**UNIVERSIDAD COMPLUTENSE DE MADRID**

**FACULTAD DE CIENCIAS MATEMÁTICAS**

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN


## TRABAJO DE FIN DE GRADO

# Entity Resolution y Deduplication con Blocking paralelo en Spark

# Entity Resolution and Deduplication with parallel blocking using Spark

Carlos Gregorio Rodríguez

**Guillermo Herranz Álvarez**

Doble Grado en Matemáticas y Física

Curso académico 2019-2020

Convocatoria de Junio

**Resumen:**

En este trabajo planteamos un algoritmo que permite identificar qué registros de un dataset, aún no siendo idénticos, se corresponden con la misma entidad real (*Entity Resolution*). El algoritmo clásico para este proceso consiste en la comparación directa de todos los registros dos a dos y, por tanto, tiene por lo menos complejidad cuadrática. Nuestra solución mejora el algoritmo clásico utilizando paralelización y, por consiguiente, garantizando la escalabilidad del mismo. Además, el diseño del algoritmo es genérico. Permite la definición de unos parámetros de configuración para adaptarlo al dataset concreto que se desee estudiar. Las ejecuciones realizadas para analizar el comportamiento de este algoritmo han resultado muy satisfactorias, obteniendo resultados muy similares al caso clásico en unos tiempos de ejecución significativamente menores. Esta diferencia temporal es aún mayor conforme aumentemos el tamaño de los datasets sobre la que se trabajen.

**Abstract:**

In this work we present and algorithm that allows the user to identify which registers from a dataset, while not being identical, represent the same real-world entity (*Entity Resolution*). The classical algorithm for this process consists of direct comparisons between all registers and, as a result, has at least quadratic complexity. Our solution improves upon this classical algorithm by using parallelization, granting its scalability. In addition, its design is generic. It allows for some configuration parameters to be defined depending on the concrete dataset that wants to be studied. The executions performed to analyse its behaviour have been very successful, obtaining very similar results to the classical algorithm using significantly less execution time. This time difference is even bigger as the dataset's size increases.

# Índice

# 1. Introducción

Este proyecto surge a raíz de un problema que aparece durante el tratamiento y análisis de datos en el ámbito de la salud. Habitualmente, los profesionales de la salud introducen los datos de sus pacientes de forma manual en sus bases de datos. Al ser este proceso manual, aparecen principalmente dos efectos que dificultan su posterior tratamiento y análisis. Son los siguientes:

1. Uso de diminutivos, acortaciones, guiones ...
2. Introducción de errores tipográficos variados.

Estos efectos provocan que los pacientes puedan tener los datos asociados a sus pruebas médicas dispersos en varios registros, lo que dificulta el acceso de los médicos a su historial. Más que una leve inconveniencia, esto puede convertirse en un hecho peligroso cuando un médico no puede acceder a información vital sobre su paciente, necesaria para tomar una decisión sobre un tratamiento que este deba seguir.

Este problema no solo surge en el ámbito médico. En toda situación donde una empresa, un gobierno o cualquier tipo de organización quiera unificar los registros que representan información sobre una misma persona o entidad real, pero que hayan sido recogidos de forma distinta o generados por distintos medios (p.e. distintos empleados escribiendo los datos de sus clientes), debemos ser capaces de determinar si dos registros, que a priori son diferentes, realmente representan a la misma persona o entidad.

Para tratar de solucionar esta cuestión, buscamos la forma de identificar y unir los registros que proceden de una misma persona. A pesar de que la información que conforma los registros no sea la misma, debemos determinar si es suficientemente similar para considerar que efectivamente los registros comparados son de la misma persona. La solución planteada es construir un algoritmo de *Entity Resolution* (concepto que definiremos de forma precisa en el marco teórico, en la sección 2).

El concepto de *Entity Resolution* (ER) ha sido aplicado a distintos problemas y con distintas metodologías y hoy en día continua siendo un campo de estudio activo [1, 2]. Podemos encontrarnos soluciones para ER que hacen uso de tecnologías tan actuales hoy en día como *deep learning* [3], sistemas de ER inteligentes, que aprenden qué normas debe usar para realizar el proceso de forma correcta [4], o transformando los datos en una estructura de grafos para usar sistemas de vertex-matching [5]. Las formas de hacer ER han sido y continuan siendo muy variadas. En nuestro caso, queremos desarrollar un algoritmo de ER que sea aplicable a datasets similares al que origina nuestro problema. Nos referimos a datasets formados por un cierto número de filas, cada una de ellas representando un registro, y otro cierto número de columnas, cada una de ellas representando un trozo de información que posee cada registro. Además, queremos ser capaces de establecer de forma clara y transparente cuando dos registros proceden de la misma entidad real.

Inicialmente, disponemos un dataset de 35284 elementos. Un algoritmo clásico para este problema está basado en la comparación de registros de cada uno de los registros con todos los demás. Podríamos hacer uso de este algoritmo clásico para resolver este problema. Es una solución bastante lenta pero termina funcionando. Sin embargo nos planteamos, ¿y si el dataset fuese más grande? ¿Y si fuese lo suficientemente grande para que no podamos usar algoritmos clásicos en un tiempo razonable? ¿Y si el dataset que queremos estudiar no tiene los mismos campos que nuestro dataset inicial?

Tratando de responder estas preguntas hemos ideado un algoritmo escalable que se aproxime lo máximo posible a una solución óptima. Además, queremos que sea versátil, con capacidad de adaptarse a modificaciones del tipo de datos que tenemos para que no sea únicamente válido para nuestro problema original.

El algoritmo más directo que podríamos construir para hacer *Entity Resolution*, como hemos comentado, sería la comparación directa una a una de los N registros que presenta un dataset para decidir si representan o no la misma entidad. Sin embargo esta opción es complicada computacionalmente hablando, pues un algoritmo con estas características es, al menos, de orden cuadrático, $O(n^2)$. Habitualmente el coste es aún mayor debido a tener que ser exhaustivo a la hora de buscar todos las posibles anexiones, lo cual dificulta su escalabilidad. Además, no es paralelizable, es decir, debe ejecutarse en una única máquina simultáneamente. Cuando los cluster de ordenadores son cada vez más comunes, un algoritmo no paralelizable pierde mucho valor. Por este motivo hemos desarrollado un algoritmo paralelizable para buscar la escalabilidad.

El trabajo sigue la siguiente estructura. En la sección 2, vamos a introducir el marco teórico sobre el que se asienta la construcción del algoritmo. Formalizaremos el proceso de *Entity Resolution* introduciendo los conceptos y las definiciones rigurosas que permiten definir con propiedad el problema. Sobre este formalismo se establecen unas propiedades que nos garantizan la existencia de un resultado óptimo y formas (relativamente abstractas) de obtenerlo.

Continuaremos en la sección 3 introduciendo los principios fundamentales con los que se han desarrollado el algoritmo, detallando beneficios e inconvenientes que surgen de su uso. En la sección 4, incluiremos y comentaremos el propio código del algoritmo para entender cómo funciona paso a paso.

Finalmente, en la sección 5, analizaremos los resultados que obtiene frente a dos tipos de datasets. El primer tipo será un conjunto de datasets de los cuales conocemos su solución, para poder observar qué tan bien realiza su función. El segundo será el dataset del cual surge este trabajo y procede de pruebas médicas. De este dataset no se tiene la solución, por lo que nos centraremos en analizar distintas ejecuciones para evaluar el tiempo de computación y los resultados obtenidos.

# 2. Marco teórico

*Entity Resolution*, Resolucion de Entidades o, como nos referiremos desde ahora, ER, es un proceso mediante el cual identificamos y unificamos los registros de un dataset que representan a la misma entidad real. Estos registros asociados a una misma entidad pueden estar inicialmente desligados por multitud de factores, que incluyen la repetición de registros, la introducción manual de información errónea, la corrupción de los datos al transformarlos, etc.

En este marco teórico vamos a comenzar por introducir una serie de cuestiones sobre algoritmos de ER para detallar después como es el marco estructural que vamos a utilizar [6].

## 2.1. Funciones match y merge

Cuando hablamos de ER, estamos considerando implícitamente dos procesos: selección de cuando dos registros son de la misma entidad y el modo en que, una vez decidido que son de la misma entidad, juntamos los dos registros. Estos dos procesos están regidos por dos funciones, la función match y la función de merge.

Sea E el conjunto de los posibles registros, que podemos considerar infinito. Decimos que M es la función de match del proceso ER si está definida sobre $E \times E$ y tiene como imagen dos posibles valores, que llamaremos *true* y *false*. Sean $r_1$ y $r_2$ registros ($r_1, r_2 \in E$), entonces $M(r_1, r_2) = true$ si $r_1$ y $r_2$ representan la misma entidad real. En caso contrario, $M(r_1, r_2) = false$. Con estas propiedades lo que estamos pidiendo es que el matching depende únicamente de los dos registros comparados y se decide si hacen match o no de forma inequívoca, sin ningún tipo de nivel de confianza.

La función de merge, que llamaremos N, también debe satisfacer dos propiedades. Debe estar definida sobre $(E \times E)|_{\{r_1, r_2 \in E | M(r_1, r_2) = true\}}$ y tener como imagen E. Para aligerar la notación futura, $M(r_1, r_2) = true$ lo denotamos por $r_1 \approx r_2$ (en caso contrario, $r_1 \not\approx r_2$ ). También denotaremos $N(r_1, r_2)$ por $\langle r_1, r_2 \rangle$, que sería el registro obtenido al aplicar la función de merge.

Estas propiedades que acabamos de definir son las estrictamente necesarias para poder llevar a cabo un proceso de ER. Dados dos registros decidimos si hacen match y generamos un registro que aglutine la información de ambos. Esta última propiedad no la hemos enunciado, pero es evidente que es necesaria para definir una buena función de merge. Si no nuestro resultado no va a ser un buen resultado.

Sin embargo, para realizar un proceso de ER de forma eficiente, hay ciertas propiedades adicionales que querríamos pedir para hacer nuestro proceso más eficiente. Son las siguientes:

1. Idempotencia: $\forall r$, $r \approx r$ y $\langle r, r \rangle = r$. Esta propiedad nos dice que un registro hace match consigo mismo y el registro obtenido de juntarlo consigo mismo es él mismo.
2. Conmutatividad: $\forall r_1, r_2$, $r_1 \approx r_2$ si y solo si $r_2 \approx r_1$. Además, si $r_1 \approx r_2$, $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.
3. Asociatividad: $\forall r_1, r_2, r_3$ de forma que existan tanto $\langle r_1, \langle r_2, r_3 \rangle \rangle$ como $\langle \langle r_1, r_2 \rangle, r_3 \rangle$, entonces tenemos que $\langle r_1, \langle r_2, r_3 \rangle \rangle = \langle \langle r_1, r_2 \rangle, r_3 \rangle$.
4. Representatividad: Si $r_3 = \langle r_1, r_2 \rangle$, entonces $\forall r_4$ tal que $r_1 \approx r_4$, se tiene también que $r_3 \approx r_4$.

