risti
Revista Ibérica de Sistemas e Tecnologias de Informação
Iberian Journal of Information Systems and Technologies

Recebido/Submission: 18/11/2019
Aceitação/Acceptance: 09/01/2020

Método optimizado basado en algoritmo K-Means como herramienta en la detección de plagio de código fuente

Michal Duracik[1], Mauro Callejas-Cuervo[2], Miroslava Mikusova[1]

michal.duracik@fri.uniza.sk, mikusova@fpedas.uniza.sk, maurocallejas@uptc.edu.co

[1] Universidad de Zilina, Univerzitná 1, 010 26 Zilina, Eslovaquia.

[2] Universidad Pedagógica y Tecnológica de Colombia, Grupo de Investigación en Software, 150003, Tunja, Boyacá, Colombia.

Pages: 620–632

Resumen: El plagio es un problema cada vez más significativo, porque en Internet se encuentra disponible enormes recursos de información. Detectar plagio en el código fuente de proyectos desarrollados por estudiantes en las universidades es aún más difícil, debido a que año tras año se generan un sin número de nuevos proyectos académicos y son pocas las herramientas disponibles que permitan analizar grandes volúmenes de líneas de código. Además, la detección de modo automático consume muchos recursos computacionales y tiempos de ejecución. En este artículo se presenta un método mejorado del algoritmo K-Means, el cual permite dividir las líneas de código en grupos y así ahorrar tiempo y operaciones innecesarias en la búsqueda de coincidencias. Los resultados reportan que el algoritmo fue acelerado sin comprometer su propósito original y se ha logrado una aceleración triple en comparación con la implementación básica sin ninguna optimización.

Palabras-clave: algoritmo K-Means, código fuente, plagio, rendimiento computacional.

Optimized method based on the K-means clustering algorithm as a tool to detect source code plagiarism

Abstract: Plagiarism is a problem of growing significance, as on the Internet there are enormous information resources available. Detecting plagiarism in the source code of projects developed by university students is even more difficult, given that year after year a countless number of new academic projects are generated and there are few tools available that allow for the analysis of large numbers of lines of code. In addition, automatic detection requires the use of many computational resources and consumes a lot of time to execute. This article presents an improved method of the K-means clustering algorithm, which permits the division of lines of code into clusters, thus, saving time and avoiding unnecessary operations in the search for coincidences. The results report that the algorithm was accelerated without

compromising its original purpose and that a triple acceleration was achieved in comparison to the basic implementation without any optimization.

Keywords: K-Means algorithm, source code, plagiarism, computational performance.

1. Introducción

Hoy en día el plagio se produce cada vez con más frecuencia en la vida cotidiana. Con la expansión de las tecnologías de la información y las comunicaciones, es cada vez más fácil acceder a varias fuentes y hacer plagio. Por otro lado, este auge también fomenta el desarrollo de herramientas que puedan detectar tales formas de plagio. El mayor énfasis se está poniendo en el desarrollo de métodos y herramientas para la detección de fraude en documentos tipo texto. Este enfoque es comprensible ya que muchos documentos académicos se crean anualmente en todo el mundo; pero encontrar plagio en el código fuente de aplicaciones informáticas es un área de interés que trae nuevos desafíos.

El problema del plagio y otras formas de hacer fraude ha sido siempre un tema muy debatido (Skalka, 2009). Se encontraron casos en los que se han otorgado títulos académicos y posterior a esto se ha evaluado sus documentos de trabajo de grado o tesis y se ha detectado una alta proporción de coincidencias con otras tesis u otras fuentes de información. Los Sistemas antiplagio o Antiplagiarism System (APS) intenta identificar partes idénticas de documentos. En Eslovaquia se utiliza para este propósito el sistema ANTIPLAG y en Colombia, es común el uso de la herramienta Turnitin.

Este trabajo se centra en optimizar la implementación del algoritmo de agrupamiento K-Means, y en nuestro trabajo ayudó a dividir el código fuente en partes para formar grupos y así ahorrar tiempo en la realización de operaciones innecesarias en la búsqueda de coincidencias largas en el código fuente.

El artículo se organiza de la siguiente, en primera instancia se describen algunas consideraciones generales sobre la temática de la investigación, luego se presenta la propuesta de optimización del algoritmo K-Means, junto con su análisis y evaluación y finalmente se presentan las conclusiones.

2. Plagion en el código fuente

En general, la percepción de plagio en el código fuente no es muy diferente del plagio en documentos tipo texto. Las razones del plagio son las mismas en ambos casos y máxime en esta época de fácil acceso a diversa información gracias a Internet. Desde nuestro punto de vista, el problema del descubrimiento de plagio en el código fuente es que actualmente no hay disponible un sistema o servicio integrado que permitan detectar el plagio a escala global. Como resultado, no hay estadísticas disponibles para describir el nivel de plagio en esta área.

