Informe Trabajo Práctico

Grupo 04
Programación Concurrente
2º Cuatrimestre de 2024

Alumnos:	Emails:	Legajo:
Bechtholdt, Cristopher	cristopherbechtholdt@gmail.com	59877
Salvanescki, Nicolás	salvanescki.unq@gmail.com	40.621.461
Salvatore, Nicolás	nicolas.salvatore@alu.unq.edu.ar	41.738.176

Índice

1. Introducción

El presente trabajo tiene por objetivo crear un programa en Java que reconozca caracteres numéricos en imágenes de 28 x 28 píxeles representadas en escala de grises y en formato **png**. Para implementarlo, se dispondrá de una base de datos de 60.000 imágenes con dichas características, ya etiquetadas con el número que representan, llamada *Modified National Institute of Standards and Technology* (MNIST).

Se hará uso del algoritmo de *machine learning*: *k-nearest neighbors* (kNN) y se mejorará la eficiencia en tiempos de ejecución aprovechando la concurrencia, utilizando múltiples threads *Workers* para el procesamiento.

El algoritmo kNN es un método de aprendizaje supervisado en el que se predice la salida de un punto de datos según las k más cercanas, es decir, determina las k imagenes cuya distancia sea mínima. Como las imagenes son de 28 x 28 *pixeles*, se pueden analizar como *Arrays* de 784 *bytes* y la "distancia" entre una imagen y otra es la diferencia *byte* a *byte* computada como la distancia euclideana entre cada elemento de ambos *Arrays*:

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{784} (x_i - y_i)^2 \tag{1}$$

Se podrá, además, ejecutar el programa en modo **test** al pasar un archivo de formato **csv** que contiene 10000 imagenes de prueba de la misma base de datos. Dicho modo permite evaluar el nivel de *precisión* del algoritmo kNN.

A continuación, se muestra el orden de entrada de datos del programa en el comando de ejecución en una terminal:

> **java -jar tp.jar k numWorkers bufferSize path dataSetPath**

Siendo:

- **k** : La cantidad de vecinos más cercanos deseados.
- **numWorkers** : La cantidad de Workers que van a estar trabajando.
- **bufferSize** : La capacidad máxima del buffer de tareas.
- **path** : La ruta en la que se encuentra la imagen o el csv a evaluar.

  Por ejemplo, "mnist_test.csv" si se encuentra en el mismo directorio que el **jar**.

  Dependiendo del formato que tenga el archivo pasado por parámetro, se determinará el modo de ejecución del programa. Si se pasa una imagen, será una ejecución normal dando por resultado el número que representa la imagen y el tiempo que tardó el algoritmo en calcularlo. Por otro lado, si se pasa un archivo de test **csv**, el programa entrará en modo de prueba. Al entrar en dicho modo se pedirá al usuario que ingrese el número de líneas que desea leer del archivo de pruebas y, luego, se imprimirá la precisión del algoritmo y el tiempo que se tardó.
- **dataSetPath** : La ruta en la que se encuentra el dataset MNIST.

Respecto a la jerarquía de la implementación del algoritmo kNN a la hora de determinar el número de una imagen, primero selecciona el *tag* del número de mayor frecuencia. Si hay empate, toma al tag que tenga el elemento con menor distancia. En caso de que haya dos o más de la misma distancia, desempata con el de menor *tag*.

2. Evaluación

2.1. Especificaciones Técnicas

Las pruebas se llevaron a cabo en un equipo con las siguientes características técnicas:

- **Procesador**: AMD Ryzen 7 5800X 8-Core 16 hilos 3,8 GHz base (Máximo 4,7 GHz).
- **Memoria RAM**: 32 GB DDR4.
- **Sistema Operativo**: Microsoft Windows 11 versión 24H2 (Compilación SO 26100, 1742).
- **Versión Java**: OpenJDK Temurin-21.0.5+11 (build 21.0.5+11-LTS) 2024-10-15.