Estas 4 propiedades las llamaremos las propiedades ICAR, por sus iniciales. Son propiedades bastante naturales y deseables en un proceso de ER. Especialmente importante es la asociatividad, pues de ella depende el hecho de que el orden en el que se realiza las operaciones sea relevante o no. También notemos que estas propiedades no implican la transitividad (si $r_1 \approx r_2$ y $r_2 \approx r_3$ no necesariamente se tiene que $r_1 \approx r_3$).

La principal fortaleza de las propiedades ICAR en un proceso ER es que cuando se satisfacen tenemos garantizado que nuestro proceso ER va a obtener el resultado más óptimo posible, es decir, vamos a conseguir anexionar la mayor cantidad de registros posibles. Para ver esto, debemos dar una serie de definiciones previas. Sea I una instancia de E, es decir, un subconjunto finito del conjunto de todos los posibles registros.

**Definicion 1** *Dada I, definimos el cierre de I, denotado por $\bar{I}$ como el menor conjunto que verifique:*

1. $I \subseteq \bar{I}$.
2. $\forall r_1, r_2 \in \bar{I}$, *si* $r_1 \approx r_2$, *entonces* $\langle r_1, r_2 \rangle \in \bar{I}$ .

Esencialmente el cierre de I es un conjunto más grande donde añadimos de forma reiterativa todos los posibles resultados de juntar registros. De este modo, si dos registros hacen match, el registro resultante de la operación de merge siempre se encuentra en este conjunto. Notemos que este conjunto $\bar{I}$ claramente existe y es único, puesto que se puede obtener como el punto fijo del proceso de añadir a I todos los posibles merges.

**Definicion 2** *Sean dos instancias de E, $I_1$ e $I_2$. Decimos que $I_1$ está dominada por $I_2$ (y lo denotamos por $I_1 \preceq I_2$) si $\forall r_1 \in I_1, \exists r_2 \in I_2$ de forma que $r_1 \approx r_2$ y además $r_2$ almacena más información que $r_1$ (denotado por $r_1 \preceq r_2$)*

Esta definición en esencia trata de establecer una especie de relación de orden entre instancias. Una instancia será menor en este sentido si para todo registro que tenga hay otro en la mayor que hace match con él y tiene más información. Un ejemplo sería que dado un registro como [{'a'}] tuviésemos otro registro como [{'a','b'}] en la mayor (y suponemos que hacen match, según la lógica del algoritmo que hemos desarrollado). Hacen match y el segundo guarda más información que el primero.

Con estas dos nociones, podemos ya definir lo que buscamos en un proceso ER, la resolución de I.

**Definicion 3** *Dada una instancia I, denotamos por $I'$ a un conjunto que verifica las siguientes 3 propiedades:*

1. $I' \subseteq \bar{I}$.
2. $\bar{I} \preceq I'$.
3. *No existe ningún subconjunto estricto de I' que verifique 1 y 2*

Este conjunto *I'* es lo que llamaremos la resolución de I, o ER(I). Esencialmente es la mínima instancia incluida en el cierre de I y que contiene toda la información que contiene el cierre de I. Esto es lo que esperaríamos intuitivamente que fuese la resolución de I, el mínimo conjunto de registros con la máxima información. Cada uno de estos registros representaría una entidad distinta. Para garantizarnos que es único tenemos la siguiente proposición.

**Proposición 1** *Dada una instancia I, la resolución de I existe, es única, y la denotaremos por ER(I)*

Una vez ya hemos introducido el concepto formal de la resolución de una instancia, vamos a definir formalmente el proceso de obtener ER(I).

**Definicion 4** *Dada una instancia I, una derivación es una transformación de I en otra instancia I' mediante dos posibles operaciones:*

- *Merge: Dados dos registros $r_1$ y $r_2$ de I, si se tiene que $r_1 \approx r_2$ y $\langle r_1, r_2 \rangle \notin I$, entonces $I' = I \cup \{\langle r_1, r_2 \rangle\}$.*
- *Purga: Dados dos registros $r_1$ y $r_2$ de I, si se tiene que $r_1 \preceq r_2$, entonces $I' = I - \{r_1\}$*

En base a esta definición, una derivación es cada uno de los procesos individuales que nos permiten acercarnos a la solución del problema. Solamente hay dos opciones, o añadimos un elemento nuevo que no teníamos, procedente de hacer merge de dos registros, o eliminamos un registro que era redundante. A partir de una derivación, podemos definir una cadena de derivaciones, $I_n$, que no es más que encadenar derivaciones para acercarnos a la solución. Decimos que una cadena es maximal si no existe ninguna posible derivación en el último elemento de la cadena.

Finalmente, podemos enunciar el teorema por el cual las propiedades ICAR resultan fundamentales para hacer un ER eficiente.

**Teorema 1** *Supongamos que un proceso ER tiene unas funciones match y merge que verifican las propiedades ICAR. Entonces, para toda instancia I, ER(I) es finita y cualquier secuencia de derivación maximal obtene ER(I).*

En base a este teorema querremos tratar de definir para nuestro algoritmo funciones match y merge que posean las propiedades ICAR para mejorar nuestra eficiencia. Para ello nos basaremos en la *Union Class*.

## 2.2. *Union Class*

La *Union Class* es una clase de funciones match y merge que cumplen las propiedades ICAR. La idea principal que la construye es que no se pierda nada de información en las operaciones. Para ello, cuando dos registros hacen match, el nuevo registro resultante se obtiene combinando los valores que definían individualmente a cada uno de los dos registros. De este modo, los nuevos registros generados contienen toda la información de sus predecesores. Un ejemplo sería el siguiente. Supongamos que tenemos dos registros que se van a juntar, *M. Teresa* y *María Teresa*. Generaríamos un nuevo registro que será {*M. Teresa, María Teresa*}. Notemos que en este caso podríamos haber juntado ambos registros en único valor: *María Teresa*. De este modo tampoco perderíamos ninguna información. No obstante, remarcamos la opción conjuntista pues es en la que vamos a basar nuestra algoritmo, pues su generalización es más sencilla. Posteriormente, cuando queramos comparar nuevos registros con el conjunto obtenido, haremos comparaciones con cada uno de los elementos del conjunto, pues todos ellos serán distintas formas de representar a una misma entidad.

Importante notar que en esta descripción de la *Union Class* no hemos hecho mención a cómo debe ser la función match, salvo que debe ser capaz de comparar registros formados por conjuntos una vez empiece a haber registros que, generados según esta lógica, aglutinan datos en conjuntos. Evidentemente debemos añadir algo más a esta función match para que se satisfagan las propiedades ICAR, y por ello tenemos la siguiente proposición.

**Proposición 2** *Dadas funciones match y merge en la* Union Class*, si la función match es además reflexiva y conmutativa, entonces se satisfacen las propiedades ICAR.*

La importancia de tener funciones match y merge que cumplan las propiedades ICAR es muy relevante para mejorar la eficiencia de un algoritmo de ER. No solo existen funciones que verifiquen estas propiedades en la *Union Class*. Sin embargo, la forma de construir estas funciones resulta una forma muy natural de abordar numerosos problemas de ER. Además, las propiedades exigidas son propiedades fáciles de conseguir al diseñar funciones de match. Otras propiedades también deseables y/o intuitivas, como la transitividad, no son tan sencillas de obtener. De hecho, el algoritmo diseñado no posee la propiedad de transitividad.

# 3. Principios fundamentales del algoritmo

El diseño de nuestro algoritmo se basa en varios principios fundamentales:

1. La escalabilidad.
2. La precisión.
3. La adaptabilidad.

## 3.1. Escalabilidad

Un objetivo fundamental a la hora de desarrollar este algoritmo es que sea escalable. Esto quiere decir que el coste computacional del algoritmo escale de forma aproximadamente lineal con el tamaño del dataset a tratar. Para hacer nuestro algoritmo escalable, hemos decidido hacer uso del lenguaje de programación Python y de Spark con su funcionamiento con RDD para paralelizar las tareas. Para poder paralelizar una tarea en un dataset, tenemos en primer lugar que generar pares clave-valor para ser capaces de ordenar distintas tareas a cada máquina que forme parte del cluster donde se ejecute el algoritmo. Como los únicos valores que tenemos son los campos o columnas de nuestro dataset, es necesario que al menos uno de ellos se convierta en nuestra clave. Nosotros escogimos usar solo uno para realizar el match sobre todos los demás campos, buscando la máxima similitud entre los registros. Esta separación de los valores por la clave que tienen es una técnica conocida como *blocking*, que trata de reducir la complejidad del proceso. Así, las comparaciones las realizaremos de forma exhaustiva en cajas o buckets de un tamaño menor y por lo tanto con menor coste. Tendremos tantos buckets como claves distintas existan. El uso de esta técnica genera dos cuestiones importantes a tener en cuenta.

- Al separar los registros en grupos donde un campo debe ser exactamente idéntico (la clave) estamos perdiendo posibles matches donde el campo escogido pueda ser solo ligeramente distinto.
- Al escoger un campo queremos separar el problema en cajas o buckets donde el tamaño de las cajas sea significativamente menor que el tamaño del dataset original para reducir notablemente la complejidad. Una mala elección de clave puede no lograr esto.

La segunda cuestión es la más fácil de solucionar. Basta hacer un estudio previo de cómo es el dataset para elegir como clave aquel campo que presente buckets de menor tamaño. Un conocimiento previo del dataset a resolver resulta muy útil para configurar adecuadamente el algoritmo. Uno de los parámetros a definir es precisamente cual es la clave a usar.

En cambio, la primera cuestión es más difícil de abordar. La solución propuesta es la siguiente. Tras aplicar la lógica de merge en los buckets generados y volver a juntar los resultados, tenemos de nuevo un dataset unificado. Podemos ahora escoger una clave distinta y con ella volver a realizar la misma operación. Esto permite que elementos que originalmente se encontraban en distintos buckets se mezclen y puedan juntarse si es que la lógica de match lo permite. Notemos que puede ocurrir que dos registros representen la misma entidad pero tengan todos los campos distintos, aunque sea por poco. En este caso, la lógica diseñada solo podría llegar a juntarlos si hay algún registro intermedio al que pueda juntarse uno de los dos. Se originaría de esta forma un registro que, al tener más información, podría terminar el match. Un ejemplo para visualizar lo que acabamos de comentar es el siguiente. Supongamos que los dos registros originales son *[1,3]* y *[2,4]*. Estos dos

registros no tienen nada en común, aunque los valores no son muy lejanos. Si existe un registro intermedio, digamos *[1,4]*, que hace match con ambos, podríamos usarlo como pivote para terminar juntando los dos registros originales.

Veremos en la sección 4 que para poder implementar la paralelización debemos renunciar a seguir fielmente el principio de la *Union Class*. En dicha sección mencionaremos una operación necesaria para poder ejecutar el algoritmo, el despliegue, y comentaremos las implicaciones que tiene a este respecto. Por lo tanto la solución obtenida no estará garantizada a ser la más óptima. No obstante, al seguir fielmente esta lógica en todos los pasos salvo en esa operación, esperamos que podamos acercarnos lo máximo posible a la solución óptima.