Los sistemas de detección de plagio más utilizados son el *Sistema de Medida de Similitud de Software* (MOSS) de la Universidad de Stanford y el *sistema JPlag* de Alemania, ambos sistemas se crearon hace más de diez años y aún se están desarrollando. Existen otras investigaciones relacionadas con el desarrollo de sistemas de detección de plagio en código fuente en ámbitos generales (Bradshaw & Chindeka, 2020; Aldabbas, 2019;

Ganguly, Jones, Ramírez-de-la-Cruz, Ramírez-de-la-Rosa, & Villatoro-Tello, 2018) y otras en el ámbito académico (Tennyson, 2019; Novak, Joy, & Kermek, 2019; Karnalim, Budi, Toba & Joy, 2019); la mayoría de estas iniciativas son individuales y se podrían colocar a disposición del público en general, como apoyo a la mitigación de este problema. Si queremos utilizar dichos sistemas en el proceso de enseñanza, nos encontramos con un problema grave, ya que el ámbito educativo se caracteriza por el hecho de que todos los años se está generando una gran cantidad de nuevos datos, la cual necesita ser comprobada año-año, y por lo tanto es necesario introducir algún tipo de análisis incremental de estos datos, ya que así se está realizado y con éxito, en los APS textual.

2.1. Deficiencias de los métodos actuales.

Los métodos utilizados usualmente para detectar plagio en el código fuente se han desarrollado junto con métodos para documentos de texto desde el siglo pasado. En el caso de documentos textuales, el desarrollo progresa con métodos más avanzados que se basan en el aprendizaje automático profundo para detectar plagio (Rakian, Safi & Rastegari, 2015) y los sistemas grandes integran estos métodos.

Durante el estudio de dicha problemática se ha encontrado algunas deficiencias:

- Obsolescencia.
- Sistemas cerrados (sin código abierto).
- Proceso complejo de evaluación del plagio.
- Incapacidad para utilizar o no una gran base de información.

Estas deficiencias pueden ser el resultado del hecho de que fueron diseñadas para detectar plagio dentro de un pequeño grupo de archivos de código fuente.

2.2. Propuesta de un nuevo método.

A continuación, se describe el método propuesto el cual permite buscar plagio en grandes volúmenes de códigos fuente y mostrar sus componentes. En la Figura 1, se esquematiza dicha propuesta.

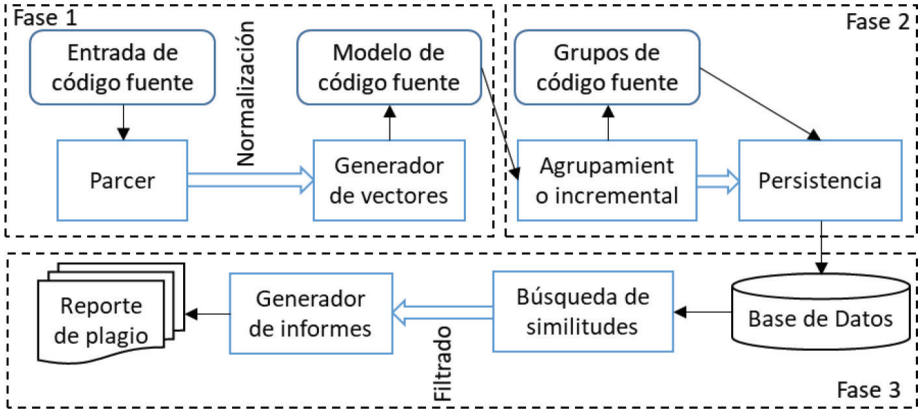

Figura 1 – Diagrama de componentes de un nuevo método de búsqueda de plagio

La principal ventaja del método propuesto es su modularidad porque se divide en 3 fases, cada una de las cuales consta de dos algoritmos. Las fases son las siguientes:

1. Procesamiento y representación del código fuente.
2. Agrupación y persistencia de datos.
3. Búsqueda de similitudes y creación de un informe.

