Optimización del Sistema de Clasificación Basado en Reglas Difusas Chi-FRBCS-BigDataCS

Mikel Elkano[1], Mikel Galar[1], José Sanz[1], and Humberto Bustince[1]

Dpto. de Automática y Computación, Universidad Pública de Navarra, Campus Arrosadía s/n, 31006 Pamplona, España,
{mikel.elkano, mikel.galar, joseantonio.sanz, bustince}@unavarra.es

Resumen. En este trabajo presentamos una optimización del único Sistema de Clasificación Basado en Reglas Difusas que es capaz de afrontar problemas Big Data hasta la fecha, esto es, Chi-FRBCS-BigDataCS. Nuestra propuesta está basada en la utilización de *Look-Up-Tables* para pre-calcular los grados de pertenencia a las diferentes etiquetas lingüísticas y ahorrar así millones de cálculos. Además, proponemos varias modificaciones que permiten explotar el potencial de Apache Hadoop, agilizando los diferentes procesos de búsqueda y ordenación llevados a cabo por el algoritmo. Todas estas adaptaciones han mostrado una mejora significativa en tiempos de ejecución sin alterar el rendimiento del método.

Palabras Clave: Sistemas de Clasificación Basado en Reglas Difusas, Big Data, Apache Hadoop, MapReduce, Datasets imbalanceados

1 Introducción

Uno de los problemas que nos encontramos en Big Data [1] es la *clasificación*, donde un algoritmo extrae conocimiento de un conjunto de datos para predecir futuros patrones. Entre las técnicas más aplicadas en este tipo de tareas están los Sistemas de Clasificación Basado en Reglas Difusas (SCBRDs). Este tipo de sistemas representan el conocimiento mediante reglas que contienen etiquetas lingüísticas interpretables por el ser humano [5]. El primer y único SCBRD que es capaz de afrontar problemas Big Data hasta la fecha es Chi-FRBCS-BigDataCS [8]. Este clasificador está basado en el algoritmo de Chi et al. [2] y consiste en el aprendizaje de varios clasificadores Chi cuyas bases de reglas son agregadas al final. Para el despliegue del sistema distribuido, Chi-FRBCS-BigDataCS emplea *Apache Hadoop*. Debido a la filosofía *divide y vencerás* que se emplea en Big Data, la fusión de información y las técnicas de *ensembles* cobran una gran importancia.

En este trabajo proponemos una serie de optimizaciones del algoritmo Chi-FRBCS-BigDataCS con el fin de agilizar la ejecución del mismo sin alterar el modelo obtenido (y por tanto los resultados). La principal adaptación que presentamos es la utilización de *Look-Up-Tables* para pre-calcular los grados de pertenencia a las diferentes etiquetas lingüísticas. De esta forma, logramos evitar en gran medida que los diferentes cálculos que se realizan a lo largo de la ejecución del algoritmo se repitan. No obstante, para que podamos emplear Look-Up-Tables, el método Chi-FRBCS-BigDataCS original debe ser re-diseñado. Para ello, proponemos también varias modificaciones que permiten además explotar el potencial de Apache Hadoop.

Para evaluar la mejora obtenida por medio de nuestra optimización, hemos empleado 15 conjuntos de datos obtenidos del repositorio UCI [7]. Dado que las adaptaciones presentadas no alteran el modelo generado (base de reglas), en este estudio experimental nos centramos en analizar las diferencias presentadas en los tiempos de ejecución. Como veremos, el nuevo diseño del algoritmo permite ejecuciones significativamente más rápidas.

La estructura del trabajo es la siguiente. En las Secciones 2-4 repasamos los conceptos básicos de Apache Hadoop, de los SCBRDs, y del método Chi-FRBCS-BigDataCS original, respectivamente. En la Sección 5 mostramos detalladamente nuestra propuesta centrándonos en cada una de las modificaciones realizadas. La Sección 6 contiene la descripción del marco experimental y el análisis de los resultados obtenidos. Finalmente, en la Sección 7 mostramos las conclusiones del trabajo.

