Bagging-RandomMiner - Un Algoritmo en MapReduce para Detección de Anomalías en Big Data

Luis Pereyra*, Diego García-Gil†, Francisco Herrera†, Luis C. González-Gurrola*, Jacinto Carrasco†, Miguel Angel Medina-Pérez‡ y Raúl Monroy‡

*Facultad de Ingeniería, Universidad Autónoma de Chihuahua,
Chihuahua, Chih. 31000, México
Email: {a261804, lcgonzalez}@uach.mx
†Departamento de Ciencias de la Computación e Inteligencia Artificial,
Universidad de Granada, Granada, España, 18071
Email: {djgarcia,herrera,jacintocc}@decsai.ugr.es
‡Tecnológico de Monterrey, Campus Estado de México,
Carretera al Lago de Guadalupe Km. 3.5, Atizapán de Zaragoza, Estado de México, 52926, México.
Email: {migue, raulm}@itesm.mx

***Index Terms*—Detección de Anomalías, Big Data, MapReduce, Bagging-RandomMiner, Apache Spark, Detección de Anomalías**

***Resumen*—La detección de anomalías es una tarea de interés en muchas aplicaciones del mundo real cuyo objetivo es identificar el comportamiento irregular de un proceso basado en el aprendizaje de cómo éste se comporta normalmente. En este trabajo, presentamos una versión MapReduce de una estrategia que ha demostrado ser exitosa para encontrar valores atípicos (anomalías) en conjuntos de datos de tamaño mediano. Esta estrategia, conocida como Bagging-RandomMiner, ahora ha sido extendida para aprovechar las bondades que Apache Spark ofrece y poder demostrar su utilidad en contextos de Big Data. Para evaluar su rendimiento, se ha utilizado el conjunto de datos PRIDE como Benchmark para identificar situaciones de estrés (anomalías) en un grupo de 23 sujetos. En este estudio comparamos Bagging-RandomMiner contra otra versión MapReduce del algoritmo que mejores resultados ha reportado sobre PRIDE, llamado OCKRA. Los experimentos indican que Bagging-RandomMiner supera claramente a OCKRA en al menos 6 % en el score AUC, más aún su tiempo de ejecución es al menos un orden de magnitud menor. Estos resultados soportan la idea de Bagging-RandomMiner como un algoritmo contendiente en tareas de Big Data. En este trabajo hacemos disponible el código de esta versión.**

I. Introducción

Hoy en día el mundo avanza implacablemente hacia la era de Big Data. Todos los días se generan quintillones de datos. La *International Data Corporation* (IDC) predijo en 2014 que para 2020, el universo digital sería diez veces más grande que en 2013[1]. La tecnología actual se ha visto superada por esta cantidad de datos. Esto ha provocado que el concepto de Big Data aparezca como un nuevo paradigma para permitir el almacenamiento y procesamiento de toda esta información. Dentro de los desafíos que presenta Big Data, hay uno de especial relevancia que ha encontrado un buen número de aplicaciones en el mundo real. La **detección de anomalías** se define como la tarea de averiguar cuándo un sistema o proceso está actuando erráticamente [1]. La importancia de esta tarea es tal que la existencia de un enfoque eficaz podría llevar a brindar soluciones a escenarios altamente sensibles, como seguridad en redes de computadoras, la detección de fraude bancario o incluso situaciones de vida o muerte [2], [3], [4].

Recientemente, se ha propuesto un algoritmo llamado Bagging-RandomMiner [5] para identificar anomalías o valores atípicos en conjuntos de datos de tamaño mediano. Motivados por la calidad de los resultados que ha registrado, incluso superando a conocidos competidores en tareas de Data Mining tales como AdaBoost.M1, Random Forrest y TreeBagger, entre otros, y su posible aplicación en escenarios de Big Data, presentamos en este artículo un diseño de MapReduce [6] de este algoritmo en el framework Apache Spark. Para evaluar este algoritmo utilizamos el conjunto de datos de Detección de Riesgos Personales (PRIDE) [7], que se creó para servir como benchmark para la detección de anomalías en el contexto de la detección de situaciones de estrés (o peligro) experimentadas por un grupo de 23 personas. Para contrastar los resultados obtenidos por Bagging-RandomMiner, también implementamos la versión MapReduce del algoritmo que mejores resultados ha reportado sobre PRIDE, conocido como OCKRA [8]. La comparación de ambas estrategias muestra que nuestro diseño supera los resultados del estado del arte en al menos $\sim 6\,\%$ en el score promedio de Área Bajo la Curva (AUC), lo cual ha sido validado por el test Bayesian Signed-Rank test. Así mismo su tiempo de ejecución es al menos un

[1]IDC: The Digital Universe of Opportunities. 2018 [Online] Disponible: http://www.emc.com/infographics/digital-universe-2014.htm

orden de magnitud menor
Ponemos a disposición de la comunidad esta implementación en el repositorio respectivo de Spark[2].

