II Workshop en Big Data y Análisis de Datos Escalable (II BigDADE)

SESIÓN 2

Smart Data: Filtrado de Ruido para Big Data

Diego García-Gil*, Julián Luengo*, Salvador García* y Francisco Herrera*
*Departamento de Ciencias de la Computación e Inteligencia Artificial, Universidad de Granada, Granada, España, 18071
Email: {djgarcia,julianlm,salvagl,herrera}@decsai.ugr.es

***Resumen*—En cualquier proceso de minería de datos, el valor del conocimiento extraído está directamente relacionado con la calidad de los datos utilizados. Los problemas Big Data, generados por el crecimiento masivo de los datos en los últimos años, siguen el mismo dictado. Un problema común que afecta a la calidad de los datos es la presencia de ruido, especialmente en los problemas de clasificación, en los que el ruido de clase se refiere al etiquetado incorrecto de las instancias de entrenamiento. Sin embargo, en la era del Big Data, las grandes cantidades de datos plantean un reto a las propuestas tradicionales creadas para hacer frente al ruido, ya que tienen dificultades para procesar tal cantidad de datos. Es necesario proponer nuevos algoritmos para el tratamiento de ruido en Big Data, obteniendo datos limpios y de calidad, también conocidos como Smart Data. En este trabajo se proponen dos enfoques de preprocesamiento de datos en Big Data para eliminar instancias ruidosas: un ensemble homogéneo y un ensemble heterogéneo, con especial énfasis en su escalabilidad y rendimiento. Los resultados obtenidos muestran que estas propuestas permiten obtener eficientemente un Smart Dataset a partir de cualquier problema de clasificación Big Data.**

***Index Terms*—Big Data, Smart Data, Clasificación, Preprocesamiento, Ruido de clase**

I. Preliminares

Hoy en día estamos rodeados por enormes cantidades de datos. Está previsto que para 2020 el universo digital sea 10 veces más grande que en 2013, totalizando unos asombrosos 44 zettabytes [1]. El volumen actual de datos ha superado las capacidades de procesamiento de los sistemas clásicos de minería de datos y ha creado la necesidad de nuevos frameworks para almacenarlos y procesarlos. Es un hecho ampliamente aceptado que hemos entrado en la era del Big Data. El Big Data es el conjunto de tecnologías que hacen posible el procesamiento de grandes cantidades de datos.

En Big Data, las técnicas preprocesamiento clásicas para mejorar la calidad de los datos consumen aún más tiempo y exigen más recursos, volviéndose inviables. La falta de técnicas de preprocesamiento eficientes y asequibles implica que las alteraciones en los datos afectarán a los modelos extraídos de ellos. Entre todos los problemas que pueden aparecer en los datos, la presencia de *ruido* es uno de los más frecuentes. El ruido puede definirse como la alteración parcial o total de la información recogida en una instancia causada por un factor externo. El ruido conduce a modelos excesivamente complejos con un rendimiento degradado.

Recientemente, se ha introducido el concepto Smart Data (centrado en la veracidad y el valor de los datos), con el objetivo de filtrar el ruido y resaltar los datos valiosos. Hay tres atributos clave para que los datos sean *inteligentes*, deben ser:

- Precisos: los datos deben ser lo que dicen que son con suficiente precisión para generar valor. La calidad de los datos es importante.
- Procesables: los datos deben ser escalables inmediatamente para maximizar objetivos empresariales.
- Rápidos: los datos deben estar disponibles en tiempo real y listos para adaptarse a las cambiantes exigencias empresariales.

El modelado y análisis avanzado en Big Data es indispensable para descubrir la estructura subyacente en los datos con el fin de obtener Smart Data. En este trabajo aportamos varias técnicas de preprocesamiento para Big Data, transformando conjuntos de datos sin procesar en Smart Data. Nos hemos centrado en la tarea de clasificación, donde se distinguen dos tipos de ruido: *ruido de clase*, cuando afecta a la etiqueta de una instancia, y *ruido de atributo*, cuando afecta a los atributos. El primero es conocido por ser el más perjudicial [1]. Como consecuencia, muchos trabajos, incluido este, han sido dedicados a resolver este problema, o al menos a minimizar sus efectos (en [2] encontramos una completa revisión).

A pesar de que se han propuesto algunos diseños para tratar el ruido en Big Data [3], no se ha propuesto ningún algoritmo para eliminar el ruido ni se ha llevado a cabo ninguna comparación en Big Data.

Por esto proponemos un framework para eliminar instancias ruidosas en Big Data sobre Apache Spark, compuesto de dos algoritmos basados en ensembles de clasificadores. El primero es un Ensemble Homogéneo (EHO), el cual usa un clasificador base (Random Forest [4]) sobre un dataset particionado. El segundo es un Ensemble Heterogéneo (EHT), el cual usa tres clasificadores para identificar instancias ruidosas: Random Forest, Regresión Logística y k-Nearest Neighbors (kNN). Para una comparativa más completa, hemos implementado un filtro basado en similaridad de instancias, llamado Edited Nearest Neighbor (ENN). ENN examina los vecinos más cercanos de cada ejemplo de entrenamiento y elimina aquellos cuyo vecindario pertenece a una clase diferente. Todas estas técnicas han sido implementadas en Apache Spark [5], y pueden ser descargadas de su repositorio comunitario [2].

Para mostrar el rendimiento de los métodos propuestos, hemos llevado a cabo experimentos con cuatro grandes datasets: *SUSY*, *HIGGS*, *Epsilon* y *ECBDL14*. Hemos creado diferentes niveles de ruido de clase para evaluar los efectos de aplicar los

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

[2]https://spark-packages.org/package/djgarcia/NoiseFramework

métodos, así como la mejora obtenida en términos de precisión para dos clasificadores: un árbol de decisión y kNN. También mostramos que, en los problemas estudiados, los clasificadores se benefician del filtrado de ruido incluso cuando no hay ruido añadido, ya que los problemas Big Data contienen ruido propio debido a la automatización en la recogida de los datos. Los resultados indican que la propuesta es capaz de eliminar exitosamente el ruido. En concreto, EHO es la primera técnica válida para tratar el ruido en problemas Big Data, mejorando la precisión con un bajo coste computacional.

El resto del trabajo está organizado como sigue: la Sección II explica el framework propuesto. La Sección III describe los experimentos llevados a cabo para probar el rendimiento del framework. Finalmente, la Sección IV concluye el trabajo.

II. Filtrado de Ruido en Big Data

En esta sección, presentamos el primer framework para Big Data sobre Apache Spark para eliminar instancias ruidosas basado en el modelo MapReduce. Se trata de un diseño MapReduce donde todos los procesos de filtrado de ruido se realizan de forma distribuida. En la Sección II-A describimos el modelo MapReduce. En la Sección II-B listamos las primitivas de Spark usadas en la implementación del framework. Hemos diseñado dos algoritmos basados en ensembles. Ambos realizan un k-fold a los datos de entrada, aprenden un modelo en cada partición de entrenamiento y limpian las instancias ruidosas de la partición de test. El primero es un ensemble homogéneo que utiliza Random Forest como clasificador, llamado EHO (Sección II-C). El segundo, llamado EHT (Sección II-D), es un ensemble heterogéneo basado en tres clasificadores: Random Forest, Regresión Logística y kNN.

II-A. Modelo MapReduce

MapReduce es un framework diseñado por Google en 2003 [6], [7]. Este modelo se compone de una función Map, que realiza una transformación, y un método Reduce, que realiza una agregación. El flujo de trabajo de una tarea MapReduce es el siguiente: el nodo máster divide el conjunto de datos y lo distribuye por el clúster. Después cada nodo aplica la función Map a sus datos. A continuación, los datos se redistribuyen por el clúster en función de los pares clave-valor generados en la fase Map. Una vez que todos los pares de una misma clave están en el mismo nodo, se procesan en paralelo.

Apache Hadoop[3] es el framework open-source más conocido para el almacenamiento y procesamiento de grandes cantidades de datos, basado en el modelo MapReduce. El framework está diseñado para tratar los errores de hardware automáticamente. A pesar de su popularidad y rendimiento, Hadoop tiene algunas limitaciones [8]: bajo rendimiento para tareas iterativas e imposibilidad de trabajar en memoria.

Apache Spark[4] es un framework open-source centrado en la velocidad, facilidad de uso y procesamiento en memoria [9]. Spark está construido sobre una estructura de datos distribuida e inmutable llamada Resilient Distributed Datasets (RDDs) [10]. Los RDDs se pueden describir como un conjunto de particiones de datos distribuidas por el clúster. Los RDDS soportan dos tipos de operaciones: acciones, que evalúan y devuelven un valor. Y transformaciones, que no son evaluadas cuando se definen, se aplican sobre un RDD y producen un nuevo RDD. Cuando se llama a una acción sobre un RDD, todas las transformaciones previas se aplican en paralelo a cada partición del RDD.

[3] Apache Hadoop Project 2018 [Online] https://hadoop.apache.org/

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

II-B. Primitivas Spark

Para la implementación del framework, hemos utilizado algunas primitivas de la API de Spark. Estas funciones ofrecen operaciones más complejas al extender el modelo MapReduce. A continuación, describimos las empleadas en los algoritmos:

- map: Aplica una transformación a cada elemento de un RDD.
- $zipWithIndex$: Para cada elemento de un RDD crea una tupla con el elemento y su índice, empezando en 0.
- $join$: Devuelve un RDD que contiene todos los pares de elementos cuyas claves coinciden entre dos RDDs.
- $filter$: Devuelve un nuevo RDD que contiene solo los elementos que satisfacen una condición.
- $union$: Devuelve un RDD resultante de la unión de dos RDDs.
- $kFold$: Devuelve una lista de k pares de RDDs, siendo el primer elemento los datos de $train$, y el segundo elemento los datos de $test$. Donde k es el número de particiones.
- $randomForest$: Método para aprender un modelo de Random Forest.
- $predict$: Devuelve un RDD que contiene las instancias y clases predichas para un dataset usando un modelo.
- $learnClassifiers$: A pesar de no ser una primitiva de Spark, la usamos para simplificar la descripción de los algoritmos. Esta primitiva aprende un modelo de Random Forest, Regresión Logística y 1NN a partir de los datos de entrada.

Estas primitivas de Spark se usan en las siguiente secciones donde describimos EHO y EHT.

II-C. Ensemble Homogéneo: EHO

El ensemble homogéneo está inspirado en Cross-Validated Committees Filter (CVCF) [11]. Este filtro elimina instancias ruidosas realizando un k-fold a los datos. Después entrena un árbol de decisión k veces, dejando fuera uno de los subconjuntos cada vez. Esto da como resultado k clasificadores que se utilizan para predecir los datos de entrenamiento k veces. Luego, utilizando una estrategia de voto, se eliminan las instancias mal clasificadas.

EHO también está basado en un esquema de particionamiento de los datos. Hay una diferencia importante con respecto a CVCF: el uso de Random Forest en lugar de un árbol de decisión como clasificador. CVCF crea un ensemble particionando los datos de entrenamiento. EHO también divide

Figura 1. Algoritmo EHO

1: **Entrada:** *data* RDD de tuplas (clase, atributos)
2: **Entrada:** *P* número de particiones
3: **Entrada:** *nTrees* número de árboles para Random Forest
4: **Salida:** RDD filtrado sin ruido
5: *partitions* ← *kFold(data, P)*
6: *filteredData* ← ∅
7: **for all** *train, test* in *partitions* **do**
8: *rfModel* ← *randomForest(train, nTrees)*
9: *rfPred* ← *predict(rfModel, test)*
10: *joinedData* ← *join(zipWithIndex(test), zipWithIndex(rfPred))*
11: *markedData* ←
12: **map** *original, prediction* ∈ *joinedData*
13: **if** *label(original) = label(prediction)* **then**
14: *original*
15: **else**
16: *(label = ∅, features(original))*
17: **end if**
18: **end map**
19: *filteredData* ← *union(filteredData, markedData)*
20: **end for**
21: *return (filter(filteredData, label ≠ ∅))*

los datos de entrenamiento, pero el uso de Random Forest nos permite mejorar el paso de la votación:

- CVCF predice el dataset entero k veces. EHO solo predice las instancias de la partición $test$ del k-fold. Este paso se repite k veces. Con este cambio no solo mejoramos el rendimiento, sino también el tiempo de cómputo, ya que solo tiene que predecir una pequeña parte de los datos de entrenamiento en cada iteración.
- No necesitamos implementar una estrategia de votación, la decisión de si una instancia es ruidosa está asociada con la predicción del Random Forest.

La Figura 1 describe el proceso de filtrado de ruido en EHO. Los parámetros de entrada son los siguientes: el dataset ($data$), el número de particiones (P) y el número de árboles para el Random Forest ($nTrees$):

- El algoritmo realiza un $kFold$ en los datos de entrada. Como se definió antes, la función $kFold$ de Spark devuelve una lista de $(train, test)$ para un P dado.
- Iteramos sobre cada partición, aprendiendo un Random Forest usando $train$ como datos de entrada, y prediciendo $test$ usando el modelo aprendido.
- Para unir los datos de $test$ y los datos predichos para comparar las clases, usamos la operación $zipWithIndex$ en ambos RDDs. Con esta operación, añadimos un índice a cada elemento de ambos RDDs. Este índice se utiliza como clave para la operación $join$.
- El siguiente paso es aplicar una función map al RDD anterior y comprobar, para cada instancia, la clase original y la predicha. Si la clase predicha y el original son diferentes, la instancia se marca como ruido y se devuelven sus atributos y la clase (línea 16).
- El resultado de la función map anterior es un RDD donde las instancias ruidosas están marcadas. Estas instancias son finalmente eliminadas usando una función $filter$ y el conjunto de datos resultante es devuelto.

Figura 2. Algoritmo EHT

1: **Entrada:** *data* RDD de tuplas (clase, atributos)
2: **Entrada:** *P* número de particiones
3: **Entrada:** *nTrees* número de árboles para Random Forest
4: **Entrada:** *vote* estrategia de voto (mayoría o consenso)
5: **Salida:** RDD filtrado sin ruido
6: *partitions* ← *kFold(data, P)*
7: *filteredData* ← ∅
8: **for all** *train, test* in *partitions* **do**
9: *classifiersModel* ← *learnClassifiers(train, nTrees)*
10: *predictions* ← *predict(classifiersModel, test)*
11: *joinedData* ← *join(zipWithIndex(predictions), zipWithIndex(test))*
12: *markedData* ←
13: **map** *rf, lr, knn, orig* ∈ *joinedData*
14: *count* ← *0*
15: **if** *rf* ≠ *label(orig)* **then** *count* ← *count + 1*
16: **if** *lr* ≠ *label(orig)* **then** *count* ← *count + 1*
17: **if** *knn* ≠ *label(orig)* **then** *count* ← *count + 1*
18: **if** *vote = majority* **then**
19: **if** *count* ≥ 2 **then** *(label = ∅, features(orig))*
20: **if** *count* < 2 **then** *orig*
21: **else**
22: **if** *count = 3* **then** *(label = ∅, features(orig))*
23: **if** *count* ≠ *3* **then** *orig*
24: **end if**
25: **end map**
26: *filteredData* ← *union(filteredData, markedData)*
27: **end for**
28: *return (filter(filteredData, label ≠ ∅))*

II-D. Ensemble Heterogéneo: EHT

El ensemble heterogéneo está inspirado en Ensemble Filter (EF) [12]. Este filtro de ruido usa tres algoritmos de aprendizaje para identificar instancias ruidosas: un árbol de decisión, kNN y un clasificador lineal. Realiza un k-fold sobre los datos y para cada una de las k particiones, aprende tres modelos en las otras $k-1$ particiones. Cada uno de los clasificadores da una etiqueta a los ejemplos de $test$. Finalmente, usando una votación, se eliminan las instancias ruidosas.

EHT sigue el mismo esquema que EF. La principal diferencia es la elección de los tres algoritmos de aprendizaje: en lugar de un árbol de decisión, usamos Random Forest. Como clasificador lineal se ha usado la Regresión Logística. Finalmente, kNN se ha mantenido como clasificador.

En la Figura 2 se describe el filtrado de ruido en EHT. Los parámetros de entrada son los siguientes: el dataset ($data$), el número de particiones (P), el número de árboles para Random Forest ($nTrees$) y la estrategia de voto ($vote$):

- Para cada partición de $train$ y $test$ del k-fold, se aprenden tres modelos: Random Forest, Regresión Logística y 1NN usando $train$ como datos de entrada.
- Después se predicen los datos de $test$ usando los tres modelos aprendidos. Esto crea un RDD de tripletas (rf, lr, knn) con la predicción de cada algoritmo para cada instancia.
- Las predicción y los datos de $test$ se unen por índice para poder comparar las predicciones con la etiqueta original.
- Se comparan las tres predicciones de cada instancia con la etiqueta original usando una función map y, dependiendo de la estrategia de voto, las instancias se marcan como ruidosas o limpias.
- Las instancias marcadas como ruidosas son eliminadas usando la función $filter$ y el dataset es devuelto.

III. Resultados Experimentales

Esta sección describe los experimentos llevados a cabo sobre cuatro problemas Big Data. En la Sección III-A mostramos los detalles de los datasets y parámetros usados por los filtros de ruido. Los estudios de precisión e instancias eliminadas se encuentran en la Sección III-B. Finalmente, la Sección III-C analiza los tiempos de cómputo de las propuestas.

III-A. Framework Experimental

En nuestros experimentos hemos utilizado cuatro datasets de clasificación:

- SUSY, compuesto por 5,000,000 de instancias y 18 atributos. El objetivo es distinguir entre una señal que produce partículas supersimétricas (SUSY) y una que no.
- HIGGS, formado por 11,000,000 de instancias y 28 atributos. Este dataset es un problema de clasificación en el que se distingue una señal que produce bosones de Higgs y otra que no.
- Epsilon, con 500,000 instancias y 2,000 atributos. Este dataset fue creado artificialmente para la Pascal Large Scale Learning Challenge en 2008.
- ECBDL14, con 32 millones de instancias y 631 atributos. Este dataset fue usado como referencia en la competición de machine learning del Evolutionary Computation for Big Data and Big Learning llevado a cabo el 14 de Julio, 2014, bajo la conferencia internacional GECCO-2014. Es un problema de clasificación binaria donde la distribución de clases está altamente desbalanceada: 98 % de instancias negativas. Para este problema, usamos una versión reducida con 1,000,000 de instancias y 30 % de instancias positivas.

Hemos llevado a cabo experimentos en cinco niveles de ruido: en cada uno, un porcentaje de las instancias de entrenamiento han sido alteradas, sustituyendo su clase por otra de las disponibles. Los niveles de ruido elegidos son 0 %, 5 %, 10 %, 15 % y 20 %. Un nivel de 0 % de ruido indica que el dataset no ha sido alterado. Dadas las limitaciones del algoritmo kNN, hemos realizado una validación hold-out.

Hemos elegido 4 particiones para EHO y EHT. Para el filtro heterogéneo, hemos usado dos estrategias de voto: *mayoría* (mismo resultado para al menos la mitad de los clasificadores) y *consenso* (mismo resultado para todos los clasificadores). Para ENN, con *k = 5* tenemos una carga de red mayor, y dada la alta redundancia de datos en Big Data, no hemos apreciado diferencia con respecto a *k = 1*. Por esto hemos elegido la opción más eficiente.

Tabla I
Precisión en test de kNN. Los valores más altos de cada dataset y nivel ruido se muestran en negrita

Dataset	Ruido	Original	EHO	EHT Mayor.	EHT Consen.	ENN
SUSY	0 %	71,79	**78,73**	77,86	74,64	72,02
	5 %	69,62	**78,68**	77,68	73,38	69,84
	10 %	67,44	**78,63**	77,44	72,01	67,66
	15 %	65,27	**78,62**	77,19	70,52	65,28
	20 %	63,10	**78,56**	76,93	69,10	63,25
HIGGS	0 %	61,21	**64,26**	63,94	62,30	60,65
	5 %	60,10	**64,06**	63,63	61,45	59,60
	10 %	58,97	**63,83**	63,29	60,65	58,56
	15 %	57,84	**63,65**	62,86	59,81	57,52
	20 %	56,69	**63,53**	62,55	58,89	56,45
Epsilon	0 %	56,55	**58,11**	57,43	55,19	56,21
	5 %	55,71	**58,64**	57,47	55,47	55,43
	10 %	55,20	**58,51**	57,26	55,25	54,79
	15 %	54,54	**58,39**	57,00	55,00	54,30
	20 %	54,05	**58,02**	56,75	54,72	53,68
ECBDL14	0 %	74,83	**76,06**	75,12	73,54	73,94
	5 %	72,36	**75,60**	74,59	72,89	72,77
	10 %	69,86	**75,31**	74,19	72,50	71,40
	15 %	67,39	**75,11**	73,99	72,11	69,68
	20 %	64,90	**74,82**	73,70	71,89	67,64

Utilizamos dos clasificadores para probar la efectividad de los filtros de ruido, un árbol de decisión y kNN. El árbol de decisión puede adaptar su profundidad para detectar mejor las instancias ruidosas, mientras que kNN es sensible al ruido cuando el número de vecinos el bajo. Para evaluar el rendimiento de los modelos usamos la precisión (número de instancias correctamente clasificadas divido por el total de instancias). Se ha aumentado la profundidad del árbol de decisión ($depth = 20$) para mejorar la detección de ruido. Para kNN usamos *k = 1* ya que es más sensible al ruido.

