ÉCOLE POLYTECHNIQUE DE MONTRÉAL

LOG8371- Ingénierie de la qualité en logiciel

TP2 : Efficacité et performance

RÉALISÉ PAR :

HIBA BAGANE - 1679837
SALOUA DJEBELI - 1719832
SAMIA JALIL - 1720168

REMIS À :
Saif-eddine Sajid
Seddik BENYAHIA

21 Mars 2019

Table des matières

Introduction

Weka (*Waikato Environment for Knowledge Analysis*) est un logiciel libre développé par un groupe de recherche de l'université Waikato et issue sous la licence **GNU**. Il s'agit d'une collection d'algorithmes d'apprentissage machine - ***Machine Learning*** pour la fouille de données - ***Data Mining***. Le logiciel contient des outils pour la préparation, la classification, la régression, le regroupement - ***Clustering***, l'association et la visualisation des données. De plus, il est possible d'utiliser Weka pour traiter des mégadonnées et mettre en pratique l'apprentissage approfondi - ***Deep Learning***.

Le code source utilisé dans ce travail est développé avec le langage de programmation Java. Les algorithmes sont accompagnés par des tests dans l'entrepôt Github du projet.

Weka possède trois composants principaux et chacun de ces derniers possède un rôle spécifique :

- ***Explorer :*** contient tous les paquetages importants de Weka tel que les algorithmes d'apprentissage, les associations, la sélection des attributs et la visualisation etc..
- ***Experimenter :*** permet de créer et d'exécuter différents algorithmes d'apprentissage sur plusieurs données en mode batch et d'analyser les résultats.
- ***Knowledge Flow :*** ce module représente graphiquement les même fonctionnalités d'Explorer.

Dans le cadre de ce travail, nous commençons par mettre à jour le plan de qualité du TP1 pour ajouter les critères et les sous-critères qui portent sur l'efficacité et la performance de Weka. Puis, nous déployons une version REST de Weka en utilisant des containeurs Docker. Ensuite, nous étudions la consommation des ressources du point de vue logiciel et de l'infrastructure. On termine avec le contrôle automatique de l'allocation dynamique de ressources.

Q1 : mise à jour du tp1

1. Exigences fonctionnelle

Les fonctionnalités disponibles dans Weka sont les suivantes :

- Apprentissage machine;
- Fouille de données;
- Prétraitement;
- Classification;
- Régression linéaire;
- Clustering;
- Règles d'association;
- Sélection d'attributs;
- Expériences et modèles;
- Flot de travail;

- Visualisation de modèles.

2. Critères et sous-critères de qualités cibles

Les tests de logiciel représentent un facteur critique pour la validation et la vérification d'un système d'apprentissage comme Weka. Dans cette section, nous allons présenter 4 critères de qualité et expliquer leur importance dans le projet.

Tableau 1 : Sous-critères de qualité pour le critère *Functional suitability*

Critère de Qualité : Functional suitability			
Sous-critères	**Description**	**Justifications**	**Objectifs**
Complétude	Ce type de test permet de mesurer le degré pour lequel l'ensemble de fonctions couvre toutes les tâches et objectifs de l'utilisateur.	Il est primordial de tester la complétude des algorithmes de classification (LinearRegression, Logistic et `BayesNet`), de clustering (SimpleKMeans) et d'association (Apriori, FPGrowth) afin d'avoir une analyse de données depuis différentes perspectives et de transformer ces données en informations utiles. Ces informations vont affecter les décisions prises par les entreprises.	La complétude des tâches des 5+1 algorithmes de Weka qu'on a choisi doit être atteindre au moins 95%.
Exactitude	Les tests d'exactitude ont pour but de vérifier que les réponses proposées par le logiciel sont bien celles attendues et avec le niveau de précision attendu. Ces informations sont généralement fournies par des spécifications fonctionnelles.	Les 5+1 algorithmes de Weka permettent de classifier, segmenter les données et d'évaluer les probabilités futures. L'utilisation de ces algorithmes touchent plusieurs domaines qui demandent plus de précisions comme le domaine de la santé, l'économie etc. Donc, les résultats doivent répondre aux besoins de leurs utilisateurs.	L'exactitude des analyses faites pour les 5+1 fonctionnalités doit être supérieure ou égale à 90%.

Tableau 2 : Sous-critères de qualité pour le critère Fiabilité

Critère de Qualité : Fiabilité			
Sous-critères	**Description**	**Justifications**	**Objectifs**
Récupérabilité	Ce type de test permet de vérifier si le système peut récupérer les données affectées directement en cas d'interruption ou de défaillance,et rétablir l'état souhaité du système	Il arrive parfois que l'un des 5+1 algorithmes de Weka ne sera plus fonctionnel. Il est important de pouvoir corriger l'erreur et rendre l'application disponible le plus rapidement possible. Les tests de récupérabilité sont importants afin de vérifier le temps nécessaire pour remettre en état de marche les algorithmes lors de certaines erreurs.	Le temps de récupération doit être au maximum 3 heures en cas de défaillance.
Maturité	Ce type de test permet de mesurer le degré pour lequel un système répond aux besoins de fiabilité en mode de fonctionnement normal et d'assurer que les requis des utilisateurs sont bien couverts.	Les 5+1 algorithmes sont des programmes libres qu'on peut les modifier c'est pour cela il est très important d'effectuer des tests de maturité afin de vérifier que la couverture des besoins des utilisateurs dans la grande majorité des cas est bien là. Par exemple, les algorithmes de clustering (SimpleKMeans) doivent être en mesure de grouper les données en sous-ensembles cohérents, les algorithmes de classification (LinearRegression, Logistic et BayesNet) doivent être capable d'utiliser les données stockées afin de localiser les données en groupes prédéterminés, filtrer les données inutiles et de les ignorer.	Weka doit doit être en mesure de s'adapter et s'ajuster avec les besoins des utilisateurs dans plus de 80% des cas.
Tolérance aux	Ce type de test permet de	Weka cible des domaines très	Le système de WEKA doit

fautes--- Robustesse	mesurer le degré pour lequel un système, un produit ou un composant fonctionne comme prévu malgré la présence des défauts matériels ou logiciels.	importants et parfois les données qu'elle traite sont très sensibles. Les 5+1 algorithmes doivent être plus ou moins capables de gérer les défauts matériels ou logiciels pour que ces derniers n'impactent pas les analyses fournis par ces algorithmes. On peut par exemple se tromper et utiliser l'algorithme Association.Apriori pour générer des règles d'association sur les attributs contenus dans un fichier d'extension «. Accdb » à la place de «. arff ». Ce scénario doit être reconnus et l'utilisateur doit en être informé. Il ne faut pas que l'application crash à cause d'une erreur.	être capable de traiter les fautes et les erreurs dans 80% des cas.

Tableau 3 : Sous-critères de qualité pour le critère Maintenabilité

Critère de Qualité : Maintainability			
Sous-critères	**Description**	**Justifications**	**Objectifs**
Réutilisabilité	Ce type de test permet de mesurer le degré pour lequel les algorithmes développés peuvent être réutilisés sur d'autres logiciels.	Les 5+1 algorithmes développés par Weka sont utilisés pour explorer et expérimenter des différentes données dans n'importe quel domaine. Avec ce type de test, on s'assure que ces algorithmes peuvent être réutilisés sur d'autres logiciels de data-mining.	Les algorithmes de Weka doivent être réutilisés sur au moins 80% des différents types de données
Modifiabilité	Ce type de test permet de mesurer le degré pour lequel un système peut être efficacement modifié sans introduire de défauts ou une dégradation de la qualité des fonctionnalités existantes.	Vu les dépendances entre les modules : « classifiers », « clusters » et « associations » de Weka, ce test est nécessaire pour s'assurer que les changements faits sur un module n'affectent pas les	Le fonctionnement de Weka doit avoir toujours les même comportements attendus suite à l'ajout ou la modification de nouvelles fonctionnalités.

		autres modules.	