2 Apache Hadoop

Apache Hadoop es uno de los frameworks más empleados en entornos Big Data. Este framework proporciona una implementación libre del algoritmo *MapReduce* [3], compuesto por las siguientes fases.

1. *Fase Map*: los datos de entrada se dividen primero en múltiples fragmentos lógicos asociados a diferentes bloques físicos. De esta forma, cada fragmento será procesado por una sola unidad de procesamiento llamada *mapper*. Cada nodo puede ejecutar múltiples mappers de manera concurrente. Durante el procesamiento llevado a cabo en el mapper, los datos de entrada se transforman en pares clave-valor <*k*, *v*>que son procesados en la función *map()* (definida por el usuario), la cual se invoca para cada par. El resultado de esta función es otro par clave-valor <*k*', *v*'>que forma parte de los denominados *datos intermedios*. Finalmente, los datos intermedios son preparados para ser enviados a la siguiente fase mediante los siguientes pasos:
   (a) *Sorting and Merging*: las salidas se ordenan respecto a las claves y se genera una lista para cada clave que contiene todos sus valores (<*k*', list (*v*')>).
   (b) *Partitioning*: se asigna un reducer a cada clave.
   (c) *Shuffle*: los datos intermedios se copian a los reducers.
2. *Fase Reduce*: el reducer se encarga de agregar las salidas de los mappers. Para ello, primero se ordenan todos los pares clave-valor recibidos respecto a las claves. Una vez finalizan todos los mappers y todos los pares clave-valor han sido ordenados, se invoca la función *reduce()* (definida por el usuario) para cada clave (*k*') y se agregan todos sus valores (list (*v*')). Finalmente, el reducer devuelve el resultado final (*v*") para cada clave.

3 Algoritmo Chi et al.

Una de las herramientas más populares para resolver problemas de clasificación son los Sistemas de Clasificación Basado en Reglas Difusas (SCBRDs). Estas técnicas proporcionan un modelo expresado mediante reglas que contienen etiquetas lingüísticas interpretables por el ser humano [5].

Para generar la base de reglas, se aplica un algoritmo de aprendizaje empleando un conjunto de entrenamiento $\mathscr{D}_T$ compuesto por P ejemplos etiquetados $x_p = (x_{p1}, \ldots, x_{pn})$ con $p \in \{1, \ldots, P\}$, donde x_{pi} es el valor del i-ésimo atributo ($i \in \{1, 2, \ldots, n\}$) del ejemplo p-ésimo. Cada ejemplo pertenece a una clase $y_p \in \mathbb{C} = \{C_1, C_2, ..., C_m\}$, donde m es el número de clases del problema. La estructura de las reglas empleadas en Chi es la siguiente:

$$\text{Regla } R_j: \text{ Si } x_1 \text{ es } A_{j1} \text{ y } \ldots \text{ y } x_n \text{ es } A_{jn} \text{ entonces Clase} = C_j \text{ con } RW_j \qquad (1)$$

donde R_j es la etiqueta de la regla j-ésima, $x = (x_1, \ldots, x_n)$ es un vector n-dimensional que representa el ejemplo, A_{ji} es una etiqueta lingüística modelada por una función de pertenencia triangular, C_j es la etiqueta de la clase y RW_j es el peso de la regla. En el caso de conjuntos de datos imbalanceados [4] (como es el caso de este trabajo), calculamos el peso de la regla utilizando el *Penalized Cost-Sensitive Certainty Factor* (PCF-CS) [8], que se trata de una modificación del *Penalized Certainty Factor* (PFC) [6]:

$$RW_j = PCF\text{-}CS = \frac{\sum_{x_p \in Class C_j} \mu_{A_j}(x_p) \cdot Cs(y_p) - \sum_{x_p \notin Class C_j} \mu_{A_j}(x_p) \cdot Cs(y_p)}{\sum_{p=1}^{P} \mu_{A_j}(x_p) \cdot Cs(y_p)} \qquad (2)$$

donde $Cs(y_p)$ es el coste asociado a la clase y_p. Los costes se definen como $Cs(min) = IR$ y $Cs(maj) = 1$, siendo min y maj la clase minoritaria y mayoritaria, respectivamente, y IR es el ratio de imbalanceo definido como P_{maj}/P_{min}, donde P_{maj} y P_{min} son el número de ejemplos pertenecientes a la clase mayoritaria y minoritaria, respectivamente.