Los experimentos se han realizado en un clúster compuesto por 20 nodos de cómputo con la siguiente configuración: 2 x Intel Xeon E5-2620, 6 núcleos, 2.00 GHz, 2 TB HDD, 64 GB RAM. Hemos utilizado la siguiente configuración software: Hadoop 2.6.0-cdh5.4.3 de la distribución open source de Apache Hadoop de Cloudera, Apache Spark y MLlib 2.2.0, 460 núcleos (23 núcleos/nodo), 960 RAM GB (48 GB/nodo).

III-B. Análisis de Precisión e Instancias Eliminadas

En esta sección se presenta el análisis de los resultados obtenidos por los clasificadores después de aplicar el framework propuesto. Denotamos con *Original* el uso del clasificador sin utilizar ninguna técnica de tratamiento de ruido.

La Tabla I muestra la precisión en test para los cuatro datasets y los cinco niveles de ruido usando kNN como clasificador. En vista de estos resultados podemos señalar que:

Tabla II
Precisión en test del árbol de decisión. Los valores más altos de cada dataset y nivel ruido se muestran en negrita

Dataset Voto	Ruido	Original	EHO	EHT Mayor.	EHT Consen.	ENN
SUSY	0 %	80,24	79,78	79,69	**80,27**	78,56
	5 %	79,94	79,99	80,07	**80,36**	77,49
	10 %	79,15	79,85	79,81	**80,04**	77,00
	15 %	78,21	**79,81**	79,32	79,47	75,81
	20 %	77,09	**79,71**	79,35	78,95	74,21
HIGGS	0 %	70,17	**71,16**	69,61	70,41	68,85
	5 %	69,61	**71,14**	69,34	69,98	68,29
	10 %	69,22	**71,06**	68,95	69,56	67,52
	15 %	68,65	**71,03**	68,52	69,04	66,93
	20 %	67,82	**71,05**	68,18	68,38	66,05
Epsilon	0 %	62,39	**66,86**	65,13	66,07	61,54
	5 %	61,10	**66,64**	65,32	66,09	60,41
	10 %	60,09	**66,87**	65,46	66,11	59,20
	15 %	59,02	**66,62**	65,33	65,99	58,09
	20 %	57,73	**66,46**	65,08	65,69	56,71
ECBDL14	0 %	73,98	**74,59**	74,21	74,51	73,66
	5 %	72,87	**74,64**	74,16	74,54	73,48
	10 %	71,67	**74,59**	73,84	74,51	72,75
	15 %	70,28	**74,61**	73,82	73,91	71,68
	20 %	68,66	**74,83**	73,78	73,82	70,16

Tabla III
Tasa de Reducción (%) para EHO, EHT y ENN

Dataset Voto	Ruido	EHO	EHT Mayoría	EHT Consenso	ENN
SUSY	0 %	20,62	21,04	8,74	49,51
	5 %	23,57	25,09	10,33	49,57
	10 %	26,50	27,94	11,68	49,66
	15 %	29,44	30,85	13,04	49,74
	20 %	32,35	33,71	14,22	49,82
HIGGS	0 %	29,08	35,13	8,20	49,71
	5 %	31,15	36,65	8,83	49,75
	10 %	33,23	38,11	9,59	49,81
	15 %	35,38	39,55	10,35	49,92
	20 %	37,34	41,05	11,11	49,88
Epsilon	0 %	34,31	22,30	2,90	49,97
	5 %	25,32	25,24	4,32	50,01
	10 %	27,80	27,88	5,83	49,97
	15 %	30,79	30,71	7,22	50,01
	20 %	33,52	33,44	8,74	50,17
ECBDL14	0 %	22,44	21,35	5,85	26,58
	5 %	25,80	24,51	8,25	31,06
	10 %	28,49	27,68	10,31	35,07
	15 %	31,13	30,71	12,07	38,63
	20 %	33,86	33,69	13,91	41,60

- El uso de cualquier técnica de filtrado de ruido siempre mejora la precisión *Original* para un mismo nivel de ruido. Hay que tener en cuenta que el uso de los filtros de ruido permiten a kNN obtener mejor rendimiento para cualquier nivel de ruido que *Original* a un nivel del 0 % para cualquier conjunto de datos.
- Si nos fijamos en la mejor estrategia de filtrado de ruido para kNN, el filtro homogéneo EHO, permite a kNN obtener los mejores valores de precisión.
- EHT obtiene un rendimiento inferior a EHO. Por otro lado, la estrategia de voto es crucial para EHT, ya que el voto por consenso alcanza una precisión menor, cercana al 2 % con respecto al voto por mayoría.
- Mientras que la precisión para *Original* cae en torno a un 2 % cada 5 % de incremento de ruido, llegando a un total de un 10 % menos precisión al 20 % de ruido, EHO decrece menos de un 1 % en total.
- El método base, ENN, es la peor opción para kNN, ya que obtiene la precisión más baja de los tres filtros de ruido. Para ENN, la precisión cae alrededor de un 2 % cada 5 % de incremento de ruido. Sin embargo, ENN es preferible a no tratar el ruido.

La Tabla II recoge los resultados de precisión en test para los tres métodos de ruido usando un árbol de decisión profundo como clasificador. De estos resultados podemos señalar:

- De nuevo, evitar tratar el ruido no es la mejor opción, ya que usar el filtro adecuado supone una mejora de precisión. Sin embargo, como el árbol de decisión es más robusto al ruido que kNN, no todos los filtros son mejores que si no tratamos el ruido (*Original*). Cuando los filtros eliminan demasiadas instancias, limpias y ruidosas, el árbol de decisión se ve más afectado ya que es capaz de tolerar pequeñas cantidades de ruido mientras explora las instancias limpias. kNN se ve más afectado que el árbol de decisión por las instancias ruidosas mantenidas. Por esto, una mala elección del filtro de ruido penalizará el rendimiento del árbol de decisión.
- Para niveles bajos de ruido, EHT puede rendir ligeramente mejor que EHO para algún dataset. Sin embargo, del 10 % en adelante, EHO supera a EHT, haciéndolo el mejor filtro para lidiar con ruido en árboles de decisión.
- Como observamos anteriormente, la precisión *Original* cae con cada incremento de ruido. En este caso, EHO rinde incluso mejor que con kNN, ya que la precisión se mantiene de 0 % al 20 % de ruido.
- En cuanto a la estrategia de voto de EHT, el voto por consenso obtiene mejores resultados que el voto por mayoría. Hay que resaltar que en kNN se ha observado lo contrario: dado que kNN es mucho más sensible y necesita fronteras más limpias (logradas con el voto por mayoría), el árbol de decisión se beneficia de la eliminación de ruido más precisa del voto por consenso.
- ENN, alcanza en torno a un 1 % menos de precisión que el resto para niveles de ruido bajos, incrementándose esta diferencia hasta un 5 % para niveles altos.

Los resultados mostrados han demostrado la importancia de aplicar un filtrado de ruido, independientemente de la cantidad de ruido presente en el dataset. Para explicar mejor por qué EHO es la mejor estrategia de filtrado, debemos estudiar la cantidad de instancias eliminadas.

En la Tabla III mostramos la tasa de reducción de instancias después de aplicar los tres filtros de ruido sobre los cuatro datasets. Cuanto mayor sea el porcentaje de ruido, mayor será la cantidad de instancias eliminadas. Sin embargo, observamos

Tabla IV
Tiempo Medio de Cómputo para EHO, EHT y ENN en segundos

Dataset Voto	EHO	EHT Mayoría	Consenso	ENN
SUSY	513,46	5.511,15	5.855,66	8.956,71
HIGGS	587,72	15.300,62	15.232,99	25,441,09
Epsilon	1.868,75	4.120,79	7.201,05	2.718,97
ECBDL14	1.228,24	9.710,70	11.217,02	14.080,03

diferentes patrones dependiendo de la técnica utilizada:

- De media, EHO elimina en torno a un 25 % de instancias al 0 % de ruido. Cada nivel de ruido, se eliminan un 3 % más de instancias.
- En EHT la estrategia de voto tiene un gran impacto en el número de instancias eliminadas. Mientras que el voto por mayoría elimina casi tantas instancias como EHO, el voto por consenso es mucho más conservativo.
- ENN es el filtro que más instancias elimina. De media elimina la mitad de las instancias del dataset al 0 % de ruido, incrementándose un 1 % cada nivel. Este comportamiento tan agresivo lacra el rendimiento de los clasificadores que toleran algo de ruido, como el árbol de decisión.
- EHO es el filtro más equilibrado en términos de instancias eliminadas. A pesar de que la cantidad de instancias eliminadas por EHT con el voto por mayoría es muy similar a EHO, las instancias seleccionadas son diferentes, afectando al clasificador usado posteriormente.

III-C. Tiempos de Cómputo

Para constituir una propuesta válida en Big Data, este framework también tiene que ser escalable. Esta sección mostramos los tiempos de cómputo de los filtros EHO, EHT y ENN.

En la Tabla IV podemos ver los tiempos medios de ejecución para los tres filtros en segundos. Como el nivel de ruido no es un factor que afecte a estos tiempos, mostramos la media para las cinco ejecuciones realizadas para cada dataset.

Los tiempos medidos muestran que EHO es también el método más rápido. EHO es unas diez veces más rápido que EHT y ENN. Esto se debe al uso del clasificador kNN en EHT y ENN, el cual es un algoritmo muy pesado computacionalmente. Como resultado, EHO es la opción más recomendada para tratar con el ruido en problemas Big Data.

En vista de estos resultados concluimos que:

- El uso de cualquier técnica de filtrado de ruido siempre mejora la precisión *Original* al mismo nivel de ruido.
- EHO ha sido el método que mejor rendimiento ha obtenido para ambos clasificadores. También es el más eficiente términos de tiempo.
- La estrategia de voto tiene un gran impacto en el número de instancias eliminadas en EHT.
- kNN es un método muy pesado computacionalmente, afectando a los tiempos de cómputo de EHT y ENN.

IV. Conclusiones

Este trabajo presenta el primer filtro de ruido para Big Data, donde la alta redundancia de las instancias y los problemas de alta dimensionalidad plantean nuevos retos a los algoritmos clásicos de preprocesamiento de ruido. Hemos propuesto dos algoritmos de filtrado de ruido, implementados en un framework Big Data: Apache Spark. Diferentes estrategias de particionamiento y ensembles de clasificadores han llevado a tres enfoques diferentes: un ensemble homogéneo, un ensemble heterogéneo y un enfoque de filtrado simple basado en similitud entre instancias.

La idoneidad de estas técnicas propuestas ha sido analizada utilizando varios conjuntos de datos. El ensemble homogéneo ha demostrado ser el enfoque más adecuado en la mayoría de los casos, tanto en la mejora de la precisión como en la mejora de los tiempos de ejecución. También muestra el mejor equilibrio entre instancias eliminadas y mantenidas.

El problema de ruido en Big Data es un paso crucial en la transformación de estos datos brutos en Smart Data. Con esta propuesta hemos permitido obtener Smart Data a partir de datos imperfectos. Nuestra propuesta puede resolver problemas con millones de instancias y miles de características en poco tiempo, obteniendo datasets limpios de ruido.

Financiación

Este trabajo está financiado por el Proyecto Nacional TIN2017-89517-P, y el Proyecto BigDaP-TOOLS - Ayudas Fundación BBVA a Equipos de Investigación Científica 2016.

Referencias

[1] X. Zhu and X. Wu, “Class Noise vs. Attribute Noise: A Quantitative Study,” *Artificial Intelligence Review*, vol. 22, pp. 177–210, 2004.
[2] B. Frénay and M. Verleysen, “Classification in the presence of label noise: A survey,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 5, pp. 845–869, 2014.
[3] B. Zerhari, “Class noise elimination approach for large datasets based on a combination of classifiers,” in *Cloud Computing Technologies and Applications (CloudTech), 2016 2nd International Conference on*. IEEE, 2016, pp. 125–130.
[4] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001.
[5] M. Hamstra, H. Karau, M. Zaharia, A. Konwinski, and P. Wendell, *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, 2015.
[6] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[7] S. Ramírez-Gallego, A. Fernández, S. García, M. Chen, and F. Herrera, “Big data: Tutorial and guidelines on information and process fusion for analytics algorithms with mapreduce,” *Information Fusion*, vol. 42, pp. 51 – 61, 2018.
[8] J. Lin, “Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail!” *Big Data*, vol. 1, no. 1, pp. 28–37, 2013.
[9] 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.
[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedins of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28.
[11] S. Verbaeten and A. Assche, “Ensemble methods for noise elimination in classification problems,” in *4th International Workshop on Multiple Classifier Systems*, ser. Lecture Notes on Computer Science, vol. 2709. Springer, 2003, pp. 317–325.
[12] C. E. Brodley and M. A. Friedl, “Identifying Mislabeled Training Data,” *Journal of Artificial Intelligence Research*, vol. 11, pp. 131–167, 1999.

Un análisis crítico del clasificador AkDE como *ensemble* y sus implicaciones para tratar con grandes volúmenes de datos

Jacinto Arias
Dpto. de Sistemas Informáticos
Universidad de Castilla-La Mancha
Albacete, 02071, España
jacinto.arias@uclm.es

José A. Gámez
Dpto. de Sistemas Informáticos
Universidad de Castilla-La Mancha
Albacete, 02071, España
jose.gamez@uclm.es

José M. Puerta
Dpto. de Sistemas Informáticos
Universidad de Castilla-La Mancha
Albacete, 02071, España
jose.puerta@uclm.es

***Abstract*—En clasificación supervisada el tamaño muestral condiciona enormemente el modelo a utilizar, especialmente para volúmenes de datos masivos donde la eficiencia del modelo y su potencia predictiva constituyen un equilibrio entre rendimiento y complejidad computacional. Los clasificadores basados en Redes Bayesianas permiten ajustarlo parametrizando su aprendizaje para estimar distribuciones de probabilidad cada vez más complejas. El rendimiento puede descomponerse en sesgo, que se reduce al aumentar la complejidad, y varianza, que aumenta de manera inversamente proporcional. El clasificador AkDE es uno de los ejemplos más estudiados ya que puede aprenderse en una única pasada sobre los datos y al tratarse de un modelo de tipo ensemble reduce la varianza agregando las predicciones de clasificadores individuales. En la práctica es necesario reducir su complejidad espacial y es utilizado junto a técnicas de selección de modelos basadas en la Teoría de la Información lo que implica pasadas adicionales sobre los datos. Este trabajo estudia el rendimiento de este clasificador comparándolo a otras técnicas de ensemble populares y cuestiona el impacto real de la agregación en la reducción del sesgo y la varianza. Comprobaremos empíricamente como en problemas con muestras grandes los resultados no se ajustan al modelo teórico y cómo la selección de modelos difiere del comportamiento básico de un ensemble. Los resultados obtenidos se utilizarán para proponer un modelo alternativo que sí capture las propiedades deseadas para un ensemble.**

***Index Terms*—Clasificación supervisada; Clasificadores basados en redes Bayesianas; Clasificadores ensemble.**

I. Introducción

La actual abundancia de datos y potencia computacional motiva a los investigadores en aprendizaje automático a desarrollar nuevos métodos buscando el balance entre escalabilidad y precisión. Sin embargo, no deberíamos caer en el error de conseguir este balance a cambio de producir algoritmos complejos de usar y que generen modelos difíciles de interpretar, o de otra manera, no tendrán aceptación por parte de la industria.

De hecho, gran parte de los algoritmos incluidos en los paquetes software que son referencia en la actualidad (e.g. [12]), fueron propuestos hace más de una década, pero todavía son competitivos. Parte de su éxito, sin duda se debe a sus fundamentos teóricos, pero también a su funcionamiento intuitivo y a la facilidad de interpretar y fijar los hiperparámetros necesarios para su uso. Un ejemplo claro son los clasificadores tipo *ensembles* [9] y, en particular, Random Forest (RF) [5], bien considerados tanto por investigadores como por usuarios. Desde el punto de vista de su usabilidad, el éxito recae en que un único parámetro, el número de modelos en el ensemble, sirve para controlar tanto la complejidad del modelo resultante como su nivel de precisión. P.e. en RF está demostrado que incrementar el número de árboles reduce la varianza en la clasificación manteniendo un sesgo estable. Esto garantiza que el rendimiento del clasificador mejorará o se estabilizará con un mayor número de árboles, lo que permite al usuario fijarlo basándose únicamente en la complejidad del problema abordado y/o en los recursos disponibles (tiempo y espacio).

En este trabajo nos centramos en los clasificadores basados en redes Bayesianas (BNCs), los cuáles al poderse aprender *out of core* son, en principio, excelentes candidatos para abordar grandes volúmenes de datos [1]. Además, en algunos de estos algoritmos, un único parámetro, k, controla la complejidad de los modelos aprendidos. Por ejemplo en el algoritmo kDB [14], incrementar el valor de k permite aumentar el número de dependencias permitidas y, por tanto, la complejidad de la red resultante. De forma similar, en el clasificador tipo ensemble AkDE [18], aumentar el valor de k implica también aumentar el número de dependencias, pero también el número de modelos, por lo que el orden de incremento en la complejidad es incluso superior al caso de kDB. La implicación, sin embargo, en términos del análisis en términos de sesgo y varianza es diferente: en kDB aumentar k permite reducir el sesgo a cambio de incrementar la varianza, mientras que en AkDB un aumento en k reduce la varianza pero aumenta mucho la complejidad del modelo resultante. Además, en AkDE la influencia de incrementar k se traslada en un aumento de las necesidades computacionales, sobre todo espaciales, que hacen obligatorio realizar una selección de modelos [6], [7], [11] para permitir su uso con bases de datos

Este artículo ha sido parcialmente financiado por fondos FEDER y la Agencia Estatal de Investigación (AEI/MINECO) mediante los proyectos TIN2016-77902-C3-1-P y TIN2016-82013-REDT. Jacinto Arias también está financiado por el MECD mediante la beca FPU13/00202.

de tamaño medio-grande. En este trabajo también estudiamos de forma crítica algunos de los principios en los que se basan estos procesos de selección de modelos.

Aunque el análisis basado en sesgo y varianza es habitual en otro tipo de ensembles, no es común en los ensembles de BNCs, por ello, la novedad de este trabajo reside en que proponemos evaluar los ensembles (y sus modelos constituyentes) de BNCs usando el enfoque de sesgo y varianza, estudiando su comportamiento en dominios de diferente tamaño. Los resultados que obtenemos indican que el algoritmo AkDE exhibe un comportamiento completamente diferente en muestras grandes que al considerar los benchmark habituales en aprendizaje automático, tradicionalmente formados por conjuntos de pequeño tamaño. Por otra parte, el estudio de los modelos constituyentes del ensemble, nos muestra discrepancias importantes con respecto al funcionamiento interno (y agregado) de otros ensembles típicos como RF. Estas observaciones podrían hacer a los usuarios replantearse la forma que se entiende el clasificador AkDE, especialmente en el caso de grandes conjuntos de datos. A partir del estudio, proponemos un nuevo ensemble de BNCs que si posee las propiedades esperadas en clasificadores clásicos tipo ensemble como bagging y RF.

El resto del trabajo contiene una breve descripción de los BNCs usados en el estudio, seguido de la descripción del análisis de sesgo y varianza seguido en el artículo y su aplicación al estudio del algoritmo AkDE desde la perspectiva *ensemble*. A continuación realizamos una experimentación usando tanto datos masivos como benchmarks clásicos en aprendizaje automático para evaluar los clasificadores objeto del estudio. Por último, a la luz de los resultados, proponemos un ensemble de BNCs cuyo comportamiento se asemeja más a los ensembles clásicos (bagging y RF).

II. Clasificadores basados en Redes Bayesianas

El problema de clasificación supervisada consiste en predecir la *etiqueta* $y \in \Omega_Y = \{y_1, \ldots, y_c\}$ para la variable de respuesta (o clase) Y, para un ejemplo $\mathbf{x} = (\mathbf{x_1}, \ldots, \mathbf{x_d})$ definido sobre d atributos $\{X_1, \ldots, X_d\}$. Para resolverlo, el objetivo es inducir un modelo (o clasificador) a partir de un conjunto de datos formado por m ejemplos previamente *etiquetados* $D = \{(x^{(1)}, y^1), \ldots, (x^{(m)}, y^{(m)})\}$.

El formalismo de las Redes Bayesianas (RBs) [13] usa un grafo dirigido acíclico (DAG) para representar de forma eficiente la distribución de probabilidad conjunta (DPC) definida sobre el conjunto de variables. En particular una RB factoriza la DPC mediante el producto $p(y, \mathbf{x}) = \prod_{\mathbf{i=1}}^{\mathbf{d}} \mathbf{p}(\mathbf{x_i} \mid \pi_{\mathbf{x_i}})$, donde π_{x_i} denota el conjunto de padres del atributo X_i en el DAG. Desde un punto de vista probabilístico, la etiqueta a asignar a un ejemplo $\mathbf{x}$ es la que maximiza la probabilidad a posteriori, es decir, $\arg\max_{y \in \Omega_Y} p(y \mid \mathbf{x})$.

En la práctica, los algoritmos que han demostrado un mejor rendimiento en esta tarea son los denominados BNCs [3], redes Bayesianas cuya estructura otorga un papel distinguido a la variable clase y que limitan de distinta forma el número de dependencias permitido. De ellos, el más popular por su simplicidad es Naive Bayes (NB), que asume que todas las variables predictoras son independientes dada la clase. Sin embargo, la suposición realizada resta capacidad discriminativa a NB lo que lo convierte en un clasificador con mucho sesgo. Otros modelos de BNCs se basan en imponer la estructura base de NB pero permitiendo algunas dependencias adicionales entre las variables predictoras.