Tableau 4 : Sous-critères de qualité pour le critère Performance/

Critère de Qualité : Performance/Efficacité			
Sous-critères	**Description**	**Justifications**	**Objectifs**
Temps de réponse	Ce type de test permet de mesurer le temps que l'application met pour répondre entre le moment où l'utilisateur demande la ressource et le moment ou le résultat sera affiché.	Il est primordial de tester le temps de réponse des algorithmes de classification (LinearRegression, Logistic et BayesNet), de clustering (SimpleKMeans) et d'association (Apriori, FPGrowth) afin de s'assurer que le système traite les données en un temps raisonnable. Les temps de réponse trop longs sont souvent objet de non utilisation du système.	Le temps de réponse pour chaque fonctionnalité de 5+1 algorithmes ne doit pas dépasser 3 secondes.
Capacité	Ce type de test permet de connaître les limites maximales d'une application ou d'un logiciel. Ce test permet de détecter combien d'utilisateurs peuvent effectuer des tâches simultanément, combien de requêtes peuvent être exécutés etc… .	Les tests de capacités sont importants pour déterminer les limites de volume d'information que les 5+1 algorithmes de Weka capable de les traiter sans que le système crache.	La capacité des 5+1 algorithmes de Weka de traiter des données volumineuses sans que le système crache doit être réalisable dans 75% des cas.
Utilisation de ressource	Ce type de test permet de calculer les ressources nécessaires à l'application pour fonctionner.	Les tests d'utilisation des ressources sont importante pour mesurer les ressources nécessaires aux 5+1 algorithmes pour s'exécuter correctement.	Utilisation des ressources par les 5+1 algorithmes sans avoir des problèmes comme les fuites de mémoires.

3. Plan de vérification et validation des objectifs

3.1. Plan de test

Portée des tests :

Nous avons choisi 5 algorithmes à tester :

1. **Association.**Apriori**:** cet algorithme permet de générer les règles d'association réduite sur les différents attributs contenus dans les fichiers d'extension .arff.

2. **Association.**FPGrowth**:** cet algorithme permet de trouver des ensembles d'articles volumineux sans génération de candidats. Il permet de réduire itérativement le support minimum jusqu'à ce qu'il trouve le nombre requis de règles avec la métrique minimale donnée.

3. **Clusters.**SimpleKMeans**:** cet algorithme permet de gérer automatiquements les jeux de données selon des critères spécifiques lors des calculs des distances. L'algorithme WEKA SimpleKMeans utilise une mesure de distance euclidienne pour calculer les distances entre les instances et les clusters.

4. **Classifiers.Functions.**LinearRegression**:** cet algorithme utilise la régression linéaire pour la sélection d'un modèle selon le critère AKAIKE et il est capable de gérer des instances pondérées.

5. **Classifiers.Functions.**Logistic**:** cet algorithme permet de construire des modèles de logistique linéaire. LogitBoost avec des fonctions de régression simples en tant que apprenants de base et est utilisé pour adapter les modèles logistiques. Le nombre optimal d'itérations LogitBoost à effectuer est validé de manière croisée, ce qui conduit à la sélection automatique d'attributs.

   Pour ce deuxième tp, nous avons ajouté un 6eme algorithme:

6. **classifiers.bayes**.BayesNet: cet algorithme utilise divers algorithmes de recherche et mesures de la qualité. il s'agit d'une classe de base pour un classifieur Bayes Network. permet d'apprendre les infrastructures de données tel que la structure de réseau, distributions de probabilités conditionnelles, etc.) et des installations communes aux algorithmes d'apprentissage de Bayes Network tels que K2 et B.

Déroulement des tests :

Nous allons commencer par exécuter les tests unitaires disponibles déjà dans weka. Ensuite, nous allons exécuter les tests d'intégration. Finalement, on exécute les tests de régression.

Environnement de test :

Pour exécuter nos tests, on aura besoin :

- Eclipse (JUnit)
- Un serveur Jenkins (configuré avec JDK8 ant/maven)
- Github

3.1.1. Description des tests

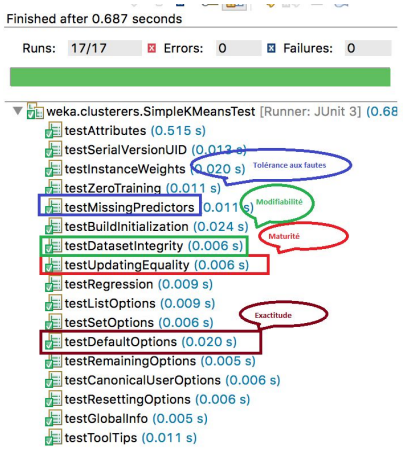
Tableau 4 : Critères de qualité couverts par les tests choisis

Tests unitaires							
Algorithmes	**Critères couverts**						
	Exactitude	**Complétude**	**Modifiabilité**	**Maturité**	**Tolérance aux fautes Robustesse**	**Réutilisabilité**	**Récupérabilité**
FPGrowth	x			x	x		
Apriori			x	x		x	x
SimpleKMeans	x			x	x		
LinearRegression		x	x				x
Logistc	x	x		x		x	

3.1.2. Rapports de tests

3.1.2.1. Résultats

➔ Tous les tests réussissent après leur exécution (voir **Figure 1**).



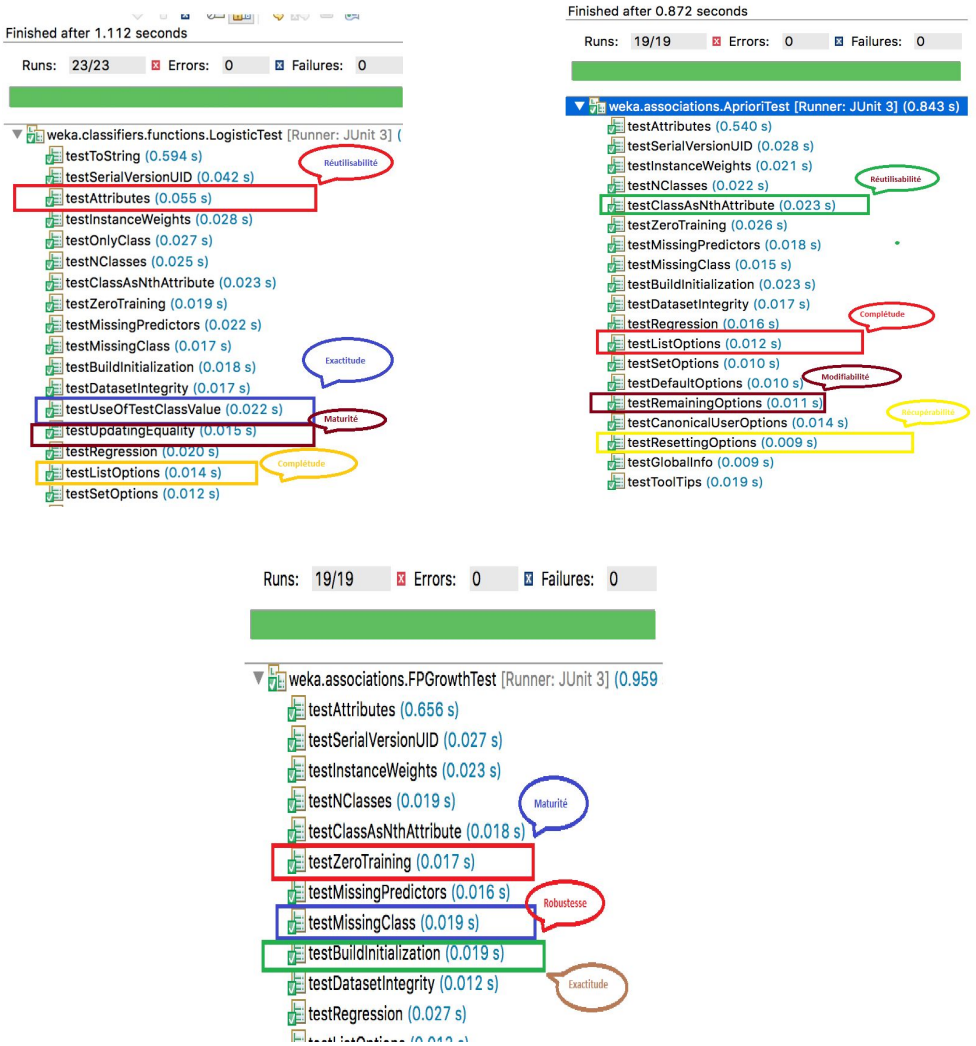


Figure 1 : Résultats de l'exécution des tests sur les 5 algorithmes choisis.