2.2. Explicación del código

A pesar de ya haberse explicado en que se basa el algoritmo kNN para medir distancias, cabe aclarar como es que el programa aprovecha la concurrencia para llevar a cabo las exigentes tareas que se demandan en el presente trabajo.

El código hace uso de un *buffer* de tareas y de una *Thread Pool* de *Workers* que van a estar tomando dichas tareas para ejecutarlas concurrentemente. Estas tareas, llamadas *MNISTasks*, constan de un rango de líneas de la base de datos y de aplicar el algoritmo kNN para obtener los "k" vecinos más cercanos a la línea del archivo de test que se está evaluando.

Los *Workers* una vez terminan con una *MNISTask*, guardan los "k" vecinos resultantes en un monitor llamado ResultadoGlobal, el cual se va a encargar de procesar los resultados que vayan entregando los *Workers*.

2.3. Evaluación: Modo prueba

En el proceso de evaluación se realizó primero con el archivo de prueba, recolectando los resultados que dio el programa al varias los siguientes parámetros:

- **Cantidad de threads** : Se evaluó el tiempo de ejecución en función de si el **numWorkers** fue de 1, 4, 8 o 16.
- **Cantidad de líneas del csv de prueba** : Se evaluó la cantidad de líneas del archivo de prueba usando 500, 1000, 2000 o 5000 cada uno con diferente cantidad de threads para determinar su impacto en el porcentaje de aciertos o *accuracy*. En esta evaluación, se comenzó siempre desde la línea 0 y solo se varió hasta que línea final se tomaban los *Array* de *bytes* del archivo de prueba.

  En el Cuadro 1, se muestran los resultados de las pruebas. Además, en la Figura 1 se representan visualmente dichos datos, siendo el eje de abscisas, la cantidad de lineas del archivo de pruebas; cada línea coloreada, una cantidad distinta de threads y el tiempo de ejecución resultante, el eje de ordenadas.

Cant. Threads	Cant. Lineas	Tiempo de ejecución (ms)	Precisión
1	500	21875	96.00 %
1	1000	40928	95.60 %
1	2000	83086	95.35 %
1	5000	199599	95.38 %
4	500	6704	96.00 %
4	1000	14042	95.60 %
4	2000	26687	95.35 %
4	5000	66705	95.38 %
8	500	4296	96.00 %
8	1000	7567	95.60 %
8	2000	15138	95.35 %
8	5000	35521	95.38 %
16	500	3506	96.00 %
16	1000	5613	95.60 %
16	2000	9637	95.35 %
16	5000	22866	95.38 %

Cuadro 1: Tiempo de ejecución según threads y líneas (k = 10 y bufferSize = 8)