En la primera fase, se implementan algoritmos para procesar el código fuente y crear un modelo en forma de árbol sintáctico; esto es específico para cada lenguaje de programación que se desee procesar. La segunda fase agrupa algoritmos que transforman el árbol de sintaxis en una estructura adecuada para su posterior procesamiento. Elegimos vectores característicos como una forma adecuada para la representación del código fuente. Entre el procesamiento y la vectorización, podemos incluir algoritmos que normalizarán el árbol de sintaxis, lo que permite detectar plagio incluso en el caso de trucos de APS de uso común (Tao, Guowei, Hu & Baojiang, 2013; Zhao, Xia, Fu & Cui, 2015). En esta misma fase, se realiza el agrupamiento de vectores similares, para esto se utiliza el algoritmo K-Means, que está ligeramente modificado para los fines perseguidos con esta investigación. El resultado de esta fase son datos listos para ser almacenados en una base de datos que permita posteriormente una forma eficiente de búsqueda. El objetivo de la agrupación es, además de preparar previamente los datos para la búsqueda, una cierta distribución lógica de los datos en grupos relacionados. Dicha distribución permite una mejor escalabilidad de todo el método. La última fase trata de obtener coincidencias individuales de la base de datos y su evaluación. En esta etapa, antes de generar el informe, un filtro sencillo de identidad puede ser utilizado para hacer los informes más fáciles de leer.

Las fases son casi independientes entre sí, el único elemento del que dependen es el formato de los datos transferidos entre fases. Un modelo de código fuente procesado, representado por vectores característicos, se transmite entre la primera y la segunda fase y una base de datos de vectores es una interfaz entre las fases 2 y 3. Tal separación de datos nos brinda más oportunidades para escalar todo el método. Una descripción detallada de las fases y otros aspectos relevantes se puede encontrar en los textos relacionados en trabajos anteriores (Duracik, Krsak, & Hrkut, 2017a; Duracik, Krsak, & Hrkut, 2017b; Duracik, Krsak, & Hrkut, 2018a; Duracik, Krsak, & Hrkut, 2018b; Duracik, Krsak, & Hrkut, 2018c; Duracik, 2019; Duracik, Krsak, & Hrkut, 2019). En este artículo, nos centraremos en los aspectos de una efectiva implementación de agrupación y una optimización de su desempeño.

3. Métodos de optimización sugeridos para la aplicación del algoritmo K-Means

La agrupación generalmente se usa para reunir diferentes muestras de datos en grupos y/o en el campo de minería de datos. En nuestro caso queremos usar la agrupación como una herramienta de clasificación previa de vectores, que luego nos permite simplificar la búsqueda de vectores similares. El conjunto de datos de entrada se divide en dos grupos: conjunto de entrenamiento o formación y el conjunto de control. El conjunto de entrenamiento se utiliza para crear grupos y los datos del conjunto de control verifican

la precisión de esta clasificación. Uno de los propósitos de nuestro trabajo se centró en diseñar un método de agrupación que permitiera agregar datos nuevos de forma gradual y mantener los grupos en una óptima configuración.

3.1. Eficiencia de la complejidad computacional del algoritmo K-means

El algoritmo básico de K-Means en su especificación solo describe las ideas principales, pero no las formas de implementación efectiva. Si queremos usar este algoritmo en nuestro método, necesitamos diseñar e implementar varias mejoras que incrementen la eficiencia del algoritmo. En la mayoría de los casos se trata de mejorar la fase inicial del algoritmo (encontrar los primeros centros (Celebi, Hassan, & Vela, 2013; Pena, Lozano & Larranaga, 1999), que es el punto de partida para tales intentos. En otros casos, la distribución inicial es importante porque una mejor distribución, acelera el algoritmo y puede eliminar el problema de soluciones subóptimas donde el algoritmo cae a un mínimo local (Selim Shokri & Mohamed, 1984). En nuestra investigación decidimos no centrarnos en los casos anteriores, sino que se realiza sólo al principio del algoritmo y los centros están siendo recalculado en su ejecución. Este cálculo se basa en el estado anterior y no es necesario reiniciar los clústeres nuevamente. El algoritmo K-Means, es relativamente simple, razón por la cual hay algunos aspectos que se pueden mejorar. Nuestro análisis mostró que la parte más ineficaz es la asignación de vectores a las agrupaciones ya que este tipo de operación tiene complejidad de $O(n * k)$, donde n es el número de vectores y k es el número de grupos. Para cada combinación de conglomerados de vectores debe calcularse la distancia, y el cálculo de la distancia euclidiana para vectores de longitud 162 también lleva algún tiempo.

En esta sección, nos centraremos en optimizar la segunda parte del algoritmo K-Means porque esta parte se ejecuta con mucha frecuencia. Nosotros sugerimos mejorar el algoritmo en los siguientes puntos:

- Añadir paralelización.
- Mejorar la estructura lógica del algoritmo.
- Mejorar las técnicas de implementación.
- Preprocesar los datos de entrada.