3.2. Stratégies

3.2.1. Intégration continue

Pour cette partie, nous avons choisi de configurer notre entrepôt Github avec l'outil d'intégration continue Jenkins. Nous avons configuré et déployé notre serveur Jenkins, ensuite nous avons ajouté un webhook dans la configuration de notre entrepôt de code. Le projet a été configuré sur Jenkins pour pouvoir effectuer un build automatique du projet Weka avec Maven et JDK 8. Les tests seront exécutés pendant le build et les résultats seront disponibles dans la vue du projet sur Jenkins. Après chaque ajout de code dans l'entrepôt, un build automatique sera lancé par Jenkins. Si tous les tests réussissent, le build sera réussi et donc stable. Dans le cas où des tests échouent, le build sera instable.

Résultats d'intégration obtenus avec Jenkins :

All Failed Tests

Test Name	Duration	Age
weka.filters.unsupervised.attribute.PrincipalComponentsTest.testBuffered	7 ms	1
weka.classifiers.functions.GaussianProcessesTest.testUseOfTestClassValue	12 ms	2
weka.attributeSelection.PrincipalComponentsTest.testRegression	26 ms	5
weka.classifiers.evaluation.EvaluationTest.testRegression	22 ms	5
weka.classifiers.functions.GaussianProcessesTest.testRegression	19 ms	5
weka.classifiers.functions.GaussianProcessesTest.testBuildInitialization	16 ms	5
weka.classifiers.functions.LinearRegressionTest.testRegression	9 ms	5
weka.classifiers.meta.ClassificationViaRegressionTest.testRegression	16 ms	5
weka.classifiers.rules.M5RulesTest.testRegression	16 ms	5
weka.classifiers.trees.M5PTest.testRegression	50 ms	5

On remarque que le build est instable sur Jenkins, car il y a 10 tests (majoritairement des tests de régression) qui échouent. Les tests échouent, car les sorties obtenues ne correspondent pas au sorties attendues par les assertions.

3.2.2. Revue et inspection

Le but des activités de revue et d'inspection est de détecter les défauts d'une manière manuelle. Les revues de code se font par une ou plusieurs personnes afin d'inspecter la qualité de code, son fonctionnement et son efficacité. Pour le projet Weka, ce genre d'activité est très important. Afin de vérifier et de valider la qualité du code produit dans ce projet, nous avons estimé qu'il serait nécessaire d'effectuer une séance de revue de code après la complétion de chaque fonctionnalité, ajout ou modification dans Weka. La revue de code peut se faire sur Github. De plus, pour les activités d'inspection, il faut toujours produire des checklists et les mettre à jour après la complétion de chaque activité.

Q2 : Un petit manuel avec les commandes utilisées pour déployer une version REST de Weka en Docker :

➔ On télécharge le code source « jguwekarest » de Github en utilisant la commande :

git clone https://github.com/jguwekarest/jguwekarest.git

➔ On rentre au répertoire « jguwekarest » en utilisant la commande :

cd jguwekarest

➔ On compile le fichier .war avec maven, qui sera utilisé pour construire l'image Docker, en utilisant la commande :

mvn clean package

➔ On construit l'image Docker en utilisant la commande :

docker build -t tp2Log8371/jguweka:OAS3 .

➔ On vérifie l'existence des images dans notre répertoire Docker local en utilisant la commande :

docker images

➔ On télécharge la dernière image Mongo à partir de Docker Hub en utilisant la commande :

docker pull mongo

➔ On lance le container qui sera identifié par le nom « mongodb » en utilisant la commande :

docker run --name mongodb -d mongo

➔ On lie notre base de données « mongodb » au container « tp2Log8371/jguweka:OA », ensuite on lance ce dernier et on expose le port 8080 à l'extérieur. On effectue tout ça en utilisant la commande :

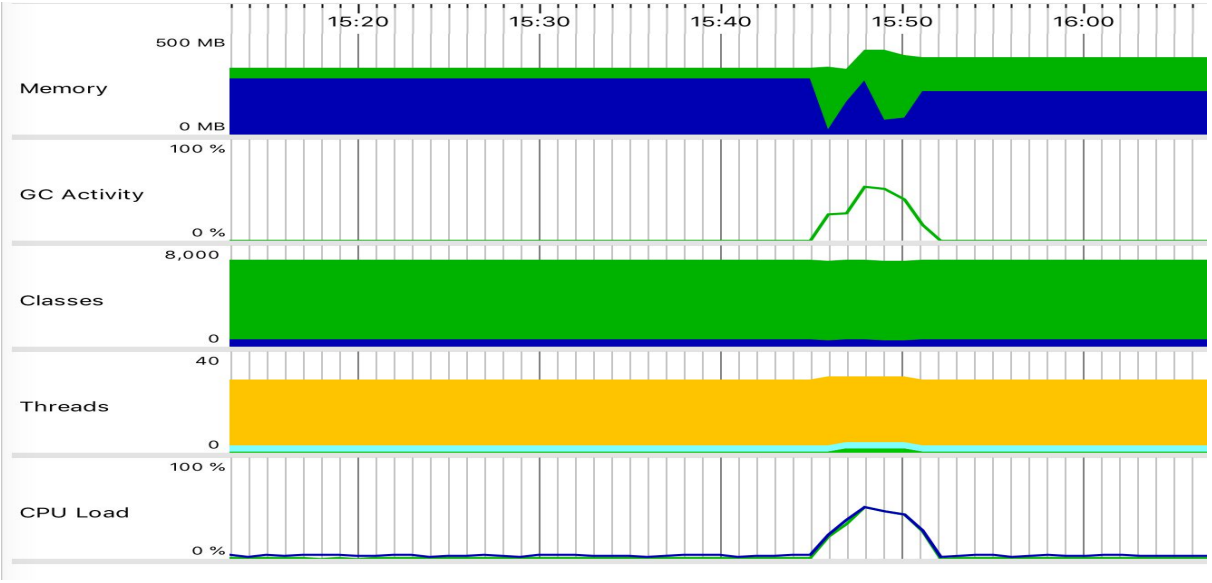
docker run -p 8080:8080 --link mongodb:mongodb tp2Log8371/jguweka:OAS3

Q3 :Profiling

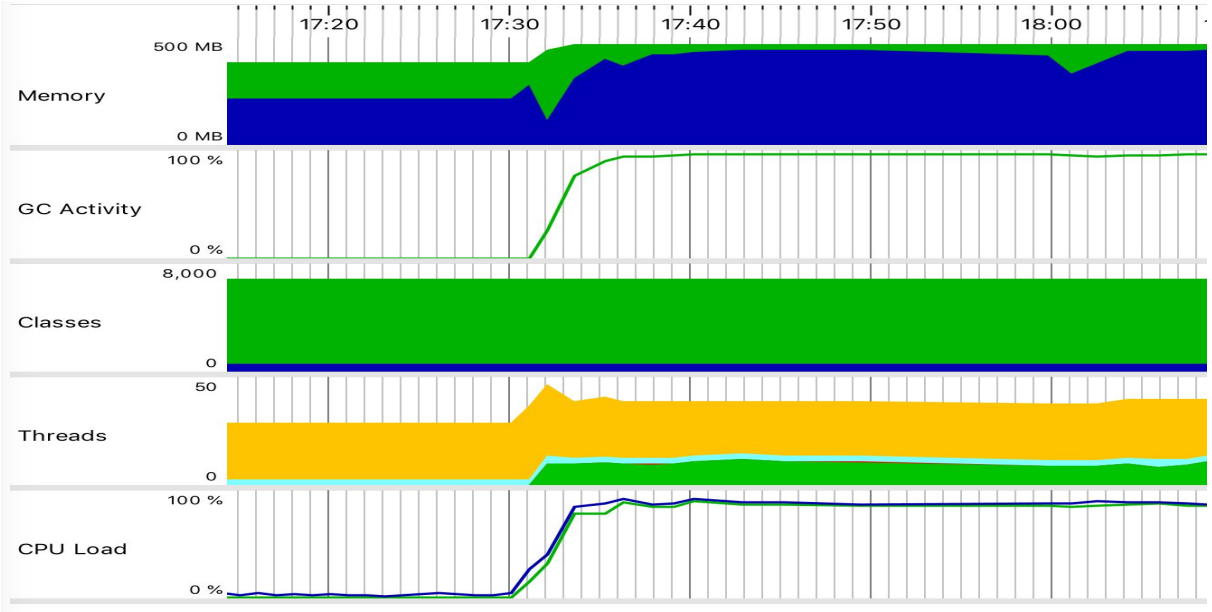
Pour les 4 scénarios suivant, nous avons utilisés les données des fichiers breast-cancer.arff, glass.arff et soybean.arff comme données d'entrée. Pour cette question, nous avons utilisé le même algorithme pour les 4 scénarios : Bayes Net (POST algorithm/BayesNet).