La organización de este trabajo se estructura de la siguiente manera: la Sección II presenta las definiciones básicas de la tarea de detección de anomalías, el conjunto de datos PRIDE y el modelo MapReduce. La Sección III describe el diseño del algoritmo distribuido propuesto. La Sección IV muestra los experimentos llevados a cabo sobre el conjunto de datos PRIDE. Finalmente, la Sección V brinda algunas conclusiones de este trabajo.

II. Antecedentes

II-A. Detección de Anomalías

En la tarea de Detección de Anomalías (también conocida como detección de valores atípicos) [1], a diferencia de la clasificación binaria (o multiclase), el clasificador aprende a reconocer solo una categoría de objeto, la que pertenece a la clase mayoritaria. En general, se supone que la clase mayoritaria representa regularidad, es decir, objetos que se expresan a menudo en el dominio del problema. Suponiendo que el clasificador es competente en esta tarea, es fácil ver que éste discriminará automáticamente un objeto que no parece pertenecer a la clase mayoritaria, es decir, una anomalía.
Teniendo este contexto se puede ver que la Detección de Anomalías es relevante en una gran cantidad de escenarios, por ejemplo, para identificar ataques en la web [2], rastrear errores de software [3] y detección de fraude [4], entre otros. En este estudio nos enfocaremos en la tarea de detección de anomalías usando como Benchmark el conjunto de datos conocido como PRIDE.

II-B. PRIDE y OCKRA

El conjunto de datos PRIDE fue diseñado originalmente por Barrera et al. [7] con el objetivo de ayudar a apoyar tareas de detección de anomalías en un escenario sensible de usuarios que experimentan situaciones estresantes. PRIDE fue creado con la ayuda de 23 usuarios que fueron supervisados durante una semana las 24 horas del día mientras realizaban sus actividades cotidianas. La recopilación de la información se logró a través de un sensor de la empresa Microsoft, llamado *Microsoft Band*. A partir de los datos generados por esta banda, se extrajeron 26 características. La Tabla I presenta las características consideradas en PRIDE para cada usuario. Para recolectar datos asociados a situaciones estresantes (condiciones de anomalía), los 23 usuarios fueron sometidos a diferentes escenarios de estrés que simulaban condiciones peligrosas que podrían enfrentarse en la vida real, por ejemplo, subiendo y bajando escaleras de un edificio, corriendo 16 metros a alta velocidad o una sesión de boxeo para simular una pelea, entre otras actividades [7]. En promedio hay 323,161 muestras por usuario y un total de 7,432,715 muestras para los 23 usuarios.

OCKRA a su vez, es el método que mejores resultados reporta usando la base de datos PRIDE, incluso superando a una Máquina de Vectores de Soporte para una sola clase (OC-SVM) [9]. El algoritmo *K-means One-Class with Randomly-projected features Algorithm* (OCKRA) [8] es un ensamble de 100 clasificadores de una sola clase, basado en múltiples proyecciones del conjunto de datos respecto a subconjuntos de datos aleatorios de características. OCKRA aplica k-means++ [10] a cada subconjunto de características para obtener una colección de centroides que representan la distribución de los datos. Para clasificar una nueva observación, cada clasificador determina una medida de similitud (0 – 1). Finalmente, se promedian las similitudes de cada clasificador y se obtiene la probabilidad de pertenencia a la clase mayoritaria.