A continuación, se detallan los puntos mencionados. Al final, evaluaremos las técnicas sugeridas y compararemos la velocidad del algoritmo antes y después de la aplicación de esta técnica.

3.2. Paralelización del algoritmo

Una de las técnicas básicas de la aceleración de algoritmos es su paralelización. En la iteración 1, podemos encontrar varios métodos que intentan paralelizar el algoritmo K-Means (Zhao, Huifang & Qing, 2009; Stoffel & Abdelkader, 1999). En nuestro trabajo decidimos explorar e implementar algunos métodos básicos de paralelización.

En general, una paralelización de algoritmos no es una tarea trivial (Wolf & Lam, 1991). El objetivo principal es utilizar la mayor parte de la potencia informática disponible de la CPU. En la implementación básica del algoritmo, observamos un 25% de uso de la CPU durante la ejecución (Ejemplo 1, Figura 2) porque la prueba se ejecutó en un

procesador que tenía 2 núcleos hiperprocesados. Dada la gran dimensión de los datos de entrada y la frecuencia de las operaciones de cálculo de distancia, decidimos primero paralelizar el cálculo de distancia euclidiana. Dividimos los vectores de entrada en varios grupos y calculamos las distancias de estas partes en paralelo y finalmente, combinamos los resultados parciales. Este principio se conoce generalmente como *MapReduce*. Utilizamos el mismo enfoque mediante el cálculo de nuevos centros de clúster, donde calculamos los elementos individuales del centro resultante en paralelo (Ejemplo 2 de la Figura 2), este método utilizó casi todos los recursos disponibles en la CPU, pero conlleva costos generales considerables; por ejemplo, la gestión de subprocesos (a través de los cuales se ha implementado la paralelización), la sincronización y otros aspectos no permiten la plena utilización de las capacidades de paralelización y la cantidad de potencia informática que consume el sistema operativo. Algunos enfoques pueden reducir parcialmente esta sobrecarga al usar el grupo de *trabajadores de subprocesos* que elimina la sobrecarga de la administración de recursos. Aunque este enfoque utilizó al máximo los recursos informáticos disponibles, el beneficio que obtuvimos fue menor que la sobrecarga adicional del sistema operativo.

Figura 2 – Utilización de la CPU durante los métodos de paralelización

Con base en estos resultados creamos nuestra propuesta final de paralelización en un nivel superior del algoritmo. Paralelamente, la operación de asignar vectores a grupos como el primer paso. Las asignaciones individuales son independientes entre sí, por lo que podemos realizar esta operación en paralelo. Al principio dividimos un conjunto de vectores que podemos asignar a n grupos (n es igual al número de núcleos de CPU disponibles) y luego realizamos la asignación de vectores en paralelo. Al implementar, se debe tener en cuenta en cómo se realiza la operación de asignación, ya que puede requerir cierto grado de sincronización. En el segundo paso dividimos los grupos en n y luego calculamos los centros para los grupos individuales en paralelo. Igual que en el caso anterior, los grupos son independientes entre sí, por lo que este enfoque no debería causar ningún problema de sincronización. Como se puede ver en el Ejemplo 3 de la Figura 2, este enfoque no alcanza el 100% de utilización de los recursos informáticos disponibles. Este problema es causado por un tamaño de clúster desigual, por lo que algunos grupos de clústeres se calculan más rápido que otros. La solución podría ser una inclusión más inteligente de los grupos en subgrupos o utilizar un enfoque en el que el cálculo de cada grupo representaría una tarea separada, y estas tareas se planificarían en función de la disponibilidad actual de recursos informáticos. Por otra parte, nuestros resultados muestran que las mejoras de rendimiento que utilizan este enfoque son mínimos, pero el aumento de complejidad de la implementación es significativo.

3.3. Heurística para la agrupación de vectores.

A continuación, se detalla cómo se puede acelerar la fase de asignación de vectores a grupos. Según el análisis de los cálculos de tiempo de ejecución, descubrimos que hasta el 68% del tiempo dedicado a la agrupación de vectores toma cálculos de distancia. Otro interesante resultado fue que, a excepción de un número insignificante de pasos iniciales, el 99% de los vectores permanecen en el mismo clúster. A pesar de este hecho, el algoritmo debe calcular la distancia a todos los demás centros, lo que cuesta mucho tiempo de computación.