### 3.2. Precisión

Para buscar la máxima precisión posible al seguir una estrategia de paralelización vamos construir funciones de match y merge que sigan las propiedades ICAR que hemos introducido en el marco teórico, sección 2. Para ello, las funciones de match y merge van a formar parte de la *Union Class* y la función de match va a satisfacer las 2 propiedades adicionales exigidas en la proposición 2. Para trabajar con la función de merge debemos trabajar con conjuntos. Por ello lo primero que haremos a nuestro dataset es transformar los campos de cada registro en un conjunto que contenga únicamente el dato que contenía. Si teníamos como valor $A$, ahora tendremos $\{A\}$. Posteriormente el funcionamiento de la función es el siguiente. Sean $r_1 = (c_1, ..., c_n)$ y $r_2 = (b_1, ..., b_n) \in E$, donde n es el número de campos del dataset. Entonces $\langle r_1, r_2 \rangle := (c_1 \cup b_1, ..., c_n \cup b_n)$.

La función de match es más delicada y configurable, pues depende en parte del problema al que queramos aplicarla. En primer lugar, al iniciar el algoritmo, asignamos a cada campo una función distancia y un factor (denotados por $d_i$ y $f_i$, respectivamente distancia y factor del campo i). La distancia definida para cada campo debe ser adecuada al tipo de campo en concreto. Veremos más sobre las distancias que pueden usarse en la sección 3.3.1. El factor es una medida de cuan relevante es encontrar discrepancias en este campo para decidir si dos registros hacen match o no. Así, calculamos la distancia entre ambos registros con la siguiente expresión.

$\text{Distancia}(\text{r}_1, r_2) = \sum_{i=1, i \neq j}^{n} f_i \cdot min\{d_i(x, y) | x \in c_i, y \in b_i\}$

donde j aquí denota el campo que estamos usando como clave. Definiendo la distancia entre dos registros de esta manera, definimos la función de match.

$$M(r_1, r_2) = \begin{cases} true & si \quad Distancia(r_1, r_2) \leq 1 \\ false & en\ caso\ contrario \end{cases}$$

### 3.3. Adaptabilidad

Una característica relevante del algoritmo diseñado es la capacidad de adaptarlo al dataset sobre el que debemos aplicarlo. El funcionamiento del algoritmo no puede ser igual cuando los campos son todos numéricos que cuando son todos strings, o cuando hay mezcla de ambos y/o de otros tipos de datos. En general, el conocimiento de los datos nos permite establecer cómo se producen los procesos de match para poder ser más eficaces. Esto es específico de cada caso.

Los parámetros que debemos introducir son los siguientes:

- Definición de distancia para cada campo.

- Factor de proporcionalidad de aplicar a cada campo.
- Orden en que tomamos los campos como clave.
- Número de iteraciones del algoritmo.

### 3.3.1. Distancias

Entendemos por la distancia de cada campo i ($d_i$) a una función que dados dos registros $r_1$ y $r_2$ y sus correspondientes valores en el campo i, $c_i$ y $b_i$, obtenga un valor numérico que mida cuan diferentes son estos campos entre si. Parecería a priori que deberíamos pedir que esta distancia verifique las propiedades básicas de una distancia, a saber:

1. $\forall r_1 = (.., c_i, ..), r_2 = (.., b_i, ..) \in E,\ d_i(c_i, b_i) = 0 \Longleftrightarrow c_i = b_i.$
2. $\forall r_1 = (.., c_i, ..), r_2 = (.., b_i, ..) \in E, d_i(c_i, b_i) = d_i(b_i, c_i).$
3. $\forall r_1 = (.., c_i, ..), r_2 = (.., b_i, ..), r_3 = (.., e_i, ..) \in E, d_i(c_i, b_i) \leq d_i(c_i, e_i) + d_i(b_i, e_i)$

No obstante, debido a la existencia de campos indefinidos (que trataremos en detalle posteriormente) y al uso de conjuntos para definir el algoritmo, es posible definir funciones distancia apropiadas para el problema a tratar y que no verifiquen estas propiedades. Un ejemplo de esto es la distancia de Levenshtein.

La distancia de Levenshtein o distancia de edición es una distancia que cumple las 3 propiedades básicas expresadas anteriormente cuando los valores de los registros son cadenas de caracteres, pero no así cuando son conjuntos de cadenas de caracteres. Recordamos en este punto que, cuando disponemos de conjuntos, la distancia que medimos es el mínimo de las distancias entre los elementos que los componen. Se define esta distancia como el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra, entendiendo por operaciones la inserción, eliminación o la sustitución de un carácter. Se trata de una distancia sumamente intuitiva y en la basaremos gran parte de la ejecución del algoritmo para probar su eficacia.

Cuando usamos conjuntos en esta distancia, podemos encontrarnos violaciones de las propiedades 1 y 3. Aún así, sigue siendo una distancia interesante a tener en cuenta. Por ejemplo:

1. d({'A','B'},{'A','C'}) = 0 a pesar de que {'A','B'}$\neq$ {'A','C'}
3. d({'A','B'},{'D','C'}) = 1 $\not\leq$ 0 = d({'A','B'},{'A','C'}) + d({'D,'C'},{'A','C'})

Si bien la distancia Levenshtein es una distancia muy intuitiva y relevante, existen multitud de distancias que pueden usarse para distintos campos. No es lo mismo medir la distancia entre campos de naturaleza tan dispar como números de identificación, fechas de nacimiento, nombres de personas, calles, ciudades, etc. En el caso particular de nombres de personas, campo de nuestro problema original, se han estudiado numerosas distancias distintas y su eficiencia [7]. Existen principalmente tres modos de medir la distancia entre dos cadenas de caracteres que hacen referencia a un nombre real, puede ser de una persona, de una calle, etc. Lo importante es que sea un nombre pronunciable. En ciertos idiomas, que no incluyen al español, la fonética resulta muy relevante, puede que incluso más, que los propios caracteres usados. Por ello podemos medir la distancia entre dos elementos en base a su distancia fonética, textual, o una mezcla de ambos.

Para medir de forma fonética la distancia entre dos elementos, primero debemos transformar el string en una forma fonética. Para ello existen distintos sistemas de codificación, como Phonex, Phonix, NYSIIS... Tras codificar los strings, se procede a medir la distancia entre las codificaciones usando distancias de strings como Levenshtein. Notemos que esto no es una forma mixta de medir, pues primero se codifica y luego se mide. Este tipo de medidas son especialmente interesantes en idiomas donde la fonética es más relevante que el string con el que se represente un nombre. Un ejemplo de esto lo encontramos en inglés, donde *Mike* podría ser el mismo nombre que *Maike*, pronunciándose igual pero introduciendo una letra por error. En cambio este tipo de distancias también tienen el problema de medir mal palabras distintas. Por ejemplo, la palabra *peace* se pronuncia igual que la palabra *piece*, pero son claramente distintas.

Los métodos textuales consisten en medir directamente la distancia entre los strings. Algunos ejemplos de distancias entre strings son la de Levenshtein, la distancia JARO, la *Longest common sub-string* o LCS, etc. Hemos mencionado solo algunas de las distancias que obtienen buenos resultados, pero en función del problema a solucionar es la persona que conoce los datos sobre los que quiere ejecutar el algoritmo quien debe seleccionar cual de todas las distancias disponibles puede medir mejor la relación entre sus datos.

Por último, los métodos mixtos consisten en algoritmos de medida que hacen uso de codificación fonética a la vez que usando funciones directas de medida entre strings. Por mencionar dos de ellos, tenemos Editex y *Syllable alignment pattern searching* o SAPS.

#### Campos indefinidos

Al tratar con bases de datos siempre solemos tener que hay cierta cantidad de registros con uno o varios campos con un valor indefinido, representado por " ", NaN u otras formas. La presencia de este tipo de campos es fundamental en como debemos definir las distancias para un campo. De no tenerlos en cuenta, podemos hacer que, debido a la presencia de indefinidos, dos registros que son exactamente iguales salvo un indefinido en uno de los dos para cierto campo no lleguen a juntarse, cuando si que debiesen hacerlo. Por ejemplo, si consideramos la distancia de Levenshtein, la distancia entre un campo " " y un nombre como *Alberto* sería 7, mayor que la distancia de *Alberto* a *Arturo*, que es 5. No consideramos que tenga sentido que la distancia a un indefinido sea mayor que a un valor que claramente es distinto.

Es por estos motivos que a la hora de tratar las distancias, debe incluirse en su lógica el tratamiento de campos indefinidos de forma explícita para evitar sinsentidos. Como veremos más adelante, en las ejecuciones para analizar el rendimiento del algoritmo hemos usado que la distancia de indefinido a cualquier otro elemento (incluyendo otros indefinidos) sea 1, independientemente de la longitud, morfología u otras características del elemento al que se mide su distancia.

### 3.3.2. Factores

El segundo conjunto de parámetros a introducir son los factores, que son valores numéricos. El límite de distancia global para que dos registros hagan match es 1, por lo que dependiendo del número de campos deberemos ajustar los factores para permitir que las distancias en los campos puedan ser mayores o menores antes de pasarnos de este límite. Además, dependiendo de nuestro conocimiento sobre los datos, podemos considerar más relevante encontrar distancias mayores en un campo o en otro, lo que nos lleva a ponderar el peso que tiene cada uno en la elección final. Por

último, la propia elección de la distancia en un campo nos puede hacer querer escoger un factor distinto en función de cómo es dicha distancia.

Un ejemplo para ilustrar la importancia de los factores es el siguiente. Supongamos que, en un primer caso, tenemos un dataset dado por un único campo. En este caso, tendremos un único factor, asociado a este campo. Si queremos que el límite de distancia para aceptar un match aumente, tendremos que reducir el factor de forma inversamente proporcional para que al final obtengamos una distancia global inferior a 1, que es el límite global. Para reducir el límite de distancia en ese campo la operación sería la opuesta.

Supongamos ahora en un segundo caso que tenemos 100 campos distintos. Ahora, si usamos el mismo factor que antes estaremos siendo mucho más restrictivos, por lo que debemos reajustar los factores que usamos. Si además queremos que distancias mayores en un cierto campo hagan más fácil que no se produzca el match, lo que haremos será hacer que el factor de ese campo sea mayor que el del resto de campos. De esta forma otorgamos a los campos de nuestro dataset una importancia relativa (y absoluta si no reescalamos) sobre el posible match.

### 3.3.3. Orden

El tercer conjunto de parámetros es el orden de tomar claves. Como hemos mencionado en la sección 3.1, el uso de una estrategia de paralelización nos dificulta la anexión de registros que caen en diferentes buckets. Para solucionar este problema y conseguir la anexión de registros que inicialmente caen en buckets distintos, haremos uso de varias claves en el orden que elijamos. Este parámetro sirve para elegir qué campos y en que orden van a usarse como clave en el algoritmo. Esta elección no es banal, pues la eficiencia del algoritmo está muy ligada al tamaño de los buckets generados. Por tanto, nuevamente, es muy importante conocer bien el dataset sobre la que usarlo.