A. k-dependence BN Classifier (kDB)

El algoritmo kDB [14] se basa en el uso de estimadores k-dependientes, es decir, cada variable puede depender de otros k atributos y de la clase. Requiere aprendizaje estructural y el modelo producido es una única RB.

El algoritmo propuesto en [14] usa los conceptos de información mutua (MI) e información mutua condicional (CMI) para guiar el proceso de aprendizaje estructural. Primero se ordenan las variables en función de $I(X_i; C)$. Después, a la variable i-ésima de dicho orden se le ponen como padres en el grafo la variable clase C y las k variables X_j con mayor $I(X_j, X_i|C)$, t.q. sólo se consideran las que preceden a X_i en el orden. La posterior estimación de parámetros requiere una pasada adicional por los datos cuando $k \geq 2$.

Fijando adecuadamente el valor de k se mejora el ajuste del modelo a los datos, sin embargo, aumentar k implica también obtener modelos más complejos, habitualmente con menor sesgo pero mayor varianza debido al sobreajuste. En [11] se estudia que en bases de datos grandes el riesgo de sobreajuste (e incremento de la varianza) disminuye considerablemente. El algoritmo kDB *selectivo* (SkDB) combate la complejidad del modelo eligiendo el k adecuado para cada variable [11], lo que requiere una pasada adicional por los datos.

B. Averaged k Dependence Estimators

El clasificador AkDE puede modelarse como un ensemble formado por un conjunto de BNCs $H = \{h_i(\pi_i), i = 1, \ldots, K\}$, donde el número de modelos K se fija restringiendo el espacio de los posibles modelos a una familia de clasificadores k-dependientes. En particular, cada modelo es un NB aumentado en el que además de depender de la clase, todos los atributos dependen de un conjunto fijo (el mismo para todos) de k padres, denotado por π_i. El ensemble consiste entonces en la familia *completa* de este tipo de clasificadores, por lo que $K = \binom{d}{k}$. La etapa de inferencia consiste en promediar la distribución de probabilidad de la clase a posteriori en todos los clasificadores del ensemble. Como caso particular, el clasificador A1DE (AODE) asume que todos los atributos dependen de la clase y otro atributo, conocido como *super padre* (SP). En AODE los modelos constituyentes se conocen como SPODE y en general como SPkDE. Al fijarse la estructura, no se requiere aprendizaje estructural, siendo un clasificador *out of core* real.

Al igual que en kDB, en AkDE podemos ajustar el balance sesgo-varianza variando el parámetro k, lo que permite representar desde clasificadores con mucho sesgo y poca varianza como NB (A0DE), hasta clasificadores con poco sesgo pero alta varianza, obtenidos al incrementar k y poder usar distribuciones de probabilidad de alta dimensionalidad

[18]. Sin embargo, en el caso de AkDE la influencia de incrementar k en la complejidad del modelo resultante se traslada al incremento del tamaño de las tablas de probabilidad y, sobre todo, al incremento en el número de modelos del ensemble. Por ejemplo, con 100 variables predictoras A1DE contendrá 100 modelos individuales pero A2DE contendrá 5000. Indudablemente esto representa un problema muy importante de cara a la escalabilidad del algoritmo en problemas de media-alta dimensionalidad (número de variables) por lo que se han propuesto diferentes soluciones basadas en realizar selección de modelos [6], [7], [11]. Es importante remarcar que si en kDB la selección de modelos tenía como objetivo un mejor ajuste de los datos, en AkDE la selección de modelos es obligatoria para convertirlo en un modelo usable en la práctica.

Estas técnicas de selección de modelos se basan en una aproximación híbrida entre el enfoque filter y wrapper, guiados por el uso de conceptos de Teoría de la Información, mayormente Información Mutua (MI). En particular, los algoritmos *Sample Attribute Selective AkDE (SASAkDE)* [6] y **Selective AkDE (SAkDE)** [7] han mostrado que es posible reducir la complejidad espacial, pero también (sorprendentemente) obtener mejor predicción (accuracy) que el ensemble formado por la familia completa de SPkDEs. Sin embargo, también hay algunos puntos débiles que deben ser estudiados: (1) este proceso añade pasadas adicionales por los datos, lo que supone un inconveniente en el caso de grandes muestras (e.g. big data); (2) la idea subyacente es asumir que la información mutua condicional del conjunto de super-padres dada la clase es un buen indicador del rendimiento en términos de clasificación del sub-modelo resultante. En este trabajo analizamos empíricamente esta hipótesis; y (3) volviendo a nuestra discusión inicial y las ventajas de tener hiperparámetros interpretables, indicar que este proceso de selección de modelos también añade cierta confusión al proceso, puesto que a igual k la complejidad del modelo resultante puede diferir mucho de un problema a otro.

III. Sesgo y Varianza en clasificación

Distintos estudios [2], [4] han analizado la capacidad predictiva de los clasificadores ensemble mediante la descomposición del error en términos de *sesgo* y *varianza*. Intuitivamente, un algoritmo de aprendizaje *sesgado* muestra un error persistente (similar) al entrenar con distintos conjuntos de datos, mientras que un algoritmo con alta varianza muestra un error que fluctúa entre los distintos conjuntos de datos.

Breiman [4] presenta una taxonomía de clasificadores estables (poca varianza a riesgo de tener mucho sesgo) e inestables (poco sesgo y alta varianza). Los clasificadores inestables, *en media* tienen un buen rendimiento, lo que les convierte en el marco ideal para los ensembles basados en voto por la mayoría, puesto que su uso consigue reducir la varianza [4]. Ejemplos son RF o bagging con árboles de decisión, que son algoritmos estado-del-arte en clasificación supervisada. Por otra parte, los modelos inestables son buenos candidatos para su aplicación en problemas grandes [11], ya que el impacto de la varianza se suaviza al disponer de más datos.

Sin embargo, la descomposición del error en sesgo y varianza proviene de la regresión numérica, y aunque su interpretación es también intuitiva en el caso de la clasificación, su aplicación no lo es. De hecho, mientras que en regresión promediar funciones independientes reduce la varianza sin modificar el sesgo, en clasificación promediar los modelos podría incrementar el error de clasificación [15].

Existen distintas formulaciones de la descomposición en sesgo/varianza [15], [16]. En este trabajo hemos optado por la implementación realizada en [16][1] de la descrita en [2], [4]. Las métricas utilizadas se obtienen a partir de la estabilidad de un clasificador $\mathcal{L}$ cuando se entrena y testea repetidamente sobre un número de conjuntos de datos $\mathcal{T}$. Definimos la tendencia central $C^{\circ}_{\mathcal{L}\mathcal{T}}(\mathbf{x})$ como la clase con máxima probabilidad de ser seleccionada para un ejemplo determinado $\mathbf{x}$ por parte de todos los clasificadores aprendidos a partir de $\mathcal{T}$: $C^{\circ}_{\mathcal{L}\mathcal{T}}(\mathbf{x}) = \arg\max_y P_{\mathcal{T}}(\mathcal{L}(\mathbf{x}) = y)$

El sesgo puede entonces medirse como el error introducido por la tendencia central del algoritmo, es decir, el error de la clasificación más frecuente, mientras que la varianza es el error introducido por la desviación de dicha tendencia central. Habitualmente se habla de estos valores en términos de *contribución* del sesgo y la varianza al error de clasificación, ya que éste puede expresarse como la suma de ambos. Para calcularlos, primero se obtiene una estimación de la tendencia central a partir de nuestra muestra $\mathcal{D}$, para lo que se ejecuta una 10x3 validación cruzada, induciendo así 30 modelos $\mathcal{L}(T^i_k)$ y el correspondiente conjunto de test f^i_k para cada una de las $i \in \{1,\ldots,10\}$ repeticiones y $k \in \{1,2,3\}$ conjuntos (folds) de entrenamiento. Por tanto, se obtienen 10 predicciones independientes para cada instancia $\mathbf{x}$ y fijamos la tendencia central $C^{\circ}_{\mathcal{L}\mathcal{T}}(\mathbf{x})$ para dicho ejemplo como la media. Formalmente, la tendencia central se obtiene como:

$$C^{\circ}_{\mathcal{L}\mathcal{T}}(\mathbf{x}) = \arg\max_y P\left(\sum_{i=1}^{10}\sum_{k=1}^{3} 1\left[\mathbf{x} \in f^i_k \wedge \mathcal{L}(T^i_k)(\mathbf{x}) = y\right]\right)$$

La contribución del sesgo y la varianza al error se calculan para cada instancia y se agregan para todo el conjunto de datos:

$$\begin{aligned} sesgo &= P_{(\mathbf{x},y),\mathcal{T}}(\mathcal{L}(\mathcal{T})(\mathbf{x}) \neq y \wedge \mathcal{L}(\mathcal{T})(\mathbf{x}) = C^{\circ}_{\mathcal{L}\mathcal{T}}(\mathbf{x}) \\ varianza &= P_{(\mathbf{x},y),\mathcal{T}}(\mathcal{L}(\mathcal{T})(\mathbf{x}) \neq y \wedge \mathcal{L}(\mathcal{T})(\mathbf{x}) \neq C^{\circ}_{\mathcal{L}\mathcal{T}}(\mathbf{x}) \end{aligned}$$

IV. AkDE Bajo la Perspectiva de un Ensemble

Entre los modelos ensemble más populares encontramos el clasificador RF [5] basado en agregar el voto de árboles de decisión aplicando Bagging: Aprender cada modelo a partir de una muestra obtenida realizando bootstraping, y Random Subspaces: Seleccionar un conjunto de nodos subóptimo a la hora de estimar la partición de cada nodo del árbol. Esto incrementa la diversidad de los modelos de un modo predecible, especialmente en árboles completamente desarrollados. El estudio original demuestra que, si se incluyen modelos

[1]Esta metodología se ha usado p.e. para estudiar la propuesta inicial del clasificador AkDE [17], [18]

suficientes el sesgo del ensemble converge asintóticamente a niveles relativos a un árbol de decisión individual.

En el caso de AkDE se aprende un número finito de modelos en función de k, imponiendo una cota para la reducción de varianza y siendo imposible estabilizar el sesgo asintóticamente. Un número fijo de modelos reducirá al mismo tiempo la diversidad, deteriorando los resultados de la agregación. Además, los modelos del ensemble no se aprenden mediante una estrategia aleatoria subóptima sino que su estructura se fija heurísticamente conforme a cada super-padre. En general un clasificador k-dependiente H reduce el sesgo ajustando su estructura de la manera más fiel posible a un modelo óptimo sin restringir H', mientras que en el caso de AkDE no hay garantía de que las distribuciones de probabilidad modeladas aproximen a dicho clasificador, por lo que podemos asumir que muchos parámetros impactarán negativamente en el sesgo.

La literatura muestra que AkDE obtiene baja varianza en la práctica, sin embargo, no existe evidencia de que esto sea un resultado directo de la agregación o es inherente a los modelos individuales. Para profundizar realizaremos un experimento, para nuestro conocimiento inédito, evaluando la contribución particular al error de sesgo y varianza para una colección de ensembles y los modelos que los componen.

A. *Sesgo y la Varianza en Problemas de Gran Tamaño*

Un BNC con bajo sesgo es idóneo para grandes volúmenes de datos ya que obtendrá distribuciones de probabilidad mejor calibradas. Para determinar el sesgo de un modelo podemos medir su estabilidad para muestras de tamaño incremental. Para ello utilizaremos la base de datos sintética *pokerhand* [8] como benchmark, evaluando la descomposición en sesgo y varianza del error sobre 20 muestreos desde 50k instancias hasta 1M [2].

En la Figura 1 podemos ver que RF obtiene menor varianza que los modelos individuales, mientras que es máxima en el caso de árbol totalmente desarrollado. Al contrario, un árbol sin aleatorizar, es un modelo menos sesgado que el ensemble por estar compuesto por modelos que no se ajustan perfectamente a los datos. No obstante, la suma confirma que el ensemble es el de menor error y mayor estabilidad, especialmente para dominios pequeños donde la varianza es más dificil de reducir. En los modelos BNC confirmamos que los modelos con mayor sesgo como naive Bayes o kDB con $k = 1$ no mejoran conforme el tamaño de la muestra aumenta, mientras que al aumentar k la mejora es clara y constante, en presencia de suficientes datos que permitan calibrar los parámetros correctamente. Respecto a la varianza, confirmamos que los modelos sencillos son más estables, mientras que los clasificadores complejos requieren de más datos para estabilizarse.

Aunque los resultados para A1DE deberían ser superiores a los de kDB con $k = 1$ o $k = 2$, vemos que empeoran al aumentar la muestra. Si utilizamos el voto por la mayoría en lugar de la agregación numérica de probabilidades, Figura 1 bajo el nombre *a1de-majority*, vemos que la agregación de probabilidades suaviza el error cometido por los modelos de peor calidad, mientras que el voto por la mayoría los imita, implicando que si hay más modelos sesgados que acertados el ensemble lo estará también. Un tamaño de muestra mayor calibra las probabilidades de dichos modelos hacia valores extremos, convirtiendo al ensemble en un modelo equivalente al voto por la mayoría para muestras grandes.

B. *Diseño Alternativo de Ensembles Basados en BNCs*

Recientemente se han propuesto ensembles de BNCs alternativos, dado que en la práctica AkDE requiere aplicar selección de modelos y con ello una fase de aprendizaje adicional, podemos utilizar otros modelos que ya la realizan como kDB. Cabe destacar el clasificador kDF (k-dependent forest) [10] basado en construir un ensemble compuesto por n modelos kDB, uno para cada atributo X donde se aplica un proceso de ordenación de los atributos más sofisticado que introduce diversidad. Aunque esta propuesta supera en rendimiento a A1DE sufre de los mismos problemas descritos anteriormente ya que solo considera un número finito de modelos obtenidos de un modo no aleatorio.

Un modelo ensemble basado en agregación o voto por mayoría debería capturar ambas propiedades, para lo que proponemos una alternativa básica, el clasificador k-dependiente aleatorio (RkDB). Este ensemble tendá $h \in [1, \inf)$ modelos independientes aprendidos por una versión alterada de kDB, considerando solo un subconjunto en una proporción $\alpha \in [0, 1]$ de los padres disponibles para cada nodo en cada modelo, tomando como inspiración el particionado subóptimo que realiza RF. Así, añadimos diversidad mediante aleatoriedad controlada, preservando el sesgo del clasificador original.

C. *Sesgo y Varianza en los Modelos Individuales*

En nuestro segundo experimento evaluaremos el comportamiento de la descomposición del error para un benchmark clásico (véase la Tabla I) obtenido a partir del repositorio UCI [8]. Este conjunto de problemas es el utilizado en la mayoría de propuestas basadas en el clasificador AkDE. Estos modelos han obtenido siempre un buen rendimiento en este contexto, como hemos podido reproducir según muestran los resultados de la Tabla II. Comprobamos que no existe diferencia significativa entre el rendimiento de A1DE y RF y la nueva propuesta RkDB cuando k=1, no obstante para sucesivos valores de k los resultados empeoran, lo que achacamos al reducido tamaño de muestra de los problemas que conforman el benchmark. Si se observa la Figura 1 podemos comprobar que ocurre lo mismo en muestras de poco tamaño.

A la vista de que los ensembles mejoran a los modelos individuales, estudiaremos el efecto de la agregación en la descomposición en sesgo y varianza. La Figura 2 muestra la distancia entre el ensemble y los modelos individuales, podemos ver como RF reduce la varianza como esperabamos seguido de RkDB, mientras que en A1DE esta reducción es

[2]Experimentos realizados en un cluster Apache Spark de 7 nodos con procesadores hexacore Intel Xeon E5-2609v3 1.90GHz y 64GB de RAM. El software está basado en [1] y el código está disponible en http://github.com/jacintoArias/pgm2018.

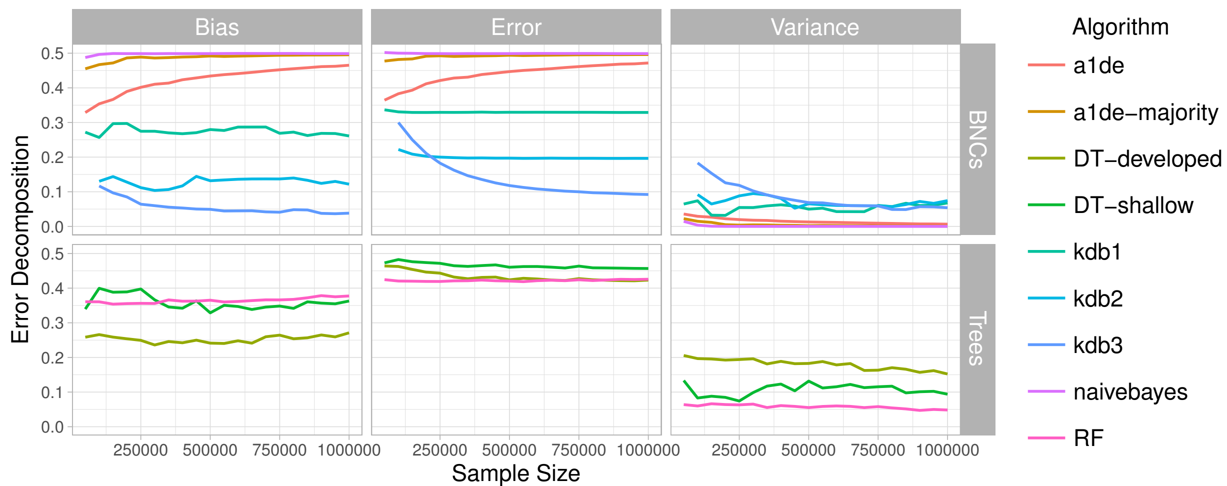


Figura 1. Evolución de sesgo y varianza para experimentos repetidos sobre muestras de mayor tamaño incrementalmente. La gráfica muestra en el eje y los valores para varianza, sesgo y error de izquierda a derecha, para la familia de modelos BNC en la parte superior y los basados en árboles en la inferior.

Problema	Casos	Atr.	Etiq.	Problema	Casos	Atr.	Etiq.	Problema	Casos	Atr.	Etiq.
Pokerhand	1000000	10	8	Car	1728	8	4	Soybean	307	35	15
Adult	48842	15	2	Contraceptive-mc	1473	10	3	Haberman	306	3	2
Chess	28056	6	18	German	1000	21	2	HeartDisease-c	303	14	2
Letter	20000	17	26	Vowel	990	14	11	Audiology	226	70	24
Nursery	12960	9	5	Tic-Tac-Toe	958	10	2	New-Thyroid	215	6	3
PenDigits	10992	17	10	Anneal	898	39	6	Glass-id	214	10	3
CensusIncome	10419	14	2	Vehicle	846	19	4	Sonar	208	61	2
Mushrooms	8124	23	2	PimaIndiansDiabetes	768	9	2	Autos	205	26	7
Musk	6598	168	2	BreastCancer-w	699	10	2	Wine	178	14	3
OpticalDigits	5620	49	10	BalanceScale	625	5	3	Hepatitis	155	20	2
PageBlocks	5473	11	5	CreditApproval	690	15	2	TeachingAssistant	151	6	3
Spambase	4601	58	2	Cylinder-bands	512	39	2	Iris	150	5	3
Hypothyroid	3772	30	4	Haberman	306	3	2	Promoters	106	58	2
Kr.vs.kp	3196	37	2	HouseVotes84	435	17	2	Zoo	101	17	7
Splice	3190	62	3	HorseColic	368	22	2	Post-operative	90	9	3
Segment	2310	20	7	Ionosphere	351	35	2	LaborNegotiations	57	17	2
Mfeat	2000	6	2	PrimaryTumor	339	18	22	LungCancer	32	57	3

Tabla I
Propiedades de los problemas utilizados en los experimentos. Los atributos continuos han sido discretizados en cuatro intervalos por igual frecuencia para los modelos BNC.

casi imperceptible, incluso la mejora parece venir de una reducción en sesgo. Observando una muestra de estas diferencias a nivel individual, Figura 3, podemos ver dos distribuciones muy diferentes, muy consistente en el caso de RF y RkDB pero casi aleatoria en A1DE. Este experimento saca a luz que A1DE al no seguir los mismos principios de diseño que otros ensembles tampoco se comporta de la manera esperada, lo que explica porque la selección de subconjuntos puede mejorar el rendimiento respecto al conjunto entero, una propiedad nada deseable y difícil de controlar en un ensemble [15].

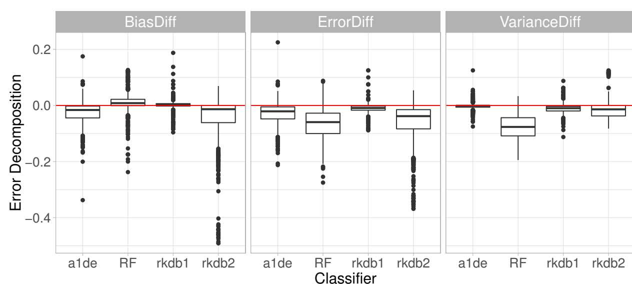


Figura 2. Distribución de las diferencias en la descomposición del error entre el modelo agregado y los diferentes modelos individuales. La línea roja en y=0 ayuda a distinguir entre diferencias positivas y negativas.