A la hora de generar las reglas, Chi emplea el siguiente algoritmo.

1. *Construcción de las etiquetas lingüísticas.* Se crean los conjuntos difusos (etiquetas lingüísticas) con la misma forma triangular manteniéndolos distribuidos de forma equitativa a lo largo del rango de valores.
2. *Generación de una regla por cada ejemplo*. Se genera una nueva regla para cada ejemplo x_p de la siguiente manera.
   (a) Se calculan los grados de pertenencia de cada valor x_{pi} a todos los conjuntos difusos de la variable i.
   (b) Se selecciona para cada variable el conjunto difuso con mayor grado de pertenencia.
   (c) Se determina la parte antecedente mediante la intersección de los conjuntos difusos seleccionados. La parte consecuente la determina la etiqueta de la clase del ejemplo (y_p).
   (d) Se calcula el peso de la regla siguiendo la Ecuación (2).

Como podemos apreciar, es posible que obtengamos reglas con los mismos antecedentes pero diferentes consecuentes. En tal caso, se selecciona únicamente la regla de mayor peso. Aquellas reglas que tengan peso negativo son eliminadas de la base de reglas.

4 Chi-FRBCS-BigDataCS

Esta aproximación fue presentada en [8] como una adaptación del algoritmo de Chi et al. para trabajar con problemas Big Data de clasificación binarios imbalanceados. Para ello, los autores hacen uso del framework Apache Hadoop. Este método está compuesto por dos fases:

1. *Generación de las bases de reglas*: en cada mapper se aprende un clasificador Chi independiente considerando únicamente los ejemplos asociados a cada mapper. Por consiguiente, las reglas obtenidas en cada clasificador se generan empleando un sub-conjunto del conjunto de entrenamiento. Una vez ha terminado la fase de aprendizaje de todos los clasificadores, se obtienen tantas bases de reglas como mappers (clasificadores) se hayan ejecutado.
2. *Agregación de las bases de reglas*: todas las bases de reglas generadas en la fase anterior se agregan en un único reducer, obteniendo la base de reglas definitiva. Si hay reglas duplicadas, se selecciona aquella con mayor peso.

Los Algoritmos 1-3 muestran el pseudo-codigo (extraído de [8]) de esta aproximación. La Tabla 1 describe el flujo de datos del algoritmo.

Algoritmo 1 Función map() del Mapper para el algoritmo Chi-BigData.

Procedure map (key, value)

Input: par <*key*, *value*>, donde *key* es el offset en bytes y *value* son los valores del ejemplo.
Begin
1: *example* ← *EXAMPLE_REPRESENTATION* (*value*)
2: *examples* ← *examples.add* (*example*) {*examples* contiene todos los ejemplos asociados al mapper}
End

Algoritmo 2 Función cleanup() del Mapper para el algoritmo Chi-BigData.

Procedure cleanup ()

Output: par <*key'*, *value'*>, donde *key'* es cualquier valor entero y *value'* contiene la base de reglas.
Begin
1: *fuzzy_ChiBuilder.build* (*examples, Cs*) {donde *Cs* contiene el coste asociado a cada clase}
2: *ruleBase* ← *fuzzy_ChiBuilder.getRuleBase* ()
3: EMIT (*key'*, *ruleBase*)
End

Tabla 1. Flujo de datos del algoritmo Chi-FRBCS-BigDataCS original.