Le snapshot de profiling est disponible dans le dossier Q3.

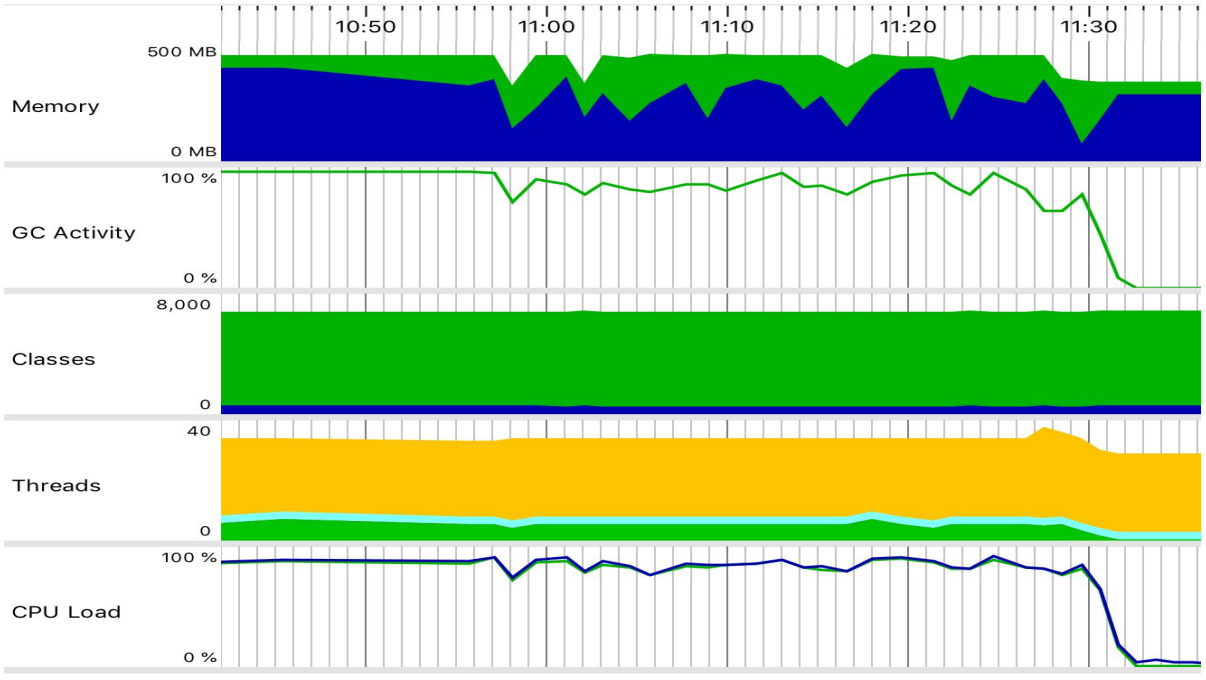
Scénario 1 : charge réduite : 1 utilisateur



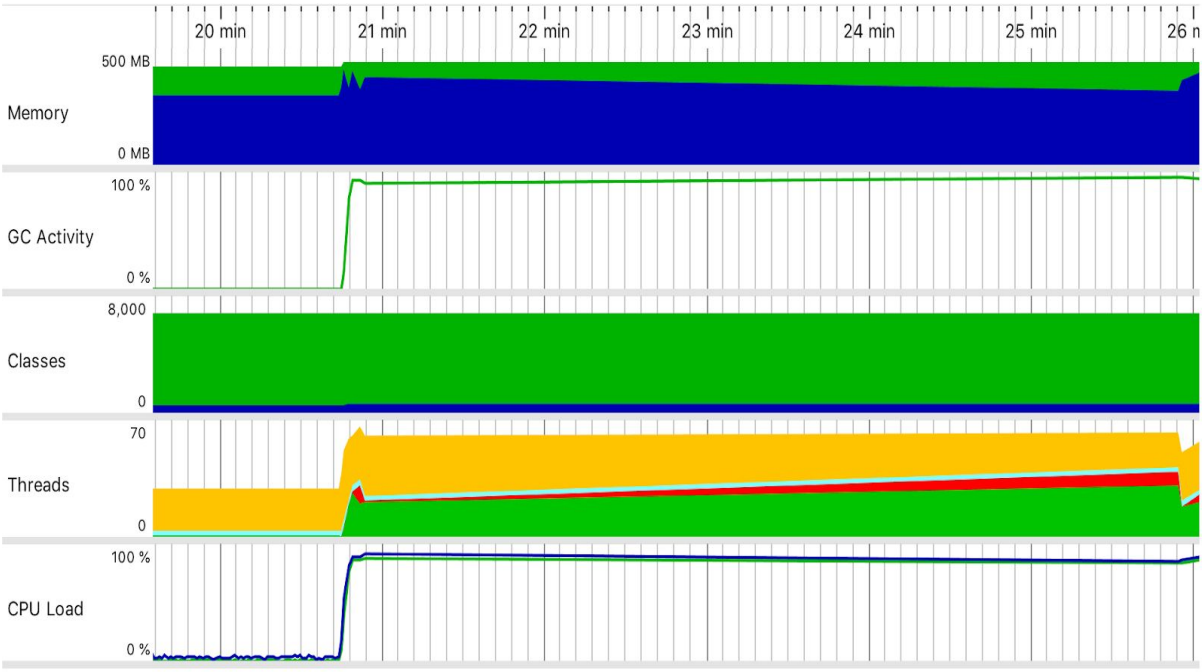
Scénario 2 : charge moyenne : 10 utilisateurs



Scénario 3 : charge augmentée : 100 utilisateurs



Scénario 4 : charge augmentée exceptionnelle : 1000 utilisateurs



Q4 :Tests de charge

A. Analyse des résultats

Scénario 1 : charge réduite : 1 utilisateur (1 thread) 500 loops

Scénario 2 : charge moyenne : 10 utilisateurs (10 threads) 1000 loops

Scénario 3 : charge augmentée : 100 utilisateurs (100 threads) 1500 loops

Scénario 4 : charge augmentée exceptionnelle : 1000 utilisateurs (1000 threads) 2000 loops

Pour analyser les performances du service REST testé, nous allons nous concentrer sur 2 paramètres.

Débit : le débit est le paramètre le plus important. Cela représente la capacité du serveur à gérer une charge importante. Plus le débit est élevé, plus la performance du serveur est meilleure.