D. Sobre la Efectividad de la Selección de Modelos

Hemos observado baja consistencia en los modelos que componen un ensemble AkDE, donde unos modelos muestran propiedades deseables como bajo sesgo mientras que otros parecen ser propensos a error y tienen poco margen de varianza que reducir. Los algoritmos de selección de modelos introducidos aprovechan esta circunstancia para reducir el espacio de modelos y al mismo tiempo mejorar su rendimiento utilizando heurísticas basadas en la hipótesis de que la Información Mutua (MI) entre los atributos predictores que guían los modelos y la clase es un buen indicador de su rendimiento.

Cabe realizar entonces la pregunta de si estos métodos seleccionan entonces aquellos clasificadores propensos a mejorar en la agregación, poco sesgo y mayor varianza, o solo aquellos que minimizan el error independientemente. Para responder empíricamente hemos evaluado diversos criterios de selección de modelos de manera incremental, es decir, añadiendo cada vez un modelo adicional. Se han evaluado tres criterios wrapper: mínimo error, bias y varianza; un criterio filter basado en MI y un criterio aleatorio para establecer una base de comparación. La Tabla III muestra la suma del

Figura 3. Distribución bidimensional de sesgo (x) y varianza (y) para los modelos individuales (cruces azules) y el ensemble (punto rojo).

error sobre todos los incrementos para cada criterio y base de datos del benchmark anterior. Los resultados muestran que los mejores criterios son aquellos que minimizan el error o su componente del sesgo, mientras que la varianza, la IM y el resultado aleatorio son equivalentes. Esta evidencia reafirma nuestra hipótesis y revela que la los criterios filter basados en Información Mutua pueden no ser los más acertados.

V. Conclusiones y Trabajo Futuro

Hemos visto que los buenos resultados de AkDE en la práctica son inconsistente con la definición de un ensemble, y que la mejora en rendimiento al seleccionar modelos solo es relevante en la aplicación de técnicas wrapper que son muy costosas de computar en un escenario real. En su lugar hemos propuesto y comparado favorablemente un clasificador sencillo de tipo ensemble basado en BNCs. Los resultados muestran que es comparable a AkDE y que proporciona un recorrido de mejora que exploraremos en trabajos posteriores, especialmente en problemas de gran tamaño, donde hemos visto una clara superioridad del clasificador básico kDB en calibración de probabilidades y reducción del sesgo.

Algorithm	rank	pvalue	win	tie	loss
A1DE	2.48	-	-	-	-
RF	2.49	**9.7929e-01**	26	0	27
RkDB1	2.76	**8.7224e-01**	30	2	21
kDB1	3.58	7.1644e-03	37	0	16
RkDB2	4.75	1.8647e-09	46	0	7
kDB2	4.93	7.4299e-11	46	1	6

Tabla II
Las columnas gana, empata y pierde comparan el error obtenido por el mejor modelo (A1DE) contra los demás (p.e, A1DE gana 24 veces contra RF y pierde 17). El rango y el p-valor corresponden al test de Friedman y post-hoc (Holm) con $\alpha = 0.05$. En negrita las hipótesis rechazadas.

Criterio	Rango	p-valor	Gana	Empata	Pierde
error	1.52	-	-	-	-
bias	1.83	**3.6273e-01**	24	3	17
mi	3.75	7.8353e-11	43	0	1
variance	3.81	3.7139e-11	41	1	2
random	4.09	1.0269e-13	43	0	1

Tabla III
Comparativa de la suma del error para distintos criterios de selección en A1DE. Métricas descritas en la Tabla II.

References

[1] Jacinto Arias, Jose A. Gamez, and Jose M. Puerta. Learning distributed discrete Bayesian Network Classifiers under MapReduce with Apache Spark. *Knowledge-Based Systems*, 117:16 – 26, 2017.
[2] Eric Bauer, Ron Kohavi, Philip Chan, Salvatore Stolfo, and David Wolpert. An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants. *Machine Learning*, 36:105–139, 1999.
[3] Concha Bielza and Pedro Larrañaga. Discrete Bayesian Network Classifiers: A Survey. *ACM Comput. Surv.*, 47(1):5:1–5:43, jul 2014.
[4] Leo Breiman. Arcing classifiers. *Annals of Statistics*, 26(3):801–849, 1998.
[5] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
[6] Shenglei Chen, Ana M. Martinez, Geoffrey I. Webb, and Limin Wang. Sample-Based Attribute Selective AnDE for Large Data. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):172–185, 2017.
[7] Shenglei Chen, Ana M. Martínez, Geoffrey I. Webb, and Limin Wang. Selective AnDE for large data learning: a low-bias memory constrained approach. *Knowledge and Information Systems*, 50(2):475–503, 2017.
[8] Dua Dheeru and Efi Karra. UCI Machine Learning Repository, 2017.
[9] Thomas G. Dietterich. An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees. *Machine Learning*, 40:139–157, 2000.
[10] Zhiyi Duan and Limin Wang. K-Dependence Bayesian Classifier Ensemble. *Entropy*, 19(12), 2017.
[11] A M Martínez, G I Webb, S Chen, and N A Zaidi. Scalable learning of Bayesian network classifiers. *Journal of Machine Learning Research*, 17:1–30, 2016.
[12] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.
[13] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. 2014.
[14] Mehran Sahami. Learning Limited Dependence Bayesian Classifiers. In *Proc. of the 2nd Int. Conf. on Knowledge Discovery and Data Mining*, KDD'96, pages 335–338. AAAI Press, 1996.
[15] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *Annals of Statistics*, 26(5):1651–1686, 1998.
[16] Geoffrey I. Webb. MultiBoosting: a technique for combining boosting and wagging. *Machine Learning*, 40(2):159–196, 2000.
[17] Geoffrey I. Webb, Janice R. Boughton, and Zhihai Wang. Not so naive Bayes: Aggregating one-dependence estimators. *Machine Learning*, 58(1):5–24, 2005.
[18] Geoffrey I. Webb, Janice R. Boughton, Fei Zheng, Kai Ming Ting, and Houssam Salem. Learning by extrapolation from marginal to full-multivariate probability distributions: Decreasingly naive Bayesian classification. *Machine Learning*, 86(2):233–272, oct 2012.

Un enfoque aproximado para acelerar el algoritmo de clasificación Fuzzy kNN para Big Data

Jesus Maillo, Julián Luengo, Salvador García, Francisco Herrera
Dept. de Ciencias de la Computación e Inteligencía Artificial
Universidad de Granada. 18071, Granada, España
Email: {jesusmh, julianlm, salvagl, herrera}@decsai.ugr.es

Isaac Triguero
School of Computer Science
University of Nottingham, Jubilee Campus
Nottingham NG8 1BB, United Kingdom
Email: Isaac.Triguero@nottingham.ac.uk

***Resumen*—El clasificador difuso de los k vecinos más cercanos (Fuzzy k Nearest Neighbor - Fuzzy kNN) es reconocido por su eficacia en problemas de aprendizaje supervisado. El algoritmo kNN clasifica comparando una nueva instancia con los datos etiquetados mediante una función de similitud. Su versión difusa se compone de dos etapas. La primera etapa calcula el grado de pertenencia a cada clase para cada instancia del problema, con el objetivo de detectar con mayor precisión las fronteras entre clases. La segunda etapa clasifica siguiendo el mismo esquema que el algoritmo kNN, aunque haciendo uso del grado de pertenencia a clase previamente calculado. La propuesta existente de Fuzzy kNN para afrontar conjuntos de datos muy grandes no es completamente escalable, debido a que replica el comportamiento de Fuzzy kNN original. Con el objetivo de agilizar este algoritmo, en esta contribución se propone un enfoque aproximado que acelera los tiempos de ejecución sin perder calidad en la clasificación, mediante el uso de árboles e implementado en Spark. En la experimentación realizada se comparan varios enfoques de Fuzzy kNN para big data, con conjuntos de datos de hasta 11 millones de instancias. Los resultados muestran una mejora en tiempo de ejecución y precisión con respecto al modelo de la literatura.**

***Palabras Clave*—Conjuntos difusos, K-vecinos más cercanos, Clasificación, MapReduce, Apache Spark, Big Data**

I. Introducción

El algoritmo de los k vecinos más cercanos (*k Nearest Neighbor* - kNN) [1] es catalogado como clasificador basado en instancias. Para clasificar, compara las instancias no vistas con aquellas etiquetadas del conjunto de entrenamiento utilizando una función de similitud. Generalmente la similitud es medida mediante una función de distancia (Euclídea o Manhattan). A pesar de su simplicidad, el clasificador kNN es uno de los diez algoritmos de clasificación más relevantes [2].

Sin embargo, kNN suponiendo todos los vecinos igual de importantes en la clasificación, considerando que las fronteras entre clases están perfectamente definidas. Existen diferentes propuestas basadas en la teoría de los conjuntos difusos que abordan este problema. En [3], el algoritmo clásico Fuzzy kNN [4] destaca como uno de los enfoques más eficaces. Fuzzy kNN se compone de dos etapas: cálculo de grado de pertenencia y clasificación. La primera etapa, cambia la etiqueta de la clase por un grado de pertenencia a cada clase, de acuerdo a las instancias más cercanas. La segunda etapa, calcula el kNN con la información del grado de pertenencia. Así, se consigue detectar con mayor precisión las fronteras, viéndose menos afectado por el ruido y mejorando al kNN en la mayoría de los problemas de clasificación.

En el ámbito del big data, tanto el algoritmo kNN como Fuzzy kNN encuentran problemas para manejar grandes conjuntos de datos con respecto al tiempo de ejecución y al consumo de memoria. Existe una propuesta exacta del algoritmo kNN para abordar problemas big data y se denomina *k Nearest Neighbor - Iterative Spark* (kNN-IS) [5]. Además de esta versión exacta, también existen versiones aproximadas que reducen los tiempos de ejecución: Metric-Tree [6] y Spill-Tree. En [7], los autores estudian estos modelos y proponen el modelo Hybrid Spill-Tree (HS) como hibridación de ambos con el objetivo de mejorar el tiempo de ejecución sin afectar a los resultados debido a la redundancia de datos implícita en big data. Por otro lado, existe una solución exacta para aplicar Fuzzy kNN en big data, *Global Exact Fuzzy k Nearest Neighbors* (GE-FkNN) [8]. Aunque es capaz de escalar hasta conjuntos de datos muy grandes, los tiempos de ejecución de la primera etapa son sustancialmente altos, provocando un cuello de botella. Sin embargo, no se han aplicado enfoques aproximados para el algoritmos Fuzzy kNN en big data.

En este trabajo se propone una variación aproximada del algoritmo Fuzzy kNN desarrollado en Spark. Para aliviar el cuello de botella de la primera etapa del algoritmo Fuzzy kNN, se estudiarán dos enfoques: local y global. El enfoque local consiste en dividir el conjunto de datos en diferentes partes y calcular el grado de pertenencia a clase internamente en cada partición (operación map), sin considerar otras particiones. El enfoque global está basado en el modelo HS. Éste genera un árbol con las instancias del conjunto de entrenamiento y lo distribuye entre todos los nodos de cómputo, considerando todas las instancias para el cálculo del grado de pertenencia. La segunda etapa es responsable de la clasificación y común para las dos propuestas. En ésta, el esquema de trabajo es igual a la versión global de cálculo de pertenencia, pero considera la pertenencia previamente calculada clase para predecir. Para comprobar el rendimiento de este modelo, los experimentos realizados se han llevado a cabo en 4 conjuntos de datos con hasta 11 millones de instancias. El estudio experimental analiza la precisión y tiempo de ejecución realizando una comparativa con otros algoritmos de la literatura.

El documento está estructurado de la siguiente manera. La

sección II introduce el estado del arte en los algoritmos Fuzzy kNN y Hybrid Spill-Tree. La sección III detalla la propuesta aproximada del algoritmo Fuzzy kNN. El estudio experimental se describe en la Sección IV. La sección V concluye el documento y esboza el trabajo futuro.

II. Preliminares

Esta sección proporciona el conocimiento básico del algoritmo Fuzzy kNN (Sección II-A), el Hybrid Spill-Tree (Sección II-B) y las tecnologías big data utilizadas (Sección II-C).

II-A. *k vecinos más cercanos difuso y su complejidad*

El algoritmo Fuzzy kNN [4] se propone como una mejora del algoritmo kNN, llegando a mejorarlo en términos de precisión para la mayoría de los problemas de clasificación. Necesita una etapa de pre-cómputo en el conjunto de entrenamiento, que calcula el grado de pertenencia a la clase. Posteriormente, calcula kNN para cada instancia no vista y decide la clase predicha con el grado de pertenencia. Una notación formal para el algoritmo Fuzzy kNN es la siguiente:

Sea CE un Conjunto de Entrenamiento y CP un Conjunto de Prueba, compuestos de **n** y **t** instancias respectivamente. Cada instancia $\mathbf{x}_i$ es un vector $(\mathbf{x}_{i1}, \mathbf{x}_{i2}, \mathbf{x}_{i3}, \ldots, \mathbf{x}_{ij})$, donde $\mathbf{x}_{ij}$ es el valor de la i-ésima instancia y j-ésima característica. Para cada instancia de CE se conoce su clase ω. Sin embargo, para las instancias de CP la clase es desconocida.

Fuzzy kNN posee dos etapas: cálculo de pertenencia y clasificación. La primera etapa calcula los k vecinos más cercanos para cada instancia del CE, manteniendo un esquema *leave-one-out* seleccionando las k instancias con una distancia menor. Finalmente, calcula el grado de pertenencia de acuerdo a la Ecuación 1. El resultado de la primera etapa será CE modificando la etiqueta de clase ω, por un vector de pertenencia a cada clase (ω_1, ω_2, …, ω_l) donde l es el número de clases. Este nuevo conjunto se llamará Conjunto de Entrenamiento Difuso, CED.

$$u_j(x) = \begin{cases} 0{,}51 + (n_j/k_{pert}) \cdot 0{,}49 & if \quad j = i \\ (n_j/k_{pert}) \cdot 0{,}49 & if \quad j \neq i \end{cases} \tag{1}$$

La etapa de clasificación calcula para cada instancia de CP sus kNN como se describe en la primera etapa, para cada instancias de CP en el CED y se obtiene el grado de pertenencia a clase. Posteriormente, obtiene la clase resultante de acuerdo a la Ecuación 2.

$$u_i(x) = \frac{\sum_{j=1}^{K} u_{ij}(1/|x - x_j|^{2/(m-1)})}{\sum_{j=1}^{K}(1/|x - x_j|^{2/(m-1)})} \tag{2}$$

La etapa extra respecto a kNN y el aumento de la complejidad computacional generan dos problemas big data:

- Tiempo de ejecución: La complejidad de calcular kNN para una instancia es $mathcalO(n \cdot c)$, donde n es el número de instancias de CE y c el número de características. Para más de un vecino, aumenta a $\mathcal{O}(n \cdot log(N))$. Además, Fuzzy kNN tiene una etapa extra de cómputo para el cálculo de grado de pertenencia.
- Consumo de memoria: Para acelerar el cálculo, se requiere disponer de los conjuntos CE y CP almacenados en memoria principal. Cuando ambos conjuntos son grandes, es fácil exceder la memoria principal disponible.

Para aliviar estas dificultades, trabajamos en el diseño de un modelo aproximado basado en Hybrid Spill-Tree desarrollado bajo las tecnologías big data de MapReduce y Spark.

II-B. *Hybrid Spill-Tree*

El algoritmo Hybrid Spill-Tree (HS) [9] es una propuesta aproximada para calcular el algoritmo kNN de forma distribuida. HS se compone de dos estructuras: Metric-Tree y Spill-Tree. La estructura de datos Metric-Tree (MT) organiza el conjunto de datos en una jerarquía espacial. Es un árbol binario cuya raíz contiene todos los elementos, y donde cada hijo representa un subconjunto de elementos. La Figura 1 ilustra como dividir los elementos entre los dos hijos, seleccionando cada hijo como las instancias más lejanas posibles (representados por ◯). Los nodos hijos de un MT no contienen instancias repetidas (representadas por $+$). El árbol tendrá una profundidad de $\mathcal{O}log(N))$. Para realizar la búsqueda de la instancia más cercana, se conserva al candidato con menor distancia C y su distancia d. Si la distancia a una rama es superior a d, la poda y continúa la búsqueda. Una vez no existe una rama del árbol con distancia menor que d, se finaliza la búsqueda y se devuelve C y d. Sin embargo, invierte tiempo y cómputo en asegurar que C es el más cercano, volviendo atrás en el árbol si fuese necesario. Con el objetivo de reducir el tiempo de búsqueda emerge Spill-Tree.

La estructura de datos Spill-Tree (SP) es una variación de MT. La diferencia principal consiste en permitir instancias compartidas entre los nodos hijos. La Figura 1 presenta cómo se dividen los datos con el mismo procedimiento que MT, permitiendo un conjunto de instancias duplicadas en los nodos hijos. El área de solapamiento es dependiente del parámetro τ. Cuando τ es 0, no se comparten instancias, sería un MT. Si τ es demasiado alto, el solapamiento es alto y la profundidad del árbol tiende a infinito. SP reduce los tiempos de búsqueda respecto a MT sacrificando la *vuelta atrás* en el árbol para comprobar que C es el más cercano. Debido al solapamiento entre hijos, la instancia que encuentra, aunque aproximada, es muy representativa del problema.

HS aparece con el objetivo de obtener un balance entre precisión y tiempo de ejecución. Para ello, fusiona los modelos MT y SP. Para construir la estructura hibrida de HS, se construye un SP, y si el número de instancias en el área de solapamiento es menor que el Umbral de Equilibrio (UE), se mantiene como SP. Si las instancias repetidas superan el UE, se reconstruye como MT. Para calcular kNN sobre esta estructura, se realizará *vuelta atrás* para asegurar el más cercano si la rama es MT, mientras que los nodos SP no harán esta comprobación. Destacar la implementación

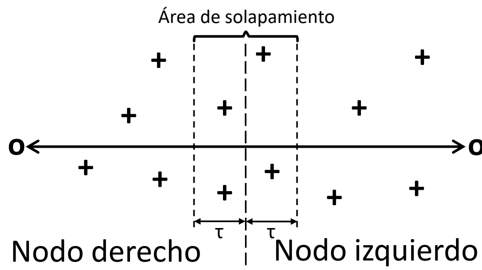


Figura 1: Construcción de Metric-Tree y Spill-Tree

disponible en el repositorio de software libre spark-package[1], que es el punto de partida en la etapa de desarrollo del presente trabajo.

II-C. Paradigma de programación MapReduce: Spark

Para el desarrollo del algoritmo propuesto en este trabajo se utilizará el paradigma de programación MapReduce [10]. MapReduce tiene como objetivo procesar grandes conjuntos de datos a través de la distribución del almacenamiento de datos y ejecución a través de un conjunto de máquinas.

La implementación de MapReduce seleccionada es Apache Spark [11]. Spark paraleliza el cálculo de forma transparente a través de una estructura de datos distribuidos llamada *Resilient Distributed Datasets* (RDD). Los RDDs permiten persistir y reutilizar estructuras de datos almacenadas en memoria principal. Adicionalmente, Spark fue desarrollado para cooperar con el sistema de archivos distribuidos de Hadoop[2] (Hadoop Distributed File System - HDFS). Con esta configuración, se puede aprovechar la división de instancias, la tolerancia a fallos y la comunicación de trabajos que proporciona Spark.

Spark incluye una librería de aprendizaje automático llamada MLlib[3]. Dispone de multitud de algoritmos de aprendizaje automático y técnicas estadísticas de diferentes áreas como clasificación, regresión o preprocesamiento de datos.

III. Fuzzy kNN acelerado para Big data

En esta sección se presenta una propuesta aproximada y distribuida del algoritmo Fuzzy kNN basado en el método HS para tratar problemas big data implementado en Spark. La presente propuesta tiene las mismas dos etapas que el modelo Fuzzy kNN: grado de pertenencia a clase (Sección III-A) y clasificación (Sección III-B).

III-A. Etapa de cálculo de grado de pertenencia a clase

Esta sección explica el flujo de trabajo de dos enfoques aproximados para calcular el grado de pertenencia a clase: Local basado en kNN (Sección III-A1) y global basado en HS (Sección III-A2) . El resultado de ambos enfoques es modificar la etiqueta de clase del Conjunto de Entrenamiento (CE) por un vector de pertenencia a clase, formando el Conjunto de Entrenamiento Difuso (CED)

[1]Hybrid Spill-Tree. https://spark-packages.org/package/saurfang/spark-knn
[2]Apache Hadoop. Web: http://hadoop.apache.org/
[3]Machine Learning Library for Spark. Web: http://spark.apache.org/mllib/

Algorithm 1 Etapa de cálculo de grado de pertenencia - Local

Require: CE, k, $\#Maps$
1: $CEP \leftarrow$ repartition(CE, #Maps)
2: $CEPD \leftarrow$ mapPartition(calcularPertenencia(CEP, k))
3: $CED \leftarrow$ unir($CEPD$)
4: **return** CED
5: **COMIENZA calcularPertenencia**
6: **for** y: CEP_i **do**
7: $Vecinos_y \leftarrow$ calcularkNNLocal ($modelo$, k, y)
8: $pertenencia_y \leftarrow$ calcularPertenencia ($Vecinos_y$)
9: $CEPD \leftarrow$ unir(y, $pertenencia_y$)
10: **end for**
11: **return** $CEPD$
12: **FIN calcularPertenencia**

Algorithm 2 Etapa de cálculo de grado de pertenencia - Global

Require: CE, k
1: $muestreas \leftarrow$ muestrear(CE,0, 2 %)
2: $arbolSuperior \leftarrow$ construirMT($muestreas$)
3: $\tau \leftarrow$ estimarTau($TopTree$)
4: $arbol \leftarrow$ repartir(CE, $arbolSuperior$, τ, $UE = 70\,\%$)
5: $modelo \leftarrow$ (difundir($arbolSuperior$),arbol)
6: **for** y: CE **do**
7: $Vecinos_y \leftarrow$ calcularkNN ($modelo$, k, y)
8: $pertenencia_y \leftarrow$ calcularPertenencia ($Vecinos_y$)
9: $resultado_y \leftarrow$ unir(y, $pertenencia_y$)
10: **end for**
11: **return** $resultado$

III-A1. Enfoque local: El enfoque local conjuntamente con la etapa de clasificación recibe el nombre *Local Hybrid Spill-Tree Fuzzy kNN* (L-HSFkNN). El Algoritmo 1 muestra los pasos y operaciones en Spark para el cálculo del grado de pertenencia. Comienza leyendo el CE de HDFS y es dividido en $\#Maps$ partes. Posteriormente, utiliza una operación mapPartition de Spark para calcular de forma distribuida el grado de pertenencia a clase para cada partición CEP_i. Para cada instancia y de cada partición CEP_i, calcula kNN y finalmente obtiene el grado de pertenencia aplicando la Ecuación 1. Una vez obtenida la pertenencia para cada partición, se agrupan los resultados y forman el CED, que será la entrada de la etapa de clasificación.

