Universidad ORT Uruguay
Facultad de Ingeniería

Mejora de la eficiencia de KNN utilizando programación paralela en F#

Entregado como requisito para la obtención del título de Licenciado en Ingeniería de Software

Franz Mayr 178236 - Franco Patrone 175993

Tutor: Sergio Yovine

2016

Nosotros, Franco Patrone y Franz Mayr, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto Integrador;

- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;

- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;

- En la obra, hemos acusado recibo de las ayudas recibidas;

- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;

- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Franco Patrone | Franz Mayr

11-04-2016

Dedicado con mucho amor a nuestros familiares y amigos en agradecimiento a su constante apoyo.

Abstract

Este trabajo se centra en la mejora de la eficiencia del algoritmo de *machine learning* denominado *k-Nearest Neighbours* (KNN) mediante el uso de programación paralela y su implementación con F# haciendo uso del *framework* Microsoft .NET, buscando identificar los mecanismos del lenguaje mejor adaptados al problema.

Debido a la creciente generación de datos y la necesidad de análisis de los mismos, se vuelve central la mejora de la *performance* de este tipo de algoritmos.

La utilización de un paradigma de programación funcional se ve motivada por el hecho de que este permite una gran abstracción al implementar problemas complejos. De lo anterior se desprende la elección de F# pues es un lenguaje que implementa dicho paradigma.

Como parte de este trabajo se realizó un estudio de las distintas alternativas de implementación de cómputo paralelo en KNN. Para ello se utilizó una especificación en el lenguaje FXML, y una investigación sobre las opciones de paralelización ofrecidas por F# y el *framework* .NET.

Se realizaron diseños e implementaciones acorde a la especificación del algoritmo antes mencionada. Se ejecutaron pruebas de eficiencia a fin de medir las mejoras obtenidas, y de eficacia con el fin de comprobar que no hubiera cambios en la misma. Estas pruebas se compararon con pruebas de una implementación secuencial de referencia.

De las implementaciones realizadas se obtuvieron dos librerías, una que implementa el algoritmo de KNN haciendo uso de la mejora conseguida mediante la programación paralela y la otra implementa el modelo de programación paralela *map-reduce.*

Se concluye que F# junto con el *framework* .NET permite hacer uso del cómputo en paralelo en busca de la mejora de la eficiencia de algoritmos, ofreciendo cuatro opciones principales de paralelización disponibles para el desarrollador.

Se obtuvo un *speedup* de 2 aproximadamente en los entornos de prueba utilizados. Esta mejora de eficiencia es visible al comparar la versión secuencial del algoritmo con la versión paralela final.

Palabras clave

optimización de KNN; programación paralela;*speedup*; *map-reduce*; *machine learning*; programación funcional; F#; FXML; Microsoft .NET;

Índice

1. Introducción **9**
1.1. Contexto y motivación 9
1.2. Estructura del documento 10

2. Marco Teórico **11**
2.1. *Machine learning* 11
2.1.1. Tipos de algoritmos 12
2.1.2. KNN 12
2.2. Programación paralela 13
2.2.1. Ley de Amdahl 14
2.2.2. Map-Reduce 15
2.3. FXML 15
2.4. Programación funcional 16
2.4.1. F# como lenguaje funcional 16

3. Diseño del algoritmo de KNN **18**
3.1. Identificación de oportunidades de paralelización 18
3.1.1. Especificación en FXML 18
3.1.2. Análisis de la especificación 21
3.2. Opciones de paralelización en F# 22
3.2.1. Paralelización en F# 22
3.2.2. Primitivas de FXML en F# 24
3.3. Diseño KNN en paralelo usando F# 26

4. Pruebas de eficiencia **28**
4.1. Contexto de las pruebas 28
4.1.1. Ambiente de ejecución 28
4.1.2. Problema utilizado 29

5. Construcción del algoritmo **30**
5.1. Algoritmo Secuencial 30

5.1.1. Implementación 30
5.1.2. Pruebas de eficiencia 30
5.2. Algoritmo Paralelo 31
5.2.1. Implementación 31
5.2.2. Pruebas de eficiencia 36

6. Abstracción de la solución **39**
6.1. MapReduce 39
6.1.1. Construcción 39
6.1.2. Pruebas de eficiencia 41
6.2. KNN 43
6.2.1. Construcción 43
6.2.2. Pruebas de eficiencia 43

7. Exploración del uso de las librerías **49**
7.1. GPU con librería KNN 49
7.1.1. Pruebas de eficiencia 50
7.2. Librería KNN con MapReduce jerárquico 50
7.2.1. Pruebas de eficiencia 52

8. Pruebas de eficacia **54**
8.1. Contexto de las pruebas 54
8.1.1. Ambiente de ejecución 54
8.1.2. Problema utilizado 54
8.2. Conclusiones de las pruebas 55

9. Conclusiones **56**

10. Trabajo a futuro **59**

11. Bibliografía **60**

12. Anexo A: Cuadros de pruebas de eficiencia **63**
12.1. Pruebas Intermedias 63
12.2. Pruebas Exploratorias 67
12.3. Pruebas Completas 69

13. Anexo B: Cuadros de pruebas de eficacia **79**

14. Anexo C: Guías de uso de las librerías **81**
14.1. MapReduce 81
14.1.1. Tipos de datos 81

14.1.2. Funciones 81
14.2. KNN 82
14.2.1. Tipos de datos 82
14.2.2. Funciones 82

1 Introducción

Gracias a los avances en las distintas áreas de la tecnología informática y el bajo costo de la construcción de dispositivos tecnológicos, cada vez más potentes y con más funcionalidades, la cantidad de datos e información generada va en aumento, un aumento que parece ser exponencial.

Para el futuro cercano, y con la masificación de la denominada Internet de las cosas, se prevé que se manejen cantidades de información muy superiores a las actuales, tal como se menciona en [1]. En este futuro cercano, la gran mayoría de dispositivos, herramientas, objetos y otras cosas que utilizamos en la actualidad en la vida cotidiana estarán conectados a Internet; transmitiendo información y comunicándose con otros dispositivos. Esta enorme cantidad de datos, generada principalmente por sensores, debe ser interpretada para poder obtener los beneficios esperados.

Como lo dice Amster [2], la interpretación y análisis de esos datos solo puede lograrse mediante el conjunto de técnicas que conforma *machine learning*, capaces de procesar grandes cantidades de datos y obtener información de interés sobre los mismos. Es por ello que los algoritmos que implementan las técnicas mencionadas cada vez se vuelven más utilizados y se hacen más complejos.