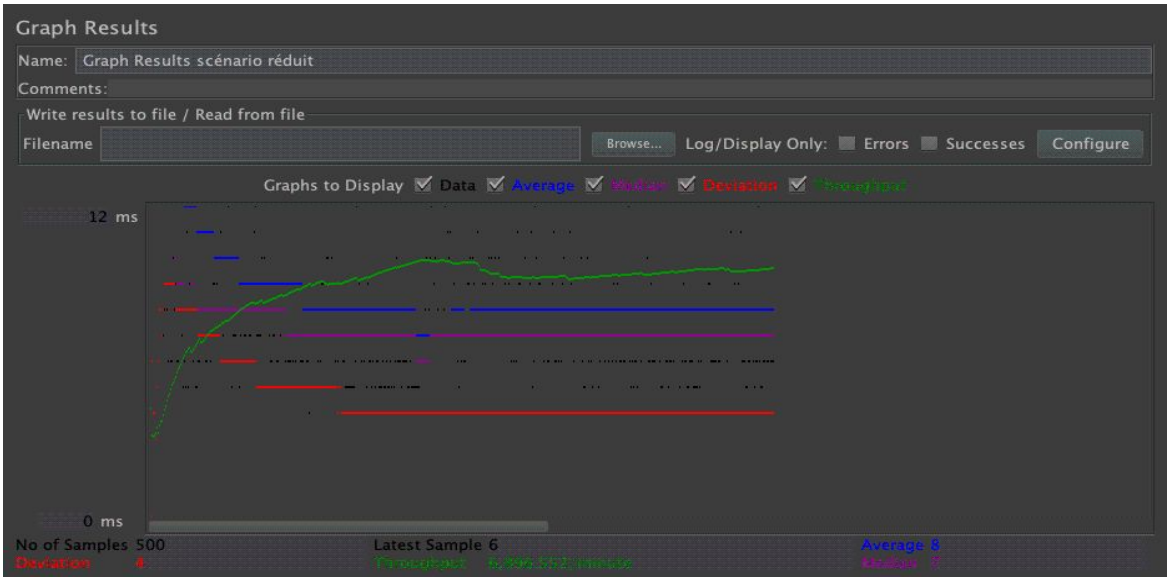
Déviation : la déviation indique l'écart par rapport à la moyenne. Plus la déviation est petite plus la performance est meilleure

Scénario 1 : charge réduite : 3 utilisateurs (3 thread) 500 loops

Algorithme Testé: Logistic

Méthode : GET

URL : /algorithm/generic?classifierName=Logistic



Pour ce scénario, le débit du notre serveur est de 6,896.552 / minute. Cela signifie que le serveur de Weka peut traiter à la fois 6896.552 requêtes par minute en utilisant l'algorithme Logistic. Cette valeur est élevée et elle prouve la haute performance de ce serveur pour le scénario réduit. Par contre, la déviation est très petite (4) ce qui confirme la satisfaction de cette performance.

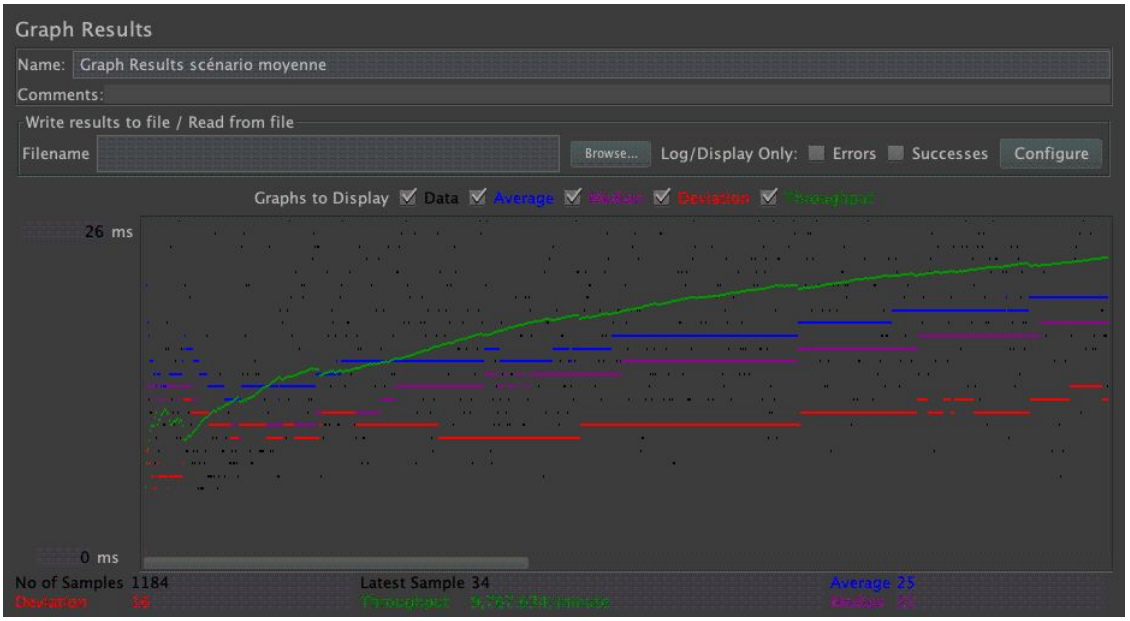
Scénario 2: charge moyenne : 10 utilisateurs (10 thread) 1000 loops

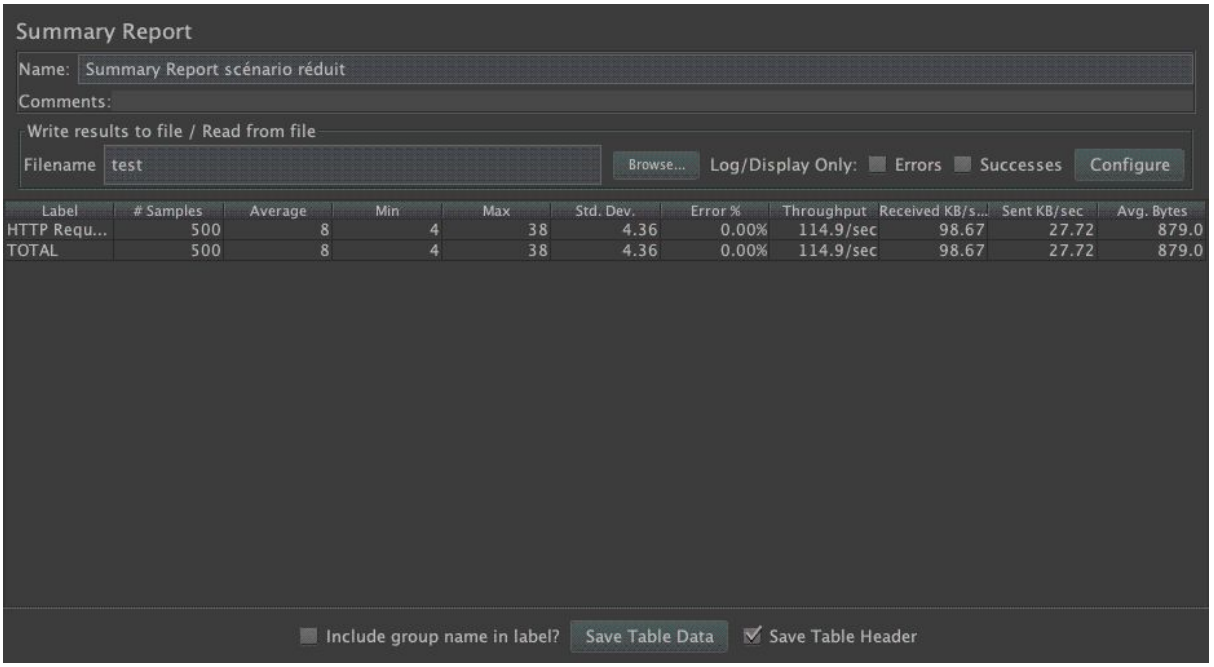
Algorithme Testé: Logistic

Méthode : GET

URL : /algorithm/generic?classifierName=Logistic

Résultats :





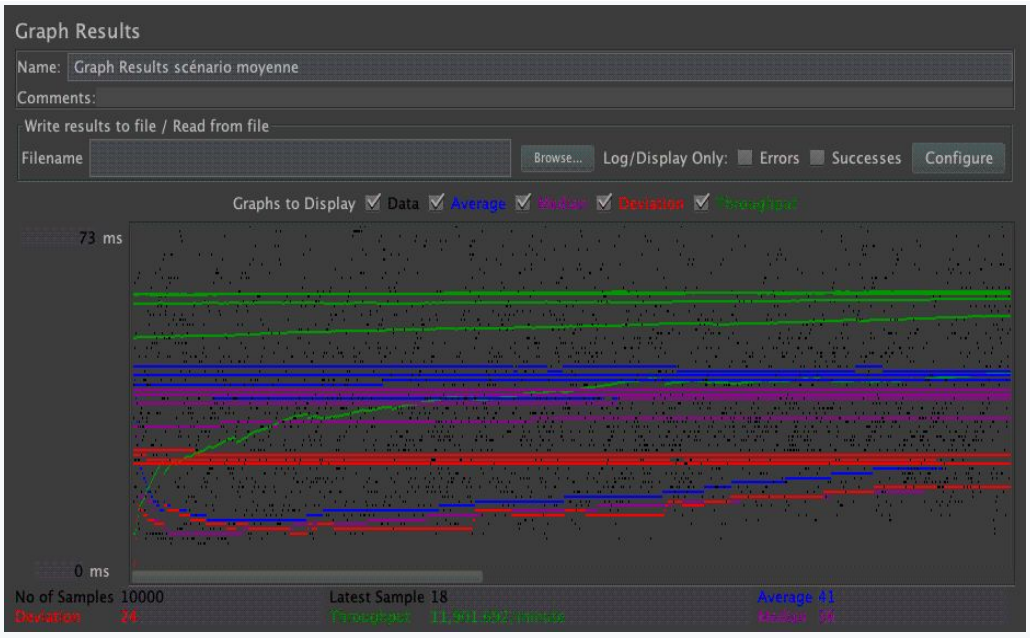
Pour ce deuxième scénario, le débit du notre serveur est de 9,767.634 / minute. Cela signifie que le serveur de Weka peut traiter à la fois 9767.634 requêtes par minute en utilisant l'algorithme Logistic. Cette valeur est plus élevée que le scénario 1 donc nous pouvons confirmer que les performances de ce serveur sont assez satisfaisantes pour le scénario moyen.