III-A2. Enfoque global basado en HS: El enfoque global sumado a la etapa de clasificación recibe el nombre *Global Approximate Hybrid Spill-Tree Fuzzy kNN* (GA-HSFkNN). El Algoritmo 1 muestra los pasos y operaciones en Spark para el cálculo del grado de pertenencia.

El algoritmo 2 muestra los pasos de la etapa de cálculo de pertenencia. Las líneas 1-5 corresponden a la etapa de ajuste del modelo, y las restantes al cálculo del grado de pertenencia.

La etapa de ajuste del modelo comienza leyendo el CE de HDFS. En primer lugar, toma un submuestreo aleatorio para construir un MT como se describe en la Sección II-B (Los autores recomiendan un 0,2 %). Este MT recibe el nombre de árbol superior (AS) y es usado para estimar el valor del parámetro τ y particionar todo el CE. La estimación de τ es la distancia media entre las instancias. Para acelerar este cálculo, se realiza con las instancias del AS.

El siguiente paso es repartir el CE. Para ello, a partir

Algorithm 3 Calcular kNN

```
Require: modelo, k, x
 1: Indices ← x.flatMap ( buscaIndices(modelo.arbol) )
 2: Vecinos ← consulta(modelo.arbol, Indices, k)
 3: return Vecinos
 4:
 5: COMIENZA buscaIndices
 6: distIzq ← nodoIzq.dist(x)
 7: distDch ← nodoDch.dist(x)
 8: if nodo! = HOJA then
 9:    if distIzq < distDch then
10:       buscaIndices(nodoIzq, ID)
11:    else
12:       buscaIndices(nodoDch, ID + hijoIzq)
13:    end if
14: else
15:    return Indices
16: end if
17: FIN buscaIndices
```

del AS se distribuyen las instancias en el espacio. El valor de τ define el área de solapamiento. Comienza construyendo un SP, y comprueba si el número de instancias en el área de solapamiento es menor que el 70 %. En otro caso, se reconstruye un MT. Al realizar la búsqueda, aquellas ramas SP realizan una búsqueda más rápida al no tener que volver atrás en el árbol. Sin embargo, las construidas como MT realizan *vuelta atrás* para asegurar el más cercano. La etapa de construcción del modelo termina distribuyendo el AS y el árbol asociado al CE.

La etapa de cálculo de grado de pertenencia se muestra en las líneas 6-10. Para cada instancia de CE, se calcula kNN siguiendo el modelo generado. El Algoritmo 3 describe como se realiza la búsqueda con operaciones nativas de Spark. Mediante una operación flatMap, se computan y obtienen los índices de las instancias más cercanas del CE. Para ello, se calcula la distancia hasta el nodo derecho e izquierdo, y continúa por el nodo con una distancia menor. Cuando alcanza un nodo hoja, devuelve el índice de la instancia seleccionada.

Dados los vecinos, se calcula el grado de pertenencia siguiendo la Ecuación 1. (Línea 8). El resultado de esta fase es el CED, pasando a ser la entrada de la etapa de clasificación.

III-B. *Clasificación con Hybrid Spill-Tree*

Esta sección describe la etapa de clasificación, la cual recibe como entrada el CED calculado en la etapa previa. La etapa de clasificación está basada en HS y sigue la misma estructura que la aproximación global de la primera etapa (Sección III-A2). Tiene dos fases diferenciadas: creación del modelo y clasificación. La primera construye el árbol y divide las instancias entre los nodos de cómputo. La segunda busca las k instancias más cercanas de CED calculado en la primera etapa, y devuelve como salida la clase predicha de acuerdo al vector de grado de pertenencia.

El Algoritmo 4 muestra los pasos de la etapa de clasificación. Se especificarán las diferencias con respecto a la sección III-A2, ya que el esquema principal es el mismo y sólo se ven afectados pequeños detalles de la estructura de datos.

Algorithm 4 Etapa de clasificación

```
Require: CED, CP, k
 1: muestreas ← muestrear(CED,0, 2 %)
 2: arbolSuperior ← construirMT(muestreas)
 3: τ ← estimarTau(TopTree)
 4: arbol ← repartir(CED, arbolSuperior, τ, UE = 70 %)
 5: modelo ← (difundir(arbolSuperior),arbol)
 6: for y: CE do
 7:    VecinosP_y ← calcularFuzzykNN (modelo, k, y)
 8:    prediccion_y ← calcularPredicción (Vecinos_y)
 9: end for
10: return prediccion_y
```

La primera diferencia se encuentra en los conjuntos de datos de entrada. En este caso, se utilizarán CED y CP. La fase de ajuste del modelo no se ve afectada pues no se modifican las características, manteniendo las distancias de las instancias. Así, se construye el modelo con la misma metodología.

El cálculo de Fuzzy kNN se hace de la misma manera, con la diferencia de que en lugar de devolver la etiqueta de clase para cada vecino, se devuelve el vector de grado de pertenencia a la clase (Línea 7). La línea 8 calcula la clase predicha aplicando la Ecuación 2. El resultado final es la clase prevista para cada instancia de CP.

IV. Estudio experimental

En esta sección se presentan las cuestiones planteadas en el estudio experimental. La Sección IV-A presenta los algoritmos de comparación utilizados para la experimentación. La Sección IV-B determina el marco de los experimentos. Finalmente, la Sección IV-C expone y analiza los resultados obtenidos.

IV-A. *Algoritmos de comparación*

Esta sección expone los algoritmos de comparación utilizados en los experimentos y sus acrónimos. Se compara con otras propuestas de Fuzzy kNN y sus análogos no *Fuzzy* siguiendo dos enfoques: local y global. El enfoque local divide las instancias y los distribuye entre los nodos de cómputo, ejecutando de forma independiente en cada partición, obteniendo resultados aproximados. El enfoque global si considera todas las instancias, aunque realice el cómputo de forma distribuida y a su vez presenta dos posibilidades: exacto y aproximado. El exacto invierte cálculo en asegurar que el resultado es igual que la variante secuencial. El aproximado, no asegura el mismo resultado acelerando el tiempo de ejecución.

Algoritmos utilizados en la experimentación para realizar la comparativa:

- *Global Exact Fuzzy kNN (GE-FkNN)* [8]: modelo exacto del algoritmo Fuzzy kNN para abordar problemas big data, obteniendo los mismos resultados que el Fuzzy kNN original. Sus dos etapas son globales y exactas.
- *Local Fuzzy kNN (L-FkNN)*: propuesta desarrollada del algoritmo Fuzzy kNN. La primera etapa es la descrita en la Sección III-A1 y su segunda etapa sería global y exacta, idéntica a la de GE-FkNN.

- *k Nearest Neighbor - Iterative Spark (kNN-IS)* [5]: propuesta exacta de kNN para abordar problemas big data, obteniendo los mismos resultados que el kNN original.
- *Hybrid Spill-Tree kNN (HS-kNN)* [9]: propuesta aproximada de kNN para big data. Aunque aproximada, considera todas las instancias en la búsqueda.

IV-B. Marco experimental

Para el estudio experimental, se han seleccionado cuatro conjuntos de datos de un elevado número de instancias. El conjunto de datos ECBDL14 se extrae de la competición [12]. A pesar de que tiene una relación de desbalanceo superior a 45, hemos seleccionado este conjunto de datos por su número relativamente alto de características. Sin embargo, en este documento no abordamos el problema de la clasificación desequilibrada, por lo que se ha submuestreado dejándolo en una relación de desbalanceo de dos. Los otros tres conjuntos han sido extraídos del repositorio UCI [13]. La Tabla I presenta el número de instancias, características y clases ($\#\omega$). Se seguirá el esquema de validación cruzada en 5 particiones.

Tabla I: Descripción de los conjuntos de datos

Conjunto de datos	#Instancias	# Características	$\#\omega$
Poker	1.025.010	10	10
ECBDL14	2.063.187	631	2
Susy	5.000.000	18	2
Higgs	11.000.000	28	2

Se tendrán en cuenta las siguientes medidas para evaluar el rendimiento de la técnica propuesta:

- *Precisión:* Contabiliza el número de clasificaciones correctas en relación con el número total de instancias.
- *Tiempo de ejecución:* Tiempo consumido considerando lecturas y comunicaciones por red de Spark.

El parámetro más conocido para el algoritmo original Fuzzy kNN es el número de vecinos (k). k podría ser diferente en cada etapa, pero mantendremos ambos iguales por simplicidad. En nuestros experimentos, el parámetro k está ajustado a 3, 5 y 7. Se necesita un parámetro adicional debido a su comportamiento distribuido. Este es el número de particiones del conjunto de entrenamiento. Para exprimir la componente distribuida, se utilizará el máximo posible para la configuración del cluster, 256 operaciones maps/particiones.

Los algoritmos basados en HS tienen dos parámetros adicionales: el número de instancias para construir el árbol principal AP y el umbral de equilibrio UE. Los valores recomendados por los autores son AP igual al 0,2 % y UE igual al 70 %.

Todos los experimentos se han realizado en un cluster compuesto por 14 nodos de cálculo gestionados por un nodo maestro. Todos los nodos tienen la misma configuración. 2 procesadores Intel Xeon CPU E5-2620, 6 núcleos (12 hilos) por procesador, 2 GHz y 64 GB de RAM. Red Infiniband 40Gb/s. Con la configuración actual, se disponen de 256 tareas map como máximo. La versión de Spark es 2.2.1.

IV-C. Resultados obtenidos y análisis

Esta sección presenta una comparativa de los modelos propuestos y los detallados en la Sección IV-A, analizando el tiempo de ejecución y la precisión de cada uno de ellos. Para realizar un análisis detallado de los algoritmos, se utilizarán los 4 conjuntos de datos expuestos en la Tabla I. La Sección IV-C1 presenta el tiempo de ejecución y precisión. La Sección IV-C2 compara los algoritmos propuestos con otros algoritmos de la literatura.

IV-C1. Estudio de tiempo y precisión: Esta sección presenta y estudia el tiempo de ejecución y precisión obtenida para los 4 conjuntos de datos y los diferentes algoritmos.

La Tabla II muestra una comparación de precisión entre los algoritmos en relación al número de vecinos (k) y para todos los conjuntos de datos.

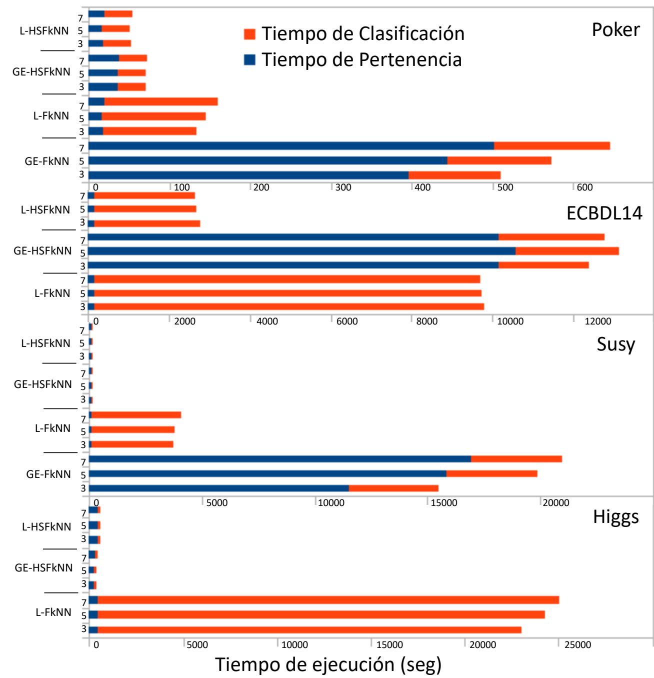


Figura 2: Tiempo de ejecución para cada conjunto y algoritmo

Tabla II: Comparativa entre algoritmos - Precisión

Algoritmo	k	Poker	ECBDL14	Susy	Higgs
GE-FkNN	3	0,5257	-	0,7301	-
	5	0,5316	-	0,7306	-
	7	0,5338	-	0,7268	-
L-FkNN	3	0,5218	0,7636	0,7178	0,5936
	5	0,5243	0,7492	0,7190	0,5953
	7	0,5243	0,7423	0,7179	0,5959
GA-HSFkNN	3	0,5233	**0,8051**	0,7302	0,5968
	5	0,5371	0,8050	0,7457	0,6081
	7	0,5459	0,8014	**0,7510**	0,6162
L-HSFkNN	3	0,5324	0,8035	0,7320	0,6038
	5	0,5432	0,8027	0,7438	0,6139
	3	**0,5487**	0,7969	0,7471	**0,6198**

La Figura 2 muestra el tiempo de la etapa de cálculo de pertenencia y el tiempo de la etapa de clasificación en segundos, para cada conjunto de datos, algoritmo y los valores de k igual a 3, 5 y 7.

De acuerdo con la tabla y figura presentadas, se observa:

Tabla III: Comparativa contra propuestas kNN

Conjunto de datos	Algoritmo	k	Tiempo Total	Precisión
Poker	kNN-IS	3	113,2370	0,4758
		5	123,8311	0,4952
		7	136,5830	0,4937
	HS-kNN	3	32,9661	0,5201
		5	33,4674	0,5305
		7	35,5575	0,5369
	GA-HSFkNN	7	72,1978	0,5459
	L-HSFkNN	7	54,3435	**0,5487**
ECBDL14	kNN-IS	3	27121,0583	0,7833
		5	28673,7015	0,7797
		7	28918,2992	0,7683
	HS-kNN	3	3151,6242	0,8020
		5	2608,1722	0,8017
		7	2691,5595	0,7986
	GA-HSFkNN	3	12413,8265	**0,8051**
	L-HSFkNN	3	2787,1230	0,8035
Susy	kNN-IS	3	2615,0150	0,6675
		5	2273,6377	0,6784
		7	2372,4100	0,6861
	HS-kNN	3	128,1860	0,7223
		5	131,0210	0,7360
		7	133,2428	0,7431
	GA-HSFkNN	7	133,2568	**0,7510**
	L-HSFkNN	7	146,2076	0,7471
Higgs	kNN-IS	3	10262,9178	0,5454
		5	13446,0519	0,5458
		7	14285,6442	0,5559
	HS-kNN	3	641,4434	0,5885
		5	651,1573	0,5936
		7	664,9204	0,5981
	GA-HSFkNN	7	429,3507	0,6162
	L-HSFkNN	7	608,2571	**0,6198**

- El valor de k no afecta significativamente a los tiempos de ejecución en ninguna de las etapas.
- El algoritmo GE-FkNN encuentra su límite de escalabilidad en su primera etapa, no llegando a ejecutar para los datasets ECBDL14 y Higgs.
- Centrándonos en los tiempos de ejecución, el modelo L-FkNN escala mejor en la primera etapa y se ve menos afectado por el número de características. El algoritmo GA-HSFkNN consigue mejores tiempos en la etapa de clasificación. Así, los mejores tiempos de ejecución son obtenidos por el algoritmo L-HSFkNN, que toma la primera etapa del algoritmo L-FkNN y la etapa de clasificación de GA-HSFkNN.
- En precisión los ganadores son GA-HSFkNN y L-HSFkNN con diferencias poco significativas. Teniendo en cuenta que los tiempos de ejecución totales son menores para L-HSFkNN, es nuestra propuesta más escalable y de mayor calidad.

IV-C2. Comparativa con versiones kNN big data: En esta sección se comparan los modelos basados en conjuntos difuso con las versiones clásicas de kNN para big data. La Tabla III muestra una comparativa entre el mejor resultado obtenido por los dos algoritmos propuestos y las dos alternativas no *Fuzzy*, explorando los mismos valores de k. Se muestra para cada conjunto de datos y algoritmo, el tiempo total de ejecución y el valor de k utilizado asociado a la precisión obtenida.

De acuerdo con la tabla presentada, se puede observar que los mejores resultados los obtienen las versiones *Fuzzy*. Además, kNN-IS obtiene resultados significativamente peores que los demás algoritmos. Aunque HS-kNN mejora respecto al algoritmo kNN-IS, siempre queda por debajo de los modelos propuestos, sin verse incrementado en exceso el tiempo de ejecución debido a la optimización realizada en la etapa de clasificación del algoritmo GA-HSFkNN.

V. Conclusiones y trabajo futuro

En esta contribución se ha propuesto un enfoque MapReduce para acelerar el algoritmo Fuzzy kNN en problemas Big data. Gracias al diseño realizado y al uso de tecnologías big data, se consigue ejecutar con conjuntos de datos muy grandes, que de otra forma sería inviable. Los experimentos realizados comparan la propuesta con otros enfoques de comportamiento exacto y aproximado para conseguir un equilibrio entre eficiencia y precisión. Asimismo, se ha comparado con las versiones no *Fuzzy* para estudiar la mejora del modelo propuesto. El algoritmo L-HSFkNN demuestra una escalabilidad muy elevada, así como resultados en precisión de alta calidad. Como trabajo futuro, se plantea adaptar el modelo desarrollado a otros problemas de minería de datos como regresión, o aprendizaje semi-supervisado.

Agradecimientos

Este trabajo se ha sustentado por el proyecto de investigación TIN2017-89517-P. J. Maillo disfruta de una beca FPU del Ministerio de Educación de España.

Referencias

[1] T. M. Cover, P. E. Hart, Nearest neighbor pattern classification, IEEE Transactions on Information Theory 13 (1) (1967) 21–27.
[2] X. Wu, V. Kumar (Eds.), The Top Ten Algorithms in Data Mining, Chapman & Hall/CRC Data Mining and Knowledge Discovery, 2009.
[3] J. Derrac, S. García, F. Herrera, Fuzzy nearest neighbor algorithms: Taxonomy, experimental analysis and prospects, Information Sciences 260 (2014) 98 – 119.
[4] J. M. Keller, M. R. Gray, J. A. Givens, A fuzzy k-nearest neighbor algorithm, IEEE Transactions on Systems, Man, and Cybernetics SMC-15 (4) (1985) 580–585.
[5] J. Maillo, S. Ramírez, I. Triguero, F. Herrera, kNN-IS: an iterative spark-based design of the k-Nearest Neighbors classifier for big data, Knowledge-Based Systems 117 (Supplement C) (2017) 3 – 15, volume, Variety and Velocity in Data Science.
[6] J. K. Uhlmann, Satisfying general proximity / similarity queries with metric trees, Information Processing Letters 40 (4) (1991) 175 – 179.
[7] T. Liu, A. W. Moore, K. Yang, A. G. Gray, An investigation of practical approximate nearest neighbor algorithms, in: Advances in neural information processing systems, 2005, pp. 825–832.
[8] J. Maillo, J. Luengo, S. García, F. Herrera, I. Triguero, Exact fuzzy k-nearest neighbor classification for big datasets, in: 2017 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), 2017, pp. 1–6.
[9] T. Liu, C. J. Rosenberg, H. A. Rowley, Performing a parallel nearest-neighbor matching operation using a parallel hybrid spill tree, uS Patent 7,475,071 (Jan. 6 2009).
[10] J. Dean, S. Ghemawat, Map reduce: A flexible data processing tool, Communications of the ACM 53 (1) (2010) 72–77.
[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, 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, 2012, pp. 1–14.
[12] ECBDL14 dataset: Protein structure prediction and contact map for the ECBDL2014 big data competition (2014). URL http://cruncher.ncl.ac.uk/bdcomp/
[13] M. Lichman, UCI machine learning repository (2013). URL http://archive.ics.uci.edu/ml

Aplicando la transformada integral de la probabilidad para reducir la complejidad de los árboles de decisión difusos multi-vía en problemas de clasificación Big Data

Mikel Elkano*†‡, Mikel Uriz*†, Humberto Bustince*†‡, Mikel Galar*†‡
*Departamento de Estadística, Informática y Matemáticas, Universidad Pública de Navarra, 31006 Pamplona, España
†GIARA, Navarrabiomed, Complejo Hospitalario de Navarra (CHN), Universidad Pública de Navarra (UPNA), IdiSNA
Irunlarrea 3, 31008 Pamplona, España
‡Institute of Smart Cities, Universidad Pública de Navarra, 31006 Pamplona, España
Emails: {mikel.elkano, mikelxabier.uriz, bustince, mikel.galar}@unavarra.es

***Resumen*—Presentamos un nuevo método de particionamiento difuso distribuido para reducir la complejidad de los árboles de decisión difusos multi-vía en problemas de clasificación Big Data. El algoritmo propuesto construye un número fijo de conjuntos difusos para todas las variables y ajusta su forma y posición a la distribución real de los datos de entrenamiento. Para ello se aplica un proceso compuesto por dos fases: 1) transformación de la distribución original en una distribución uniforme estándar mediante la transformada integral de la probabilidad. Dado que generalmente la distribución original se desconoce, se calcula una aproximación de la función de distribución acumulada extrayendo los q-cuantiles del conjunto de entrenamiento; 2) construcción de una partición difusa de Ruspini en el espacio de características transformado empleando un número fijo de funciones de pertenencia triangulares uniformemente distribuidas. A pesar de realizar una transformación sobre los datos originales, la definición de los conjuntos difusos en el espacio original se puede recuperar aplicando la función cuantil. Los resultados experimentales revelan que el método de particionamiento propuesto permite mantener la precisión del árbol de decisión difuso multi-vía del estado del arte FMDT empleando hasta 6 millones menos de hojas.**

***Index Terms*—Árboles de Decisión Difusos; Transformada Integral de la Probabilidad; Función Cuantil; MapReduce; Apache Spark; Big Data**

I. Introducción

Los árboles de decisión (ADs) [1] son uno de los algoritmos de aprendizaje automático más populares, habiendo sido aplicados en una gran variedad de problemas como las finanzas [2], la clasificación de imágenes [3], la bioinformática [4], o la medicina [5]. La principal característica de los ADs es la habilidad para explicar el razonamiento de sus predicciones mediante grafos en forma de árboles. Cada nodo es una pregunta o test en un atributo determinado (por ejemplo, ¿$x > 0{,}5$?), cada rama representa la respuesta o el resultado del test, y los nodos terminales (hojas) contienen la decisión final. Generalmente los árboles se construyen aplicando una estrategia *top-down* llamada *particionamiento recursivo* [1], en donde los datos de entrada se dividen recursivamente en dos o más sub-espacios que aumentan la homogeneidad de las distribución de las clases.