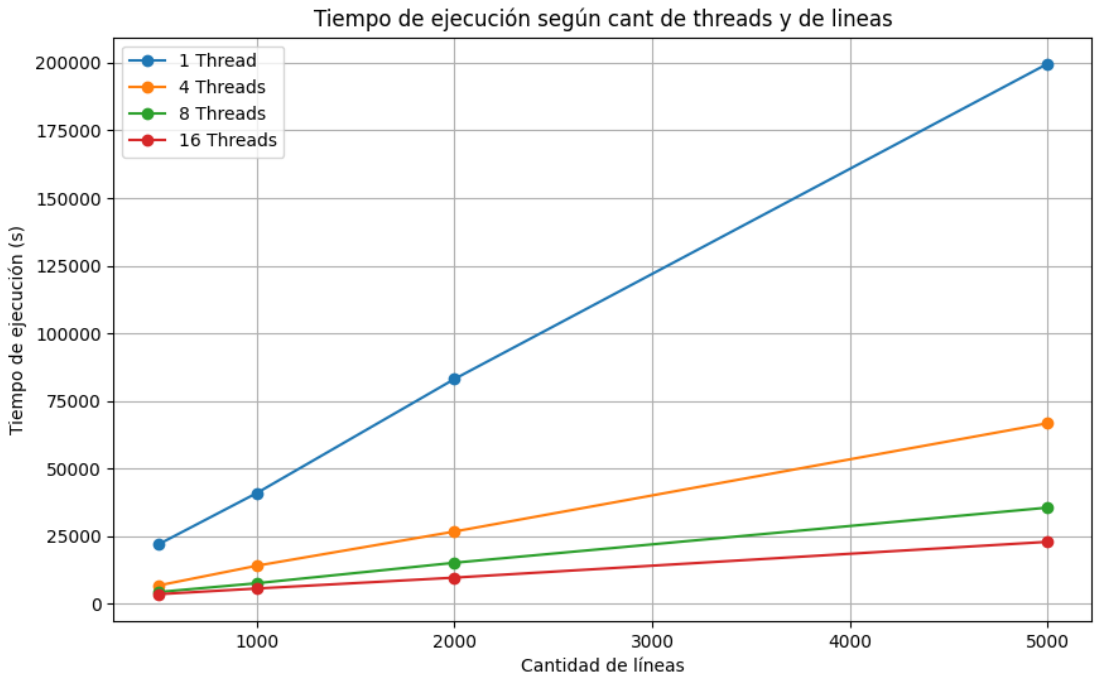

Figura 1: Tiempo de ejecución según cantidad de threads y líneas de prueba

- **k** : Fijando el valor de **numWorkers** a 16 y de cantidad de líneas del csv de prueba a 5000,

se varió la cantidad k-vecinos usando valores como 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 y 1024, para determinar si esto impactaba en el *accuracy* y en el tiempo de ejecución. Los resultados se encuentran a continuación en el Cuadro 2.3 y, visualmente, en la Figura 2. En esta última, el eje horizontal representa los "k" vecinos, mientras que el eje vertical mide el tiempo de ejecución (en milisegundos) y el porcentaje de precisión.

"k" vecinos	Tiempo de ejecución	Precisión
1	23045	95.58 %
2	22321	95.58 %
4	21718	95.92 %
8	22723	95.64 %
16	22853	94.84 %
32	23292	94.18 %
64	23486	92.96 %
128	23870	91.30 %
256	24352	89.12 %
512	26427	86.48 %
1024	29775	82.80 %

Cuadro 2: Relación entre k, tiempo y precisión (16 threads, 5000 lineas y bufferSize = 8)