Para resolver este problema, propusimos una modificación que eliminará las operaciones innecesarias de cálculo de distancia, el cual se muestra en el Algoritmo 1.

```
1: function K_Means(V : vector list, k : int, C : list of clusters): list of clusters
2:     repeat
3:         change ← 0
4:         m ← {}
5:         for all cluster c in C do
6:             m(c) ← Sort(Distance(c, C_j)), ∀ j∈{1..k} ∧ C_j != c)
7:         end for
8:         for all vector v in V do
9:              c ← LastCluster(v)
10:             dToLast ← Distance(v, c)
11:             c ← Min(Distance(v, b), dToLast), ∀ b∈m(c)∧Distance(v, m(c))<2*dToLast
12:             Assign(v,c)
13:         end for
14:         for all cluster c in C do
15:            change ← change + Recalculate(c)
16:         end for
17:    while change > 0
18:    return C
19: end function
```

Algoritmo 1 – Algoritmo para reducir el número de cálculos de distancia vectorial

Al comienzo de cada paso, el algoritmo modificado calcula una matriz de distancias entre el grupo y asigna a cada grupo una lista de pares: distancia y el otro grupo, ordenados por distancia de forma ascendente. Gracias a esto, podemos saber para cada grupo qué tan lejos están los otros grupos. Usaremos esta lista en el procedimiento de asignación de vectores, así: para cada vector, primero calculamos la distancia al clúster original (indicado como *dToLast* en el algoritmo), entonces secuencialmente itera a través de la lista de los grupos más cercanos de la agrupación donde el vector era original, hasta que la distancia entre los grupos es mayor que *2*dToLa*st. Durante esta iteración, comprobamos la distancia del vector a otro grupo, y si es necesario, actualizamos la distancia menor.

Podemos calcular cuántas operaciones de cálculo de distancia se guardan. Donde k denota el número de grupos y n el número de vectores. En nuestra implementación propuesta,

solo se requieren cálculos para la matriz de la distancia entre grupos. Se necesitan otros *n* cálculos para encontrar las distancias a los centros de los grupos originales (esto se puede reducir recordando la distancia original y usándola si el centro del grupo no ha cambiado). La última parte para ser contada es el número medio de grupos (denotada por *x*), que debe ser examinada cerca del clúster original. Según nuestras mediciones para el conjunto de datos probado, este valor fue igual a 3.2., el cual es menor que el original.

3.4. Eficiencia de implementación

Además de la optimización mencionada anteriormente en un nivel superior, también examinamos la optimización de los detalles específicos de implementación del algoritmo dependiendo del lenguaje de programación y el hardware físico. Nuestro algoritmo se implementó en el lenguaje de programación *C#*. Este lenguaje de programación es de alto nivel y no permite afectar directamente las operaciones que el hardware realiza físicamente. Además, no podemos configurar directamente el compilador para usar, por ejemplo, instrucciones vectoriales.

La implementación inicial del algoritmo utilizó varios LINQ. Como ejemplo, podemos usar el zip[2] método para calcular la distancia euclidiana. Cuando reemplazamos este método con un bucle esto ligeramente acelera el algoritmo. Se utilizó un enfoque similar para calcular los centros y la sustitución de la expresión de un bucle sencillo y esto también aceleró ligeramente el algoritmo. Sin embargo, estos cambios trajeron una aceleración solo en un pequeño porcentaje.

El análisis mostró aún otra manera de acelerar el algoritmo de cálculo de los centros. El algoritmo de cálculo original (Algoritmo 2), calculó cada componente del vector por separado, esto puede no parecer un problema a primera vista, pero cuando nos damos cuenta de cómo un único vector se almacena en la memoria, es una cuestión crítica, ya que el algoritmo debe cargar una gran cantidad de datos de la memoria operativa en el procesador para el cálculo de cada componente.

```
1: mean ← [sizeof(vector)]
2: for i in 1..sizeof(vector) do
3:     mean(i) ← average(v(i))∀ v ∈ V
4: end for
```

Algoritmo 2 – El algoritmo para el cálculo de nuevos centros de clúster

Además de eso, los procesadores actuales usan la memoria caché de forma masiva e incluyen varias técnicas para preparar los datos que puedan necesitar en el futuro (*captación previa de la CPU*). Puesto que los vectores se almacenan en lugares al azar en la memoria, la CPU no puede predecir qué datos necesita, y el procesamiento se hace más lento ya que se debe leer los datos de la memoria RAM. Para mejorar este comportamiento, modificamos el algoritmo para calcular el promedio de todos los componentes del vector a la vez. Aunque este enfoque tiene el mismo número de operaciones que el anterior, los resultados muestran que es un poco más rápido.

```
1: mean ← [sizeof(vector)]
2: for all vector v in V do
3:     for i in 1..sizeof(vector) do
4:         mean(i) ← mean(i) + v(i)
5:     end for
6: end for
7: mean(i) ← mean(i) / count(V) ∀i ∈ {1..sizeof(vector)]}
```