Fase	Entrada <*key*, *value*>	Salida <*key*, *value*>
Mapper	<offset, ejemplo>	<valor_entero, base_reglas>
Reducer	<valor_entero, list(base_reglas)>	<nulo, base_reglas>

5 Optimizando Chi-FRBCS-BigDataCS

En este trabajo presentamos un nuevo diseño del método Chi-FRBCS-BigDataCS con el fin de optimizar el algoritmo de aprendizaje y agilizar el tiempo de ejecución sin alterar el modelo obtenido (y por tanto los resultados). Los Algoritmos 4-6 muestran el pseudo-código de nuestra optimización. El flujo de datos del nuevo diseño se muestra en la Tabla 2.

Algoritmo 3 Función reduce() del Reducer para el algoritmo Chi-BigData.

Procedure reduce (key, values)

Input: par <*key'*, *values*>, donde *key'* es cualquier valor entero y *values* son las bases de reglas generadas en los mappers.
Output: par <*key"*, *value"*>, donde *key"* es un valor nulo y *value"* es la base de reglas definitiva.
Begin
1: **while** *values.hasNext* () **do**
2: *ruleBase* ← *values.getValue* ()
3: **for** *i* = 0 **to** *ruleBase.size* () − 1 **do**
4: *rule* ← *ruleBase.get* (*i*)
5: **if** *finalRuleBase.contains* (*rule*) **then**
6: *finalRuleBase.solveDuplicates* (*rule*) {*solveDuplicates*() añade la regla *rule* a la base de reglas *finalRuleBase*. Si hay alguna regla duplicada, se selecciona la de mayor peso}
7: **else**
8: *finalRuleBase.add* (*rule*)
9: **end if**
10: **end for**
11: **end while**
12: EMIT (null, *finalRuleBase*)
End

Algoritmo 4 Función map() del Mapper para nuestra propuesta.

Procedure map (key, value)

Input: par <*key*, *value*>, donde *key* representa los valores de entrada del ejemplo y *value* es la clase del ejemplo.
Begin
1: *newRule* ← *generateRuleFromExample* (*key*) {donde *newRule* son los antecedentes de la nueva regla}
2: **if not** *ruleBase.contains* (*newRule*) **then**
3: *listOfClasses* ← *{}*
4: **else**
5: *listOfClasses* ← *ruleBase.get* (*newRule*)
6: **end if**
7: *listOfClasses* ← *listOfClasses.add* (*value*)
8: *ruleBase.put* (*newRule, listOfClasses*) {siendo *ruleBase* un hash map, donde la clave son los antecedentes de la regla y el valor es un vector que contiene las clases de aquellas reglas con los mismos antecedentes que la clave}
9: *examples* ← *examples.add* (*key*)
10: *classes* ← *classes.add* (*value*)
End

Tabla 2. Flujo de datos de nuestra propuesta.

Fase	Entrada <*key*, *value*>	Salida <*key*, *value*>
Mapper	<valores_ejemplo, clase_ejemplo>	<antecedentes, (clase, peso)>
Reducer	<antecedentes, list((clase, peso))>	<antecedentes, (clase, peso)>

Algoritmo 5 Función cleanup() del Mapper para nuestra propuesta.

Procedure cleanup ()

Output: par <*key'*, *value'*>, donde *key'* es la parte antecedente de una regla dada y *value'* es el consecuente (clase) junto con el peso.

Begin

{Calculamos el grado de emparejamiento de todos los ejemplos con la base de reglas entera}

1: **for** *i* = 0 **to** *examples.size* () − 1 **do**
2: *currentExample* ← *examples.get* (*i*)
3: *currentClass* ← *classes.get* (*i*)
{Pre-calculamos el grado de pertenencia a cada etiqueta lingüística}
4: *preComputedMemberships* ← *computeMembershipDegrees* (*currentExample*)
{Calculamos el grado de emparejamiento con todas las reglas}
5: **for** *j* = 0 **to** *ruleBase.size* () − 1 **do**
6: *ruleAnts* ← *ruleBase.getKey* (*j*)
7: *matchingDegreesSum* [*j, currentClass*] ← *matchingDegreesSum* [*j, currentClass*] + *computeMatchingDegree* (*ruleAnts, currentExample, preComputedMemberships*)
8: **end for**
9: **end for**
{Calculamos los pesos de las reglas y resolvemos los conflictos}
10: **for** *j* = 0 **to** *ruleBase.size* () − 1 **do**
11: *ruleAnts* ← *ruleBase.getKey* (*j*)
12: *ruleClasses* ← *ruleBase.getValue* (*j*)
13: (*class,weight*) ← *computeRuleWeightsAndSolveConflicts* (*matchingDegreesSum* [*j*], *ruleClasses*)
14: EMIT (*ruleAnts*, (*class*, *weight*))
15: **end for**

