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 ≠ ∅))
```

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 ≠ ∅))
```

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.

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		ENN
Voto				Mayor.	Consen.	
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	Ruido	Original	EHO	EHT		ENN
Voto				Mayor.	Consen.	
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	Ruido	EHO	EHT		ENN
Voto			Mayoría	Consenso	
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	EHO	EHT		ENN
Voto		Mayoría	Consenso	
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.