1.1. Contexto y motivación

En los algoritmos que aplican el conjunto de técnicas de *machine learning* siempre es un factor determinante la eficiencia con la que se ejecutan debido a la gran cantidad de datos que deben procesar, que en ocasiones son más de millones.

Las estrategias de optimización elegidas, al igual que las plataformas, lenguajes, herramientas, hardware, etc. son de gran importancia para poder garantizar tiempos de ejecución cada vez más ajustados y optimizados.

Teniendo en cuenta lo mencionado anteriormente, surgió la motivación en la búsqueda e investigación de la factibilidad de implementación ofrecida por un lenguaje funcional, como F# [3], para la aplicación de algoritmos de *machine learning* que aprovechen el poder de procesamiento en paralelo, buscando de este modo optimizar los tiempos de ejecución, como es propuesto en [4].

Los lenguajes funcionales aumentan la abstracción en búsqueda de una mayor facilidad a la hora de construcción de los algoritmos, sacrificando eficiencia de los mismos al alejar al desarrollador de la posibilidad de tener mayor control sobre cómo se realizan ciertas acciones, dejando que el compilador utilizado sea el responsable de elegir cómo resolver algunas decisiones. De todos modos, la elección de un lenguaje funcional viene motivada por ese mismo poder de abstracción que brindan al momento de implementar soluciones a problemas complejos.

Se seleccionó específicamente F# como el lenguaje a utilizar por ser *open source* y por tener una gran integración con los lenguajes de la plataforma .NET, permitiendo integrar módulos creados en este lenguaje con una amplia diversidad de programas.

1.2. Estructura del documento

Este documento contiene la información de la investigación y productos de software derivados de la misma. La estructura del mismo se detalla a continuación:

El siguiente capítulo esta compuesto por el marco teórico necesario para que un lector técnico, pero sin conocimientos en los conceptos claves de la investigación, pueda obtener la información teórica necesaria para la comprensión de la misma. En el capítulo 3 se describen las etapas de diseño del algoritmo y los resultados obtenidos en cada una de ellas. El capítulo 4 se enfoca en describir el contexto de pruebas de eficiencia. En el capítulo 5 se explica la construcción del algoritmo, las decisiones tomadas y las pruebas realizadas para cada una de las versiones desarrolladas. En el capítulo 6 se expresan las decisiones y cambios realizados para crear una mayor abstracción del algoritmo, en busca de una utilización del mismo en diversos problemas. Luego, en el capítulo 7, se detallan exploraciones del uso de la abstracción comentada. En los capítulos 8, 9 y 10 se comentan las pruebas de eficacia realizadas, las conclusiones extraídas y el trabajo a futuro a realizar respectivamente.

2 Marco Teórico

2.1. *Machine learning*

En español se lo conoce como aprendizaje de máquina, se lo puede definir de distintas formas, dependiendo del autor que lo haga.

Por ejemplo, Mitchell en su libro *machine learning* [5] define a esta rama de la inteligencia artificial como aquella que se preocupa en contestar la pregunta de como construir programas de computadoras que mejoren automáticamente mediante la experiencia.

Desde un punto de vista estadístico, la definición cambia para centrarse en la importancia de extraer patrones y tendencias en la gran cantidad de datos que se generan en la actualidad, logrando entender qué es lo que nos dice dicha información. A esto se le denomina aprender de los datos, así lo definieron Hastie, Tibshirani y Friedman, tres estudiantes de estadística de Stanford, en su libro [6].

Adoptando la definición provista por Mitchell, otro autor, Marsland [7], resalta lo que para él es la característica más importante de *machine learning*, el hecho de que es una disciplina que se encuentra en los límites de la estadística, las matemáticas, la ingeniería y la ciencia computacional. Con esto se puede entender la naturaleza multidisciplinaria que requieren estos tipos de algoritmos.

Se puede concluir que *machine learning* es una rama de la inteligencia artificial, por la cual se entrenan programas de computadora para que mejoren mediante la experiencia. Estos tipos de programas o algoritmos son generalmente utilizados para comprender datos y extraer información de interés de los mismos. Para poder lograr esto se requieren conocimientos de distintas áreas, cada una aportando un enfoque complementario a la solución de estos problemas complejos.

2.1.1. Tipos de algoritmos

Según Mohri [8], una manera de clasificar los algoritmos de *machine learning* es por el tipo de aprendizaje, las categorías principales en esta taxonomía son dos: aprendizaje supervisado y aprendizaje no supervisado.

Aprendizaje supervisado Se recibe un conjunto de datos con etiquetas y el algoritmo clasifica los datos nuevos en alguna de las clases representadas por dichas etiquetas. Dentro de esta categoría existen dos tipos de problemas principales:

- Clasificación: Consiste en asignar una clase a un dato recibido.
- Regresión: Consiste en predecir el valor de un dato recibido.

Aprendizaje no supervisado Se reciben datos sin etiquetar y se hacen predicciones sobre los datos nuevos. Los tipos principales de problemas en esta categoría son *clustering* y reducción de dimensionalidad.

2.1.2. KNN

Dentro del conjunto de algoritmos disponibles en *machine learning*, encontramos a KNN, por sus siglas en inglés *k-Nearest Neighbours*, cuya traducción sería los K vecinos más cercanos.

Tal como lo define Thirumuruganathan [9], KNN es un tipo de algoritmo que no hace asunciones en cuanto a la distribución de los datos con los que se está trabajando, en otras palabras no espera que los datos recibidos obedezcan un patrón teórico matemático. También es un algoritmo que se caracteriza por carecer de una etapa de generalización del conjunto de entrenamiento, es decir que mantiene casi completo dicho conjunto para realizar los cómputos con cada dato de prueba.

La casi inexistencia de una etapa previa de entrenamiento del algoritmo se traduce en un costo de tiempos y recursos en la etapa de clasificación. Y dichos costos se verán directamente afectados por el tamaño del conjunto de entrenamiento utilizado.

KNN realiza el supuesto de que los datos a procesar tienen un atributo o característica espacial, en otras palabras los datos tienen un atributo escalar o multidimensional que permiten trabajar con una noción de distancia entre cada dato.

De forma muy resumida el funcionamiento de KNN se puede explicar de la siguiente manera. Se toma un conjunto de datos de prueba, y para cada uno de los datos de prueba se calcula la distancia que hay entre ese dato de prueba y todo el conjunto de entrenamiento. Luego, se buscan los k datos de entrenamiento que se encuentran más cercanos al dato de prueba en cuestión. Se termina por realizar una votación en base a la clase de los K datos de entrenamiento seleccionados, los cuales se encuentran ya clasificados. La clase que fue más votada es la clase que se le asignará al dato de prueba que se estaba clasificando.