Un ejemplo de la relevancia en el orden de elección de claves lo veremos en la sección 5.2, concretamente en las tablas 5, 4 y 3. Como podemos observar, tanto el tiempo de ejecución como la eficacia a la hora de anexionar registros está fuertemente influenciado por este parámetro.

Por último, por número de iteraciones del algoritmo nos referimos a cuantos campos vamos a usar como clave. Introducimos este parámetro en esta sección puesto que en esencia el número de iteraciones es la longitud de la lista que determina el orden. Así, si escogemos como orden los campos 1,3,4 estaremos haciendo 3 iteraciones, mientras que si escogemos 1,2,4,3,5 estaremos realizando 5 iteraciones. Cuanto mayor sea el número de iteraciones, más tiempo tardará en computar, pero más nos aseguraremos de que se mezclen elementos en distintos buckets.

## 4. La lógica del algoritmo

Vamos a proceder a explicar paso por paso como se ejecuta el algoritmo que proponemos analizando el código usado para el dataset procedente del ámbito sanitario. El código completo está disponible en un repositorio de GitHub: `https://github.com/Drisktor/TFG_Matem-ticas_ER`. Se encuentra separado en 3 archivos para tener una estructura clara de qué elementos son propios de la estructura del algoritmo y qué elementos son propios de una ejecución concreta del mismo. Notemos que el código mostrado ha sido modificado para que pueda ser mostrado de forma legible

```
def distancia_para_sets(set1,set2,distance):
    distancia_temp=[1000]
    for element in set1:
            for element2 in set2:
                distancia_temp.append(distance(element,element2))
    return min(distancia_temp)

def distancia(list_of_sets_1,list_of_sets_2, list_of_factors,list_of_distances, excluyentes = 1):
    distancia = 0

    for i in range(len(list_of_sets_1)-excluyentes):

        temp = distancia_para_sets(list_of_sets_1[i],list_of_sets_2[i],list_of_distances[i])*
                *list_of_factors[i]

        distancia = distancia + temp
    return distancia
```

Figura 1: Distancia global

en este trabajo.

El primer archivo es estructura.py, donde se encuentran las funciones que definen al algoritmo en su estructura, sin tener en cuenta una posible ejecución. Comenzamos con el código de la figura 1.

En él se encuentra la función distancia global que definimos. Tiene como argumentos dos registros, los factores para ponderar la distancia y las distancias que usa cada campo. Además, tiene un parámetro opcional llamado excluyentes, que determina cuantos de los últimos campos no se usan para comparar. Esto es especialmente importante, pues antes de iniciar el algoritmo, cada registro debe tener un identificador numérico como último valor. Esto se usará al final del algoritmo para evitar repeticiones. Si el dataset no tiene este identificador, debemos añadirlo como un campo más al final de cada registro. El parámetro excluyentes toma como valor predefinido 1, considerando que este identificador no se usa para comparar. Esta función recorre campo por campo, añadiendo el valor del mínimo de la distancia elegida entre todos los elementos de los dos conjuntos comparados multiplicada por su factor correspondiente, tal y como describimos en la sección 3.2. Por tanto una llamada sobre esta función nos genera la distancia entre dos registros.

A excepción de dos, el resto de funciones de este archivo no merece ser comentado, pues se trata de operaciones de reestructuración de los registros, moviendo campos de posición para que se ajusten a la forma que se espera a la hora de ejecutar. Una de las dos relevantes es la función de match y merge. El código lo encontramos en la figura 2.

La idea de esta función es tomar los elementos que se encuentran en un bucket y aplicar sobre ellos lo que sería el equivalente a un algoritmo clásico de ER. Por este motivo es interesante no usar como clave campos que generen buckets muy grandes, pues hace la computación más compleja. Para tratar de aligerar la computación, llevamos contadores que nos permiten saber cuando el elemento actual ya se le ha aplicado un merge en algún paso anterior del bucle for (lista de comprobados). De habérselo aplicado, no es necesario tenerlo en consideración en la iteración actual.

El proceso lógico es el siguiente. Sea un elemento del bucket. Si ya ha sido anexionado a otro

```
def match_and_merge(x,factors,distances, excluyentes = 1):
    threshold = 1
    return_list = []
    key, iterador = x
    data = list(iterador)
    data.sort(key=lambda x: len(x[3]),reverse=True)
    comprobados = []
    contador = 0
    result=dict()
    i=0
    for element in data[:]:
        seguimos = True
        if i not in comprobados:
            result[contador] = (element)
            while seguimos:
                j=0
                seguimos = False
                for element2 in data[:]:
                    if j not in comprobados and distancia(result[contador],element2,factors,
                                                          distances, excluyentes) <=  threshold:
                        comprobados.append(j)
                        seguimos = True
                        temp = result[contador]
                        for cont in range(len(element2)):
                            temp[cont] = temp[cont].union(element2[cont])
                        result[contador] = (temp)
                    j+=1
            contador+=1
        i+=1
    for i in range(len(result)):
        return_list.append((key,result[i]))
    return return_list
```

Figura 2: Función de match y merge.

```python
from Levenshtein import distance as L_distance

######Distancias definidas

def L_distance_vacios(element1,element2):
    if element1 == '' or element2 == '':
        return 1
    else:
        return L_distance(element1,element2)

def distancia_levenstein_separando_vacios(element1,element2):
    set1= set(element1.split(' '))
    set2= set(element2.split(' '))
    distancia_temp = [1000]
    if '' in set1 or '' in set2:
        distancia_temp.append(1)
    for element in list(set1):
            for element_2 in list(set2):
                distancia_temp.append(L_distance_vacios(element,element_2))
    return min(distancia_temp)
```

Figura 3: Ejemplos de distancias que podemos definir.

lo ignoramos. En caso contrario, lo añadimos a nuestro diccionario de resultados. A continuación, buscamos en todos los registros que no han sido anexionados a otros si hay match con alguno. De haberlo, modificamos la entrada del diccionario de resultados con el merge de ambos registros. Por la falta de transitividad de nuestra función, puede haber registros que inicialmente no hiciesen match, pero que tras haber convertido nuestro registro en uno que almacena más información si que lo haga. Para ello, introducimos un bucle while que hace que este proceso se repita mientras obtengamos al menos un match nuevo. Por la finitud del bucket este proceso termina necesariamente. Una vez no encontramos más posibles matches, pasamos al siguiente elemento del bucket y repetimos el proceso hasta terminarlo.

La otra función de este archivo que merece la pena comentar es la función que propiamente ejecuta el algoritmo. No obstante, creemos que es mejor para entender por completo su funcionamiento comentar antes los otros dos archivos que componen el algoritmo.

El archivo `distancias.py` es un archivo donde se cargan las distancias que se van a querer usar en la ejecución del algoritmo. En nuestro caso, se encuentra cargada la distancia Levenshtein en dos versiones. La primera de ellas es la distancia de Levenshtein propiamente dicha añadiendo la condición de que la distancia de un indefinido a cualquier otro elemento sea 1. La segunda se trata de una modificación hecha sobre esta distancia para medir mejor la distancia entre nombres compuestos, separando los nombres como *Jose Carlos* en $\{Jose,\ Carlos\}$ y aplicando el mínimo de las distancias de este conjunto con otro elemento. El código sería el observado en la figura 3.

En cuanto a `Algoritmo.py`, se trata de un archivo donde se definen los detalles específicos de la ejecución a realizar y se realiza esa ejecución. En primer lugar, cargamos las funciones que están definidas en los otros 2 archivos mencionados y definimos los parámetros tratados en la sección 3.3, así como los preparamos para que tengan una forma adecuada para la ejecución. El código completo se encuentra en la figura 4.

```python
import numpy as np
from estructura import *
from distancias import *


#######Espacio de parametros

archivo = '*****************'
orden = [1,3,0,2]
factores_origen = [1,0.25,0.25,0.5]
distancias_origen = [L_distance_vacios,L_distance_vacios,
                     distancia_levenstein_separando_vacios,L_distance_vacios]
multiplicador = 2/3


#### Preparación parámetros

longitud = len(orden)
factores=[0 for x in range(len(orden))]
for i in range(len(orden)):
    factores[i] = factores_origen[:orden[i]]+factores_origen[orden[i]+1:]

distancias=[0 for x in range(len(orden))]
for i in range(len(orden)):
    distancias[i] = distancias_origen[:orden[i]]+distancias_origen[orden[i]+1:]

for i in range(len(factores)):
    factores[i] = [multiplicador*x for x in factores[i]]

list_of_maps = [seleccionar_map(orden[0])]
list_of_matchs = []
for i in range(len(orden)-1):
    list_of_maps.append(seleccionar_map_intermedio(orden[i+1],orden[i]))

for i in range(len(orden)):
    list_of_matchs.append(seleccionar_match(factores[i],distancias[i]))

last_match = seleccionar_match(factores[0],distancias[0], excluyentes = 2)
```

Figura 4: Definición y preparación de parámetros.

```python
##### Función de preparación del dataset

def mapeo_desde_archivo(x, l = 5):
    datos = x.split(';')
    for i in range(len(datos)):
        if i == l-1:
            datos[i] = set(eval(datos[i]))
        else:
            datos[i] = set([datos[i]])
    return datos
```

Figura 5: Ejemplo de función de preparación del dataset.

Comenzamos por el orden en que los campos van a ser usados como clave. Seguimos con la generación de los factores y distancias a usar en cada iteración del algoritmo. En el siguiente paso introducimos un multiplicador. La función de este multiplicador es reescalar los factores. De esta forma podemos escribir inicialmente cual el peso relativo de cada campo, lo cual es más sencillo de realizar. Reducir este multiplicador tiene el mismo efecto que aumentar el límite que no puede sobrepasar la distancia entre dos registros para juntarlos, mientras que aumentarlo tiene el efecto contrario. Así, reducirlo es equivalente a ser más conservadores a la hora de juntar registros y aumentarlo es ser menos conservador.

A continuación, guardamos en listas una serie de funciones para poder realizar de golpe varias iteraciones del algoritmo. Los maps corresponden a las transformaciones que tenemos que hacer entre iteraciones, que consisten en retornar el campo que estamos usando como clave al resto del registro y separar el nuevo campo clave del registro. Claramente este proceso está definido en función del orden escogido de campos. Notemos que la primera función de map es distinta al resto, pues solo tiene como argumento el elemento a ser clave, ya que no había ninguno siendo clave anteriormente. Respecto a los match que se observan, lo que se hace es añadir una función conjunta de match y merge, que en cada iteración es distinta al cambiar los factores y las distancias usadas. Queda por comentar la funcion *last_match*, pero para entender bien su relevancia la comentaremos en la propia ejecución.