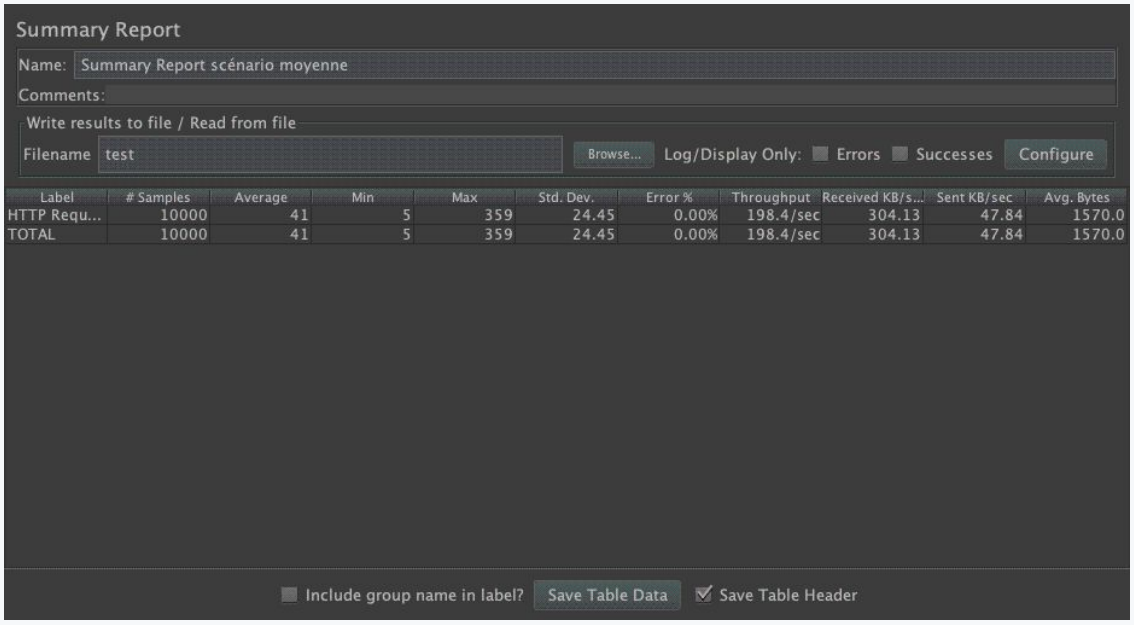
Scénario 2: charge moyenne : 10 utilisateurs (10 threads) 1000 loops

Algorithme Testé: BayesNet

Méthode : GET
URL : /algorithm/generic?classifierName=BayesNet

Résultats :





Pour ce deuxième scénario, le débit du notre serveur est de 11,901.692 / minute. Cela signifie que le serveur de Weka peut traiter à la fois 11901.692 requêtes par minute en utilisant l'algorithme BayesNet. Cette valeur est plus élevée que le scénario 2 en utilisant l'algorithme Logistic, donc nous pouvons confirmer que les performances de ce serveur sont très satisfaisantes pour le scénario moyen en utilisant l'algorithme BayesNet.

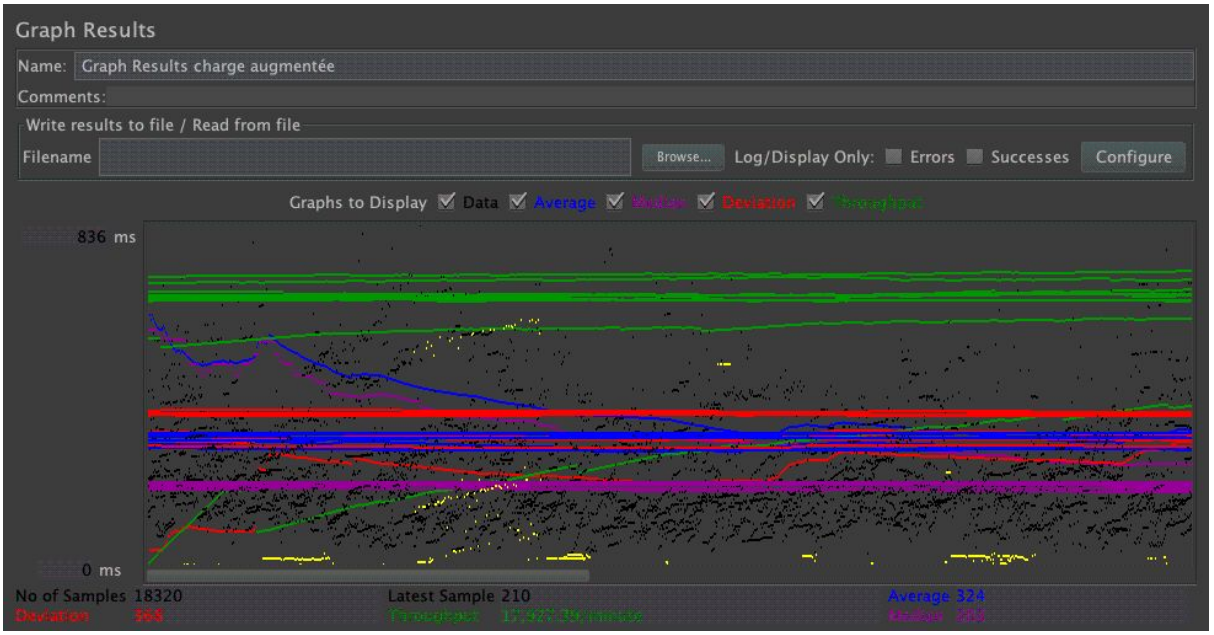
Scénario 3: charge augmentée : 100 utilisateurs (100 thread) 1500 loops

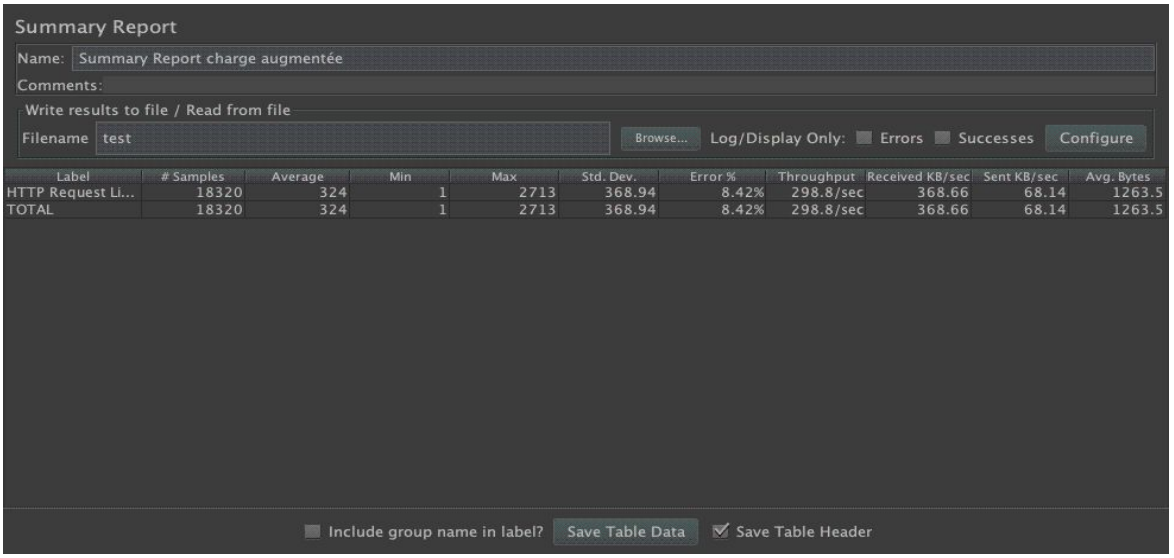
Algorithme Testé: LinearRegression

Méthode : GET

URL : /algorithm/generic?classifierName=LinearRegression

Résultats :