La variante más simple de KNN es tomando un valor de K igual a 1. De esta forma, el funcionamiento del algoritmo pasa a ser el detallado a continuación. Luego del calculo de las distancias de todos los datos de entrenamiento con el dato de prueba a clasificar, se selecciona el dato de entrenamiento más cercano. La clase que tiene el dato de entrenamiento seleccionado es la clase que se le asignará al dato de prueba en cuestión.

KNN es un algoritmo que puede utilizarse tanto para regresión como para clasificación, esto depende de la utilización que se le de a la función del cálculo de distancia. En el contexto de este trabajo será utilizado para clasificación.

Si el lector lo desea, puede referirse al artículo de investigación original realizado por Fix y Hodges [10] en la Universidad de California, para ampliar la lectura sobre este tipo de algoritmos.

2.2. Programación paralela

Barney en su tutorial realizado para el Lawrence Livermore National Laboratory [11] define de manera sencilla el concepto de programación paralela. El autor dice que la misma implica el desarrollo de código que será procesado en paralelo por distintas unidades de cómputo y que para ello es necesario dividir el código en porciones que se ejecutarán en simultaneo en distintas unidades de procesamiento, manejando concurrencia sobre recursos compartidos como pueden ser CPU o memoria RAM.

El objetivo de la programación paralela es que el tiempo total de ejecución del programa se vea reducido. Este objetivo se vuelve central para las cada vez más demandantes aplicaciones desarrolladas, con gran enfoque en la *performance*. Si bien es verdad que los costos de computadoras potentes están en descenso, para aplicaciones en servidores con gran demanda de procesamiento ya sea por cálculos complejos, gran cantidad de cálculos, gran demanda de usuarios, etc. la necesidad es evidente.

Si el lector le interesa continuar la lectura sobre programación paralela, puede realizarlo en el libro introductorio realizado por Grama, Gupta, Karypis y Kumar [12].

2.2.1. Ley de Amdahl

Amdahl presenta en la conferencia citada en [13] las bases teóricas del concepto de que se puede obtener un *speedup* o mejora en tiempo de ejecución de una tarea con una carga de trabajo fija al mejorar el contexto de ejecución de la misma.

Este concepto se puede escribir matemáticamente de la siguiente manera:

$$Speedup(n) = \frac{1}{(1-p) + \frac{p}{n}} \tag{2.1}$$

Donde:

Speedup(n) es el *speedup* teórico de la ejecución de la tarea entera.

n es el *speedup* de la ejecución de la porción paralelizable de la tarea.

p es el porcentaje de tiempo de ejecución de la sección paralelizable de la tarea (valor entre 0 y 1).

Y además:

$$\begin{cases} Speedup(n) \leq \frac{1}{1-p} \\ \lim_{n\to\infty} Speedup(n) = \frac{1}{1-p} \end{cases} \tag{2.2}$$

A su vez el tiempo de ejecución de una tarea T en un sistema se puede desglosar de la siguiente manera:

$$T = (1-p)T + pT \tag{2.3}$$

Donde p es el porcentaje anteriormente definido y $(1-p)T$ el tiempo que demora la porción del código que no se beneficiaría de una mejora en el sistema y pT el tiempo que demora la porción que si.

Si el sistema es mejorado, entonces el tiempo de ejecución $T(n)$ se puede desglosar de la siguiente manera:

$$T(n) = (1-p)T + \frac{p}{n}T \tag{2.4}$$

En este caso se puede observar que lo que varía es la ecuación del tiempo de ejecución de la parte paralelizable, que se ve disminuida por el factor n (el *speedup*).

Finalmente, pensando en una carga fija de trabajo se puede decir que:

$$Speedup(n) = \frac{T}{T(n)} = \frac{1}{(1-p)+\frac{p}{n}} \tag{2.5}$$

La ecuación 2.5 nos sirve para conocer el *speedup* cuando no se conocen los porcentajes de código paralelizable, pues se calcula como resultado de los tiempos de ejecución con y sin mejora.

2.2.2. Map-Reduce

Es un modelo de programación utilizado para computación paralela que permite procesar grandes cantidades de datos. Map-Reduce fue popularizado a principios del siglo XXI por Dean y Ghemawat [14] de Google.

El modelo utiliza dos funciones definidas por el desarrollador: una función de mapeo y otra de reducción que se describen a continuación.

Función de mapeo es definida por el desarrollador, procesa un conjunto de entrada y devuelve un conjunto de pares clave/valor intermediarios.

Función de reducción es definida por el desarrollador, procesa el conjunto de pares clave/valor intermediarios y los mezcla, para obtener un subconjunto más acotado. Comúnmente el subconjunto tiene una cantidad igual a cero o uno.

Los resultados obtenidos por la función reductora son los que devuelve el modelo Map-Reduce.

2.3. FXML

FXML es un lenguaje algebraico que ofrece una cantidad mínima de primitivas que permiten especificar algoritmos con un alto nivel de abstracción, ofreciendo una fácil identificación de las oportunidades de procesamiento en paralelo en dichos algoritmos.

El lenguaje se caracteriza por ser independiente de cualquier plataforma, poseer un alto grado de abstracción, y proveer expresividad semántica y

sintáctica para expresar el algoritmo que se desea especificar, apoyándose en un modelo de ejecución abstracto.

Mediante el uso de dependencias de control y de datos, el lenguaje permite especificar las restricciones de paralelismo determinadas por restricciones de tiempo o precedencia.

Para ampliar la lectura sobre el lenguaje, su sintaxis o sus aplicaciones se puede leer el artículo de investigación original de Yovine [15] o el libro de Alexander y Gardner [16].

2.4. Programación funcional

El paradigma de programación funcional es aquel que se basa en la utilización de funciones, y cuya ejecución consiste en la evaluación de las mismas para obtener un resultado.

Como es definido en el artículo de Rivadera [17], la programación funcional intenta alejarse de los lenguajes procedimentales inspirados en el funcionamiento interno de las computadoras (que realizan abstracciones de recursos como la memoria) y pasa a acercarse a el modelo matemático abstracto para resolver cualquier tipo de problemas.

Actualmente los lenguajes de programación funcional cada vez son más populares por las ventajas que ofrecen y por la facilidad de implementación de soluciones a problemas complejos.