Tabla I
Estructura del conjunto de datos PRIDE

Característica			Número
Acelerómetro Giroscópico	**Eje X**	$\bar{x}$	1
		s	2
	Eje Y	$\bar{x}$	3
		s	4
	Eje Z	$\bar{x}$	5
		s	6
Velocidad Angular	**Eje X**	$\bar{x}$	7
		s	8
	Eje Y	$\bar{x}$	9
		s	10
	Eje Z	$\bar{x}$	11
		s	12
Acelerómetro	**Eje X**	$\bar{x}$	13
		s	14
	Eje Y	$\bar{x}$	15
		s	16
	Eje Z	$\bar{x}$	17
		s	18
Ritmo Cardíaco			19
Temperatura de la piel			20
Pasos			21
Velocidad			22
UV			23
Δ Podómetro			24
Δ Distancia			25
Δ Calorías			26

II-C. Modelo MapReduce

MapReduce es un framework introducido por Google en 2003 [6]. Este modelo se compone de dos procedimientos para procesar datos de forma paralela: *Map* y *Reduce*. La operación de *Map* realiza una transformación a los datos de entrada, mientras que el método *Reduce* consiste en una operación de agregación. El flujo de trabajo de un programa MapReduce se compone de cuatro pasos: primero, el nodo maestro divide los datos de entrada y los distribuye en todos los nodos. En segundo lugar, la operación *Map* aplica una transformación a los datos presentes localmente. A continuación, los resultados se redistribuyen en el clúster en función de los pares clave-valor generados en la operación *Map*. Después de que este proceso haya finalizado, todos los pares que pertenecen a una misma clave están en el mismo nodo. Finalmente, los pares clave-valor se procesan en paralelo.
Apache Spark [3] es un framework de código abierto basado en

[2]https://spark-packages.org/package/wuicho-pereyra/Bagging-RandomMiner

[3]Apache Spark Project 2018 [Online] Disponible: https://spark.apache.org/

el modelo MapReduce construido para favorecer la eficiencia, la facilidad de uso y el cálculo en memoria interna [11], [12]. La característica central de Spark son los *Resilient Distributed Datasets* (RDDs) [13]. Los RDDs se pueden describir como una colección distribuida e inmutable de particiones de los datos, distribuidos por el clúster. Los RDDs implementan dos tipos de operaciones: *(i)* transformaciones, que no se evalúan cuando se definen y devuelven un nuevo RDD después de aplicar una función; y *(ii)* acciones, que devuelven un valor y desencadenan todas las transformaciones previas del RDD. Todas las transformaciones se aplican en paralelo a cada partición.

III. Una versión MapReduce de Bagging-RandomMiner

Para una descripción de la versión local del algoritmo Bagging-RandomMiner se sugiere al lector consultar el trabajo [5]. Bajo el modelo de MapReduce se usaron las siguientes funciones de Spark: (*i*) **mapPartitions** (aplica una función a cada partición de un RDD y devuelve un nuevo RDD), (*ii*) **zipWithIndex** (agrega a un RDD sus índices de elementos) y, (*iii*) **join** (devuelve un RDD que contiene todos los pares de elementos con claves coincidentes en éste y otro RDD). Bagging-RandomMiner está compuesto por dos fases: aprendizaje y clasificación. En la fase de aprendizaje, se selecciona de manera aleatoria un porcentaje RS de los datos. A partir de esta muestra, se realiza otro sub muestreo para seleccionar aquellos objetos que representarán la distribución completa de los objetos originales, los cuales son llamados **Objetos Más Representativos de la población (MRO)**. A partir de aquí, se calcula un umbral de decisión (δ) promediando la matriz de distancia de los MRO con la ayuda de la función *mapPartitions*. Esto se hace iterativamente para un número T de clasificadores. En la fase de clasificación, debido a que los RDD se distribuyen aleatoriamente de forma natural, la función *zipWithIndex* se aplica para agregar un índice a cada instancia. Luego, cada instancia de test se compara con los MRO, eligiendo solo el que tenga la distancia más cercana a la instancia de test. Finalmente, la similitud se calcula con la distancia más cercana y el umbral (δ), esto corresponde a la probabilidad del comportamiento genuino. Este proceso se repite hasta que se completen los T clasificadores, y, mediante la función *join*, se almacena el voto de cada clasificador, lo que nos permite promediar un resultado final. La Figura 1 muestra un diagrama del funcionamiento de ambas fases.