Por último, debemos definir una función que prepare el dataset desde el archivo a la forma que debe tener (a saber, una lista de listas donde cada elemento es un conjunto de valores). Esta función dependerá de cómo esté guardado el dataset inicialmente. Un ejemplo de función de preparación sería la mostrada en la figura 5.

La ejecución del algoritmo se realiza en la última función del archivo `estructura.py`, y es ejecutada como mostramos en la figura 6.

Tras ejecutar el algoritmo, mostramos el tiempo de ejecución y guardamos el resultado en un archivo *csv* para su posterior análisis.

Antes de comentar cómo se realiza la ejecución, vamos a describir una operación que se lleva a cabo durante la misma: el despliegue. Recordemos que todos los elementos del dataset deben ser conjuntos. Por ello, cuando usamos un elemento como clave, solo formará un bucket con otros registros con exactamente el mismo conjunto como clave. Esto en la primera iteración no es pro-

```
#### Ejecución



result, tiempo =
↪  algoritmo(orden,list_of_maps,list_of_matchs,last_match,archivo,mapeo_desde_archivo)

print('Fin: {} s'.format(tiempo))

with open(archivo.split('.')[0]+'_resultado.csv','w') as f:
    for item in result:
        f.write('%s\n' % item)
```

Figura 6: Llamada de ejecución del algoritmo.

```
0  def algoritmo(orden,list_of_maps,list_of_matchs,last_match,archivo,mapeo_preparacion):
1      start = t.time()
2      rdd_op = sc.textFile(archivo).map(mapeo_preparacion)
3      for i in range(len(orden)):
4          rdd_op =
           ↪  rdd_op.map(list_of_maps[i]).flatMap(separar_keys).groupByKey().flatMap(list_of_matchs[i])
5      rdd_op = rdd_op.map(seleccionar_map_intermedio(orden[0],orden[i]))
6      rdd_op = rdd_op.map(copiar_leading_set).flatMap(separar_keys)
7      rdd_op = rdd_op.groupByKey().flatMap(last_match).map(orden_final)
8      rdd_op = rdd_op.map(map_registros).groupByKey().map(merge_registros)
9      result = rdd_op.collect()
10     end= t.time()
11     return result, end-start
```

Figura 7: Pasos del algoritmo.

blemático, pero en las sucesivas es desastroso, pues dos elementos que tienen distancia 0 como son $\{A,\ B\}$ y $\{A\}$ no acabarían en el mismo bucket. Esencialmente son iguales, por lo que queremos que sus registros asociados acaben en el mismo bucket. Por ello, lo que haremos será desplegarlos en varios registros iguales, donde la clave de cada registro será un elemento de dicho conjunto.

Como adelantábamos en la sección 3.1, esta operación de despliegue nos hace desviarnos de la lógica de la *Union Class*. Es inevitable usarla si queremos generar buenos buckets donde comparar elementos, pero a cambio nos vemos forzados a generar registros donde se pierde información al realizarla. Esto implica que el orden de ejecución será relevante a la hora de obtener resultados. Como también mencionábamos en dicha sección, al seguir el resto del tiempo la lógica de la *Union Class*, esperamos que vernos forzados a esta operación no nos distancie demasiado de la solución óptima.

Finalmente, el código de la ejecución se encuentra dentro de la siguiente función. Es importante remarcar que esta sección de código es en la que realmente se usa Spark como herramienta de paralelización. El resto de funciones son puro código en Python que se ejecutará de forma secuencial en cada uno de las máquinas que compongan el cluster. Vamos a ir comentando este código linea por linea para facilitar su comprensión. El código se muestra en la figura 7.

Linea 1: Iniciamos el reloj para llevar la cuenta del tiempo de la ejecución del algoritmo.

Linea 2: Definimos el RDD cargando desde archivo el dataset sobre el que ejecutamos. A la vez, usamos la función para cargar desde el archivo los datos en un formato útil para ejecutar (incluyendo la transformación en conjuntos). Recordamos en este punto que los datos deben venir con un identificador único de cada registro como último campo del respectivo registro. De no tenerlo habría que añadirlo previamente.

Lineas 3 y 4: En cada iteración de este bucle colocamos el elemento que va a ser clave en la iteración y lo desplegamos. A continuación agrupamos los elementos por su clave para formar los buckets y ejecutamos las funciones de match y merge. Iteramos este proceso mientras el parámetro orden nos lo ordene.

Lineas 5 y 6: Tras realizar todas las operaciones, colocamos el dataset de forma que la nueva clave vuelva a ser el campo fue clave en el primer paso. En este punto tenemos el problema de que tras desplegar, hay campos cuyas valores no son lo completos que deberían ser como consecuencia de ese despliegue. Por ello volvemos a ejecutar usando como clave la misma que en el primer paso, pero en esta ocasión guardando qué conjunto es el que antes de desplegar estaba actuando como clave. Lo guardamos al final de nuestro registro y desplegamos.

Linea 7: Agrupamos por la clave, y ejecutamos last match. Esta función tiene en cuenta que hemos copiado el conjunto clave antes del despliegue, y por ello tiene como parámetro excluyentes 2. Además, usa los factores y las distancias de la primera iteración. El último map observado, que usa la función orden final, elimina la existencia de una clave y reordena el registro, dejándolo de la forma [campo[orden[0]],resto de campos ordenados según su orden inicial, excluyendo el orden[0]].

Linea 8: En este punto hay varios registros que no son copias unos de otros, pero que han juntado exactamente los mismos identificadores de registro. Por este punto es importante tener estos identificadores. Debido al funcionamiento del algoritmo, los elementos del campo que es clave en último lugar no se agrupan correctamente, si no que se reparten en varias copias del mismo registro pero cambiando ese campo. Para unificarlo, haremos una última agrupación. En este caso, la clave será el conjunto de identificadores sin desplegar. De este modo solo se unirán registros que procedan de los mismos registros iniciales. En este caso, no hacemos ningún match, unimos directamente con merge registros, definida en `estructura.py` .

Linea 9: Finalmente, llamamos al método collect para realizar las operaciones definidas sobre el RDD y llevar a cabo la ejecución.

Linea 10: Terminamos el reloj para obtener el tiempo de ejecución.

Linea 11: Devolvemos el resultado obtenido y el tiempo de ejecución.

# 5. Aplicación sobre datasets concretos

En esta sección pretendemos analizar los resultados que se obtienen al ejecutar el algoritmo frente a los dos tipos de datasets mencionados en la introducción. Comenzaremos en primer lugar con los datasets de los cuales disponemos la solución y continuaremos con el dataset de nuestro problema original.

## 5.1. FEBRL

Estos datasets están generados a partir de paquete *Freely Extensible Biomedical Record Linkage* (*FEBRL*). Los 4 datasets asociados los hemos obtenido del paquete de Python *recordlinkage*. De estos 4 datasets, el número 4 (del cual obtenemos realmente 2 datasets) tiene todos sus registros correspondientes a entidades distintas, por lo que lo usaremos para comprobar que el algoritmo no esté resolviendo mal algunas entidades. Como parámetros al algoritmo, usaremos como factores 0.5 si el campo es numérico y 0.25 si el factor es un string. Además, usaremos un multiplicador de 0.2 para ser poco conservadores a la hora de hacer match. Como función distancia usaremos en todos la distancia Levenstein, considerando que la distancia a cualquier string vacío es 1. Con respecto al orden, haremos 3 iteraciones sobre los campos 5, 6 y 3, en orden de enumeración. Esta elección no está realizada al azar. Se ha observado cómo es la distribución de frecuencias de todos los campos para buscar cuales son los más apropiados. Un análisis de cómo afecta esta distribución a la ejecución del algoritmo lo veremos en la sección 5.2. Los resultados obtenidos los presentamos en la tabla 1.

| | Nº elementos | Tiempo (s) | Duplicados | Detectados | Detectados ( %) |
|---|---|---|---|---|---|
| fbrl1 | 1000 | 0.76 | 500 | 456 | 91.2 |
| fbrl2 | 5000 | 2.28 | 1000 | 934 | 93.4 |
| fbrl3 | 5000 | 2.10 | 3000 | 2783 | 92.8 |
| fbrl4a | 5000 | 2.73 | 0 | 0 | - |
| fbrl4b | 5000 | 3.24 | 0 | 0 | - |

Tabla 1: Ejecución de nuestro algoritmo sobre los dataset obtenidos de FEBRL

Como podemos observar de la tabla 1, el algoritmo no parece haber cometido el error de juntar registros que no deben juntarse, lo cual puede ser (dependiendo del contexto) una propiedad muy importante que debe poseer. Además, ha obtenido muy buenos resultados de detección, por encima del 90 % en los 3 datasets que tenían duplicidades.

En cuanto al tiempo de la ejecución, es remarcable que el tiempo se triplica aproximadamente en un dataset que multiplica el número N de registros por 5 (fbrl1 vs fbrl 2 y 3). Este crecimiento pequeño en tiempo de ejecución se debe a la elección de un buen orden de ejecución del algoritmo, pues los campos elegidos tienen una buena distribución para no generar buckets muy grandes (lo que aumentaría la complejidad) ni muy pequeños (lo que haría difícil realizar la operación de merge).

Para poder poner en contexto estos resultados obtenidos para nuestro algoritmo sería interesante poder compararlo con un algoritmo clásico de ER que use las mismas funciones de match y merge. Este algoritmo tendría a su favor que no separa en buckets, por lo que a priori no puede haber dos registros que no lleguen a juntarse si el match lo permite y que sigue la *Union Class*

tal y como se definió inicialmente. Por este motivo tiene garantizado alcanzar el mejor resultado que puede alcanzarse con dicha función de match. Los resultados de ejecutar este algoritmo clásico sobre los datasets anteriores quedan reflejados en la tabla 2.

| | Nº elementos | Tiempo (s) | Duplicados | Detectados | Detectados ( %) |
|---|---|---|---|---|---|
| fbrl1 | 1000 | 17.72 | 500 | 461 | 92.2 |
| fbrl2 | 5000 | 1475 | 1000 | 947 | 94.7 |
| fbrl3 | 5000 | 880 | 3000 | 2808 | 93.6 |
| fbrl4a | 5000 | 1669 | 0 | 0 | 0 |
| fbrl4b | 5000 | 1603 | 0 | 0 | 0 |

Tabla 2: Ejecución del algoritmo clásico sobre los dataset obtenidos de FEBRL.

Como podemos observar con una simple comparación entre ambas tablas, el algoritmo clásico obtiene una mayor precisión. De hecho, al seguir la lógica de la *Union Class*, este es en teoría el mejor resultado que podemos obtener con esta función match. No obstante resulta muy llamativo el aumento en el tiempo de ejecución. Los tiempos pasan de unos 2 segundos a valores del orden de 1000 segundos. En nuestra opinión, este aumento del coste computacional no es justificable para obtener la poca precisión adicional que se obtiene, incluso en el caso de datasets pequeños. Recordemos que el dataset fbrl1 tiene 1000 elementos mientras que los otros 4 datasets tienen 5000 elementos. Cuando consideramos ejecutarlo sobre el dataset sobre el cual planteamos inicialmente este TFG, de 35284 elementos, el uso de un algoritmo clásico podría hacer uso de mucho tiempo de ejecución.