El lector interesado puede ampliar la lectura sobre el paradigma y un lenguaje funcional en específico en el libro de Ruiz, Gutiérrez, Guerrero y Gallardo [18].

2.4.1. F# como lenguaje funcional

F# [3], leído *F Sharp*, es un lenguaje de programación fuertemente tipado y multiparadigma: imperativo, funcional y orientado a objetos. Es un lenguaje *open source* desarrollado y actualizado por la fundación F# Software Foundation, Microsoft y colaboradores. Un compilador multiplataforma es distribuido por la fundación que permite desarrollar aplicaciones en F# en Windows, Linux y Mac.

Visual F# [19] es la versión de F# que agrega la integración en el *framework* .NET de Microsoft, permitiendo utilizar las librerías de dicho *framework*, además de posibilitar la compatibilidad entre aplicaciones escritas en distintos lenguajes de la plataforma, como pueden ser C#, Visual Basic, entre otros.

3 Diseño del algoritmo de KNN

3.1. Identificación de oportunidades de paralelización

Para comenzar a diseñar el algoritmo se identificaron las oportunidades de computación paralela que este ofrece. Primero se especificó el algoritmo de forma abstracta sin apegarse al lenguaje de implementación, y posteriormente se realizó un análisis con el fin de encontrar dichas oportunidades.

3.1.1. Especificación en FXML

Se comenzó por realizar una especificación del algoritmo de KNN utilizando el lenguaje FXML para reconocer oportunidades de paralelización que de otra forma no se podrían identificar o podrían ser pasadas por alto.

A continuación se detalla la especificación del algoritmo realizada, cabe aclarar que las funciones que se encuentran subrayadas en la especificación de FXML no fueron especificadas debido a que son una parte variable del algoritmo de KNN y dependerán del dominio de aplicación especifico en que se utilice el algoritmo.

Las funciones de la especificación 3.1 son descritas a continuación:

Votar Función encargada de decidir la clase más representativa del conjunto.

DarVoto Cada elemento emite un voto con el valor de su clase.

ReducirContandoVotos Cuenta la cantidad de votos de cada clase y devuelve una estructura con pares (valor, cantidad de votos).

ObtenerMaximo Obtiene el valor más votado del conjunto.

```
1  DEP kMenor_i → voto_i, ∀i
2  DEP voto_j → votos, ∀j
3  DEP votos → maximo
4
5  Votar (kMenores)
6  {
7      par
8      {
9          forall  kMenor ∈ kMenores
10         {
11             voto = DarVoto kMenor
12         }
13         votos = ReducirContandoVotos voto
14     }
15     maximo = ObtenerMaximo votos
16 }
```

Código 3.1: Votar

```
1  DEP d_i → distancia_i,∀i
2  DEP distanica_j → resultado, ∀j
3
4  ObtenerDistancia(datoEntrenamiento, datoPrueba)
5  {
6        par
7      {
8          forall  d  ∈ CANT_DIMENSIONES
9          {
10             d1 = datoPrueba.Dimensiones[d]
11             d2 = datoEntrenamiento.Dimensiones[d]
12             distancia  = CalcularDistancia d1 d2
13         }
14         resultado  = ReducirDimensiones distancia
15     }
16 }
```

Código 3.2: ObtenerDistancia

Las funciones de la especificación 3.2 son descritas a continuación:

ObtenerDistancia Función que obtiene un valor de distancia entre el dato de prueba y un dato de entrenamiento evaluando todas las dimensiones de los datos, comparándolas y finalmente reduciendo estas comparaciones a un solo valor.

ReducirDimensiones Su responsabilidad, como lo dice su nombre, es reducir los valores de distancia entre dimensiones de un dato de prueba y uno de entrenamiento a un único valor representativo.

CalcularDistancia Función que calcula la distancia entre dos valores de un mismo tipo de dimensión.

```
 1 DEP datoPrueba, datoEntrenamiento_i distancia_i, ∀i
 2 DEP distancia_j, K → kMenores, ∀j
 3 DEP kMenores →resultado
 4
 5 KNN (K, datosEntrenamiento, datoPrueba)
 6 {
 7     par
 8     {
 9         forall  datoEntrenamiento  ∈ datosEntrenamiento
10         {
11             distancia  = ObtenerDistancia datoEntrenamiento datoPrueba
12         }
13         kMenores = ObtenerKMenores distancia
14         resultado = Votar kMenores
15     }
16 }
```

Código 3.3: KNN

Las funciones de la especificación 3.3 son descritas a continuación:

KNN Función responsable de clasificar el dato de prueba recibido comparando su distancia a todos los datos de entrenamiento.

```
 1 DEP datoPrueba_i r_i, ∀i
 2 DEP r _jresultados,∀j
 3
 4 CalculoKNNParaTodosLosDatosTest(K, datosEntrenamiento, datosPrueba)
 5 {
 6     forall  datoPrueba  ∈ datosPrueba
 7     {
 8         resultado  = KNN K datosEntrenamiento datoPrueba
 9     }
10     resultados  = ConcatenarResultados resultado
11 }
```

Código 3.4: CalculoKNNParaTodosLosDatosTest

Las funciones de la especificación 3.4 son descritas a continuación:

CalculoKNNParaTodosLosDatosTest Función encargada de computar KNN para cada uno de los datos de prueba. Obteniéndose al final la clasificación de los mismos.

ConcatenarResultados Se encarga de agrupar los resultados de la clasificación en una estructura.

3.1.2. Análisis de la especificación

Observando la especificación realizada del algoritmo de KNN se pueden realizar algunas observaciones. A continuación se detallan todas las oportunidades de paralelización identificadas para este algoritmo, agrupadas por función.

CalculoKNNParaTodosLosDatosTest

Oportunidad 1 Existe la posibilidad de aplicar el algoritmo de KNN para el conjunto de datos de prueba de forma paralela, en otras palabras, ejecutar KNN sobre todo el conjunto de pruebas al mismo tiempo.

KNN

Oportunidad 2 Se puede realizar el cómputo en paralelo del cálculo de las distancias entre un dato de prueba y todo el conjunto de datos de entrenamiento.

Oportunidad 3 A medida que se van realizando las operaciones para el cálculo de distancia entre un dato de prueba y un dato de entrenamiento, se puede ir seleccionando si ese dato de prueba puede estar o no dentro de los K más cercanos.

ObtenerDistancia

Oportunidad 4 Existe una oportunidad de paralelización en el cálculo de la distancia entre un dato de prueba y un dato de entrenamiento. Dicha distancia consiste en realizar el cálculo de distancias entre cada una de las dimensiones.