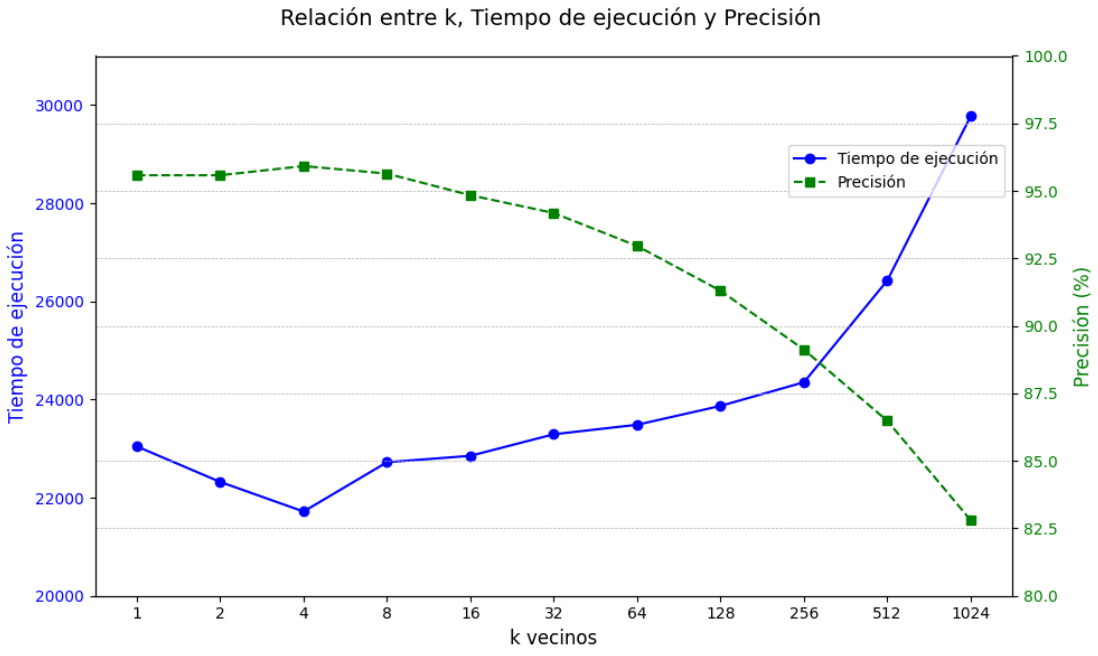

Figura 2: Relación entre k vecinos, tiempo de ejecución y precisión obtenida

2.4. Evaluación: Modo imagen

El proceso de evaluación continuó con varios casos del modo de ejecución normal del programa. Este modo, como se hizo mención anteriormente, consta de utilizar la base de datos y el algoritmo kNN para reconocer el número representado en la imagen pasada por parámetro. A continuación, se mostrarán algunos casos de interés en los cuales se dibujó un número en una imagen (de 28 x 28 píxeles con perfil de color en blanco y negro) y se introdujo en el programa.

Para la imagen de un número seis que se muestra en la Figura 3, los resultados obtenidos fueron los mostrados en el Cuadro 3, revelando un acierto por parte del programa.

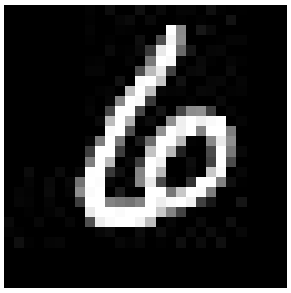

Figura 3: Número seis dibujado por el grupo

"k" vecinos	Cant. Threads	Resultado	Tiempo de Ejecución
1	1	6	1312
5	1	6	1328
10	1	6	1325
1	4	6	1314
5	4	6	1287
10	4	6	1314

Cuadro 3: Resultados al pasar el número seis

No obstante, para el número nueve de la Figura 4, los resultados del Cuadro 4 indican que pueden haber fallos. Además, se muestra que variar el número de "k", al menos en este caso, no generó ningún cambio en el resultado.

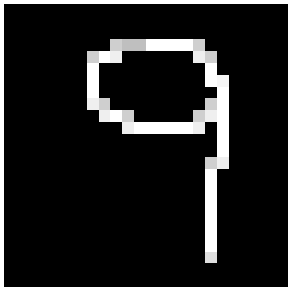

Figura 4: Número nueve dibujado por el grupo

"k"vecinos	Cant. Threads	Resultado	Tiempo de Ejecución
1	1	6	1283
5	1	6	1279
10	1	6	1300
256	1	6	1295

Cuadro 4: Resultados al pasar el número nueve

A diferencia del caso anterior, se obtuvieron dos casos con mucha variación en el resultado dependiendo del número de "k" vecinos de la prueba, estos fueron los siguientes:

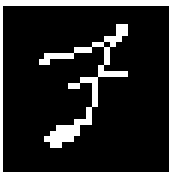

Figura 5: Un siete parecido a una semicorchea

Figura 6: Un cinco algo extraño, similar a una clave musical.

En dichos casos, los resultados obtenidos fueron: (Véase Cuadro 5 y Figura 7).