Algoritmo 3 – El algoritmo más eficiente para calcular centros de clúster

Si analizamos por qué esta versión del algoritmo es más eficiente, encontraremos que esta versión utiliza el caché de la CPU y la captación previamente mencionada, de manera más efectiva. Acceder al caché de la CPU es mucho más rápido que acceder a los datos en la RAM. Otra razón es que la CPU está cargando datos de la memoria, en bloques de su memoria caché, por lo que cuando intentamos acceder a un componente del vector, hay una buena posibilidad de que el componente ya esté en la memoria caché. Además, este enfoque secuencial (paso a paso a través de todos los componentes del vector) también facilita la predicción de la captación previa de la CPU, que prepara los datos que se necesitarán en el futuro. Como hemos visto, la correcta aplicación de los algoritmos y la utilización de cachés de la CPU pueden a menudo ayuda a acelerar los algoritmos sin modificar su funcionamiento.

3.5. Reducción del tamaño de los datos

Hasta ahora, hemos descrito métodos de optimización que se centraron en mejorar los algoritmos o su implementación. Ninguno de estos métodos ha tenido algún efecto sobre el "output" del algoritmo. En este apartado describiremos un enfoque ligeramente diferente para aumentar la efectividad.

En Duracik, Krsak & Hrkut (2019), discutimos la selección de elementos adecuados para la indexación. Se ha demostrado que nuestros vectores contienen muchos componentes dependientes cuya eliminación haría que el algoritmo sea más eficiente. Como esta tarea es más compleja, no la trataremos directamente en esta sección. En cambio, mostraremos que incluso si reducimos el número de vectores, la razón principal del uso de la agrupación no se verá afectada.

La primera opción que veremos es la eliminación de vectores largos. Dado que nuestro algoritmo no limita la longitud máxima de los vectores generados, a menudo sucede que los vectores creados cubren todo el código fuente. No tiene sentido comparar tales vectores porque siempre serán demasiado específicos, y es más eficiente comparar las partes más pequeñas del código fuente y luego combinar estas partes más pequeñas en otras más grandes. Es importante agregar que al eliminar algunos vectores largos no perdemos casi información del código fuente, ya que dichos vectores siempre se han formado uniendo los más pequeños, y estos vectores más pequeños cubren casi todo el código fuente. La pregunta clave es: ¿cómo determinar la longitud máxima del vector?, Con base en nuestros experimentos, hemos fijado este valor en 250.

La segunda opción para hacer que la agrupación sea más eficiente es reducir la dimensión de los vectores. Consideramos dos enfoques para reducir el número de componentes. En el primer enfoque, decidimos eliminar cualquier componente que tenga una baja entropía, ya que de todos modos no nos ayudarán a buscar vectores. Y en el segundo enfoque, se seleccionaron los componentes en función de su frecuencia; entonces decidimos usar el segundo enfoque. Con base en el conjunto de datos probados, se eliminaron 80 componentes que ocurrieron en menos del 0.1% de los vectores. Sin embargo, en este enfoque propuesto, es importante darse cuenta de que la eliminación de los 80 componentes se aplica solo a la fase de agrupamiento. Trabajaremos con los vectores originales en otros algoritmos (especialmente al buscar coincidencias).

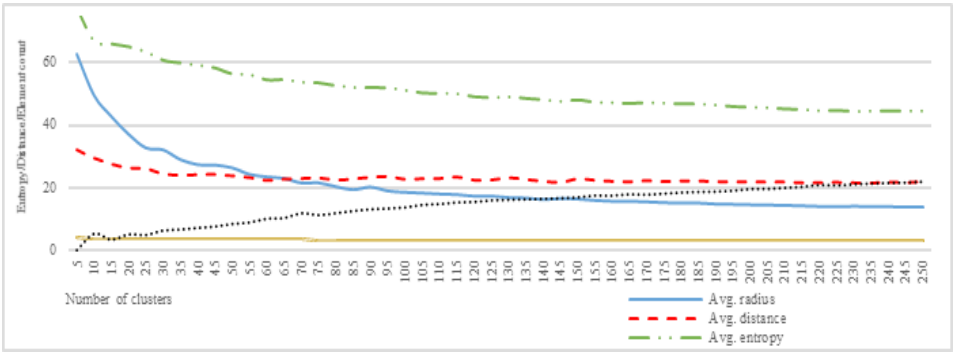

Figura 3 – Características seleccionadas de clusters usando vectores simplificados

Como puede verse en la Figura 3, las características individuales se comportan de manera similar al método de agrupamiento, donde todos los vectores se han utilizado (sin reducir). La única diferencia está en los valores absolutos, donde la entropía promedio es menor en un 25-30%. La distancia mínima promedio es aproximadamente un 50% más baja, que fue causada por la eliminación de los componentes "sobresalientes" del vector. Otras características también han disminuido, pero su disminución no es tan significativa. Con base en estos resultados podemos decir que los grupos creados son más pequeños y más compactos. No evaluamos el efecto de esta reducción en la eficiencia de búsqueda de plagio, ya que no depende de manera crítica de la forma de agrupamiento. Por otro lado, podemos decir que reducir la cantidad de datos acelerará el proceso de agrupación.