Oportunidad 5 Otra oportunidad esta en la reducción de las distancias calculadas para cada dimensión, o mejor dicho a medida que se realiza el cálculo de las distancias para cada dimensión, se pueden comenzar a reducir, sin tener que esperar a que estén todas las distancias para empezar la reducción.

Votar

Oportunidad 6 Existe la oportunidad de paralelizar la invocación de la función "DarVoto"sobre cada "kMenor"seleccionado.

Oportunidad 7 Se puede comenzar a contar los votos sin que se haya terminado de dar el voto de cada uno de los "kMenores".

A partir de ahora toda alternativa o posibilidad de paralelización identificada anteriormente mediante el análisis de la especificación de FXML será denominada "oportinudad".

3.2. Opciones de paralelización en F#

En esta sección se explicarán las herramientas que F# provee a nivel de paralelismo y se hará un mapeo entre estas herramientas y las primitivas provistas por el lenguaje FXML.

3.2.1. Paralelización en F#

El lenguaje cuenta con herramientas de paralelismo propias además de las provistas por el *framework* .NET.

Dentro de .NET existen dos grandes opciones:

PLINQ La librería es una implementación paralela de LINQ to Objects que habilita consultas paralelas sobre colecciones de datos.

Dentro de PLINQ se pueden encontrar implementaciones de los siguientes patrones:

- Parallel Loops: permite la ejecución de una función independiente sobre todos los datos de un conjunto, en otras palabras posibilita la paralelización de datos.
- Parallel Aggregation: posibilita realizar una operación de agregación o reducción sobre un conjunto de datos, para obtener un único resultado.

Task Parallel Library (TPL) Es una librería que se basa en el concepto de *task* o tarea que representa una acción asíncrona.

Dentro de TPL se pueden implementar los siguientes patrones:

- Parallel Tasks: permite ejecutar acciones en paralelo, posibilita la paralelización de tareas. A veces es conocido como el patrón Fork/Join o Master/Worker.
- Futures: haciendo uso de tareas, se puede implementar *futures*. Un *future* es una variable cuyo valor es desconocido y será conocido más tarde en la ejecución.
- Dynamic Task Parallelism: permite realizar tareas en paralelo y agregar tareas dinámicamente a una cola de trabajo para que sean ejecutadas. Aplica principalmente para problemas del estilo de *divide and conquer* y/o para aquellos problemas cuya versión secuencial utiliza recursión.
- Pipelines: haciendo uso de *tasks* y colecciones concurrentes se puede implementar este patrón. El mismo consiste en una secuencia de productores-consumidores que realizan transformaciones sobre un conjunto de datos.

Para realizar las sincronizaciones sobre estructuras de datos existe implementaciones de colecciones concurrentes de .NET, que son colecciones que permiten ser accedidas por *threads* de manera concurrente, habilitando de ese modo sincronización entre *threads* ejecutando en paralelo.

Para continuar la lectura sobre las opciones de paralelización de .NET, se recomienda al lector el artículo ofrecido en MSDN [20] sobre programación paralela con el *framework*.

Por otro lado, F# ofrece dos opciones principales:

Array.Parallel Module Es un módulo que cuenta con implementaciones de operaciones paralelas sobre *arrays*.

Asynchronous Workflows habilita la realización de cómputos asíncronos y es generalmente usado para ejecutar operaciones en segundo plano y continuar la ejecución en el *thread* principal sin esperar el resultado.

A partir de ahora se utiliza el término "opción de paralelización" para referirse a las alternativas antes mencionadas ofrecidas por F# y/o .NET que posibilitan la ejecución en paralelo.

3.2.2. Primitivas de FXML en F#

Par No existe una primitiva del lenguaje que mapee directamente pero se puede implementar haciendo uso de:

- Asynchronous Workflows, para cada bloque de código que se quiera ejecutar en paralelo, se escribe dentro de un bloque async (ver ejemplo en código 3.5).
- Task Parallel Library de .NET, creando un *task* para cada bloque de código que se quiera ejecutar en paralelo, y lanzando las tareas en paralelo (ver ejemplo en código 3.6).

```
1 let sleep x = async {
2                       printfn "empezo sleep %i" x
3                       do! Async.Sleep x
4                       printfn "termino sleep %i" x
5                   }
6
7
8 let par = [sleep 20000; sleep 5000]
9            |> Async.Parallel
10           |> Async.RunSynchronously
```

Código 3.5: Primitiva Par utilizando Asynchronous Workflows

```
1 let funA () = printfn "empezo funA"
2               Thread.Sleep(2000)
3               printfn "termino funA"
4
5 let funB () = printfn "empezo funB"
6               Thread.Sleep(5000)
7               printfn "termino funB"
8
9 let par = Parallel.Invoke([|(Action funA); (Action funB)|])
```

Código 3.6: Primitiva Par utilizando Task Parallel Library

Forall Existen dos maneras primarias de representar la primitiva:

- El forall ofrecido por PLINQ se mapea directamente con la primitiva definida en FXML (ver ejemplo en código 3.7), pero con un poco más de inteligencia, al tomar las decisiones de paralelización y particionamiento de la colección sobre la cual se trabaja. De todas formas se puede especificar como se quiere que trabaje, por ejemplo definiendo particionadores propios.

- El parallel map perteneciente a Array.Parallel Module también permite implementar esta primitiva (ver ejemplo en código 3.8) pero con menor inteligencia que el forall de PLINQ.

```
1 let doSomething x = printfn "empezo doSomething %i" x
2                     Thread.Sleep(x*1000+1000)
3                     printfn "termino doSomething %i" x
4                     x+1
5
6 let anArray: int[] = Array.init 5 (fun index -> index+1)
7
8 let plinqForAllExample = anArray.AsParallel().ForAll(fun x ->
    doSomething x|>ignore)
```

Código 3.7: Primitiva ForAll utilizando PLINQ

```
1 let doSomething x = printfn "empezo doSomething %i" x
2                    Thread.Sleep(x*1000+1000)
3                    printfn "termino doSomething %i" x
4                    x+1
5
6 let anArray = Array.init 5 (fun index -> index+1)
7
8 let parallelMapForAllExample = Array.Parallel.map doSomething
    anArray
```

Código 3.8: Primitiva ForAll utilizando Array.Parallel.map

Reduce Se puede implementar en PLINQ utilizando las funciones de agregación, o utilizando una función construida por el desarrollador y especificándola mediante Aggregate, siempre que dicha función cumpla con las restricciones establecidas (ver ejemplo en código 3.9).

```
1 let anArray: int[] = Array.init 5 (fun index -> index+1)
2
3 let plinqReduceExample = anArray.AsParallel().Aggregate(1, fun x y
    -> x * y)
```

Código 3.9: Primitiva Reduce utilizando PLINQ

3.3. Diseño KNN en paralelo usando F#

En esta sección se utilizarán las oportunidades de paralelización identificadas en el análisis de la especificación en FXML previamente realizado y las opciones de paralelización del lenguaje a utilizar para realizar un diseño de la implementación del algoritmo KNN.

