)
    list_rewards = ['ref_energy',
                    'ref_energy.1','ref_energy.2','ref_energy.3',
                    'ref_energy.4','ref_energy.5','ref_energy.6','ref_energy.
 ↪7',
                    'ref_energy.8','ref_energy.9','ref_energy.10',
                    'ref_energy.11','ref_energy.12']
```

```
    rewards = rewards.drop(['app','basename'],axis=1)
    rewards.columns = list_rewards

    context = pd.get_dummies(dataframe0['kmeans_labels'])
    context['duration'] = dataframe0['duration']
    list_durations = ['duration','duration.1','duration.2',
                      'duration.3','duration.4','duration.5',
                      'duration.6','duration.7','duration.8','duration.9',
                      'duration.10','duration.11','duration.12']
    durations = pd.DataFrame(durations,columns=list_durations)

    dt = dataframe0['kmeans_labels'].copy()
    dt = pd.concat([dt,durations,rewards],axis=1)

    rewards = (-1)*rewards
    alloc_linucb = ba.LINUCB(context,rewards)

    recap,temps_execution,consommations = calcul_freq(dt,alloc_linucb[1])

    return recap
```

```
def clustering_bandits(freq,t,c):

    df = load_data(freq)

    dataframe0,Kmeans0 = clustering(df,t,c)

    recap = bandits(dataframe0)

    return recap,Kmeans0
```

```
def attribue_cluster(Kmeans0,app,n):
    app = app.median()
    dist = np.zeros(n)
    centroids = kmeans0.cluster_centers_

    for i in range(n):
        dist[i] = np.linalg.norm(test - centroids[i])

    return np.argmin(dist)
```