3.6. Evaluación experimental

En este apartado evaluamos el efecto de los enfoques de optimización individuales en la velocidad resultante del algoritmo K-Means. Todos los valores de tiempo presentados se basan en un promedio de 5 mediciones usando datos de la base de datos de prueba. La inicialización de los grupos fue la misma para cada uno de los casos medidos. No medimos la inicialización y la asignación inicial de vectores a grupos. En la medición,

examinamos la duración promedio de un ciclo del algoritmo, lo que significa asignar vectores a grupos y un recalculo de los centros. Establecemos el número de grupos en 100. Cada medición adicional también incluye el uso de técnicas de la fase anterior. La configuración del sistema en el que se realizaron las mediciones fue la siguiente:

- CPU Intel Core I5 6200U,
- 16 GB de RAM,
- Disco duro SSD.

	Inserción de vectores	Cálculo de centros	Distancia	Total
Versión original	153,6	1.7	-	155,3
Paralización agregada	49,9	1.1	-	51,0
Heurística para asignación de vectores	16,3	1.1	-	17.4
Implementación de optimización	5.1	0.1	-	5.2

Tabla 1 – Comparación de velocidad computacional del algoritmo K-Means (en segundos)

4. Conclusiones

La optimización del algoritmo es a menudo muy importante cuando se crean sistemas con grandes volúmenes de datos. En este caso, el algoritmo se aceleró sin comprometer su propósito original, tal como lo muestran los resultados de la tabla 1, en donde las modificaciones han logrado una aceleración triple en comparación con la implementación básica sin ninguna optimización.

La implementación del sistema reveló que el componente más importante del sistema será la base de datos, aunque inicialmente sea necesaria alguna potencia para calcular "clusters" a partir de los datos iniciales. Y durante la ejecución del sistema, el componente principal usará una base de datos donde se almacenan los vectores para una configuración de clúster dada y la base de datos se usará para buscar coincidencias de código fuente.

Tal diseño del sistema, junto con el hecho de que la necesidad de "re-clusterización" disminuye a medida que aumenta la cantidad de datos, permitirá que el sistema utilice los recursos informáticos de manera efectiva. Como ya se mencionó, en el funcionamiento normal del sistema solo será necesario ejecutar la base de datos, y si es necesario volver a realizar "re-clusterización", esto agregará temporalmente potencia de cálculo.

Referencias

Aldabbas, H. (2019). An IoT based framework for students' interaction and plagiarism detection in programming assignments. *Journal of Theoretical and Applied Information Technology*, 97(18), 4723-4737. Retreived from: http://www.jatit.org/volumes/Vol97No18/1Vol97No18.pdf.

Bradshaw, K. & Chindeka, V. (2020). Detecting similarity in multi-procedure student programs using only static code structure. *Communications in Computer and Information Science*, (pp. 211-226). Switzerland: Springer International Publishing. doi: 10.1007/978-3-030-35629-3_14.

Celebi, M. E., Hassan, A. K., & Vela, A. (2013). A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert systems with applications,* 40(1), 200-210.

Duracik, M. (2019). Semi-automatic identification of non-significant source code parts using clustering. *Mathematics in science and technologies: proceedings of the MIST conference 2019*, (pp. 17-21). Slovakia: Slovak Mathematical Society.

Duracik, M., Krsak, E. & Hrkut, P. (2017)a. Using concepts of text based plagiarism detection in source code plagiarism analysis. *Plagiarism across Europe and beyond 2017: conference proceedings*. (pp. 177-186). Brno, Mendel University. Retrieved from: https://plagiarism.pefka.mendelu.cz/files/proceedings17.pdf.

Duracik, M., Krsak, E. & Hrkut, P. (2017)b. Current trends in source code analysis, plagiarism detection and issues of analysis big datasets. *Procedia Engineering*, 192(1), 136-141. doi.org/10.1016/j.proeng.2017.06.024.