III-A. Fase de Aprendizaje

El Algoritmo 1 es la clase principal que orquesta la operación de Bagging-RandomMiner. Para iniciar el procedimiento, es necesario proporcionar el conjunto de entrenamiento ($dataTrain$) y el conjunto de prueba ($dataTest$) en la estructura RDD de tipo $LabeledPoint$, el número de clasificadores (T), el porcentaje de objetos seleccionados para cada clasificador (RS), el porcentaje de objetos más representativos ($MRO_{percent}$) y el tipo de distancia que se aplicará ($disType; 1 \rightarrow$ euclidea, $2 \rightarrow$ manhattan y $3 \rightarrow$ chebyshev). En el paso 5, un clasificador se entrena aplicando el Algoritmo 2, pasando los parámetros de $dataTrain$, RS y $MRO_{percent}$ como argumentos. En el paso 7, se obtiene una predicción del conjunto de test aplicando el Algoritmo 3, pasando como parámetros el conjunto de prueba y el modelo entrenado. Luego, cada vez que se hace una predicción del conjunto de test, la función de $union$ se usa para almacenar las predicciones de cada clasificador (paso 8). Finalmente, en el paso 10, se promedian los resultados del clasificador para obtener la probabilidad de pertenencia al comportamiento genuino de cada instancia de test, y se devuelve en una variable con estructura RDD [índice, probabilidad, clase real].

El Algoritmo 2 es la parte principal de la fase de aprendizaje. Aquí es donde se determinan los MRO y el umbral de decisión para la clasificación de un nuevo objeto. Los parámetros necesarios para comenzar con esta etapa son: $dataTrain$, RS, $MRO_{percent}$ y $disType$. Primero, debemos determinar los MRO del conjunto. Para lograr esto, en el paso 3 elegimos aleatoriamente el porcentaje de RS de los datos que serán usados por el clasificador para representar la distribución de los datos originales. Luego, en el paso 4, el porcentaje de MRO indicado se selecciona aleatoriamente, siendo estos objetos los prototipos que representan el clasificador. Ahora, para determinar el umbral de decisión (δ) es necesario calcular la matriz de distancias. Con la ayuda de la función $mapPartitions$, la matriz de distancias se calcula para cada nodo, es decir, en paralelo, cada nodo es responsable de calcular el promedio de la matriz de distancias con los MRO correspondientes. Al final, cada uno de los promedios se recopila y se calcula el promedio de estos resultados, que es el umbral de decisión δ (pasos 5-15). Finalmente, se crea un modelo y se devuelven los MRO, el umbral calculado δ y el tipo de distancia utilizada (paso 16).

III-B. Fase de predicción

El Algoritmo 3 es responsable de la clasificación de los nuevos objetos. Para lograr esto, necesitamos el $DataTest$ y el modelo entrenado por el Algoritmo 2. El proceso comienza aplicando la función $zipWithIndex$ (paso 3) que asigna un índice a cada instancia para no perder el orden y poder aplicar la función de unión en el algoritmo 1. Después, con la ayuda de la función $mapPartitions$, la distancia de cada objeto de test se calcula con todas las MRO distribuidos en cada nodo. Al final, se elige la distancia más cercana ($dmin$) al objeto de test (paso 10) y la medida de similitud se calcula con la $dmin$ obtenida y el umbral de decisión δ (paso 11). Cuando todos los nodos terminan sus respectivas ejecuciones, el Algoritmo 3 devuelve un RDD [índice, probabilidad] (paso 14).

IV. Resultados Experimentales

Los experimentos se diseñaron utilizando *5-fold cross-validation* y dos versiones de Bagging-RandomMiner, variando el número de clasificadores en el ensamble: 10 y 50, respectivamente. Este último aspecto es interesante de evaluar ya que OCKRA también es un método de ensamble con un número fijo de clasificadores igual a 100. En cuanto a los