End

Algoritmo 6 Función reduce() del Reducer para nuestra propuesta.

Procedure reduce (key, values)

Input: par <*key'*, *values*>, donde *key'* es la parte antecedente de una regla dada y *values* son pares (*clase*, *peso*) de aquellas reglas que tienen *key'* como parte antecedente.

Output: par <*key"*, *value"*>, donde *key"* es la parte antecedente de una regla dada y *value"* es un par (*clase*, *peso*) que indica la clase y el peso de la regla una vez resuelto los conflictos.

Begin

1: *maxWeight* ← 0
2: **while** *values.hasNext* () **do**
3: (*class, weight*) ← *values.getValue* ()
4: **if** *weight* > *maxWeight* **then**
5: *maxWeight* ← *weight*
6: *maxClass* ← *class*
7: **end if**
8: **end while**
9: **if** *maxWeight* > 0 **then**
10: EMIT (*key'*, (*maxClass*, *maxWeight*))
11: **end if**

End

A continuación presentamos las diferentes modificaciones que hemos llevado a cabo de la siguiente manera. Primero, describimos el problema existente en una parte específica del algoritmo Chi-FRBCS-BigDataCS original. Posteriormente, mostramos la solución propuesta para optimizar dicha parte.

- *Aprovechando los pares clave-valor*
  - *Problema*: Tal y como se muestra en la Tabla 1, el algoritmo original no aprovecha los pares clave-valor empleados en todas las fases de MapReduce. La función map() (Algoritmo 1) se encarga únicamente de almacenar los ejemplos de entrada, desaprovechando el hecho de que Chi genera una nueva regla por cada ejemplo. De esta forma, la base de reglas entera se genera en la función cleanup() (Algoritmo 2). Por otro lado, la clave de salida de los mappers es cualquier valor entero (Algoritmo 2). Esto quiere decir que el reducer no considera las claves para la fase de agregación. Por consiguiente, no se aprovechan las fases de sorting y merging llevadas a cabo automáticamente por MapReduce. Esto implica que debemos de realizar una búsqueda exhaustiva sobre todas las bases de reglas en la fase reduce para encontrar las reglas duplicadas y resolver los conflictos. Este proceso puede realizarse directamente por las fases de sorting y merging de MapReduce.
  - *Solución*: Observando el pseudo-código y el flujo de datos de nuestra optimización (Algoritmos 4-6 y Tabla 2), podemos apreciar dos diferencias respecto al algoritmo original. Primero, nuestra propuesta genera una nueva regla en cada llamada a la función map() (Algoritmo 4, linea 1), aprovechando la concurrencia de esta fase. La segunda diferencia es que cada mapper devuelve una nueva regla cada vez que se calcula su peso (Algoritmo 5, line 14), en lugar de devolver la base de reglas entera de una vez. De esta forma, la clave de salida de los mappers es la parte antecedente de la regla, mientras que el valor es el consequente (clase) de la regla junto con su peso. Por consiguiente, en cada mapper obtenemos tantos pares clave-valor como reglas generadas (una vez eliminadas las reglas duplicadas) en ese mismo mapper. Esta modificación del flujo de datos nos permite aprovechar las fases de sorting y merging para agrupar todas aquellas reglas que compartan antecedentes (reglas duplicadas). Por tanto, no tenemos la necesidad de realizar una búsqueda exhaustiva sobre la base de reglas para resolver los conflictos, acelerando significativamente la fase reduce.
- *Permitiendo la ejecución de múltiples reducers*
  - *Problema*: El flujo de datos original de Chi-FRBCS-BigDataCS no permite la ejecución de más de un reducer, provocando un cuello de botella en la fase reduce y disminuyendo su escalabilidad.
  - *Solución*: En nuestra propuesta, el hecho de que todos los conflictos de una regla dada sean asignados a un único reducer nos permite ejecutar tantos reducers como deseemos, eliminando el cuello de botella previamente mencionado.
- *Pre-cálculo de los grados de pertenencia mediante Look-Up-Tables*
  - *Problema*: De acuerdo con la Ecuación (2) mostrada en la Sección 3, un elevado número de cálculos de grados de pertenencia se repiten a lo largo del