Así pues observamos que nuestro algoritmo presenta muy buenos resultados en un número bajo de elementos en el dataset. Su tiempo de ejecución es muy rápido en comparación al algoritmo clásico y la precisión obtenida es muy similar a la mejor que puede obtenerse. Es de esperar que cuando aumentemos el número de elementos en el dataset nuestro algoritmo funcione aún mejor que el clásico, pues su crecimiento debe ser menor, si hacemos una buena elección de los parámetros, al hacer uso del blocking.

A continuación pues, estudiaremos como se comporta nuestro algoritmo sobre nuestro dataset problema. Comencemos por describir el dataset y los parámetros usados para configurar el algoritmo.

## 5.2. Dataset médico

El dataset se compone de 5 columnas: ID, apellidos, nombre, fecha de nacimiento y registros de pruebas. De estos 5 campos, el de registros de pruebas actúa como identificador de registro, pues cada elemento es único en el dataset. Sobre la elección de parámetros, sobre ID, apellidos y fecha de nacimiento usamos la misma distancia que en los datasets anteriores. No obstante, en los nombres usamos esta distancia ligeramente modificada. Debido a la presencia de nombres compuestos, decidimos que la distancia entre dos nombres era el mínimo de la distancia entre los constituyentes de ese nombre. Así, la distancia entre, por ejemplo, *María Teresa* y *María* sería el mínimo entre las distancias *María* a *María* y *Teresa* a *María*, 0 en este caso. La elección de los factores fue: 0.25 para nombre y apellido, 0.5 para fecha y 1 para ID. Como en este caso estamos tratando con un dataset real, no disponemos de un dataset similar para saber si estamos haciendo

match de registros que no queremos que hagan match. Por ello, usaremos un multiplicador bastante más conservador que en el caso de los datasets *FEBRL* para tratar de evitar esta situación: $\frac{2}{3}$.

Para tratar el orden, como el tiempo de ejecución del algoritmo en este dataset no era demasiado elevado, pudimos ejecutarlo con todas las posibles permutaciones de los 4 campos. Además, para todas estas permutaciones hicimos el algoritmo usando los 2,3 y 4 primeros elementos de orden para observar como evolucionaba el coste y la precisión al iterar más sobre él. Usaremos la siguiente notación para los campos de este dataset: el campo 0 corresponde a ID, el campo 1 a apellido, el 2 a nombre y el 3 a fecha de nacimiento. Todos estos datos obtenidos los vamos a presentar en las tablas 5, 4 y 3 . Estos datos fueron obtenidos usando como procesador un i7-10510U.

| Orden | Tiempo (s) | Detecciones | Detecciones/total ( %) |
|---|---|---|---|
| [0,1,2,3] | 53.10 | 11984 | 33.96 |
| [0,1,3,2] | 53.86 | 11984 | 33.96 |
| [0,2,1,3] | 50.34 | 11984 | 33.96 |
| [0,2,3,1] | 48.30 | 11984 | 33.96 |
| [0,3,1,2] | 53.15 | 11984 | 33.96 |
| [0,3,2,1] | 49.64 | 11984 | 33.96 |
| **[1,0,2,3]** | **30.39** | **11985** | **33.97** |
| **[1,0,3,2]** | **31.45** | **11985** | **33.97** |
| [1,2,0,3] | 36.63 | 11986 | 33.97 |
| [1,2,3,0] | 35.92 | 11986 | 33.97 |
| [1,3,0,2] | 35.51 | 11986 | 33.97 |
| [1,3,2,0] | 39.21 | 11986 | 33.97 |
| [2,0,1,3] | 41.12 | 11982 | 33.96 |
| [2,0,3,1] | 41.55 | 11985 | 33.97 |
| [2,1,0,3] | 45.76 | 11985 | 33.97 |
| [2,1,3,0] | 48.09 | 11985 | 33.97 |
| [2,3,0,1] | 46.23 | 11986 | 33.97 |
| [2,3,1,0] | 45.93 | 11985 | 33.97 |
| **[3,0,1,2]** | **29.75** | **11985** | **33.97** |
| **[3,0,2,1]** | **29.68** | **11985** | **33.97** |
| [3,1,0,2] | 33.42 | 11986 | 33.97 |
| [3,1,2,0] | 38.16 | 11986 | 33.97 |
| [3,2,0,1] | 37.12 | 11986 | 33.97 |
| [3,2,1,0] | 34.79 | 11986 | 33.97 |

Tabla 3: Ejecución de nuestro algoritmo con 4 iteraciones para todas las combinaciones de orden posibles. Remarcamos en negrita los mejores resultados en tiempo de ejecución.

Para comentar estos datos, debemos primero mencionar que el tiempo de ejecución no era el mismo siempre, si no que dependía de los procesos del ordenador, variando alrededor de un 20 % de los valores proporcionados. Observando el algoritmo con 4 iteraciones, observamos que, independientemente del orden, siempre llegamos a una situación similar, con un número casi igual de elementos anexionados. Lo que si varía son los tiempos de ejecución, variando desde 29 segundos en las iteraciones más rápidas a 53 segundos en las más lentas. Esta diferencia no se debe a la variabilidad que hemos comentado, de alrededor del 20 %, si no que se debe al tamaño de los buckets

| Orden | Tiempo (s) | Detecciones | Detecciones/total ( %) |
|---|---|---|---|
| [0,1,2] | 54.12 | 11973 | 33.93 |
| [0,1,3] | 40.18 | 11984 | 33.96 |
| [0,2,1] | 46.42 | 11973 | 33.93 |
| [0,2,3] | 47.73 | 11977 | 33.94 |
| [0,3,1] | 37.15 | 11984 | 33.96 |
| [0,3,2] | 52.13 | 11977 | 33.94 |
| [1,0,2] | 29.45 | 11973 | 33.93 |
| [1,0,3] | 20.10 | 11985 | 33.97 |
| [1,2,0] | 36.10 | 11973 | 33.93 |
| **[1,2,3]** | **13.79** | **11955** | **33.88** |
| [1,3,0] | 24.96 | 11985 | 33.97 |
| **[1,3,2]** | **15.84** | **11955** | **33.88** |
| [2,0,1] | 39.43 | 11971 | 33.93 |
| [2,0,3] | 40.15 | 11975 | 33.94 |
| [2,1,0] | 43.82 | 11971 | 33.93 |
| [2,1,3] | 26.85 | 11954 | 33.88 |
| [2,3,0] | 45.77 | 11976 | 33.94 |
| [2,3,1] | 25.86 | 11955 | 33.88 |
| [3,0,1] | 19.15 | 11985 | 33.97 |
| [3,0,2] | 28.99 | 11978 | 33.95 |
| [3,1,0] | 23.03 | 11986 | 33.97 |
| **[3,1,2]** | **13.91** | **11955** | **33.88** |
| [3,2,0] | 36.23 | 11974 | 33.94 |
| **[3,2,1]** | **14.51** | **11955** | **33.88** |

Tabla 4: Ejecución de nuestro algoritmo con 3 iteraciones para todas las combinaciones de orden posibles. Remarcamos en negrita los mejores resultados en tiempo de ejecución.

generados durante el algoritmo. Por ello hemos insistido en lo relevante de una buena elección de orden de ejecución.

Esta relevancia queda aún más palpable en el caso de ejecuciones con 3 iteraciones. En este caso, observamos que la diferencia en las ejecuciones más rápida y más lenta es mayor, unos 14 segundos frente a 54 segundos. Además, también observamos que las ejecuciones más rápidas se producen cuando el primer elemento del orden es o bien 1 o bien 3, lo que nos indica que son mejores campos para trabajar este algoritmo. También observamos que con 3 iteraciones, los números de registros anexionados, si bien muy similares, ya no son tan homogéneos. Hay varias diferencias que nos hacen ver que el orden es relevante siempre y cuando no estemos iterando sobre todos los campos, donde su efecto es prácticamente despreciable.

Finalmente si observamos las ejecuciones con 2 iteraciones, vemos amplificada la variabilidad en tiempo de ejecución y resultado obtenido. Nuevamente los más rápidos son aquellos con los campos 1 y 3, que son además los que obtienen mejores resultados en 2 iteraciones.

Para ver por qué un campo resulta mejor que otro para realizar las ejecuciones del algoritmo podemos analizar la distribución de las entradas de dichos campos. Tomando como distribución a

| Orden | Tiempo (s) | Detecciones | Detecciones/total (%) |
|---|---|---|---|
| [0,1] | 35.75 | 11839 | 33.55 |
| [0,2] | 40.70 | 11742 | 33.28 |
| [0,3] | 33.08 | 11795 | 33.43 |
| [1,0] | 18.99 | 11836 | 33.54 |
| [1,2] | 12.97 | 11759 | 33.33 |
| **[1,3]** | **3.38** | **11858** | **33.61** |
| [2,0] | 38.38 | 11748 | 33.30 |
| [2,1] | 24.80 | 11750 | 33.30 |
| [2,3] | 25.66 | 11782 | 33.39 |
| [3,0] | 18.38 | 11796 | 33.43 |
| **[3,1]** | **3.34** | **11858** | **33.61** |
| [3,2] | 13.65 | 11748 | 33.30 |

Tabla 5: Ejecución de nuestro algoritmo con 2 iteraciones para todas las combinaciones de orden posibles. Remarcamos en negrita los mejores resultados en tiempo de ejecución.

analizar el número de veces que se repite cada valor (y por tanto el tamaño del bucket que genera cuando lo usamos como clave), obtenemos tanto la media, como la desviación típica y el valor máximo observado de dicha distribución. Estos datos los mostramos en la tabla 6. Como podemos observar los campos que ofrecen peores resultados son aquellos que tienen una distribución con una mayor desviación típica y un mayor bucket máximo.

| Campo | Media | Desv. Tipica | Maximo |
|---|---|---|---|
| ID | 1.47 | 7.62 | 1171 |
| Apellido | 1.67 | 2.01 | 71 |
| Nombre | 7.17 | 30.36 | 693 |
| Fecha | 2.32 | 1.78 | 17 |

Tabla 6: Distribución de los campos en el dataset problema.

Con estas observaciones, podemos afirmar, ahora con evidencias, que para la ejecución de este algoritmo es muy necesario hacer un breve estudio previo del dataset para determinar cual es el mejor orden de ejecución. También observamos que aumentar el número de elementos de orden, si bien lleva a un aumento del tiempo de ejecución, mejora los resultados y los hace menos dependiente del orden escogido. También un preprocesado previo sobre los datos puede ayudar a eliminar ciertos errores en las registros del dataset que sabemos a priori que lo son o permitir unificar distintas formas de escribir un cierto valor (como por ejemplo en los nombres de este dataset). La eliminación de estos errores conocidos ayuda al algoritmo a hacer match de registros que queremos juntar.