La lógica difusa [6] ha mostrado ser una forma efectiva para mejorar el rendimiento de clasificación de los algoritmos de aprendizaje automático cuando se trabaja con datos imprecisos que generan incertidumbre [7], [8], incluyendo los árboles de decisión difusos (ADDs). Sin embargo, en entornos Big Data los excesivos requerimientos de tiempo y espacio de los ADDs afectan seriamente la escalabilidad de estos algoritmos. Segatori et al. propusieron una solución MapReduce que consiste en un nuevo método de particionamiento difuso y en un algoritmo de aprendizaje de ADD distribuido [9]. El discretizador genera una partición difusa fuerte y triangular para cada atributo continuo basándose en la entropía difusa, las cuales son empleadas posteriormente para construir el árbol. Los autores propusieron dos versiones de ADD que difieren en la estrategia de particionamiento recursivo: el ADD binario (o bi-vía) (FBDT) y el ADD multi-vía (FMDT). El ADD binario divide el espacio del atributo en dos sub-espacios (nodos hijo), mientras que el ADD multi-vía puede generar más de dos sub-espacios. A pesar de que la solución propuesta por Segatori et al. consigue buenos resultados en términos de precisión, normalmente los árboles construidos por este algoritmo están compuestos por miles e incluso millones de hojas, por lo que la complejidad del modelo es generalmente elevada.

En este trabajo presentamos un nuevo método de particionamiento difuso que reduce la complejidad de los árboles construidos por FMDT en términos de número de conjuntos difusos empleado por variable y número de hojas. El algoritmo propuesto aplica la *transformada integral de la probabilidad* [10] para ajustar un número fijo de conjuntos difusos a la distribución real del conjunto de entrenamiento. Esta transformación permite convertir las variables del conjunto de entrenamiento en variables aleatorias uniformes independientemente de su distribución original. Posteriormente, se construyen la particiones difusas de Ruspini [11] en el nuevo espacio transformado empleando funciones de pertenencia

triangulares uniformemente distribuidas. Los conjuntos difusos resultantes se utilizan posteriormente por el FMDT original para construir el árbol.

Para evaluar los beneficios de nuestra propuesta, hemos llevado a cabo un estudio empírico que consiste en 4 problemas de clasificación Big Data disponibles en los repositorios de UCI [12] y OpenML[1]. En este estudio comparamos la precisión y la complejidad del modelo de FMDT cuando se considera el método de particionamiento difuso original y el propuesto. Los resultados experimentales muestran una reducción significativa en la complejidad del modelo cuando se aplica nuestro método.

La estructura de este trabajo es la siguiente. La Sección II repasa los conceptos básicos de los ADDs y describe brevemente la solución distribuida de Segatori et al. para construir ADDs para Big Data. En la Sección III introducimos el método de particionamiento difuso propuesto. El marco y el estudio experimental se muestran en las Secciones IV y V. Finalmente, la Sección VI presenta las conclusiones de este trabajo.

II. Preliminares: árboles de decisión difusos para Big Data

Los árboles de decisión (ADs) [1] son algoritmos de aprendizaje automático supervisado no-paramétricos empleados para tareas de clasificación y regresión. En este trabajo nos centramos en las tareas de clasificación, las cuales consisten en construir un modelo predictivo llamado *clasificador* que es capaz de clasificar ejemplos no etiquetados (desconocidos) en base a un conjunto de entrenamiento formado por ejemplos previamente etiquetados. Cada ejemplo $x = (x_1, \ldots, x_F)$ contenido en el conjunto de entrenamiento TR pertenece a una clase $y \in \mathbb{C} = \{C_1, ..., C_M\}$ (siendo M el número de clases del problema) y se caracteriza por un conjunto de F variables (también llamadas atributos o características), donde cada variable x_f puede tomar cualquier valor contenido en el conjunto $\mathcal{F}_f$. Por tanto, la construcción de un clasificador consiste en encontrar una función de decisión $h : \mathcal{F}_1 \times \ldots \times \mathcal{F}_F \rightarrow \mathbb{C}$ que maximice la precisión en la clasificación.

Un AD es un grafo dirigido acíclico donde cada nodo interno es un test sobre un atributo, cada rama representa el resultado del test, y cada nodo terminal (hoja) contiene la decisión final (etiqueta de la clase). Los ADs se construyen aplicando un *particionamiento recursivo* [1] del espacio de atributos. La selección del atributo considerado en el nodo de decisión está basada en métricas que miden la diferencia entre el nivel de homogeneidad de las clases contenidas en el nodo padre y en los nodos hijos. Para los atributos continuos se pueden aplicar soluciones basadas tanto en fuerza bruta como en estrategias de discretización. Las primeras comprueban todos los posibles puntos de corte en el conjunto de entrenamiento, mientras que las últimas dividen el dominio del atributo en un conjunto discreto de intervalos (también llamados *bins*). Dado que las soluciones por fuerza bruta pueden ser computacionalmente costosas, los ADs diseñados para Big Data suelen aplicar estrategias de discretización para acelerar la ejecución del algoritmo y reducir la complejidad del modelo.

[1] https://www.openml.org/search?type=data

Los árboles de decisión difusos (ADDs) [7] hacen uso de la lógica difusa [6] para manejar mejor la incertidumbre y crear fronteras de decisión suaves que mejoran la precisión de la clasificación. En este caso los atributos continuos están caracterizados por particiones difusas en lugar de un conjunto discreto de intervalos. Por consiguiente, un valor de entrada determinado puede pertenecer a uno o más conjuntos difusos con un grado de pertenencia determinado y activar múltiples ramas al mismo tiempo. Las particiones difusas permiten manejar transiciones progresivas entre los intervalos adyacentes y mejorar así la precisión de las predicciones cuando se manejan datos numéricos. A la hora de clasificar un nuevo ejemplo x, se calcula el grado de activación de cada hoja. Para ello se debe calcular el grado de activación de todos los nodos internos que pertenecen a la ruta de la hoja correspondiente de la siguiente forma. Dado el nodo actual CN que considera x_f como el atributo a dividir, se calcula el grado de activación $md^{CN}(x)$ de CN para x:

$$md^{CN}(x) = T\left(\mu^{CN}(x_f), md^{PN}(x)\right), \tag{1}$$

donde T es una T-norma, $\mu^{CN}(x_f)$ es el grado de pertenencia de x_f al conjunto difuso asociado con el nodo CN, y $md^{PN}(x)$ es el grado de activación del nodo padre PN para x. Posteriormente, se obtiene el grado de asociación $AD_m^{LN}(x)$ de la clase C_m en la hoja LN para x:

$$AD_m^{LN}(x) = md^{LN}(x) \cdot w_m^{LN}, \tag{2}$$

donde $md^{LN}(x)$ es el grado de activación de la hoja LN para x y w_m^{LN} es el peso de la clase C_m en LN. En la literatura se han propuesto diferentes definiciones para w_m^{LN} [13]. En este trabajo consideramos

$$w_m^{LN} = \frac{\sum_{x \in TR_{C_m}} md^{LN}(x)}{\sum_{x \in TR} md^{LN}(x)}, \tag{3}$$

donde TR_{C_m} es el conjunto de todos los ejemplos de entrenamiento que pertenecen a la clase C_m. Finalmente, se predice la clase de x siguiendo un determinado criterio, siendo los siguientes los más comunes:

- *Máxima activación*: se predice la clase que corresponde al máximo grado de asociación.
- *Voto ponderado*: para cada clase se calcula la suma de todos los grados de asociación correspondientes a la clase y se predice aquella que obtiene la mayor suma.

Los requerimientos de tiempo y memoria de los ADDs pueden causar serios problemas de escalabilidad cuando se tratan grandes conjuntos de datos. En este trabajo consideramos la solución distribuida propuesta por Segatori et al. en [9] para construir ADDs en Big Data, la cual consta de dos fases:

1. *Particionamiento difuso*. Se genera una partición difusa triangular fuerte para cada atributo continuo basándose en la entropía difusa. Para ello, el algoritmo selecciona la partición difusa candidata que minimiza la entropía difusa y divide el dominio del atributo en dos subconjuntos de forma recursiva hasta que se cumple una cierta condición de parada. A pesar de que las particiones difusas construidas por este método suelen ser precisas, el alto número de conjuntos difusos que habitualmente contienen aumenta la complejidad del modelo.
2. *Aprendizaje del ADD*. Se construye un ADD aplicando una de las dos estrategias de división consideradas por los autores: binaria (FBDT), que genera dos nodos hijos, o multi-vía (FMDT), que puede generar más de dos hijos. Ambos métodos emplean la ganancia de información difusa [14] para la selección del atributo. En este trabajo nos centramos en los árboles FMDT.

III. Aplicando la transformada integral de la probabilidad para reducir la complejidad de los árboles de decisión difusos multi-vía

En este trabajo proponemos un nuevo método de particionamiento difuso distribuido que reduce la complejidad de los ADDs generados por el algoritmo FMDT [9]. La solución propuesta reemplaza el método de particionamiento difuso original empleado por FMDT sin alterar el algoritmo de aprendizaje del ADD. Los objetivos de nuestra propuesta son los siguientes:

- Construir un número reducido de conjuntos difusos por atributo. El método original añade conjuntos difusos a la partición hasta que la ganancia de información difusa es menor que un cierto umbral, aumentando la complejidad del modelo. Nuestro método emplea un número fijo de conjuntos difusos para todos los atributos.
- Ajustar los conjuntos difusos a la distribución real de los atributos. La solución propuesta modifica tanto la forma como la posición de los conjuntos difusos para mejorar la capacidad de discriminación del modelo

Para cumplir estos objetivos proponemos un algoritmo que consiste en una fase de pre-procesamiento que resulta en un particionamiento difuso auto-adaptativo.

- Pre-procesamiento: las variables del conjunto de entrenamiento son transformadas en variables aleatorias uniformes aplicando la *transformada integral de la probabilidad* [10] descrita en el Teorema 1. Este teorema afirma que cualquier variable aleatoria continua puede ser convertida en una variable aleatoria uniforme estándar.

  Teorema 1. *Si X es una variable aleatoria continua con una función de distribución acumulada (FDA) $F_X(x)$ y si $Y = F_X(X)$, entonces Y es una variable aleatoria uniforme en el intervalo [0,1].*

  Demostración. Supongamos que $Y = g(X)$ es una función de X donde g es derivable y estrictamente creciente. Por tanto, su inversa g^{-1} existe únicamente. La FDA de Y se puede derivar usando

  $$F_Y(y) = Prob\,(Y \leq y) = Prob\left(X \leq g^{-1}(y)\right) \\ = F_X\left(g^{-1}(y)\right)$$

  y su densidad viene dada por

  $$f_Y(y) = \frac{d}{dy}F_Y(y) = \frac{d}{dy}F_X(g^{-1}(y)) \\ = f_X(g^{-1}(y)) \cdot \frac{d}{dy}g^{-1}(y).$$

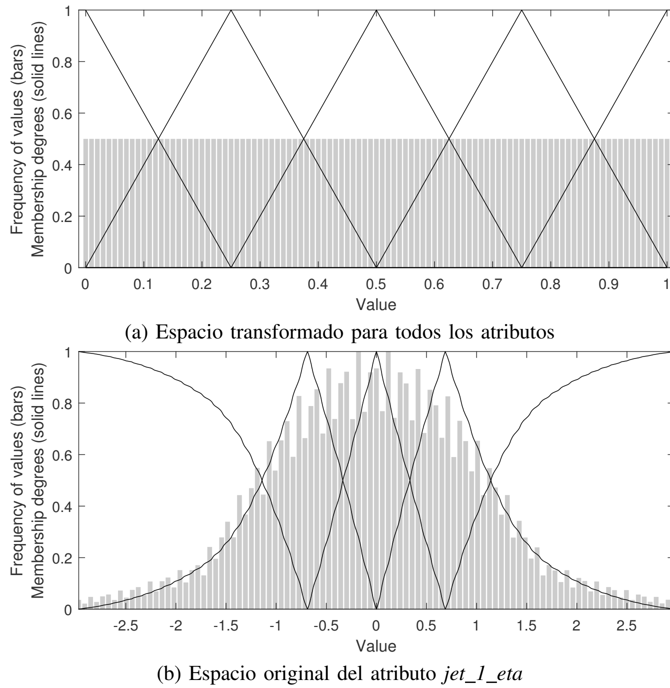
  Este procedimiento se llama la técnica de la FDA y permite que la distribución de Y se derive de la siguiente forma:

  $$F_Y(y) = Prob\,(Y \leq y) = Prob\left(X \leq F_X^{-1}(y)\right) \\ = F_X\left(F_X^{-1}(y)\right) = y$$

  ■

  Sin embargo, dado que la distribución original del conjunto de entrenamiento es desconocida, la FDA no puede ser calculada exactamente. Como solución proponemos calcular los q-cuantiles del conjunto de entrenamiento para obtener una FDA aproximada. Para ello, para cada variable, se ordenan todos los valores y se extrae cada cuantil. Si q es menor que el número de ejemplos en el conjunto de entrenamiento, el valor de la FDA para un valor dado se interpola linealmente en el intervalo [Q_{i-1}, Q_i], siendo Q_i el primer cuantil mayor que el valor. Si el valor es menor que el primer cuantil (Q_1) o mayor que el último cuantil (Q_{q-1}), el valor de la FDA será 0 o 1, respectivamente. De esta forma, las variables del nuevo conjunto de datos transformado seguirán una distribución aproximadamente uniforme independientemente de su distribución original. Por supuesto, la transformación sobre el conjunto de test se realiza interpolando la FDA con los cuantiles extraídos del conjunto de entrenamiento.
- Particionamiento: se construye una partición difusa fuerte de Ruspini [11] distribuyendo uniformemente un número fijo de funciones de pertenencia triangulares a lo largo del intervalo [0,1]. Cabe destacar que la definición de cada uno de los conjuntos difusos en el espacio original puede recuperarse aplicando la función cuantil [15]. En este caso, para cada punto que define la función de pertenencia triangular, el valor correspondiente se interpolaría calculando la inversa de la función lineal empleada para calcular la FDA entre los dos cuantiles más próximos. La Figura 1 muestra un ejemplo ilustrativo de cómo se distribuyen los conjuntos difusos en el espacio original y en el transformado del atributo *jet_1_eta* del conjunto de datos HIGGS. Las líneas sólidas y las barras representan las funciones de pertenencia y la distribución original de las variables, respectivamente.

Nótese que ambas fases (pre-procesamiento y particionamiento) están estrechamente relacionadas. Dado que, bajo nuestro punto de vista, una partición difusa de Ruspini con



(a) Espacio transformado para todos los atributos

(b) Espacio original del atributo *jet_1_eta*

Figura 1: Conjuntos difusos construidos para *jet_1_eta* en HIGGS.

funciones de pertenencia uniformemente distribuidas es una buena forma de modelar una distribución uniforme, nuestra hipótesis es que si es posible transformar la distribución de cualquier atributo en una distribución uniforme y al mismo tiempo poder realizar el proceso inverso, se obtiene una partición auto-adaptada para la distribución original del atributo. El hecho más interesante es que este resultado se obtiene sin necesidad de diseñar un nuevo método de particionamiento específico. El código fuente de nuestra propuesta está escrito en Scala 2.11[2] para Apache Spark 2.0.2 y está disponible en GitHub[3] bajo la licencia GPL.

IV. MARCO EXPERIMENTAL

En esta sección primero describimos los conjuntos de datos y las medidas de rendimiento empleadas para evaluar los métodos considerados en los experimentos (Sección IV-A). Posteriormente detallamos los parámetros y la configuración del entorno utilizados para la ejecución de los algoritmos (Sección IV-B).

IV-A. *Conjuntos de datos y medidas de rendimiento*

Para llevar a cabo el estudio experimental hemos considerado 4 problemas de clasificación Big Data disponibles en los repositorios de UCI [12] y OpenML[4]. La Tabla I muestra la descripción de los conjuntos de datos indicando el número de instancias (#Instancias), atributos (#Atributos) reales (R)/enteros(E)/categóricos(C)/totales(T), y clases (#Clases). Los nombres de BNG Australian (BNG) y HEPMASS (HEPM) han sido abreviados. En todos los experimentos se ha empleado un esquema de validación cruzada estratificada de 5 particiones.

[2]http://www.scala-lang.org/

[3]https://github.com/melkano/uniform-fuzzy-partitioning

[4]https://www.openml.org/search?type=data

Tabla I: Descripción de los conjuntos de datos.

Dataset	#Instancias	#Atributos				#Clases
		R	E	C	T	
BNG	1,000,000	8	6	0	14	2
HEPM	10,500,000	28	0	0	28	2
HIGGS	11,000,000	28	0	0	28	2
SUSY	5,000,000	18	0	0	18	2

El rendimiento en clasificación se ha medido empleando la denominada matriz de confusión (Tabla II), la cual almacena el número de ejemplos clasificados correcta e incorrectamente para cada clase. De esta matriz podemos obtener las siguientes cuatro medidas:

Tabla II: Matriz de confusión para un problema binario.

	Predicción positiva	Predicción negativa
Clase positiva	Verdadero positivo (TP)	Falso Negativo (FN)
Clase negativa	Falso Positivo (FP)	Verdadero Negativo (TN)

- Ratio de verdaderos positivos: porcentaje de ejemplos positivos clasificados correctamente.
  $TP_{rate} = TP \ / \ (TP + FN)$.
- Ratio de verdaderos negativos: porcentaje de ejemplos negativos clasificados correctamente.
  $TN_{rate} = TN \ / \ (TN + FP)$.
- Ratio de falsos positivos: porcentaje de ejemplos negativos clasificados incorrectamente.
  $FP_{rate} = FP \ / \ (FP + TN)$.
- Ratio de falsos negativos: porcentaje de ejemplos positivos clasificados incorrectamente.
  $FN_{rate} = FN \ / \ (FN + TP)$.

Basados en estas métricas, el rendimiento de clasificación de cada método se ha medido empleando el porcentaje de precisión y el Área Bajo la Curva ROC (AUC) definidas como:

$$\%Precision = (TP{+}TN) \ / \ (TP{+}FN{+}FP{+}TN) \quad (4)$$

$$AUC = (1 + TP_{rate} + FP_{rate}) \ / \ 2 \quad (5)$$

IV-B. *Parámetros y configuración del entorno*

En cuanto a los parámetros utilizados para FMDT, hemos fijado los valores sugeridos por los autores en el artículo original [9]:

- Medida para calcular la impureza de los nodos: entropía difusa
- T-norma: producto
- Máximo número de bins para los atributos numéricos: 32

- Máxima profundidad del árbol: 5
- $\gamma = 0.1\,\%$; $\phi = 0.02 \cdot N$; $\lambda = 10^{-4} \cdot N$

Todos los métodos han sido ejecutados en un cluster de 8 nodos conectados a una Red de área Local Ethernet a 1Gb/s. La mitad de estos nodos están compuestos por 2 procesadores Intel Xeon E5-2620 a 2.4 GHz (3.2 GHz con Turbo Boost) con 12 núcleos virtuales en cada uno (de los cuales 6 son físicos). Tres de los nodos restantes están equipados con 2 procesadores Intel Xeon E5-2620 a 2.1 GHz con el mismo número de núcleos que los anteriores. El último nodo es el master, compuesto por un procesador Intel Xeon E5-2609 con 4 núcleos físicos a 2.4 GHz. Todos los nodos esclavos están equipados con 32GB de RAM, mientras que el maestro trabaja con 8GB de RAM. Con respecto al almacenamiento, todos los nodos emplean discos duros con velocidades de lectura/escritura de 128 MB/s. Todo el cluster funciona sobre CentOS 6.5, Apache Hadoop 2.6.0, y Apache Spark 2.0.2.