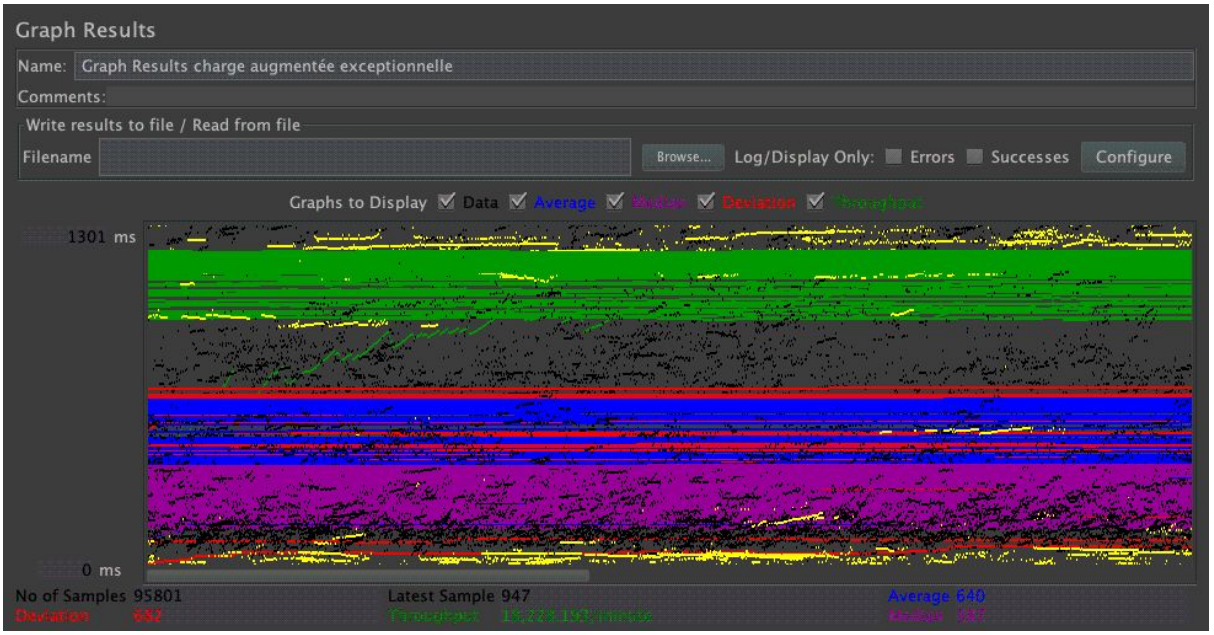
Pour le scénario 3 (charge augmentée) ,le débit du notre serveur est de 17,901.39 / minute. Cela signifie que le serveur de Weka peut traiter à la fois 17901.39 requêtes par minute en utilisant l'algorithme LinearRegression. Cette valeur est très élevée, donc nous pouvons confirmer que les performances de ce serveur sont satisfaisantes et efficaces pour un scénario moyenne en utilisant l'algorithme LinearRegression.

Scénario 3: charge augmentée exceptionnelle : 1000 utilisateurs (1000 threads) 2000 loops

Algorithme Testé: LinearRegression

Méthode : GET

URL : /algorithm/generic?classifierName=LinearRegression

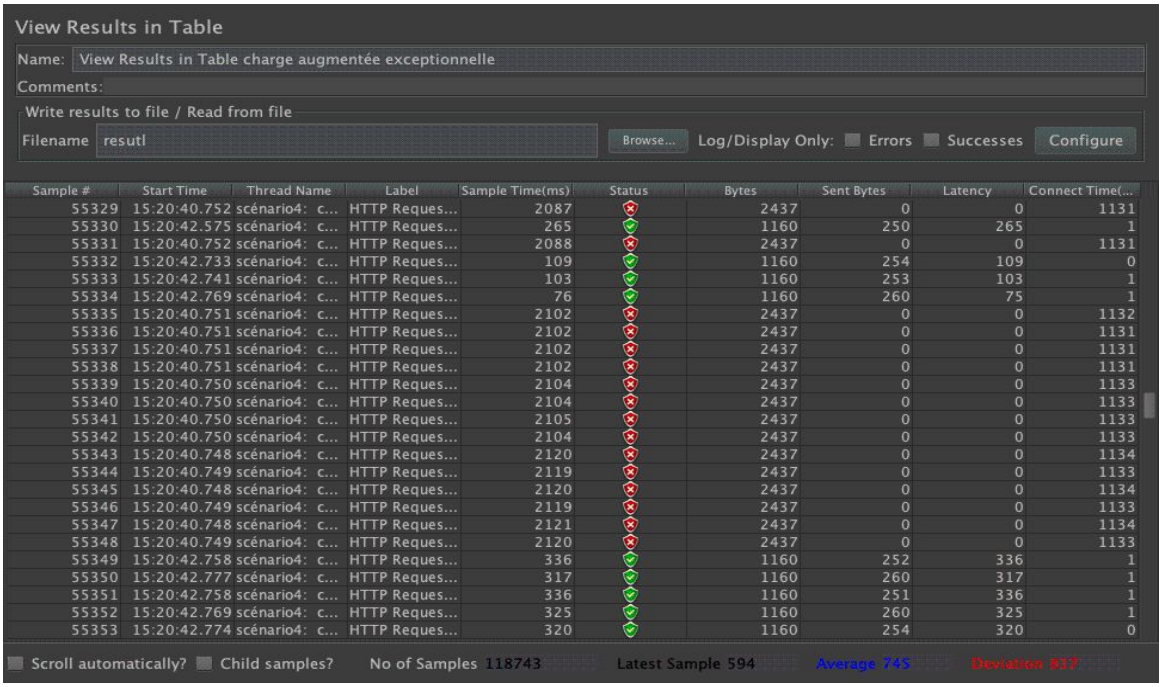


Pour le scénario 4 (charge augmentée exceptionnelle), le débit du notre serveur est de 18,228.193 / minute. Cela signifie que le serveur de Weka peut traiter à la fois 18228.193 requêtes par minute en utilisant l'algorithme LinearRegression. Cette valeur est très élevée, cependant la déviation est élevé aussi (une valeur de 682) donc nous pouvons confirmer que les performances sont plus ou moins satisfaisantes pour ce scénario en utilisant l'algorithme LinearRegression.

B. Les Limites de Weka

Le temps pris par les scénarios 1 et 2 pour exécuter les requêtes était entre 15 secondes et 30 secondes. Tandis que les scénarios 3 et 4 ont pris énormément de temps (entre 13 à 15 min) pour exécuter toutes les requêtes.