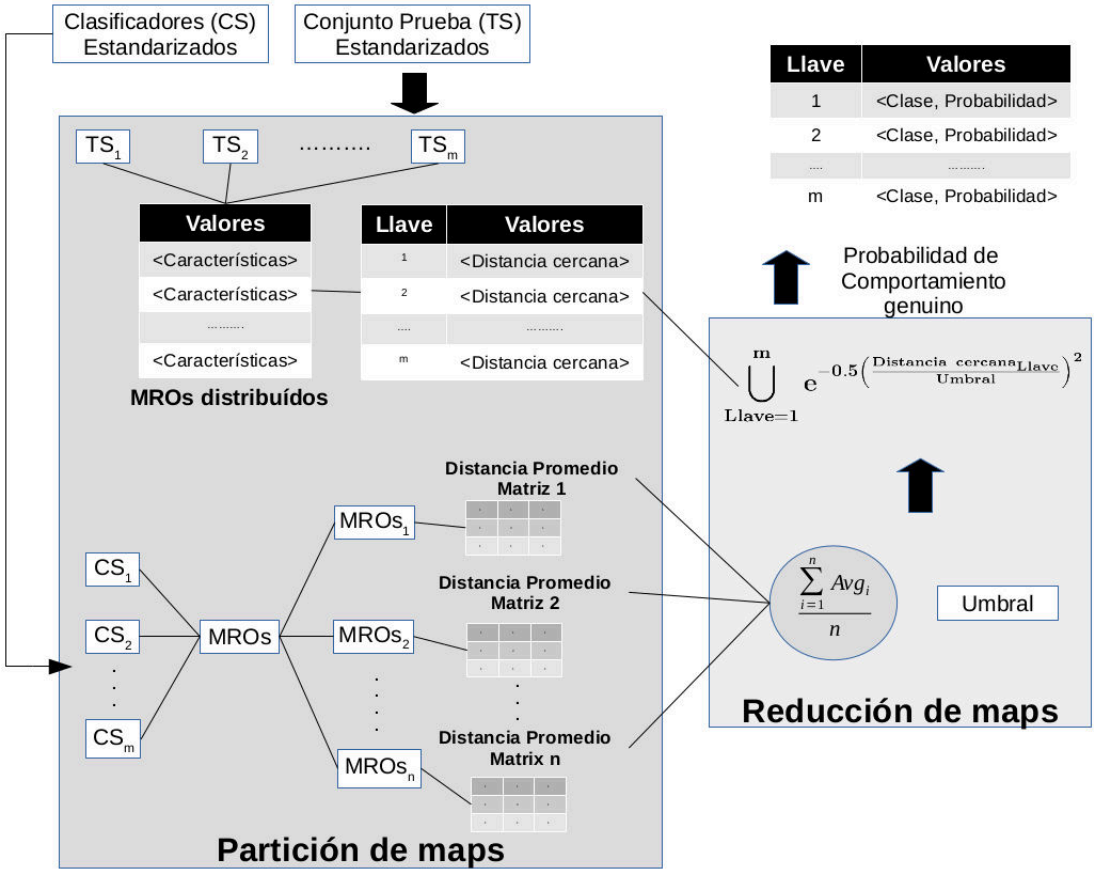

Figura 1. Diagrama de aprendizaje y predicción de Bagging-RandomMiner en MapReduce.

Algoritmo 1 Main Bagging-RandomMiner

Entrada: *datos:* RDD de tipo *LabeledPoint* (características, clase), *prueba:* RDD de tipo LabeledPoint (características, clase) *T:* número de clasificadores, *RS:* porcentaje de muestreo, $MRO_{percent}$: porcentaje de *MROs* y *distType:* tipo de distancia.
Salida: Probabilidad de pertenencia al comportamiento genuino.
for $i = 0...T$ **do** || Itera para crear un modelo de cada clasificador.
    || Se manda llamar el algoritmo 2.
    $Model \leftarrow$ Aprendizaje$(dataTrain, RS, MRO_{percent}, distType)$
    || Se manda llamar el algoritmo 3
    $pred\_tmp \leftarrow$ Predicción$(dataTest, Model)$
    $predicciones$.join$(pred_tmp)$ || Con la función *join*, las predicciones se almacenan con la llave índice.
end for
return Promedio$(predicciones)$ || Los resultados de cada modelo son promediados, esto corresponde a la probabilidad del comportamiento genuino

Algoritmo 2 Aprendizaje

Entrada: *dataTrain:* RDD de tipo LabeledPoint (características, clase), *RS:* porcentaje de muestreo, $MRO_{percent}$: porcentaje de *MROs* y *distType:* tipo de distancia.
Salida: El modelo entrenado, un objeto de clase RandomMiner
$dataTrain' \leftarrow$ Clasificador$(dataTrain, RS)$ || Un porcentaje de los datos es seleccionado aleatoriamente.
$MROs \leftarrow$ Seleccionar$(MRO_{percent},\ dataTrain')$ || Un porcentaje de MRO es seleccionado aleatoriamente.
mapPartition $l \in MROs$ || Computar la distancia de cada MRO.
    $Distance \leftarrow 0$ || Se inicializa la distancia.
    for $i = 0$ To $size(l)$ **do** || Itera sobre cada MRO de la partición.
        for $j = i + 1$ To $size(l)$ **do** || Itera sobre cada MRO de la partición.
            || Se calcula la distancia entre cada par de MRO
            $Distance \leftarrow Distance +$ CalcDist$(l(i),\ l(j),\ distType)$
        end for
    end for
    $umbral$.append(Avg$(Distance)$) || El promedio de cada partición.
end mapPartition
$\delta \leftarrow$ Avg$(umbral)$ || Promedio global de las particiones.
return $(MROs,\ \delta,\ distType)$ || Se retorna un modelo con los MRO, el umbral δ y el tipo de distancia.