De acuerdo con los autores de FMDT, emplear más de 24 núcleos de CPU tiene un impacto negativo en el tiempo de ejecución de este método. Por consiguiente, hemos empleado 6 *executors* con 4 núcleos cada uno para ejecutar FMDT.

V. Estudio experimental

Para evaluar el rendimiento de nuestra propuesta hemos llevado a cabo un estudio empírico que cubre tres aspectos: rendimiento en la clasificación (Tabla III), complejidad del modelo (Tabla IV), y tiempo de ejecución (Tabla V). En todos los casos hemos considerado cuatro métodos: el FMDT original [9] y tres configuraciones diferentes del método propuesto que difieren en el número de conjuntos difusos (X) utilizados para los atributos numéricos (denotado como FMDT_X). Cabe señalar que el método FMDT original se quedó sin memoria mientras procesaba HEPMASS debido al gran número de hojas que se generaron a lo largo del entrenamiento, y por tanto no se muestran resultados para este método en HEPMASS.

Las Tablas III y IV revelan que el método de particionamiento difuso propuesto (FMDT_X) es capaz de mantener el rendimiento de clasificación de FMDT con modelos notablemente más simples. Las diferentes configuraciones de nuestra aproximación rinden de forma similar en términos de precisión y AUC (excepto en HIGGS), a pesar de que hay una tendencia positiva a favor del uso de más conjuntos difusos. Sin embargo, emplear más conjuntos difusos suele causar que el algoritmo de aprendizaje genere más hojas, aumentando la complejidad del modelo. A continuación analizamos los resultados obtenidos en cada uno de los conjuntos de datos:

- BNG: nuestro método mejora la precisión y el AUC de FMDT un 6 % y un 8 %, respectivamente. A pesar de que los árboles construidos por FMDT_X son más profundos, éstos tienen entre 8 mil y 80 mil veces menos de hojas que FMDT.
- HEPM: el FMDT original construye demasiadas hojas como para poder afrontar este problema con el cluster descrito en la Sección IV-B, quedándose sin memoria durante la ejecución. Este hecho sugiere que nuestra aproximación es una solución candidata para evitar la explosión del número de hojas durante la inducción de los ADDs.
- HIGGS: el rendimiento de clasificación de FMDT_5 en este problema cae casi un 1 % con respecto al resto de métodos, revelando que 5 conjuntos difusos no son suficientes para capturar la complejidad de este problema. Sin embargo, el resto de configuraciones (FMDT_7 y FMDT_9) son capaces de mantener el rendimiento de clasificación de FMDT con 15 mil y 50 mil hojas, respectivamente, mientras que FMDT genera 6 millones de hojas. Además, el método de particionamiento difuso original genera casi el doble de conjuntos difusos que FMDT_7.
- SUSY: todas las configuraciones rinden de forma similar a FMDT en términos de capacidad de discriminación. Sin embargo, nuestro método construye árboles de 3 mil, 15 mil, y 50 mil hojas, mientras que FMDT genera 5 millones de hojas. En este caso, la diferencia en el número de conjuntos difusos empleados por cada método es aún mayor, ya que FMDT utiliza casi 23 conjuntos de datos por atributo.

Tabla III: Rendimiento de clasificación de cada método.

		%Precision		
Dataset	FMDT	FMDT_5	FMDT_7	FMDT_9
BNG	$80.23^{\pm 0,05}$	$86.79^{\pm 0,06}$	$86.93^{\pm 0,07}$	$86.97^{\pm 0,06}$
HEPM	-	$91.13^{\pm 0,02}$	$91.25^{\pm 0,02}$	$91.33^{\pm 0,02}$
HIGGS	$71.54^{\pm 0,02}$	$70.61^{\pm 0,02}$	$71.32^{\pm 0,03}$	$71.69^{\pm 0,03}$
SUSY	$79.29^{\pm 0,05}$	$79.15^{\pm 0,04}$	$79.49^{\pm 0,04}$	$79.66^{\pm 0,04}$
		AUC		
Dataset	FMDT	FMDT_5	FMDT_7	FMDT_9
BNG	$.7896^{\pm ,0004}$	$.8649^{\pm ,0006}$	$.8658^{\pm ,0007}$	$.8662^{\pm ,0007}$
HEPM	-	$.9113^{\pm ,0002}$	$.9125^{\pm ,0002}$	$.9133^{\pm ,0002}$
HIGGS	$.7143^{\pm ,0001}$	$.7033^{\pm ,0002}$	$.7114^{\pm ,0003}$	$.7155^{\pm ,0003}$
SUSY	$.7859^{\pm ,0004}$	$.7847^{\pm ,0004}$	$.7880^{\pm ,0004}$	$.7898^{\pm ,0004}$

La Tabla V muestra el tiempo requerido por cada método para las tres fases: particionamiento difuso, la inducción del ADD, y el tiempo total. En general, no hay diferencias significativas entre los diferentes métodos cuando se trata de la fase de particionamiento, a pesar de que el algoritmo propuesto es un 30 % más rápido que el método original en SUSY. Sin embargo, cuando se considera la inducción del ADD, la reducción en la complejidad del modelo cuando se emplea nuestro método resulta en tiempos de ejecución mucho más rápidos.

VI. Conclusiones

En este trabajo hemos presentado un nuevo método de particionamiento difuso distribuido que reduce la complejidad del modelo de los árboles de decisión difusos (ADDs) multivía en problemas de clasificación Big Data. El algoritmo

Tabla IV: Complejidad de cada modelo.

Número de hojas				
Dataset	FMDT	$FMDT_5$	$FMDT_7$	$FMDT_9$
BNG	83,044	1,211	4,807	9,492
HEPM	-	2,854	13,472	43,339
HIGGS	6,414,575	3,005	15,876	53,489
SUSY	5,225,134	2,977	14,989	49,038
Profundidad media				
Dataset	FMDT	$FMDT_5$	$FMDT_7$	$FMDT_9$
BNG	3.02	4.67	5.00	4.35
HEPM	-	4.52	4.03	3.93
HIGGS	3.25	5.00	5.00	4.89
SUSY	3.68	5.00	5.00	4.76
Promedio de número de conjuntos difusos				
Dataset	FMDT	$FMDT_5$	$FMDT_7$	$FMDT_9$
BNG	6.04	5.00	7.00	9.00
HEPM	-	5.00	7.00	9.00
HIGGS	13.01	5.00	7.00	9.00
SUSY	22.60	5.00	7.00	9.00

Tabla V: Tiempos de ejecución(s) de cada método.

Particionamiento				
Dataset	FMDT	$FMDT_5$	$FMDT_7$	$FMDT_9$
BNG	58	41	40	40
HEPM	-	295	292	294
HIGGS	252	273	274	276
SUSY	110	77	72	77
Aprendizaje				
Dataset	FMDT	$FMDT_5$	$FMDT_7$	$FMDT_9$
BNG	25	23	22	24
HEPM	-	149	158	153
HIGGS	4,984	176	167	158
SUSY	1,282	76	75	77
Tiempo total				
Dataset	FMDT	$FMDT_5$	$FMDT_7$	$FMDT_9$
BNG	84	65	63	65
HEPM	-	445	450	448
HIGGS	5,238	450	441	435
SUSY	1,392	154	148	155

propuesto consiste en transformar el conjunto de entrenamiento original de tal forma que todas las variables numéricas sigan una distribución uniforme estándar. Para ello se aplica la transformada integral de la probabilidad, la cual afirma que cualquier variable aleatoria continua puede ser convertida en una variable aleatoria uniforme en base a la función de distribución acumulada (FDA) original. Dado que la FDA es generalmente desconocida, proponemos aproximar esta función calculando los q-cuantiles del conjunto de entrenamiento e interpolando linealmente entre dichos cuantiles. Después de esta transformación, se crean las particiones difusas de Ruspini distribuyendo un número fijo de funciones de pertenencia triangulares de forma uniforme a lo largo del intervalo [0,1]. Para recuperar los puntos que definen los conjuntos difusos en el espacio original se aplica la función cuantil. El método de particionamiento propuesto es capaz de ajustar tanto la posición como la forma de los conjuntos difusos a la distribución real del conjunto de entrenamiento.

Para evaluar el rendimiento de nuestra propuesta, hemos realizado un estudio empírico que se centra en el algoritmo de inducción de ADDs multi-vía propuesto por Segatori et al. para Big Data (FMDT). Para ello, hemos reemplazado el método de particionamiento difuso original por el método propuesto, sin modificar la fase de inducción del ADD. Los resultados experimentales revelan que nuestro algoritmo genera árboles significativamente más simples que son capaces de mantener el rendimiento de clasificación del método original.

Agradecimientos

Este trabajo ha sido financiado parcialmente por el Ministerio de Ciencia, Innovación y Universidades de España con el proyecto TIN2016-77356-P.

Referencias

[1] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[2] M.-Y. Chen, “Predicting corporate financial distress based on integration of decision tree classification and logistic regression,” *Expert Systems with Applications*, vol. 38, no. 9, pp. 11 261–11 272, 2011.

[3] C.-C. Yang, S. Prasher, P. Enright, C. Madramootoo, M. Burgess, P. Goel, and I. Callum, “Application of decision tree technology for image classification using remote sensing data,” *Agricultural Systems*, vol. 76, no. 3, pp. 1101–1117, 2003.

[4] D. Che, Q. Liu, K. Rasheed, and X. Tao, “Decision tree and ensemble learning algorithms with their applications in bioinformatics,” *Advances in Experimental Medicine and Biology*, vol. 696, pp. 191–199, 2011.

[5] J. Sanz, D. Paternain, M. Galar, J. Fernandez, D. Reyero, and T. Belzunegui, “A New Survival Status Prediction System for Severe Trauma Patients Based on a Multiple Classifier System,” *Computer Methods and Programs in Biomedicine*, vol. 142, no. C, pp. 1–8, 2017.

[6] L. Zadeh, “Fuzzy sets,” *Information and Control*, vol. 8, no. 3, pp. 338 – 353, 1965.

[7] C. Janikow, “Fuzzy decision trees: Issues and methods,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 28, no. 1, pp. 1–14, 1998.

[8] J. Sanz, H. Bustince, A. Fernández, and F. Herrera, “IIVFDT: Ignorance functions based interval-valued fuzzy decision tree with genetic tuning,” *International Journal of Uncertainty, Fuzziness and Knowlege-Based Systems*, vol. 20, no. SUPPL. 2, pp. 1–30, 2012.

[9] A. Segatori, F. Marcelloni, and W. Pedrycz, “On Distributed Fuzzy Decision Trees for Big Data,” *IEEE Transactions on Fuzzy Systems*, vol. 26, no. 1, pp. 174–192, 2018.

[10] J. E. Angus, “The Probability Integral Transform and Related Results,” *SIAM Review*, vol. 36, no. 4, pp. 652–654, 1994.

[11] E. H. Ruspini, “A new approach to clustering,” *Information and Control*, vol. 15, no. 1, pp. 22–32, 1969.

[12] D. Dheeru and E. Karra Taniskidou, “UCI machine learning repository,” 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[13] H. Ishibuchi, T. Nakashima, and M. Nii, *Classification and Modeling with Linguistic Information Granules: Advanced Approaches to Linguistic Data Mining (Advanced Information Processing)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2004.

[14] M. Zeinalkhani and M. Eftekhari, “Fuzzy partitioning of continuous attributes through discretization methods to construct fuzzy decision tree classifiers,” *Information Sciences*, vol. 278, pp. 715–735, 2014.

[15] N. U. Nair, P. G. Sankaran, and N. Balakrishnan, *Quantile-Based Reliability Analysis*. New York, NY: Springer New York, 2013, ch. Quantile Functions, pp. 1–28.

EDGAR-MR: un algoritmo evolutivo distribuido escalable para la obtención de reglas de clasificación

Miguel Ángel Rodríguez
Departamento de Tecnologías de la Información
Universidad de Huelva
Huelva, España

Antonio Peregrín
Centro de Estudios Avanzados en Física, Matemáticas y Computación
Universidad de Huelva
Huelva, España

Resumen. *El modelo evolutivo para la obtención de reglas de clasificación, EDGAR, cuenta entre sus especificidades con diferentes estrategias para resolver el problema del aprendizaje distribuido de reglas cuando el conjunto de entrenamiento está dividido en diferentes subconjuntos. Sin embargo, el modelo encuentra limitaciones prácticas a su escalabilidad con un número creciente de particiones de datos. Las actuales tecnologías derivadas de los principios de MapReduce en cambio, pueden ayudar a resolver dicha limitación facilitando una implementación más escalable del modelo evolutivo distribuido EDGAR, manteniendo aspectos de su esencia. Es este trabajo se desarrolla una primera aproximación práctica a dicho modelo.*

Palabras clave: Big Data, Algoritmos Genéticos Distribuidos, EDGAR, MapReduce, Clasificación, Programación Distribuida.

I. Introduccion

Actualmente, enormes cantidades de datos son recogidas y almacenadas en bases de datos. El fin último de la Minería de Datos es obtener información de interés en forma de modelos precisos y entendibles de estas bases de datos. Dentro de esta disciplina, los sistemas de clasificación basados en reglas son un instrumento tradicional de gran utilidad. La extracción de reglas de un conjunto de datos depende en gran medida de la topología de los datos y del volumen de éstos. Adicionalmente, a medida que el volumen de datos crece, su manejo es más complejo, el tiempo necesario para su tratamiento se incrementa de manera exponencial, y además crece la dificultad de aprendizaje del algoritmo.

Los algoritmos evolutivos han demostrado una gran capacidad como recurso para extraer conocimiento, siendo robustos al ruido y a otras características inherentes a los datos; sin embargo, suele ser difícil escalar un algoritmo evolutivo eficientemente, debido al cálculo reiterativo de la bondad de los individuos que implica la evaluación de un conjunto de reglas sobre el conjunto de datos.

En este sentido, el algoritmo evolutivo distribuido para obtener reglas de clasificación, EDGAR [1], propone un sistema de distribución de poblaciones y repartición de datos que permite escalar en varias magnitudes el número de instancias que es capaz de manejar eficientemente un algoritmo evolutivo de sus características. No sólo eso, EDGAR también emplea otras estrategias dirigidas al manejo de conjuntos de datos con clases no balanceadas sin preprocesamiento específico. Sin embargo, existen límites inherentes a la arquitectura que utiliza, las cuales se pueden confirmar en la práctica.

En los últimos años, la aparición del paradigma *MapReduce* [2] y el sistema de almacenamiento HDFS [3], ha impulsado relevantes mejoras en la escalabilidad. No obstante, no todos los algoritmos son directamente aptos para implementarse directamente de forma equivalente, y por tanto, en tales casos es necesario establecer estrategias que conserven la calidad de los resultados y mantengan el tiempo de ejecución a niveles competitivos en relación a otras alternativas distribuidas.

Este trabajo propone un algoritmo evolutivo escalable para la obtención de reglas de clasificación basado en los principios de EDGAR, implementado sobre el paradigma *MapReduce*, y realiza una experimentación preliminar para comprobar la validez los conceptos propuestos.

La organización de este trabajo es la siguiente: en la primera Sección se introduce el esquema general del algoritmo evolutivo EDGAR. La Sección 2.1 se dedica a exponer las alternativas de implementación del modelo distribuido sobre *MapReduce*. La Sección 3 se centra específicamente en la adaptación de las estrategias para conjuntos de datos no balanceados. Finalmente, la Sección 4 analiza los resultados de precisión, calidad y escalabilidad sobre una experimentación preliminar con conjuntos de datos de diversas características.

II. Antecedentes

En esta Sección se repasan, para el aprendizaje de clasificadores, en primer lugar, las principales referencias en el área de los algoritmos genéticos distribuidos, y en segundo lugar, las propuestas que emplean *MapReduce*.

A. *Modelos evolutivos distribuidos para el aprendizaje de clasificadores*

Una de las primeras referencias reseñables en este ámbito es REGAL [4]. Éste propone una división de datos en nodos que contienen algoritmos genéticos (AGs) para el aprendizaje de reglas. Posteriormente refina dichas reglas asignando cada una de ellas y sus datos asociados a diferentes nodos. REGAL-TC [5] es una propuesta que mejora la precisión de REGAL a través de un ponderado de contra-ejemplos y otras estrategias. Now-GNet [6] por su parte, propone una mejora sobre el mismo esquema utilizando un buffer de reglas a evaluar que separa las

funciones de evaluación del individuo y del AG, lo cual permite un mayor grado de paralelización. Estos tres algoritmos tienen en común la presencia de un proceso supervisor síncrono que en base a los resultados parciales de los nodos redistribuye datos y reglas a los nodos subordinados.

EDGAR utiliza un modelo mixto de nodos *aprendedores* independientes, que emplean un AG local (AGL), con un subconjunto de los datos y supervisor central. El sistema es asíncrono, utilizando una copia del conjunto global de datos para evaluar las reglas generadas por los AGLs y generar un clasificador final basado en la cobertura y calidad de ellas sobre el conjunto de datos de entrenamiento (Figura 1). El AGL utiliza una codificación de tipo *un individuo = una regla*, dentro del paradigma GCCL; la población representa un conjunto redundante de reglas y la solución del mismo es un clasificador compuesto por un subconjunto de las mismas que clasifica al conjunto de los datos asignados.

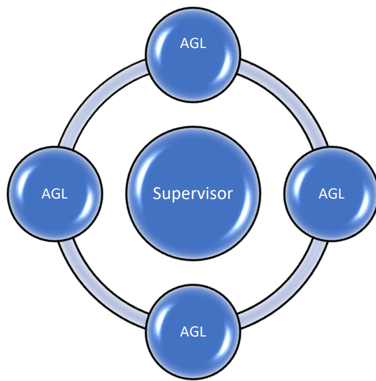


Fig. 1. Modelo distribuido de EDGAR: poblaciones de algoritmos geneticos y datos con particiones en nodos y nodo central con conjunto completo

La regla se representa como una cadena binaria donde cada posible valor de un atributo está asociado a un bit. Esta representación puede tener varios valores activos en cada atributo, consiguiendo un lenguaje de descripción de conceptos compacto, pero por otra parte requiere que los conjuntos de datos continuos sean discretizados previamente. La clase también está representada en el cromosoma con un único valor a la vez (Figura 2).

c_1			c_2		c_3			*Clase*	
v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
1	0	1	0	0	1	0	0	0	1

si c_1 en (v_1,v_3) y c_3 en (v_1) entonces clase es v_2

Fig. 2. Ejemplo de representación de regla en EDGAR a formato binario

Las mejores reglas, por calidad y cobertura, se intercambian periódicamente entre nodos para favorecer la cooperación entre los AGLs de modo similar al modelo de AGs distribuidos basado en Islas [9] y se envian a un nodo central que hace las veces de supervisor. Este nodo dispone de una copia del conjunto global de datos que utiliza para reevaluar las reglas recibidas y generar un clasificador final basado en la cobertura y calidad de ellas sobre el conjunto completo de datos de entrenamiento. El supervisor manda una señal de terminación a los nodos AGL cuando el clasificador generado no mejora durante un periodo de tiempo y genera el clasificador como una lista ordenada en base al valor de las reglas sobre el conjunto completo de datos.

B. MapReduce en el ámbito del aprendizaje de clasificadores

MapReduce [1] permite un modelo de diseño paralelo basado en la división de datos. Este modelo se basa en la existencia de un conjunto de datos distribuidos que permiten una alta escalabilidad de datos y la codificación de tareas sobre estos basados fundamentalmente en dos operaciones *Map* y *Reduce*.

Centrándonos en el aprendizaje de modelos de clasificación, hay técnicas que se prestan mejor que otras para este paradigma porque no necesitan de un acceso repetido a los datos. En este sentido, la generación de clasificadores con *Random Forest* [7] implementado bajo *MapReduce* realiza la evaluación de múltiples árboles de decisión eficientemente. Un ejemplo reciente de este uso puede verse en [8].

Otro algoritmo directamente compatible con esta arquitectura es el de generación de reglas basadas en ejemplos como semillas que posteriormente formarán el clasificador. En [10] se generan reglas en la fase *Map* en base a un conjunto de etiquetas lingüísticas que son evaluadas en la fase de construcción y posteriormente seleccionadas en la fase *Reduce* para generar el conjunto de reglas final teniendo en cuenta el coste de la clasificación en caso de existir clases no balanceadas.

III. El Modelo EDGAR-MR

En este apartado proponemos una versión del modelo EDGAR sobre el paradigma *MapReduce*. Esta versión, a la que denominamos EDGAR-MR, aprovecha las mejoras en eficiencia, robustez y facilidad de ejecución con conjuntos de datos de alta cardinalidad que proporciona *MapReduce* manteniendo un modelo de calidad y precisión en conjuntos de datos complejos.

Como se ha indicado anteriormente, EDGAR lleva a cabo el aprendizaje de reglas empleando un conjunto de AGLs, en lo que cada uno de ellos trabaja sobre una fracción del conjunto completo de datos. Este modelo de datos distribuido es parcialmente compatible con el paradigma *MapReduce*, ya que el aprendizaje de reglas está asociado a las particiones de datos, y la ulterior extracción del clasificador final es una tarea independiente. Sin embargo, EDGAR emplea mecanismos cuya adaptación al modelo *MapReduce*, por los principios de éste, no pueden llevarse cabo, como son por ejemplo aquellos relacionados con la cooperación entre AGLs basado en el intercambio de reglas.