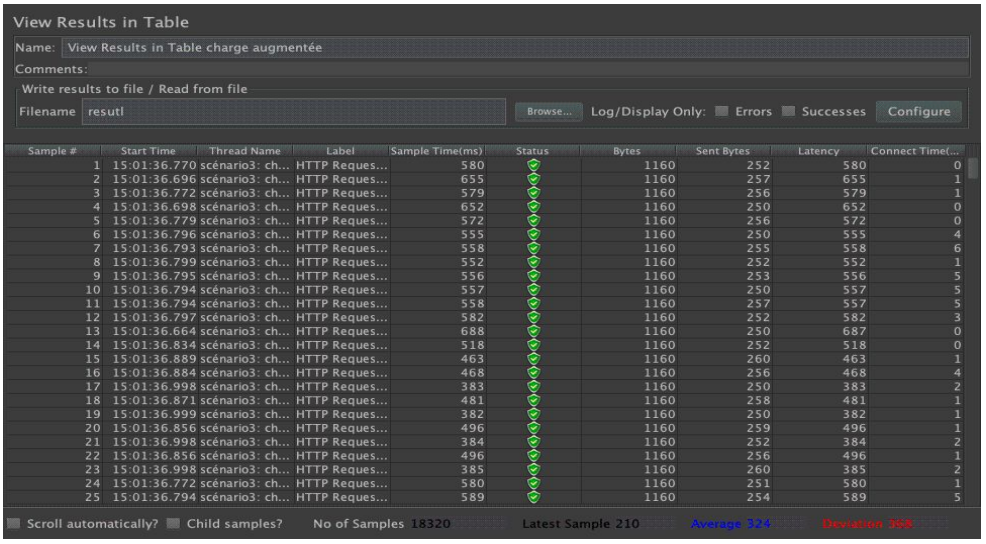
Résultats obtenus (scénario 4 : charge augmentée exceptionnelle)



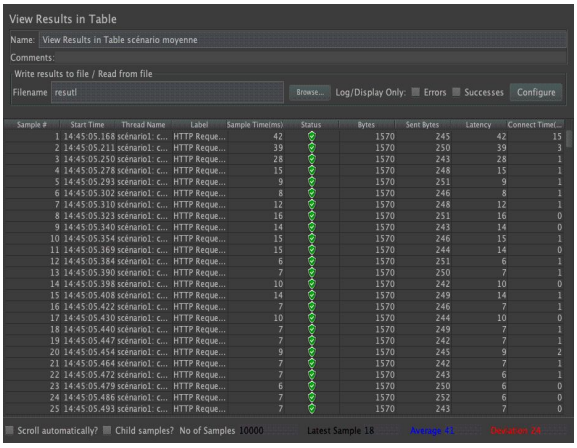
Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(...
55329	15:20:40.752	scénario4: c...	HTTP Reques...	2087		2437	0	0	1131
55330	15:20:42.575	scénario4: c...	HTTP Reques...	265		1160	250	265	1
55331	15:20:40.752	scénario4: c...	HTTP Reques...	2088		2437	0	0	1131
55332	15:20:42.733	scénario4: c...	HTTP Reques...	109		1160	254	109	0
55333	15:20:42.741	scénario4: c...	HTTP Reques...	103		1160	253	103	1
55334	15:20:42.769	scénario4: c...	HTTP Reques...	76		1160	260	75	1
55335	15:20:40.751	scénario4: c...	HTTP Reques...	2102		2437	0	0	1132
55336	15:20:40.751	scénario4: c...	HTTP Reques...	2102		2437	0	0	1131
55337	15:20:40.751	scénario4: c...	HTTP Reques...	2102		2437	0	0	1131
55338	15:20:40.751	scénario4: c...	HTTP Reques...	2102		2437	0	0	1131
55339	15:20:40.750	scénario4: c...	HTTP Reques...	2104		2437	0	0	1133
55340	15:20:40.750	scénario4: c...	HTTP Reques...	2104		2437	0	0	1133
55341	15:20:40.750	scénario4: c...	HTTP Reques...	2105		2437	0	0	1133
55342	15:20:40.750	scénario4: c...	HTTP Reques...	2104		2437	0	0	1133
55343	15:20:40.748	scénario4: c...	HTTP Reques...	2120		2437	0	0	1134
55344	15:20:40.749	scénario4: c...	HTTP Reques...	2119		2437	0	0	1133
55345	15:20:40.748	scénario4: c...	HTTP Reques...	2120		2437	0	0	1134
55346	15:20:40.749	scénario4: c...	HTTP Reques...	2119		2437	0	0	1133
55347	15:20:40.748	scénario4: c...	HTTP Reques...	2121		2437	0	0	1134
55348	15:20:40.749	scénario4: c...	HTTP Reques...	2120		2437	0	0	1133
55349	15:20:42.758	scénario4: c...	HTTP Reques...	336		1160	252	336	1
55350	15:20:42.777	scénario4: c...	HTTP Reques...	317		1160	260	317	1
55351	15:20:42.758	scénario4: c...	HTTP Reques...	336		1160	251	336	1
55352	15:20:42.769	scénario4: c...	HTTP Reques...	325		1160	260	325	1
55353	15:20:42.774	scénario4: c...	HTTP Reques...	320		1160	254	320	0

Comme le montre la figure ci-dessus, certaines requêtes envoyées par les utilisateurs ont échoué à cause de la grande demande sur le réseau, l'échec est dû à un overtime et la latence du réseau. La figure ci-dessus montre bien que toutes les requêtes dont la durée est supérieure à 2000 ms ont échoué.

Résultats obtenus (scénario 3 : charge augmentée)



Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(...
1	15:01:36.770	scénario3: ch...	HTTP Reques...	580		1160	252	580	0
2	15:01:36.696	scénario3: ch...	HTTP Reques...	655		1160	257	655	1
3	15:01:36.772	scénario3: ch...	HTTP Reques...	579		1160	256	579	1
4	15:01:36.698	scénario3: ch...	HTTP Reques...	652		1160	250	652	0
5	15:01:36.779	scénario3: ch...	HTTP Reques...	572		1160	256	572	0
6	15:01:36.796	scénario3: ch...	HTTP Reques...	555		1160	250	555	4
7	15:01:36.793	scénario3: ch...	HTTP Reques...	558		1160	255	558	6
8	15:01:36.799	scénario3: ch...	HTTP Reques...	552		1160	252	552	1
9	15:01:36.795	scénario3: ch...	HTTP Reques...	556		1160	253	556	5
10	15:01:36.794	scénario3: ch...	HTTP Reques...	557		1160	250	557	5
11	15:01:36.794	scénario3: ch...	HTTP Reques...	558		1160	257	557	5
12	15:01:36.797	scénario3: ch...	HTTP Reques...	582		1160	252	582	3
13	15:01:36.664	scénario3: ch...	HTTP Reques...	688		1160	250	687	0
14	15:01:36.834	scénario3: ch...	HTTP Reques...	518		1160	252	518	0
15	15:01:36.889	scénario3: ch...	HTTP Reques...	463		1160	260	463	1
16	15:01:36.884	scénario3: ch...	HTTP Reques...	468		1160	256	468	4
17	15:01:36.998	scénario3: ch...	HTTP Reques...	383		1160	250	383	2
18	15:01:36.871	scénario3: ch...	HTTP Reques...	481		1160	258	481	1
19	15:01:36.999	scénario3: ch...	HTTP Reques...	382		1160	250	382	1
20	15:01:36.856	scénario3: ch...	HTTP Reques...	496		1160	259	496	1
21	15:01:36.998	scénario3: ch...	HTTP Reques...	384		1160	252	384	2
22	15:01:36.856	scénario3: ch...	HTTP Reques...	496		1160	256	496	1
23	15:01:36.998	scénario3: ch...	HTTP Reques...	385		1160	260	385	2
24	15:01:36.772	scénario3: ch...	HTTP Reques...	580		1160	251	580	1
25	15:01:36.794	scénario3: ch...	HTTP Reques...	589		1160	254	589	5



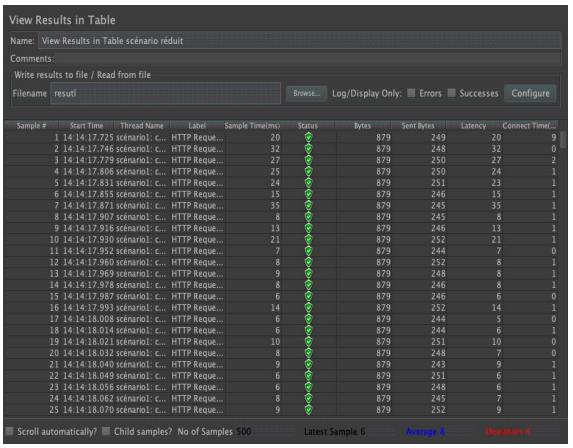


Figure 3: résultats obtenus (scénario 2) Figure 4: résultats obtenus (scénario 1)

Ces trois dernières figures montrent bien la divergence de la latence entre les scénarios à charge réduite et moyenne et celui à charge augmentée.