parámetros de RandomMiner, se usaron la distancia chebyshev y los valores del $1\,\%$ para muestreo de los datos y $1\,\%$ de objetos MRO.

La métrica para evaluar los modelos es el área bajo la curva (AUC) de la tasa de detección positiva (comportamiento genuino) (TP) versus la tasa de detección de falso positivo (comportamiento anormal) (FP). Este indicador de acuerdo con [14] es una excelente medida para evaluar clasificadores

Algoritmo 3 Predicción

1: **Entrada:** *dataTest:* RDD de tipo *LabeledPoint* (características, clase), *Model:* modelo entrenado (*MRO, δ, distType*).
2: **Salida:** Probabilidad de pertenencia al comportamiento genuino.
3: $testIndex \leftarrow$ zipWithIndex$(dataTest)$ ‖ Un índice es añadido a cada instancia.
4: **mapPartition** $l \in testIndex$ ‖ Se evalua por partición el conjunto de prueba (maps)
5: **for** $i = 0$ To $size(l)$ **do** ‖ Itera sobre cada instancia de la partición.
6: **for** $j = 0$ To $size(MROs)$ **do** ‖ Itera sobre cada MRO de la partición.
7: ‖ Se calcula la distancia entre cada instancia de prueba y los MRO.
8: $Distance(j) \leftarrow$ CalcDist$(l(i),\ MRO(j),\ disType)$
9: **end for**
10: $dmin \leftarrow \min(Distance)$ ‖ Para cada instancia de prueba, se determina la más cercana a los MRO.
11: $similaridad \leftarrow e^{-0{,}5\left(\frac{dmin}{\delta}\right)^2}$ ‖ Se determina la similaridad de la distancia más cercana.
12: **end for**
13: **end mapPartition**
14: **return** $similaridad$ de tipo RDD ‖ Probabilidad del comportamiento genuino.

de una clase. Finalmente, parar tener mejor evidencia que la ofrecida por test frecuentistas [15], se aplicó la prueba estadística *Bayesian Signed Rank Test* [16] para comparar la métrica AUC de la mejor versión de Bagging-RandomMiner vs OCKRA.

La infraestructura de cómputo (clúster) utilizado para todos los experimentos en este trabajo está compuesto por 14 nodos gestionados por un nodo maestro. Todos los nodos tienen la misma configuración de $hardware$ y $software$. En cuanto al $hardware$, cada nodo tiene 2 procesadores $IntelXeonE5-2620$, 6 núcleos (12 hilos) por procesador, 2 GHz y 64 GB de RAM. La red utilizada es $Infiniband$ de $40Gb/s$. El sistema operativo es Cent OS 6.5, con Apache Spark y MLlib 2.2.0, 336 núcleos (24 núcleos / nodo), 728 GB de RAM (52 GB / nodo).
La Figura 2 muestra el porcentaje del score AUC para cada uno de los 23 sujetos que integran el dataset PRIDE obtenido por RandomMiner (2 configuraciones) y OCKRA. De acuerdo a esta figura se pueden obtener 2 conclusiones: (1) los resultados de RandomMiner mejoran la identificación de anomalías en prácticamente todos los usuarios, y (2) No hay una diferencia marcada entre usar 10 clasificadores o 50 para RandomMiner. Este último punto sugiere que el método es robusto.

Para analizar estadísticamente los resultados se aplicó el test no paramétrico *Bayesian Signed Rank* a un nivel de confianza del $95\,\%$. Este test, a diferencia de los test frecuentistas (t-test), nos permite ser más concluyentes respecto a las diferencias de los algoritmos comparados [15]. En la Figura 3 se muestra a través de coordinadas baricéntricas la distribución de diferencias en el desempeño de los algoritmos en tres regiones: $left$ (OCKRA es mejor que RandomMiner), $rope$ (no hay diferencia) y $right$ (RandomMiner es mejor que OCKRA). Observando esta figura es evidente que la mayoría de los casos se inclina más a RandomMiner que a la región de no diferencia y definitivamente más que a soportar la probabilidad de que OCKRA sea el mejor algoritmo.