Para poder valorar correctamente los valores absolutos de registros anexionados lo más lógico es compararlo con cuántos registros logramos anexionar usando los mismos principios sobre el dataset completo, sin hacer uso del blocking. Para ello hemos ejecutado ese principio de anexión siguiendo el algoritmo clásico sobre los 35284 elementos, con comparaciones 1 a 1. Hemos realizado esta ejecución sobre un servidor con un hardware de menor calidad que el resto de ejecuciones mos-

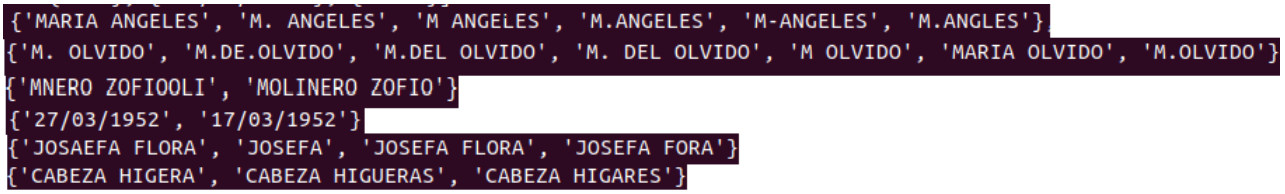


Figura 8: Ejemplos de anexión reales.

tradas, por lo que se ha tardado una semana en terminar el cómputo. Finalmente, se han realizado 11958 anexiones. Es muy remarcable el hecho de que este número, que a priori debería ser superior al número de anexiones realizadas con el algoritmo desarrollado, es inferior. Esto se debe a que al medir distancias en el algoritmo desarrollado se comparan 3 columnas, mientras que el clásico hace uso de las 4. Esto puede generar una pequeña desviación de los resultados, como observamos, aunque es de esperar que no muy relevante.

Por estos datos obtenidos, podemos afirmar que el número de anexiones que realiza el algoritmo desarrollado es muy similar al máximo de anexiones posibles con los parámetros proporcionados en un tiempo notablemente inferior al algoritmo sin hacer uso de blocking.

Para terminar esta sección, vamos a mostrar en una figura algunos ejemplo de anexiones reales de registros. Por motivos de confidencialidad solo mostraremos un único campo del registro anexionado, pero es suficiente para mostrar las capacidades que tiene y los resultados que puede lograr, así como las diferentes formas que hay de designar a una misma entidad. Los mostramos en la figura 8.

### 5.2.1. Escalado en el tiempo de ejecución

Una cuestión interesante de analizar es cuán rápido escala el tiempo de ejecución del algoritmo desarrollado conforme aumentamos el tamaño del dataset. Para ello hemos troceado el dataset original en datasets de menor tamaño y hemos ejecutado tanto el algoritmo clásico como el que hemos desarrollado sobre estos datasets reducidos para ver este escalado. Para cuantificarlo, vamos a usar la librería Scipy de Python para ajustar los datos a una función del tipo $a \cdot x^b$. Los resultados los mostramos en la tabla 7. Este análisis también lo hemos realizado sobre distintas versiones del algoritmo desarrollado para ver cómo ejecutar el algoritmo con más o menos iteraciones (mayor o menor longitud de la lista *orden*) afecta al crecimiento del coste computacional. Los ajustes obtenidos se muestran en las figuras 9, 10, 11 y 12.

Como podemos observar en estos resultados, el coste computacional del algoritmo se reduce al hacer uso de la técnica de blocking, reduciendo el tamaño de los buckets en los que se realizan las comparaciones 1 a 1. Observamos que existe una notable diferencia entre el exponente para 3 y 4 iteraciones frente al que observamos con 2 iteraciones. Eso se debe a que en las primeras iteraciones estamos usando como clave un campo que no genera buckets muy grandes, pero al entrar en la tercera iteración sí que se usa una clave que tiene este problema, aumentando notablemente el coste. Concretamente usamos el campo 0 en esta tercera iteración. A pesar del aumento

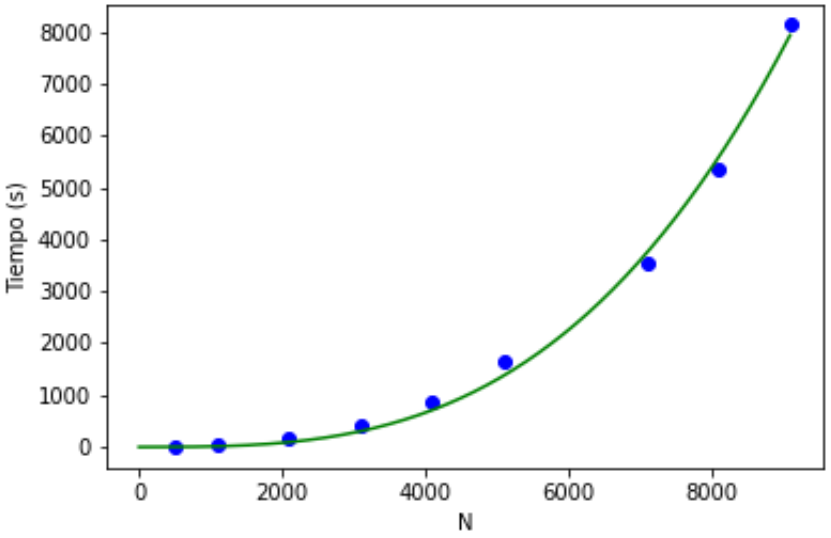


Figura 9: Ajuste a los tiempos de ejecución obtenidos para el algoritmo clásico.

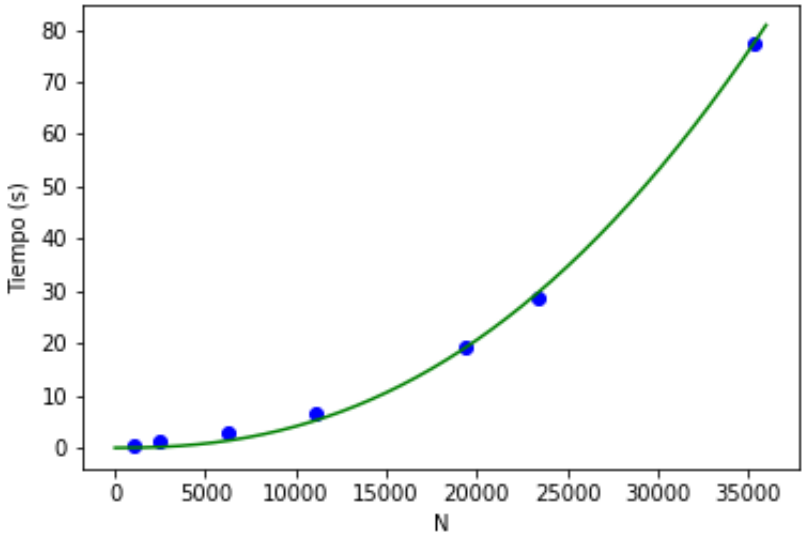


Figura 10: Ajuste a los tiempos de ejecución obtenidos para el algoritmo desarrollado con 4 iteraciones.

| Algoritmo | Exponente |
|---|---|
| Clásico | 3.02 |
| 4 iteraciones | 2.32 |
| 3 iteraciones | 2.24 |
| 2 iteraciones | 1.11 |

Tabla 7: Exponente de crecimiento de distintas versiones del algoritmo.

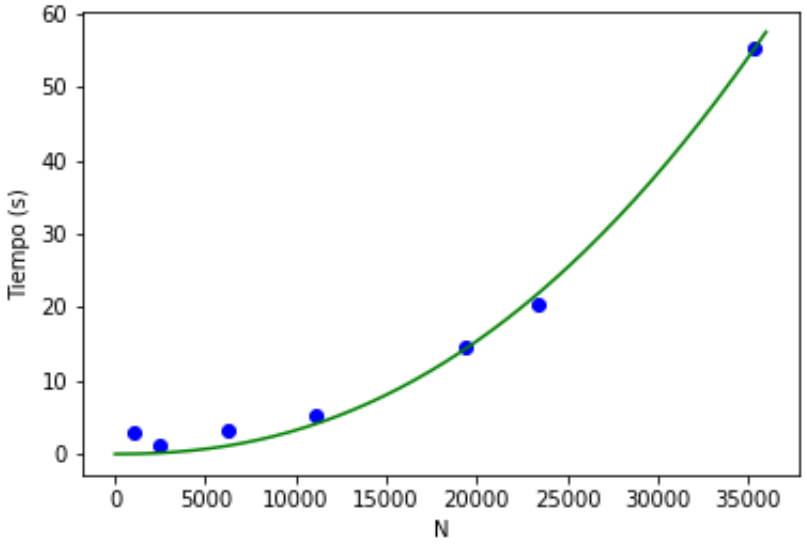


Figura 11: Ajuste a los tiempos de ejecución obtenidos para el algoritmo desarrollado con 3 iteraciones.

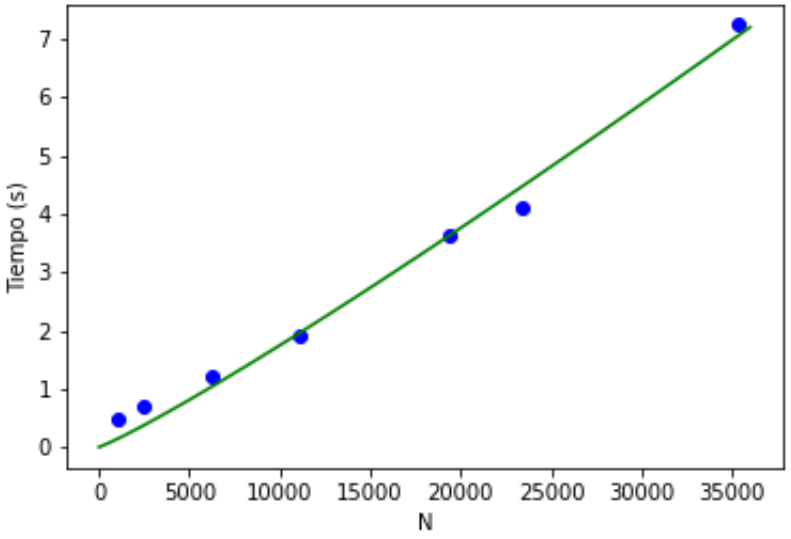


Figura 12: Ajuste a los tiempos de ejecución obtenidos para el algoritmo desarrollado con 2 iteraciones.