No todas las oportunidades son aprovechables puesto que algunas no tienen sentido en el contexto del algoritmo. Se descartaron las oportunidades 6 y 7 debido a que el valor de K seleccionado siempre es más chico que el tamaño de los conjuntos de prueba y de entrenamiento, por consiguiente la cantidad de datos a procesar en dichas oportunidades es poca y la mejora en *performance* no supera el *overhead* en tiempo de ejecución que requiere dicha paralelización.

Por otro lado, algunas opciones de paralelismo provistas por el lenguaje no son realizables para ciertas oportunidades identificadas. Las opciones de paralelismo pertenecientes a la librería TPL fueron descartadas debido a que dichas opciones están enfocadas en la paralelización de tareas, y el algoritmo de KNN, por las características del mismo, esta enfocado en la paralelización de datos.

El diseño realizado para el algoritmo de KNN es el siguiente:

Oportunidad 1 Para implementar esta oportunidad se puede hacer uso de Parallel Loops, ya sea mediante for, foreach o forall paralelo o utilizando el AsParallel de PLINQ. Otra alternativa es utilizar el map paralelo de Array.Parallel.

Oportunidad 2 Para esta oportunidad se puede seguir la misma estrategia mencionada anteriormente, ya sea con Parallel Loops o el map de Array.Parallel.

Oportunidad 3 En esta oportunidad se puede hacer uso de async de Asynchronous Workflows junto con una estructura concurrente. De esta forma se ejecutan las funciones de calculo de distancias de forma asíncrona y sus resultados son guardados en la estructura concurrente, de la cual se obtienen los resultados para ir seleccionando los K más cercanos al dato de prueba. Para la selección de los K menores, se puede utilizar una función reductora de Parallel Aggregation.

Oportunidad 4 Al igual que en las oportunidades 1 y 2 se puede hacer uso de Parallel Loops o el map de Array.Parallel.

Oportunidad 5 En esta oportunidad se puede hacer uso de Parallel Aggregation para realizar la reducción de las distancias.

En el capítulo 5 se estudiarán todas las alternativas planteadas anteriormente en mayor detalle.

4 Pruebas de eficiencia

Antes de comenzar con los capítulos de construcción del algoritmo, abstracción de la soluciones y las exploraciones realizadas, se comentará información de importancia referente a las pruebas de eficiencia realizadas en dichos capítulos.

4.1. Contexto de las pruebas

4.1.1. Ambiente de ejecución

El ambiente de pruebas utilizado para todas las pruebas de eficiencia del presente documento (a excepción que se especifique lo contrario) es el denominado PC1:

PC1 Computadora corriendo Windows 10 Pro en un microprocesador Intel Core i5-3470 de 3.20 Ghz con cuatro *cores*, junto con una memoria RAM de 8 GB.

PC2 Computadora corriendo Windows 7 en un microprocesador Intel Core i7-2600K de 3.40 Ghz con cuatro *cores* con dos *threads* o subprocesos cada uno, junto con una memoria RAM de 8 GB.

El resultado de las pruebas es el promedio de tres ejecuciones de cada prueba.

4.1.2. Problema utilizado

Para realizar las pruebas se utilizó una versión del conjunto de datos del MNIST [21] ("Modified National Institute of Standards and Technology"), ampliamente utilizado por la comunidad de *machine learning*, en formato ".CSV".

El problema consiste en el reconocimiento de dígitos escritos a mano, estos dígitos están definidos por imágenes de 28 píxeles de alto por 28 de ancho, resultando en un conjunto de 784 píxeles con valores enteros de 0 a 255 representando el color.

Número de datos totales de entrenamiento: 42.000. Número de datos totales de test: 28.000.

5 Construcción del algoritmo

Una vez establecida la mejor combinación de primitivas, herramientas y librerías ofrecidas por F# realizada en el diseño del algoritmo, se procedió a la construcción del mismo utilizando el lenguaje mencionado.

5.1. Algoritmo Secuencial

Se comenzó especificando una versión del algoritmo evitando que en su ejecución se realizara cómputo en paralelo. Este algoritmo es idéntico a las versiones paralelas en cuanto a lo funcional, sin embargo su implementación en serie tiene el fin de trazar una línea base con la cual comparar los resultados obtenidos al ejecutar las distintas versiones paralelas del mismo.

5.1.1. Implementación

El algoritmo comenzó siendo un primer acercamiento de solución al problema, pero fue siendo optimizado en la medida que se optimizaba su versión paralela, siempre respetando la premisa de no ejecutar código en paralelo. Estas optimizaciones se fueron realizando para que la versión secuencial estuviera en igualdad de condiciones que su contraparte paralela.

Básicamente se computan todas las distancias, luego se obtienen las menores, y finalmente se vota la clase del dato de prueba, todos estos pasos esperan a que el anterior finalice para comenzar.

5.1.2. Pruebas de eficiencia

Se realizaron pruebas de eficiencia sobre el algoritmo secuencial construido para poder luego comparar el desempeño de las versiones paralelas del algoritmo. Para ver los resultados completos, ver el cuadro 12.7 en el Anexo A (Pruebas de eficiencia).

En la figura 5.1 se observa la eficiencia del algoritmo secuencial para dos valores distintos de K. Se puede ver como al aumentar la cantidad de datos de prueba a clasificar, los tiempos de ejecución aumentan de forma lineal, tal como es de esperarse.