aprendizaje de Chi. Dado que el cálculo del peso de la regla requiere del grado de emparejamiento de todos los ejemplos y que este proceso se repite para cada regla, vemos que el cálculo del grado de pertenencia de cada ejemplo a las diferentes etiquetas lingüísticas se realiza cada vez que la etiqueta aparece en una regla. Por ejemplo, podemos disponer de miles de reglas que contienen la misma etiqueta lingüística asociada al antecedente i. Esto quiere decir que el grado de pertenencia de un ejemplo dado a dicha etiqueta lingüística será exactamente el mismo en miles de casos, resultando en millones de cálculos repetidos.

- *Solución*: Con el objetivo de evitar que el mismo grado de pertenencia se calcule más de una vez, llevamos a cabo una fase de pre-cálculo (Algoritmo 5, linea 4) donde se calculan los grados de pertenencia de un ejemplo dado a todas las etiquetas lingüísticas. De esta forma, nos aseguramos de que cada grado de pertenencia se calcula una sola vez. Para todo este proceso, empleamos una Look-Up-Table cuyos valores se van actualizando para cada ejemplo.

6 Estudio experimental

En este estudio queremos comprobar la mejora que ofrece la optimización que hemos propuesto del algoritmo Chi-FRBCS-BigDataCS respecto al original. Dado que nuestra aproximación no afecta al modelo generado (y por tanto a los resultados), en este estudio analizaremos únicamente los tiempos de ejecución de cada método.

La Sección 6.1 describe los conjuntos de datos y parámetros empleados en el estudio, mientras que en la Sección 6.2 analizamos los resultados obtenidos.

6.1 Marco experimental

Para el desarrollo del estudio experimental, hemos considerado 6 conjuntos de datos diferentes del repositorio UCI[7]. Con el objetivo de obtener más conjuntos de datos y de trabajar con el mismo marco experimental que en [8], hemos seleccionado 15 problemas de clasificación binarios diferentes convirtiendo los conjuntos de datos originales multi-clase en múltiples problemas binarios *One-vs-Rest*. Para ello, en cada conjunto de datos original seleccionamos una clase positiva y consideramos el resto como la clase negativa. La Tabla 3 muestra la descripción de los conjuntos de datos indicando el número de ejemplos (#Ejemplos), número de ejemplos de la clase mayoritaria y minoritaria (P_{maj}:P_{min}), y el número de atributos (#Atributos) reales (R), enteros (E), nominales (N), y totales (T).

Todos los experimentos han sido realizados aplicando una validación cruzada de 5 particiones. De esta manera, dividimos aleatoriamente el conjunto de datos en 5 particiones, cada una conteniendo el 20% de los ejemplos, y empleamos una combinación de cuatro de ellas (80%) para entrenar el sistema y el resto para testearlo. Por tanto, el resultado de cada conjunto de datos se calcula como la media de las 5 particiones.

Respecto a los parámetros considerados para la ejecución de los métodos, en ambos casos (Chi-FRBCS-BigDataCS original y la optimización propuesta) hemos empleado

Tabla 3. Descripción de los conjuntos de datos.