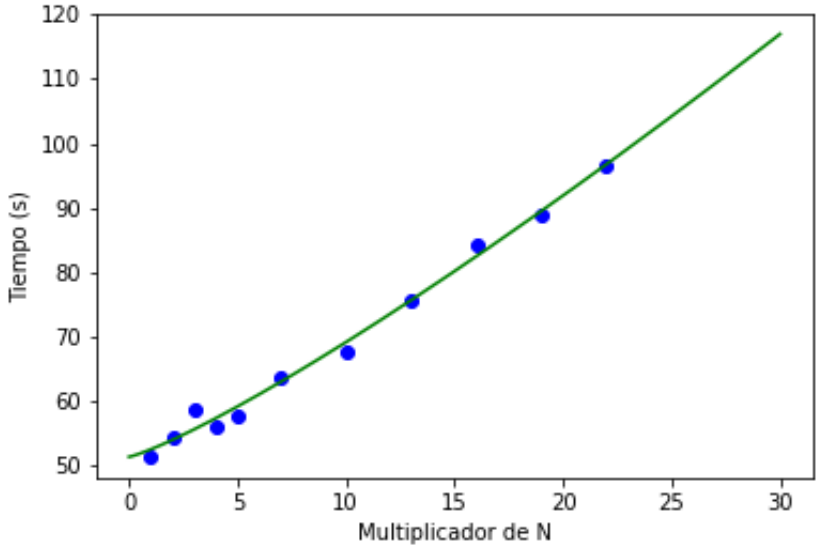


Figura 13: Ajuste a los tiempos de ejecución obtenidos para el algoritmo desarrollado con 4 iteraciones en un cluster de ordenadores.

significativo en el coste computacional, los resultados obtenidos usando 2 o 3 iteraciones son similares (11858 frente a 11985 elementos anexionados). Por ello dependiendo del tamaño del dataset que tengamos y el objetivo de precisión que tengamos nos interesará hacer más o menos iteraciones.

Como último análisis del escalado en tiempo de ejecución, hemos querido operar el algoritmo en el entorno para el cual está diseñado: un cluster de ordenadores. Hemos hecho uso del algoritmo en su versión de 4 iteraciones en distintos tamaños del dataset para ver cómo evolucionaban los tiempos de ejecución. Una observación que hemos realizado es la existencia de un tiempo de ejecución mínimo. Para poder llevar a cabo la ejecución y que todos los datos se distribuyan correctamente en el cluster parece que la ejecución no puede bajar de cierto tiempo, independientemente del tamaño del dataset. Por ello, ajustaremos los datos a la expresión $a \cdot x^b + c$.

Otro efecto que hemos observado es como los tiempos para los tamaños de dataset que tenemos no varían. Es decir, tarda aproximadamente lo mismo en uno 20000 elementos que en 40000. Para observar bien el escalado, hemos aumentado el tamaño de nuestro dataset haciendo copias del mismo y añadiéndolas. De esta forma, representaremos el escalado como tiempo de ejecución frente a número de veces que hemos ampliado el dataset, mientras que el ajuste se hará con el número de elementos N. La representación de los datos obtenidos es la figura 13. El exponente que se obtiene para este ajuste es 1.19. Como podemos notar, es un valor notablemente inferior al que habíamos obtenido cuando ejecutábamos este mismo algoritmo en una sola máquina con varios núcleos, donde obteníamos un exponente de 2.32. Así, el buen aprovechamiento de la paralelización convierte el algoritmo en escalable.

# 6. Conclusiones

En este trabajo, hemos querido desarrollar un algoritmo que nos permitiese discernir en un dataset cuando dos o más registros representaban a la misma entidad o persona. Para ello, hemos desarrollado un marco teórico para el tratamiento del proceso ER (*Entity Resolution*) con el objetivo de desarrollar un algoritmo que realice este proceso sobre nuestro dataset inicial. Este algoritmo lo hemos basado en tres principios fundamentales. El primero es la escalabilidad del tiempo de computación con el tamaño del dataset que queremos tratar, cuestión que hemos abordado construyendo un algoritmo paralelo con Spark.

En segundo lugar, queríamos alcanzar el resultado óptimo del proceso. Es decir, obtener un conjunto de registros donde hubiese un registro y solo uno por cada entidad real presente en el dataset a tratar. Esto lo hemos planteado siguiendo la lógica de la *Union Class*, introducida en la sección 2.2, estableciendo unas funciones de match y merge con unas propiedades (ICAR) que nos garantizasen que se obtiene el mejor resultado posible. Sin embargo, debido a la necesidad de usar claves para lograr la paralelización, se tuvo que renunciar a seguir fielmente este principio, siguiendo una lógica que solo se desviaba de la planteada en el marco teórico en lo estrictamente necesario. Con esto esperábamos que el resultado obtenido fuese lo más similar posible al óptimo.

Y en tercer lugar, queríamos que el algoritmo fuese adaptable a distintos datasets, lo cual hemos conseguido definiendo un conjunto de parámetros modificables que proporcionan la libertad necesaria para poder ejecutar el algoritmo sobre distintos tipos de datasets.

Tras construir el algoritmo y presentarlo en la sección 4, hemos procedido a analizar si los resultados de su ejecución están conformes a las expectativas. En primer lugar, analizamos la ejecución del algoritmo sobre datasets de los cuales ya se conocía la solución, los datasets *FEBRL*. Hemos comparado los resultados con un algoritmo que usa la misma lógica que el planteado, pero que no paraleliza y, por tanto, sigue completamente el marco teórico presentado. La conclusión es clara, nuestro algoritmo obtiene una precisión muy similar (por debajo, como era de esperar, al no seguir nuestro algoritmo fielmente la *Union Class*) a la que obtiene el algoritmo clásico, pero con tiempos de ejecución significativamente inferiores.

A continuación, ejecutamos el algoritmo con un dataset sin resolver y con un tamaño significativamente superior (pasamos de 5000 a 35284 elementos). En este caso la diferencia en tiempo de ejecución con el algoritmo sin paralelizar resulta aún más notable que con los datasets anteriores sin una notable pérdida de precisión, independientemente del parámetro de orden usado o del número de iteraciones. En todos los casos mejoramos el resultado en tiempo con poca pérdida de precisión.

Finalmente, hemos realizado un análisis del coste computacional asociado tanto al algoritmo desarrollado como a la versión sin paralelizar. Hemos obtenido que el uso de la paralelización reduce significativamente el coste computacional, especialmente si usamos como claves únicamente campos que no repitan muchas veces el mismo elemento a lo largo de los registros. Es decir, que no generen buckets demasiado grandes, lo que aumenta el coste.

Con estos datos podemos afirmar que el resultado del algoritmo planteado es parcialmente satisfactorio. Hemos logrado nuestro objetivo inicial, que era desarrollar un algoritmo escalable que se acerque lo máximo posible a la solución óptima y que además fuese suficientemente versátil para adaptarlo a datasets de naturaleza distinta a nuestro dataset problema inicial. Sin embargo también hemos observado que su eficacia es dependiente fuertemente de los parámetros escogidos para

ejecutarlo. Buscamos usar como clave campos que tengan una distribución con poca desviación con respecto a su valor medio, de forma que generen buckets no muy grandes pero lo suficientemente grandes para que puedan realizarse comparaciones y se consigan realizar anexiones de registros. Además, el coste computacional obtenido para 3 y 4 iteraciones no es suficientemente escalable cuando ejecutamos en una única máquina, al tener un exponente de crecimiento alrededor de 2. Únicamente resulta verdaderamente escalable en el caso de 2 iteraciones, donde se hace uso precisamente de este tipo de campos, logrando un exponente alrededor de 1. Así, la eficacia del algoritmo queda subordinada a la existencia de este tipo de campos en el dataset.

Pese a que en una única máquina el algoritmo no sea escalable salvo haciendo uso de los campos con buenas propiedades, cuando lo ejecutamos en el entorno para el cual está diseñado, un cluster de ordenadores, el exponente de crecimiento cercano a 1 hace que se vuelva escalable incluso cuando usamos 4 iteraciones. Esto nos indica que si hacemos un uso completo de una de las características fundamentales del algoritmo como es la paralelización, el objetivo de escalabilidad se cumple.

Como posibles mejoras al algoritmo presentado, hay una muy clara. El algoritmo no paralelizado se espera que tenga un coste computacional cercano a un exponente 2. En cambio, el que hemos obtenido nosotros es aproximadamente 3. Esto nos indica que la función que realiza las operaciones de match y merge aún tiene cabida a ser optimizado para reducir el coste global del algoritmo. Esperamos que tras realizar una mejora de este tipo, el coste del algoritmo desarrollado se verá reducido igualmente, aunque en menor proporción que lo haga la versión no paralelizada.

Otra posible mejora que podría realizarse es tratar de automatizar la selección de los parámetros usados, en especial del orden. Mediante el análisis estadístico del dataset, por ejemplo, podríamos ser capaces de hacer esa selección de forma óptima. Al ser tan sensible a una correcta elección del parámetro de orden, dar la opción de basar esta elección en una serie de datos objetivos permite que el usuario final que quiera hacer uso del algoritmo no tenga por qué tener un conocimiento tan profundo del dataset. El establecimiento de las reglas estadísticas concretas que deben regir esa elección de parámetros queda fuera de los objetivos de este trabajo, pero puede resultar un estudio teórico interesante para optimizar el funcionamiento del algoritmo.

# 7. Bibliografía


[1] Vassilis Christophides y col. "End-to-end entity resolution for big data: A survey". En: *arXiv preprint arXiv:1905.06397* (2019).

[2] Dimas Cassimiro do Nascimento, Carlos Eduardo Santos Pires y Demetrio Gomes Mestre. "Exploiting block co-occurrence to control block sizes for entity resolution". En: *Knowl. Inf. Syst.* 62.1 (2020), págs. 359-400. DOI: 10.1007/s10115-019-01347-0. URL: https://doi.org/10.1007/s10115-019-01347-0.

[3] Luciano Barbosa. "Learning representations of Web entities for entity resolution". En: *IJWIS* 15.3 (2019), págs. 346-358. DOI: 10.1108/IJWIS-07-2018-0059. URL: https://doi.org/10.1108/IJWIS-07-2018-0059.

[4] Chenchen Sun y col. "A genetic algorithm based entity resolution approach with active learning". En: *Frontiers Comput. Sci.* 11.1 (2017), págs. 147-159. DOI: 10.1007/s11704-015-5276-6. URL: https://doi.org/10.1007/s11704-015-5276-6.

[5] Muhammad Sadiq y col. "A Vertex Matcher for Entity Resolution on Graphs". En: *14th International Conference on Ubiquitous Information Management and Communication, IMCOM 2020, Taichung, Taiwan, January 3-5, 2020*. IEEE, 2020, págs. 1-4. DOI: 10.1109/IMCOM48794.2020.9001799. URL: https://doi.org/10.1109/IMCOM48794.2020.9001799.

[6] Omar Benjelloun y col. "Swoosh: a generic approach to entity resolution". En: *VLDB J.* 18.1 (2009), págs. 255-276. DOI: 10.1007/s00778-008-0098-x. URL: https://doi.org/10.1007/s00778-008-0098-x.

[7] Peter Christen. "A Comparison of Personal Name Matching: Techniques and Practical Issues". En: *Workshops Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China*. IEEE Computer Society, 2006, págs. 290-294. DOI: 10.1109/ICDMW.2006.2. URL: https://doi.org/10.1109/ICDMW.2006.2.