"k"vecinos	Siete semicorchea		Cinco extraño	
	Resultado	Tiempo de Ejecución	Resultado	Tiempo de Ejecución
1	3	1296	1	1290
5	6	1286	5	1288
10	6	1288	1	1275
50	6	1289	6	1290
100	2	1289	6	1292
256	2	1276	6	1295
500	2	1304	6	1309
5000	7	1297	7	1332
20000	1	1352	1	1306
50000	1	1362	1	1344
60000	1	1370	1	1348

Cuadro 5: Resultados obtenidos al pasar el siete y el cinco

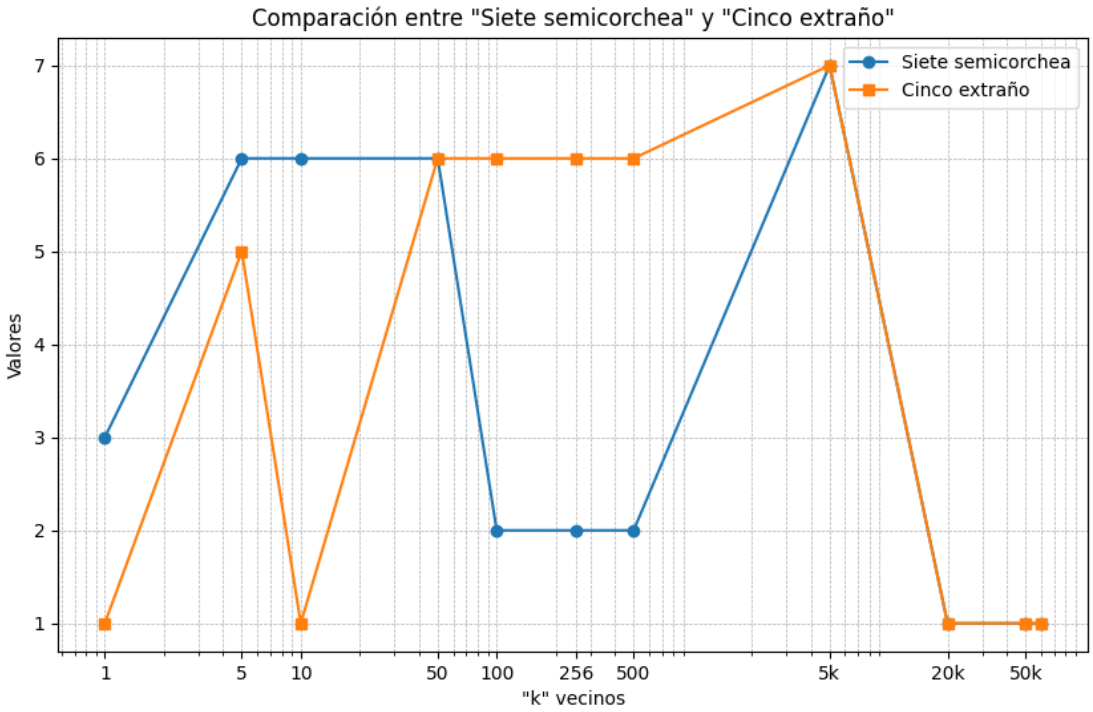

Figura 7: Variación en el resultado según "k" vecinos

3. Análisis

Se puede concluir, a través de los datos del Cuadro 1 que, dada una cantidad de líneas fijas, el menor tiempo de ejecución lo va a tener la mayor cantidad de threads. Es decir, que el tiempo de ejecución lo tienen los 16 threads al leer solo 500 líneas del archivo de *test*. Por otro lado, el mayor tiempo de ejecución lo tiene un thread único al leer 5000 líneas del archivo. En principio, siguiendo la tendencias lineales de la Figura 1, esto supone una clara conclusión: a mayor número de threads, menor tiempo de ejecución.