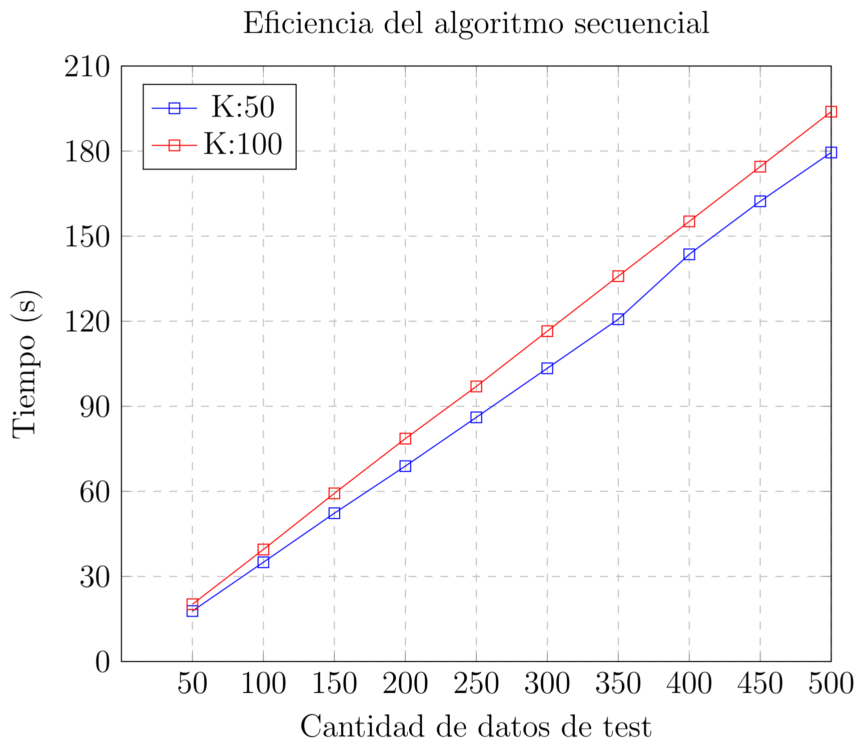

Figura 5.1: Tiempo de ejecución en segundos para el algoritmo secuencial en función de la cantidad de datos de prueba

5.2. Algoritmo Paralelo

En esta sección se detalla la construcción del algoritmo en paralelo haciendo uso de la especificación y diseño realizado en el capitulo anterior. Luego se procede a realizar las pruebas de eficiencia correspondientes, comparándolas con las del algoritmo secuencial mencionado en la sección previa.

5.2.1. Implementación

El funcionamiento de la primera versión de la implementación paralela del algoritmo se explica en el diagrama 5.2.

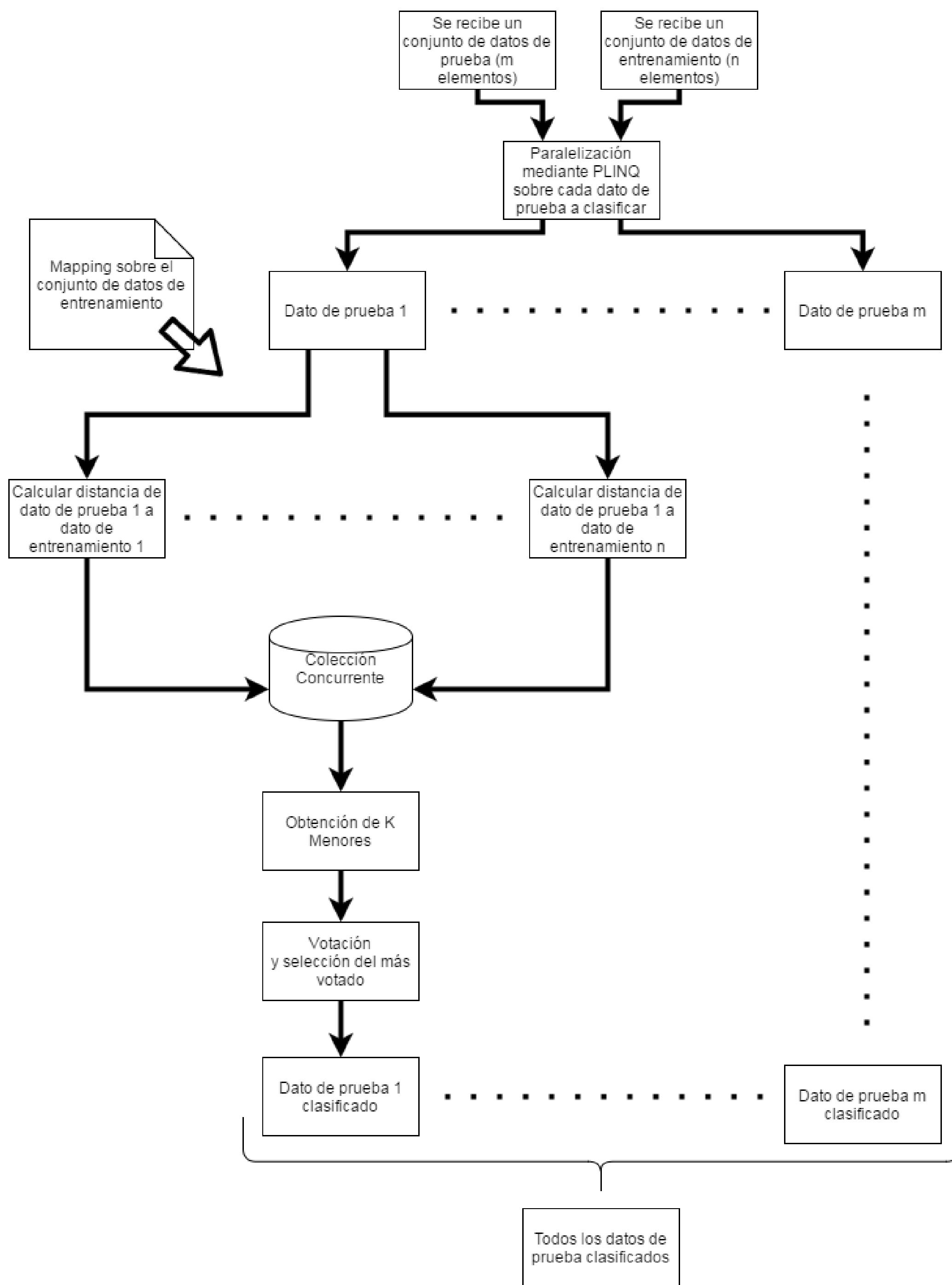

Figura 5.2: Diagrama del algoritmo KNN paralelo

A continuación se explica con mayor detalle el funcionamiento del algoritmo, cuya implementación utiliza algunas opciones comentadas en el diseño del algoritmo de KNN en la sección 3.3.