Duracik, M., Krsak, E. & Hrkut, P. (2018)a: Source code representations for plagiarism detection. Learning Technology for Education Challenges. LTEC 2018. Communications in Computer and Information Science, vol 870. (pp. 61-69). 1. Ed, Cham: Springer International Publishing AG.

Duracik, M., Krsak, E. & Hrkut, P. (2018)b: Issues with the detection of plagiarism in programming courses on a larger scale. *Proceedings 16th IEEE International Conference on Emerging eLearning Technologies and Applications*. (pp. 141-147). New Jersey: Institute of Electrical and Electronics Engineers. doi: 10.1109/iceta.2018.8572260

Duracik, M., Krsak, E. & Hrkut, P. (2018)c. Scalable source code plagiarism detection using source code vectors clustering. *Proceedings of 2018 IEEE 9th International Conference on Software Engineering and Service Science*. (pp. 499-502). Institute of Electrical and Electronics Engineers. doi: 10.1109/icsess.2018.8663708.

Duracik, M., Krsak, E. & Hrkut, P. (2019). Searching source code fragments using incremental clustering. In Geoffrey C. Fox and David W. Walker. *Concurrency and Computation: Practice and Experience*. Retreived from: https://doi.org/10.1002/cpe.5416.

Ganguly, D., Jones, G.J.F., Ramírez-de-la-Cruz, A., Ramírez-de-la-Rosa, G. & Villatoro-Tello, E. (2018). Retrieving and classifying instances of source code plagiarism. *Information Retrieval Journal*, 21 (1). 1-23. DOI: 10.1007/s10791-017-9313-y.

Karnalim, O., Budi, S., Toba, H. & Joy, M. (2019). Source code plagiarism detection in academia with information Retrieval: Dataset and the observation. *Informatics in Education*, 18 (2), 321-344. DOI: 10.15388/infedu.2019.15.

Novak, M., Joy, M. & Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Transactions on Computing Education*, 19 (3), art. No. A27. DOI: 10.1145/3313290.

Pena, J. M., Lozano, J. A. & Larranaga, P. (1999). An empirical comparison of four initialization methods for the k-means algorithm. *Pattern recognition letters*, 20(10), 1027-1040. doi: https://doi.org/10.1016/S0167-8655(99)00069-0.

Rakian, S., Safi, E. F. & Rastegari, H. (2015). A Persian fuzzy plagiarism detection approach. *Journal of Information systems and telecommunication*, 3(11), 182-190. doi: 10.7508/jist.2015.03.007.

Selim Shokri, Z. & Mohamed, A. I. (1984). K-means-type algorithms: A generalized convergence theorem and characterization of local optimality. *IEEE Transactions on pattern analysis and machine intelligence*, 6(1), 81-87. doi: 10.1109/tpami.1984.4767478.

Skalka, J. (2009). *Prevencia a odhaľovanie plagiátorstva*. (Zber prác za účelom obmedzenia porušovania autorských práv v kvalifikačných prácach na vysokých školách), Fakulta prírodných vied, Univerzita konštatnína filozofa v Nitre, Nitra.

Stoffel, K. & Abdelkader, B. (1999). Parallel k/h-means clustering for large data sets. *European Conference on Parallel Processing*. *Lecture Notes in Computer Science*, vol 1685, (pp. 1451-1454). Berlin, Germany: Springer. doi.org/10.1007/3-540-48311-X_205.

Tao, G., Guowei, D., Hu, Q. & Baojiang, C. (2013). Improved plagiarism detection algorithm based on abstract syntax tree. *Proceedings of the 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*,(pp. 714-718). Washington, USA, IEEE Computer Society.

Tennyson, M.F. (2019). ASAP: A Source Code Authorship Program. *International Journal on Software Tools for Technology Transfer*, 21(4), 471-484. DOI: 10.1007/s10009-019-00517-3.

Wolf, M. E. & Lam, M. S. (1991). A loop transformation theory and an algorithm to maximize parallelism. *IEEE transactions on parallel and distributed systems*, 2(4), 452-471. doi: 10.1109/71.97902.

Zhao, J., Xia, K., Fu, Y. & Cui, B. (2015). An AST-based code plagiarism detection algorithm. *10th International Conference on Broadband and Wireless Computing, Communication and Applications*, (pp. 178-182). Krakow, Poland: IEEE Computer Society. doi: 10.1109/BWCCA.2015.52.

Zhao, W., Huifang, M. & Qing, H. (2009). Parallel k-means clustering based on mapreduce. *IEEE International Conference on Cloud Computing*, (pp. 674-679). Berlin, Germany: Springer. doi.org/10.1007/978-3-642-10665-1_71.