La Figura 4 muestra el tiempo de ejecución total para las dos versiones de RandomMiner, 10 y 50 clasificadores con 256 $maps$ para cada ejecución y OCKRA. Ambas versiones de RandomMiner tienen un mejor tiempo de ejecución que OCKRA, ahora, si consideramos que usando 10 clasificadores en RandomMiner es suficiente para obtener resultados competitivos esta diferencia se vuelve sustancial.

Dado que la contribución de este artículo es un nuevo diseño escalado de Bagging-RandomMiner, es importante contextualizar el tiempo de ejecución con respecto a la versión original del mismo algoritmo, llamado Local, esto se puede observar en la Figura 5. La escalabilidad de Bagging-RandomMiner es notable, ya que el tiempo de ejecución del algoritmo por usuario pasó de 14.4 horas a 3.66 minutos en promedio.

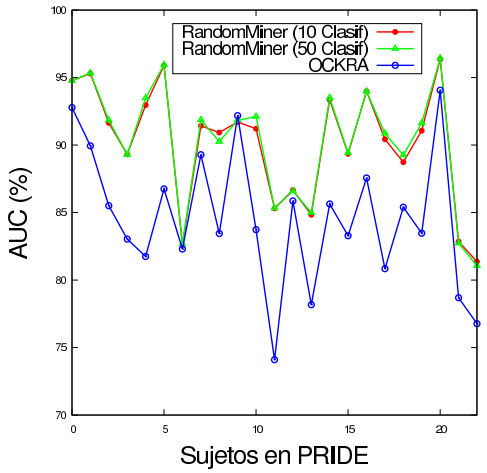

Figura 2. Porcentaje de Área bajo la curva (AUC) para RandomMiner (2 configuraciones) y OCKRA para cada uno de los 23 sujetos en PRIDE

V. Conclusiones

En este trabajo, se ha propuesto un diseño escalable y distribuido del algoritmo Bagging-RandomMiner desde la perspectiva MapReduce, basada en la filosofía de ensambles y aleatoriedad, que permite abordar problemas con alta dimensionalidad en el área de clasificación de una sola clase, para la detección de anomalías y conjuntos de datos altamente desequilibrados. Los resultados sugieren que Bagging-RandomMiner supera a OCKRA, la mejor estrategia para la detección de anomalías en el conjunto de datos PRIDE. Además, el método parece ser robusto con respecto al número de clasificadores que considera, ya que con un número muy pequeño de clasificadores (10) se logra un alto rendimiento y mejora sustancialmente el tiempo de ejecución.

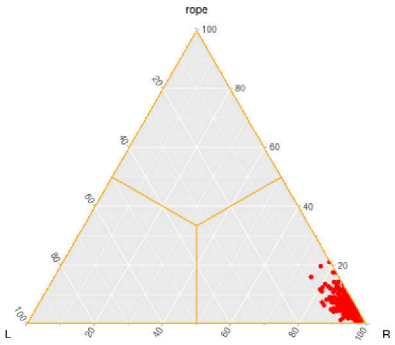

Figura 3. Resultados del Bayesian Signed Rank test sobre los resultados de OCKRA y Bagging-RandomMiner. La aglutinación de datos en la esquina inferior derecha (asociada a RandomMiner) sugiere fuertemente que éste es mejor algoritmo que OCKRA.

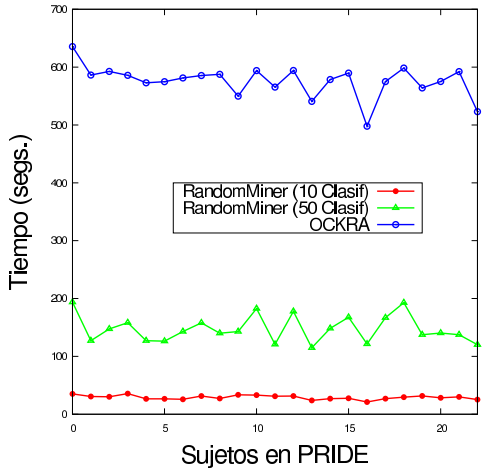

Figura 4. Tiempo de ejecución de los algoritmos RandomMiner (2 configuraciones) y OCKRA para cada uno de los 23 sujetos en PRIDE

Agradecimientos: Se agradece al Consejo Nacional de Ciencia y Tecnología de México (CONACYT) por la beca 620097 para estudios de posgrado otorgada al primer autor.