Sin embargo, al analizar el gráfico verticalmente podemos notar que la eficiencia que supone aumentar el número de threads no crece linealmente, sino de manera logarítmica (asintótica). La mejora de aumentar de 8 a 16 *threads* es mucho menor que la de 4 a 8, aunque la relación sea duplicar en ambos casos. Esto implica, matemáticamente hablando, que en algún momento no va a ser rentable aumentar los threads para mejorar los tiempos de ejecución. Podría medirse para todo el archivo de pruebas (10.000 lineas) que pasaría con 32, 64 o 128 *threads* para ver la relación asintótica pero, no es necesario hacerlo, en la Figura 1 puede verse como no cambia usar 8 o 16 threads para 500 o 1000 líneas.

Respecto al *buffer*, para todas las pruebas se dejó fijo el número. Pensamos que reducir el tamaño del buffer podría generar un *Bottleneck* (cuello de botella), ya que, por ejemplo, para un **bufferSize** igual a 1, los *Workers* estarían tomando las tareas del buffer de a una y la carga también sería de a una. Es decir, el *thread* principal tiene que estar esperando que un worker tome una tarea para cargar otra. Esto, en consecuencia generaría una reducción de performance pero, como no lo medimos, no sabemos que tanto afecta al tiempo de ejecución.

El número de "k" vecinos que quieren obtenerse influye negativamente en la precisión y aumenta el tiempo de ejecución, según el Cuadro 2.3. Esto puede notarse también en el Cuadro 5, ya que al aumentar el número de "k" fluctúa el resultado obtenido. No obstante, que fluctúe no implica que se aleje del resultado deseado, ya que en el caso del siete, el resultado correcto lo retorna con 5000 vecinos. Sin embargo, al aumentar más el "k" el resultado se torna al número uno, en ambos casos

de dicho Cuadro. Este patrón lo notamos en algunas oportunidades, con números más similares al uno, por ejemplo, cuatro o siete. Respecto al tiempo de ejecución, aumentar el número k no lo aumenta significativamente, ya que la recolección de los "k" vecinos se guarda en un tiempo $O(log(K))$, siendo $K = 60{,}000/numWorkers$.

La cantidad de líneas del archivo de prueba hace, inevitablemente, que aumente el tiempo de ejecución del programa, ya que se están agregando más operaciones a realizar. Pero puede aumentar la precisión. Para esta última afirmación nos basamos en que con 10000 líneas hemos obtenido una precisión de 97 %, un punto y medio porcentual mayor a los resultados mostrados en este trabajo. Por lo que, si es posible que aumente la precisión, pero no siempre es así como puede verse en el Cuadro 1, donde a veces disminuye.

Mediante el análisis de todos los datos recolectados se concluye que el punto óptimo respecto a los parámetros de entrada es:

- **k**: rondando el número 10.
- **numWorkers**: 8 (usar más threads consume recursos innecesariamente).
- **bufferSize**: 8 (coincidiendo con el número de *Workers*).
- Número de líneas de prueba: 10000.

Respecto a las imagenes de prueba, los resultados no siempre fueron consistentes y la razón se debe a la base de datos de imagenes empleada de entrenamiento y a la distribución de pixeles de cada una de ellas: El desempeño del algoritmo kNN para clasificar imágenes de números depende en gran medida de la similitud de los pixeles entre la imagen de prueba y las imágenes del conjunto de datos MNIST. Si la imagen de entrada tiene características similares, como contraste claro entre dígitos y fondo, el algoritmo puede identificar correctamente el número como el número seis de la Figura 3 independientemente del número "k.^{el}egido. Sin embargo, si la imagen presenta diferencias significativas, como el número nueve de la Figura 4, como variaciones en los tonos de blanco el algoritmo experimenta dificultades para determinar el número de manera confiable. Más aún, si la forma de los dígitos da una distribución de los pixeles cuyo desempate para que indique cual número puede ser es "k"dependiente, como ocurrió en la Figura 5 o Figura 6. Por lo cual, si se desea utilizar el programa para predecir los dígitos de forma fehaciente, hay que preprocesar las imagenes para que sean adecuadas en formato, constrate y tonalidad.