Conjunto de datos (dataset)		#Ejemplos	(P_{maj}:P_{min})	#Atributos			
Id	Nombre			R	E	N	Total
Censu	Census	142521	(134359:8162)	1	12	28	41
Cov_1	Covtype_1	581012	(369172:211840)	10	0	44	54
Cov_2	Covtype_2	581012	(297711:283301)	10	0	44	54
Cov_3	Covtype_3	581012	(545258:35754)	10	0	44	54
Cov_7	Covtype_7	581012	(560502:20510)	10	0	44	54
Fa_FI	Fars_Fatal_Injury	100968	(58852:42116)	5	0	24	29
Fa_II	Fars_Incapaciting_Injury	100968	(85896:15072)	5	0	24	29
Fa_NI	Fars_No_Injury	100968	(80961:20007)	5	0	24	29
Fa_NE	Fars_Noninc._Ev._Injury	100968	(87078:13890)	5	0	24	29
Pok_0	Poker_0	1025009	(513701:511308)	0	10	0	10
Pok_1	Poker_1	1025009	(591912:433097)	0	10	0	10
Pok_2	Poker_2	1025009	(976181:48828)	0	10	0	10
Pok_3	Poker_3	1025009	(1003375:21634)	0	10	0	10
Skin	Skin	245057	(194198:50859)	0	3	0	3
Susy	Susy	5000000	(2712173:2287827)	18	0	0	18

3 etiquetas lingüísticas por variable y hemos aplicado el PCF-CS (Ecuación 2 en la Sección 3) para realizar el cálculo de los pesos de las reglas.

En cuanto a la infraestructura usada para los experimentos, 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 y Apache Hadoop 2.6.0. Esta configuración ofrece 42 containers de YARN concurrentes, donde cada uno puede ser un mapper, un reducer, o el Application Master.

6.2 Resultados experimentales

En esta sección vamos a analizar el efecto de nuestra optimización en el tiempo de ejecución del algoritmo Chi-FRBCS-BigDataCS. La Tabla 4 muestra los promedios de los tiempos de ejecución de los mappers. La razón de no incluir los tiempos totales es que la mayor parte del procesamiento del algoritmo se realiza en los mappers (cuando se generan las bases de reglas). Por consiguiente, si incluyéramos el tiempo total en conjuntos de datos no demasiado grandes, el sobre-coste del algoritmo MapReduce representaría gran parte de ese tiempo y la mejora no podría apreciarse de forma correcta. Por el contrario, si consideráramos el tiempo total en conjuntos de datos grandes, la mejora

obtenida en los mappers eclipsaría la obtenida en el resto de fases de MapReduce. Por otro lado, hemos omitido Susy de la media debido al tiempo requerido para ejecutar todas las combinaciones de mappers y particiones. No obstante, en la Tabla 4 incluimos una comparativa para Susy empleando 32 mappers (el máximo que podemos ejecutar de forma concurrente en el cluster utilizado).

Tabla 4. Promedio del tiempo de ejecución de los mappers (mm:ss).

Dataset	32 mappers		64 mappers		128 mappers		256 mappers	
	Original	Propuesta	Original	Propuesta	Original	Propuesta	Original	Propuesta
Censu	00:02	00:00	00:00	00:00	00:00	00:00	00:00	00:00
Cov_1	03:03	00:33	00:45	00:08	00:11	00:02	00:02	00:00
Cov_2	03:01	00:33	00:45	00:08	00:11	00:02	00:02	00:00
Cov_3	03:01	00:33	00:45	00:08	00:11	00:02	00:02	00:00
Cov_7	03:01	00:33	00:45	00:08	00:11	00:02	00:02	00:00
Fa_FI	00:00	00:00	00:00	00:00	00:00	00:00	00:00	00:00
Fa_II	00:00	00:00	00:00	00:00	00:00	00:00	00:00	00:00
Fa_NI	00:00	00:00	00:00	00:00	00:00	00:00	00:00	00:00
Fa_NE	00:00	00:00	00:00	00:00	00:00	00:00	00:00	00:00
Pok_0	07:41	00:25	01:55	00:06	00:29	00:01	00:07	00:00
Pok_1	07:45	00:25	01:56	00:06	00:29	00:01	00:07	00:00
Pok_2	07:48	00:26	01:56	00:06	00:29	00:01	00:07	00:00
Pok_3	07:42	00:25	01:56	00:06	00:29	00:01	00:07	00:00
Skin	00:08	00:01	00:02	00:00	00:00	00:00	00:00	00:00
AVG.	**03:05**	**00:16**	**00:46**	**00:04**	**00:11**	**00:00**	**00:02**	**00:00**
Susy	2562:07	26:49	-	-	-	-	-	-