En el resto de esta Sección pues, se va a describir el diseño concreto empleado para el modelo EDGAR-MR.

Las fases principales de EDGAR-MR (fig. 3) son:

- Inicial, carga de datos: Preparación de conjuntos de datos para iniciar el proceso distribuido.
- Fase de aprendizaje de reglas: Es llevada a cabo por los AGLs implementados en los *Map*, y en ella, cada uno aprende un clasificador local basado en su partición de datos.
- Fase de agregación de reglas: Las reglas provenientes de los *Map* son procesadas en nodos *Reduce* para generar una única regla que agrega los casos positivos y negativos encontrados en los *Map*.
- Fase de generación del clasificador: el clasificador se genera como una lista ordenada de reglas por mayor cobertura y calidad.
- Fase de *test*: se clasifican los conjuntos de *test* con el clasificador generado para tener una medida de precisión del mismo.

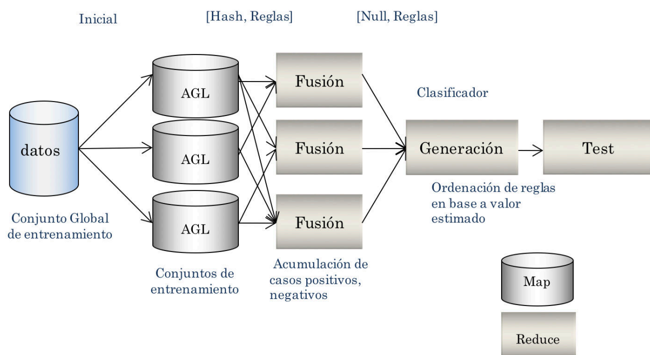


Fig. 3. Esquema *MapReduce* de las fases de aprendizaje, agregación, generacion de clasificador y *test*

A continuación se describe en detalle en cada una de las fases :

A. *Inicial, carga de datos*

El proceso de aprendizaje requiere una partición de los datos en los nodos *Map* en bloques HDFS independientes que son replicados y transferidos a otras máquinas para ser procesados por cada tarea *Map* independiente. Esta partición es realizada mediante un muestreo aleatorio de cada clase teniendo en cuenta equilibrar las clases en los AGL en caso de existir desbalanceo (figura 2). Otro paso de esta preparación inicial consiste en la recodificación en el formato de cromosoma binario que será utilizado por los AGL en los *Map*.

B. *Fase de aprendizaje de reglas (Map)*

Cada AGL genera un conjunto de de reglas que cubre completamente el conjunto de los datos en el nodo. La salida del *Map* son conjuntos (*Key, Value*), donde *Value* corresponde a la regla concatenada con el número de casos positivos y negativos que cubre y *Key* contiene un valor generado mediante una función hash para repartir la carga sobre los nodos *Reduce*. Esta función utiliza una codificación sobre los antecedentes de la regla para reunir en el mismo *Reduce* las reglas con los mismos antecedentes. La función hash codifica en un número en base 10 la codificación binaria de la clase. Por ejemplo, en un conjunto de datos con 3 atributos de tres valores cada uno y dos clases, EDGAR usa 11 bits para su codificación; por tanto el rango de la clave se encuentra entre 20 y 211. Si se implementa con 2 nodos *Reduce*. Las reglas con clave entre 0 y 210 serán asignados al primero y las claves entre 210 y 211 se asignan al segundo. El pseudocódigo 1 muestra el detalle de este *Map*.

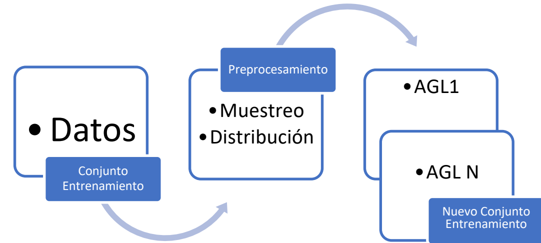


Fig. 4. Esquema de rebalanceo inicial de clases en fase de carga inicial

```
Input (Key, Value)
Key, Valor no utilizados, Map inicial.
AGL.generarClasificador(ConjuntoDatosLocal)
Para cada Regla en el clasificador
        Key' : Valor decimal (Regla)
Valor: (Regla, casos positivos, casos negativos)
Emit (Key', Valor)
Fin Para
```

Pseudocódigo 1. Fase de aprendizaje de reglas (*Map*)

C. *Fase de agregación de reglas (Reduce)*

Este proceso combina los conjuntos de reglas recibidos de cada *Map*, fusionando las reglas con los mismos antecedentes. El proceso de fusión es como sigue: acumula los casos positivos y negativos de las reglas con los mismos antecedentes para cada clase y crea una nueva regla fusionada. La nueva regla tendrá la clase de la regla con mayor valor ponderado de cobertura y calidad, denominado Π (1). Éste criterio minimiza la longitud de la regla y cobertura de casos negativos y maximiza la cobertura de casos positivos.

$$\Pi = (1+ longitud^{-1})^{-CasosNeg} * CasosPos \quad (1)$$

```
Input (Key, Values)
        Key es una clave hash calculada en base a la regla
        Values: las reglas generadas por los AGLs en los Map
ArrayList auxReglas= new ArrayList();
Para cada Regla en values
        Si (Regla not in auxReglas) auxReglas.add(Regla)
        Sino
            AuxReglas.get(Regla).AcumulaCasos(Regla)
        Fin Si
Fin Para
Para cada Regla en auxReglas
        Emit (null, auxReglas)
Fin Para
```

Pseudocódigo 2. *Reduce*, aprendizaje. Fusión de reglas

La salida del proceso *Reduce* es un conjunto de reglas con información necesaria para el proceso de generación del clasificador: casos positivos, casos negativos (pseudocódigo 2).

D. Fase de generación del clasificador:

El clasificador producido EDGAR originalmente, es generado mediante la evaluación de las reglas sobre el conjunto completo de datos de entrenamiento y la posterior ordenación en una lista ordenada donde cada regla elimina las instancias cubiertas por las predecesoras.

Sin embargo, en la propuesta de este trabajo, la agregación de reglas aprendidas sobre modelos locales puede no tener la misma validez sobre el conjunto completo de datos, por ello se han implementado dos variantes dependiendo del método de cálculo del valor de la regla:

- Estimado: toma como valor de ordenación para la generación del clasificador los casos positivos y negativos calculados por la fase de agregación de reglas.
- Evaluación global: efectúa una re-evaluación de las reglas generadas en la fase de agregación de reglas mediante otro ciclo *MapReduce* con el conjunto de datos preexistente en cada *Map* (pseudocódigo 3). La salida de los *Map* será acumulada en un *Reduce* que tendrá como resultado una lista de reglas con valores positivos y negativos relativos al conjunto completo de datos (fig. 5).

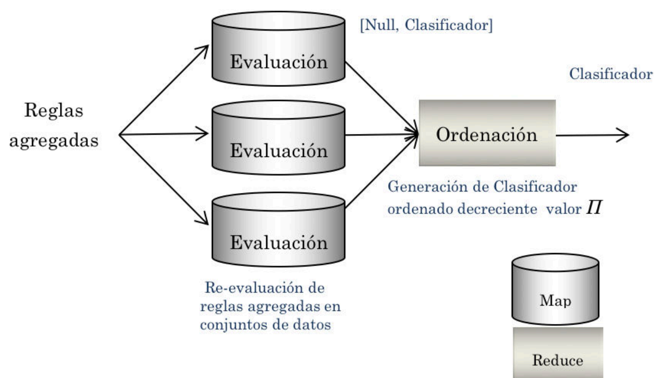


Fig. 5. Esquema *MapReduce* de la re-evaluación de reglas en variante de evaluación global en la fase de generación del clasificador

Finalmente se genera un clasificador como una lista ordenada de reglas en base al valor decreciente de Π (1).

```
Input (Key, Value)
Value es conjunto de reglas generado por el Reduce de agregación de reglas
ArrayList auxReglas = new ArrayList()
auxReglas, auxpi = GenerarClasificadorGredy(Values, Datos de test Local)
posición=0;
Para cada Regla en auxReglas
   posición ++
   Valor = Regla + Auxpi + casos positivos y negativos + posición
Fin Para
```

Pseudocódigo 3. Map de fase de evaluación global

E. Fase de test

Habitualmente se siguen esquemas de aprendizaje que reservan un 10% o un 20% del tamaño del conjunto de datos para *test*, en alguna de las configuraciones propuestas para minimizar la fractura de datos en el proceso de aprendizaje. En conjuntos de datos grandes, la medida de la precisión del clasificador generado implica en si misma la asignación de grupos de datos siguiendo un esquema de distribuido de datos y por tanto la ejecución de un nuevo ciclo *MapReduce*.

Inicialmente se carga en los nodos el conjunto de *test*. El *Map* se encarga de aplicar el clasificador generado sobre cada ejemplo generando como salida una línea para cada instancia con el valor real y el predicho (Fig. 6). El proceso *Reduce* acumula los aciertos y fallos en cada ejemplo respecto de la clase predicha generando un informe que puede ser utilizado para realizar distintas variantes de cálculo de la precisión sobre clasificadores.

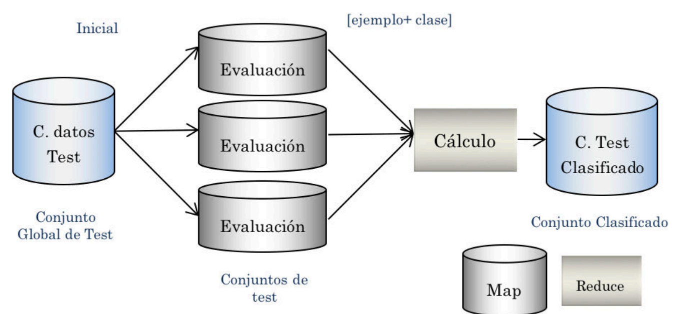


Fig. 6. Cálculo de la precisión en fase *test*

IV. ESTUDIO EXPERIMENTAL

Esta sección muestra una experimentación tentativa sobre un conjunto de *datasets* representativos para valorar la validez de la propuesta y un algoritmo de referencia no distribuido. Se estimará la calidad del clasificador generado en base al la interpretabilidad del mismo (número de reglas) y precisión (media geométrica). Asimismo se mostrará la escalabilidad del mismo según el rendimiento alcanzado respecto del número de *Maps*.

A. Entorno de Experimentación

El entorno de experimentación se basa en un cluster Hadoop compuesto por 13 nodos. Cada nodo es una máquina virtual sobre un servidor basado en memoria común y multiprocesador: 192GB de memoria RAM, 40 núcleos con *hyperthreading* y sistema operativo CentOS 7 de 64 bits. Todo el desarrollo se ha realizado en Java. Las características de las máquinas virtuales son:

Nodo maestro:

- Sistema operativo CentOS 7 64 bits, y 8 núcleos.
- 16 GB de memoria RAM.
- 20 GB de almacenamiento en disco configurado en modo dinámico.

Nodos esclavos:

- Sistema operativo CentOS 7 64 bits, y 4 núcleos.
- 12 GB de memoria RAM.
- 20 GB de almacenamiento en disco configurado en modo dinámico.

Respecto de los conjuntos de prueba se han seleccionado *datasets* balanceados y no balanceados con distintas cardinalidades y dimensionalidades. Éstos se encuentran disponibles en los repositorios UCI [12] y KEEL [13]. Los conjunto de datos con atributos continuos han sido discretizados para EDGAR-MR usando Chi2-Merge [14]. Las tablas muestran valores medios de ejecuciones con 10 semillas y 5x2 particiones de datos.

TABLA I. CARACTERÍSTICAS DE LOS CONJUNTO DE DATOS

Nombre	Instancias	Atributos	IR
Mushroom	8124	23	0.93
Page-blocks0	5472	10	8.77
Segment0	2308	19	6.01
Yeast3	1484	8	8.11

Los algoritmos genéticos utilizados como aprendedores en los *Map* (AGLs) siguen la estructura de operadores de EDGAR con la siguiente configuración:

- Probabilidad de Mutación = 0.01
- Tamaño de población = 40
- Número de iteraciones de parada = 500

B. *Análisis*

En la tabla II se observa que la precisión en *test* con media geométrica de las dos modalidades de aprendizaje, con la opción de balanceo de clases a los nodos mediante redistribución aleatoria de las instancias, sin balanceo y con aplicación del método de balaceo SMOTE [11] respecto del algoritmo de referencia C4.5 para una distribución de 12 nodos. El acrónimo NA (*No Aplica*) en la columna del conjunto de datos Mushroom refleja el hecho de que no se aplica el método de rebalanceo por tratarse de un dataset balanceado en origen. Para facilitar la visualización de los datos se ha sombreado levemente la fila correspondiente a la variante EDGAR-MR Global en las tablas II a IV.

En relación de la comparación entre las dos variantes, EDGAR-MR Estimado y Edgar-MR Global, se aprecia que la precisión de la variante con evaluación global alcanza en general las mismas cotas de precisión que el estimado. Acerca de la eficacia del método de rebalanceo, se observa que si bien la aplicación previa de balanceado con SMOTE, permite que EDGAR-MR alcance mejores niveles de precisión (muy apreciable en la precisión alcanzada sobre *Yeast3* para la variante global), el método de redistribución puede ser una alternativa válida, que para algunos conjunto de datos, puede llegar a superar al estándar de facto en el preprocesamiento para desbalanceadas, SMOTE. Se puede observar este caso en las precisiones alcanzadas sobre *Page-bloks0*, que son superadas por la variante de rebalanceo mediante la distribución balanceada de clases a los nodos (redistribución).

Mushroom es típicamente un conjunto de datos robusto frente al particionamiento de datos para su aprendizaje, o dicho de otra forma no tiene un gran riesgo de fractura de datos por la división de los mismos previa al proceso de aprendizaje. Sin embargo, se observa una diferencia apreciable entre las variantes estimada y global en el número de reglas sobre este conjunto de datos, aunque la precisión sea similar en ambos. Esto puede ser debido a la representación del mismo concepto en diferentes Maps por reglas equivalentes, que en la variante global elimina, eligiendo sólo la mejor representante del concepto y eliminando las reglas parcial o totalmente redundantes.

TABLA II. PRECISIÓN COMPARADA EN MEDIA GEOMÉTRICA

Algoritmo	Balanceo	Mushroom	Page-blocks0	Segment	Yeast3
EDGAR- MR Estimado	Sin Banlanceo	99,8	80.84	96.55	88.97
	SMOTE	NA	83.96	97.85	88.00
	Redistribución	NA	86.39	97.02	91.01
EDGAR- MR Global	Sin Balanceo	99,9	76.08	97.23	52.07
	SMOTE	NA	81.62	98.59	87.15
	Redistribución	NA	84.96	97.14	91.30
C4.5	Sin	100	97.33	99.18	94.14
	SMOTE	NA	94.33	99.22	93.53

Relativo a la comparación con otros algoritmos de clasificación, se comprueba que el algoritmo alcanza niveles próximos a al algoritmo de referencia C4.5 , sin llegar a superarlo. Probablemente el uso de conjunto de datos continuos no sea el más apropiado para un algoritmo de representación discreta y la discretización pueda afectar a la calidad del mismo.

En cuanto al número de reglas se puede observar en la tabla III, que el número de reglas generada por la variante global es claramente inferior al de la variante estimada, resultando en un clasificador más compacto. El número de reglas no se ve afectado de manera general por el método de desbalanceo, aunque se aprecia una ligera mejora del numero de reglas en el balanceo por redistribución frente a los otros.

TABLA III. NÚMERO DE REGLAS DEL CLASIFICADOR

Algoritmo	Balanceo	Mushroom	Page-blocks0	Segment	Yeast3
EDGAR-MR Estimado	Sin Balanceo	20	530	316	240
	SMOTE	NA	559	320	220
	Replica	NA	510	285	230
EDGAR-MR Global	Sin Balanceo	15	142	46	52
	SMOTE	NA	150	43	56
	Réplica	NA	144	45	49

Finalmente , en cuanto a la escalabilidad, se puede observar en la tabla IV, que existe una mejora o *speed-up* (ratio tiempo proceso original–paralelo) que dista mucho del ideal, 1/#nodos. Los tiempos indican minutos por cada configuración de nodos.

En el detalle de las ejecuciones observamos que la mayor parte del tiempo de proceso (en torno a un 70%) se consume en las tareas propias de preparación de datos, codificación en el formato de instancias y aquellos procesos en los reduce, y que en esta experimentación es un proceso único: agregación y fusión de reglas, generación de clasificador global y generación de los valores de *test* con el clasificador, siendo este tiempo similar en todas las configuraciones.

TABLA IV. TIEMPOS MEDIOS POR EJECUCIÓN

Algoritmo	#Nodos	Mushroom	Page-blocks0	Segment	Yeast3
EDGAR-MR Estimado	4	5,30	7,34	5,30	4,23
	8	4,45	6,68	4,45	3,81
	12	3,87	6,23	3,87	3,24
EDGAR-MR Global	4	5,81	7,71	5,81	4,65
	8	5,03	6,75	5,03	4,28
	12	4,34	6,34	4,34	4,02

V. CONCLUSIONES

EDGAR propuso un método eficiente para trabajar sobre conjuntos de datos particionados mediante la colaboración de algoritmos genéticos y un clasificador compacto basado en una lista ordenada validada sobre un conjunto de datos global. Sin embargo, con un número de particiones elevado disminuye la influencia de la comunicación en tiempo de aprendizaje. Asimismo la validación central, aunque de menor entidad algorítmica, supone un cuello de botella directamente ligado a la capacidad de la memoria del nodo central para albergar el conjunto completo de datos. Esta propuesta mejora la escalabilidad del modelo mediante un proceso de evaluación escalable mientras conserva el algoritmo genético de aprendizaje del original. Por otro lado no implementa la comunicación entre nodos incoherente con la implementación *MapReduce* seguida e ineficiente con un numero creciente de particiones.

Este trabajo presenta una arquitectura novedosa para la implementación de algoritmos evolutivos de clasificación basados en reglas, en el que el antecedente de la regla aprendida en distintas particiones forma una clave que permitirá su posterior fusión. Se propone asimismo un proceso de reevaluación que aprovecha los datos ya particionados para mejorar la calidad del clasificador.

Los resultados muestran una escalabilidad creciente, aunque mejorable, con el uso de un mayor numero de fases *Reduce* en las etapas de agregación. La comparación con un algoritmo de referencia parecen mostrar que la discretización afecta a la precisión del clasificador, siendo más adecuado para conjunto de datos nominales, por lo que en futuros trabajos se planteará la posibilidad de trabajar con datos continuos en los nodos para mejorar su precisión.

AGRADECIMIENTOS

Este trabajo ha sido financiado por el Ministerio de Educación y Ciencia dentro del Proyecto TIN2017-89517-P.

REFERENCIAS

[1] Rodríguez M., Escalante D. M., Peregrín A. (2011). Efficient distributed genetic algorithm for rule extraction. Applied soft computing, 11(1), 733-743.

[2] Dean J., Ghemawat S. (2008): MapReduce: Simplified Data Processing on Large Clusters. Commun ACM 51, 107–113.

[3] White T.(2009): Hadoop: The Definitive Guide. 1st ed.Sebastopol, CA: O'Reilly .

[4] Giordana A., Saitta L. (1994). Learning disjunctive concepts by means of genetic algorithms. Proceedings of the International Conference on Machine Learning, 96-104.

[5] Lopez L. I., Bardallo J. M., De Vega M. A., Peregrin A. (2011). REGAL-TC: a distributed genetic algorithm for concept learning based on REGAL and the treatment of counterexamples. Soft Computing, 15(7), 1389-1403.

[6] Anglano C., Botta M. (2002). NOW G-Net: learning classification programs on networks of workstations. IEEE Transactions on Evolutionary Computation, 6(5), 463-480.

[7] Svetnik V., Liaw A., Tong C., Culberson J. C., Sheridan, R. P., Feuston, B. P. (2003). Random forest: a classification and regression tool for compound classification and QSAR modeling. Journal of chemical information and computer sciences, 43(6), 1947-1958.

[8] Río S., López S., Benítez J.M., Herrera F. (2014): On The Use of MapReduce for Imbalanced Big Data using Random Forest. Information Sciences 285 112-137.

[9] Cantú-Paz E. (1998). A survey of parallel genetic algorithms. Calculateurs paralleles, reseaux et systems repartis, 10(2), 141-171.

[10] Peralta D., del Río S., Ramírez-Gallego S., Triguero I., Benitez J. M., Herrera F. (2015). Evolutionary feature selection for big data classification: A MapReduce approach. Mathematical Problems in Engineering.

[11] Chawla N. V., Bowyer K. W., Hall L. O., Kegelmeyer W. P. (2002). SMOTE: synthetic minority over-sampling technique. Journal of artificial intelligence research, 16, 321-357.

[12] Asuncion A., Newman D. (2007). UCI machine learning repository.

[13] Alcalá-Fdez J., Fernández A., Luengo J., Derrac J., García S., Sánchez L., Herrera F. (2011). Keel data-mining software tool: data set repository, integration of algorithms and experimental analysis framework. Journal of Multiple-Valued Logic & Soft Computing, 17.

[14] Liu H., Setiono R. (1995). Chi2: Feature selection and discretization of numeric attributes. In Tools with artificial intelligence, 1995. proceedings., seventh international conference on IEEE, 388-391.