Referencias

[1] B. Krawczyk and B. Cyganek, "Selecting locally specialised classifiers for one-class classification ensembles," *Pattern Anal. Appl.*, vol. 20, no. 2, pp. 427–439, May 2017. [Online]. Available: https://doi.org/10.1007/s10044-015-0505-z

[2] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 251–261. [Online]. Available: http://doi.acm.org/10.1145/948109.948144

[3] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 291–301. [Online]. Available: http://doi.acm.org/10.1145/581339.581377

[4] T. Fawcett and F. Provost, "Adaptive fraud detection," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 291–316, Sep 1997. [Online]. Available: https://doi.org/10.1023/A:1009700419189

[5] J. B. Camiña, M. A. Medina-Pérez, R. Monroy, O. Loyola-González, L. A. P. Villanueva, and L. C. González-Gurrola, "Bagging-randomminer: a one-class classifier for file access-based masquerade detection," *Machine Vision and Applications*, Jul 2018. [Online]. Available: https://doi.org/10.1007/s00138-018-0957-4

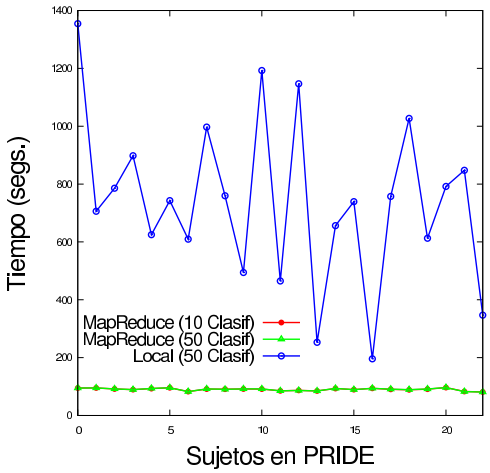

Figura 5. Tiempo de ejecución de los algoritmos RandomMiner en versión MapReduce y local.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04, 2004, pp. 10–10.

[7] A. Y. Barrera-Animas, L. A. Trejo, S. García, M. A. Medina-Pérez, and R. Monroy, "Online Personal Risk Detection Based on Behavioural and Physiological Patterns," *Information Sciences*, August 2016.

[8] J. Rodríguez, A. Y. Barrera-Animas, L. A. Trejo, M. A. Medina-Pérez, and R. Monroy, "Ensemble of one-class classifiers for personal risk detection based on wearable sensor data," *Sensors*, vol. 16, no. 10, 2016.

[9] L. Trejo and A. Barrera-Animas, "Towards and efficient one-class classifier for mobile devices and wearable sensors on the context of personal risk detection," *Sensors*, vol. 18, no. 9, 2018.

[10] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. [Online]. Available: http://dl.acm.org/citation.cfm?id=1283383.1283494

[11] D. García-Gil, S. Ramírez-Gallego, S. García, and F. Herrera, "A comparison on scalability for batch big data processing on Apache Spark and Apache Flink," *Big Data Analytics*, vol. 2, no. 1, p. 11, Mar 2017.

[12] D. García-Gil, S. Ramírez-Gallego, S. García, and F. Herrera, "Principal components analysis random discretization ensemble for big data," *Knowledge-Based Systems*, vol. 150, pp. 166 – 174, 2018.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[14] T. Fawcett, "An introduction to roc analysis," *Pattern Recogn. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.patrec.2005.10.010

[15] A. Benavoli, G. Corani, J. Demsar, and M. Zaffalon, "Time for a change: a tutorial for comparing multiple classifiers through bayesian analysis," *Journal of Machine Learning Research*, vol. 18, pp. 1–36, 2017.

[16] J. Carrasco, S. García, M. del Mar Rueda, and F. Herrera, "rnpbst: An r package covering non-parametric and bayesian statistical tests," in *Hybrid Artificial Intelligent Systems*, F. J. Martínez de Pisón, R. Urraca, H. Quintián, and E. Corchado, Eds. Cham: Springer International Publishing, 2017, pp. 281–292.