De acuerdo con la Tabla 4, la ejecución de los mappers cuando pre-calculamos los grados de pertenencia es aproximadamente 10 veces más rápida en los conjuntos de datos considerados. Cabe destacar que la mejora obtenida será cada vez mayor conforme el número de ejemplos y reglas aumenta. De hecho, si consideramos el conjunto de datos Susy (Tabla 4), vemos que cuando empleamos 32 mappers nuestra propuesta se ejecuta 100 veces más rápido que el algoritmo original.

Como podemos observar la mejora en tiempos de ejecución ofrecida por nuestra aproximación es significativa respecto al método Chi-FRBCS-BigDataCS original.

7 Conclusiones

En este trabajo hemos presentado una optimización del único Sistema de Clasificación Basado en Reglas Difusas que está disponible hasta la fecha para problemas de clasificación Big Data (Chi-FRBCS-BigDataCS). Para realizar las mejoras, por un lado hemos empleado Look-Up-Tables para realizar el pre-cálculo de los grados de pertenencia a la hora de generar las reglas con los clasificadores Chi. Gracias a esta adaptación, cuando calculamos el peso de las reglas, el grado de pertenencia de un valor a una etiqueta lingüística se calcula una sola vez por cada ejemplo, en vez de calcularse tantas veces

como reglas se hayan generado. Por otro lado, el uso de Look-Up-Tables requiere que el método Chi-FRBCS-BigDataCS tenga que ser rediseñado para trabajar con el nuevo flujo de datos. Por ello, hemos introducido varias adaptaciones que, además de permitir la fase de pre-cálculo, explotan el potencial de MapReduce.

Todas estas mejoras nos han permitido ejecutar el algoritmo de aprendizaje de Chi-FRBCS-BigDataCS entre 10 y 100 veces más rápido que el original, sin alterar el modelo obtenido (y por tanto los resultados).

Agradecimientos. Este trabajo ha sido financiado parcialmente por el Ministerio de Ciencia y Tecnología de España con el proyecto TIN2013-40765-P y por la red TIN2014-56381-REDT.

Referencias

1. Arthur, L.: What is big data? Forbes (2013), `http://www.forbes.com/sites/lisaarthur/2013/08/15/what-is-big-data/`
2. Chi, Z., Yan, H., Pham, T.: Fuzzy algorithms with applications to image processing and pattern recognition. World Scientific (1996)
3. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
4. Galar, M., Fernández, A., Barrenechea, E., Bustince, H., Herrera, F.: A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches. IEE Transactions on Systems, Man, and Cybernetics 42(4), 463–484 (2011)
5. Ishibuchi, H., Nakashima, T., Nii, M.: Classification and modeling with linguistic information granules: Advanced approaches to linguistic Data Mining. Springer-Verlag (2004)
6. Ishibuchi, H., Yamamoto, T.: Rule weight specification in fuzzy rule-based classification systems. IEEE Transactions on Fuzzy Systems 13(4), 428–435 (2005)
7. Lichman, M.: UCI machine learning repository (2013), `http://archive.ics.uci.edu/ml`
8. López, V., del Río, S., Benítez, M., Herrera, F.: Cost-sensitive linguistic fuzzy rule based classification systems under the mapreduce framework for imbalanced big data. Fuzzy Sets and Systems 258(0), 5 – 38 (2015)