- La oportunidad 1 se decidió implementar haciendo uso de la ejecución paralela sobre conjunto de datos ofrecida por PLINQ denominada AsParallel, como se muestra en el código 5.1.

```
1   let CalculoKNNParaTodosLosDatosTest(K: int) (cantTest: int) (
        train: DatoTrain[]) =
2       let tests = CargarDatosTest(rutaArchivoPrueba).[0..cantTest]
3       let resultados = query {
4                           for t in tests.AsParallel().AsOrdered() do
5                           select (KNN t K train)
6                        }
7       resultados
```

Código 5.1: CalculoKNNParaTodosLosDatosTest

  Durante la instancia de pruebas se comprobó que esta oportunidad de paralelización puede ser aprovechada en computadoras con gran potencia de cómputo paralelo, o en nodos de cómputo distribuidos. En computadoras con baja cantidad de *cores*, la paralelización mencionada no aporta una ganancia en la eficiencia al algoritmo.

  Como se explica en la documentación de PLINQ [22], y al igual que lo demuestran nuestras pruebas (ver cuadro 12.4 en Anexo A (Pruebas de eficiencia)), PLINQ tiene capacidad de tomar decisiones de paralelización en base al contexto de ejecución en el que se encuentra. Se decidió entonces dejar la responsabilidad de tomar la decisión a PLINQ en tiempo de ejecución.

- Para la oportunidad 2 se utilizó la opción de Parallel Loops, específicamete la primitiva ForAll ofrecida por PLINQ, tal como lo muestra el código 5.2.

- En cuanto a la oportunidad 3 se implementó la única opción identificada, aquella que hace uso de Asynchronous Workflows y estructuras concurrentes, que para este caso se decidió utilizar una Concurrent-Bag. El código se puede ver en los bloques 5.2, 5.3, 5.4 y 5.5.

```
1   let ObtenerDistancias (datoTest: DatoTest) (datosEntrenamiento :
        DatoTrain[]) (concurrentBag: ConcurrentBag<ParProcesado>) =
2       async { datosEntrenamiento.AsParallel().ForAll( fun e ->
            concurrentBag.Add(ProcesarPar datoTest e)) }
```

Código 5.2: ObtenerDistancias

```
1   let KNN (test: DatoTest) (k: int) (trains: DatoTrain[]) =
2       ObtenerDistanciasYKMenores test trains k
3       |> Votar
```

Código 5.3: KNN

```
1   let ObtenerDistanciasYKMenores (datoTest: DatoTest) (
        datosEntrenamiento : DatoTrain[]) (K: int) =
2       let concurrentBag = new ConcurrentBag<ParProcesado>()
3       Async.Start(ObtenerDistancias datoTest datosEntrenamiento
            concurrentBag)
4       let resultado = ObtenerKMenores concurrentBag K (datosEntrenamiento
            .Length)
5       resultado
```

Código 5.4: ObtenerDistanciasYKMenores

```
1   let ObtenerKMenores (cb :ConcurrentBag<ParProcesado>) (K: int)(
        totalElementos: int) =
2       let menores = Array.create K {Etiqueta = -1; Distancia = Int32.
            MaxValue}
3       let mutable cont = totalElementos
4       let mutable item = {Etiqueta = -1; Distancia = -1}
5       while (cont > 0) do
6           if (cb.TryTake(&item)) then
7               cont <- cont-1
8               let mayor = menores.Max()
9               if (TieneMenorDistancia item mayor) then
10                  let posMayor = Array.findIndex (fun x -> x = mayor)
                        menores
11                  menores.[posMayor] <- item
12      menores
```

Código 5.5: ObtenerKMenores

Como se puede apreciar en el código, el funcionamiento de la implementación de la opción de paralelización identificada en la oportunidad 3 está muy ligado con la oportunidad 2. A continuación se explica dicha combinación entre ambas:

Se instancia una Concurrent-Bag (estructura ofrecida por el *framework* .NET para ofrecer concurrencia sobre la colección de datos) y se lanza de forma asincrónica la ejecución del cálculo de distancias entre el dato de prueba en cuestión y el conjunto de datos de entrenamiento. La ejecución de dicho cálculo se hace de forma paralela y en varios hilos de ejecución (para contextos de ejecución que lo permitan) gracias al Forall ofrecido por PLINQ. Mientras se van calculando las distancias, éstas se almacenan en la estructura concurrente, y de forma paralela son recolectadas y procesadas por la función ObtenerKMenores, que fue ejecutada instantáneamente después de la función de cálculo de distancias. Para mayor claridad respecto a lo explicado anteriormente ver diagrama 5.2.

Opciones de implementación no utilizadas

De todas las opciones de paralelización que se diseñaron para las oportunidades identificadas, no todas se utilizaron. Las descartadas se detallan a continuación:

- Oportunidad 1: Se implementaron ambas opciones, pero se terminó por descartar la opción del map paralelo de Array.Parallel debido a la baja *performance* con respecto a la opción que finalmente se utilizó. Ver cuadro 12.2.

- Oportunidad 2: Al igual que en el caso anterior, la eficiencia del algoritmo era mejor cuando se utilizaba la opción de Parallel Loops con PLINQ, en vez de utilizar el map paralelo ofrecido por F#. Ver cuadro 12.1.

- Oportunidad 3: La opción de utilizar el patrón Parallel Aggregation fue implementada mediante Parallel Loops junto con una función reductora externa, y no desde el contexto de la consulta PLINQ, puesto que esto quitaba control sobre el código y la colección de datos intermedia que se va obteniendo como resultado de la reducción.

- Oportunidad 4 y 5: Se realizaron implementaciones con las opciones diseñadas, pero la *performance* del algoritmo empeoró. Se pudo concluir que esto es debido al *overhead* que existe para realizar la paralelización.

Además son operaciones matemáticas sencillas las realizadas para calcular las distancias y eso llevó a que en este caso no existiera beneficio al realizar las operaciones en cuestión de forma paralela. Ver cuadro 12.1.

Durante la construcción se realizaron pruebas de eficiencia para comprobar el rendimiento ofrecido por las opciones de paralelización de las oportunidades 2 y 3. Luego de ver los tiempos de ejecución de cada una de las opciones por separado, se determinó que la mejor optimización que se podía realizar era hacer una combinación de las opciones de ambas, tal como fue detallado anteriormente en esta sección. Para ver los resultados de las pruebas ver el cuadro 12.1 en el Anexo A (Pruebas de eficiencia), en donde se puede apreciar que los tiempos de ejecución de las opciones por separado eran peores que los tiempos obtenidos por la combinación mencionada.

5.2.2. Pruebas de eficiencia

Se realizaron pruebas de eficiencia sobre el algoritmo paralelo construido a fin de comparar el desempeño con la versión secuencial del algoritmo. Para ver los resultados, ver el cuadro 12.8 en el Anexo A (Pruebas de eficiencia).

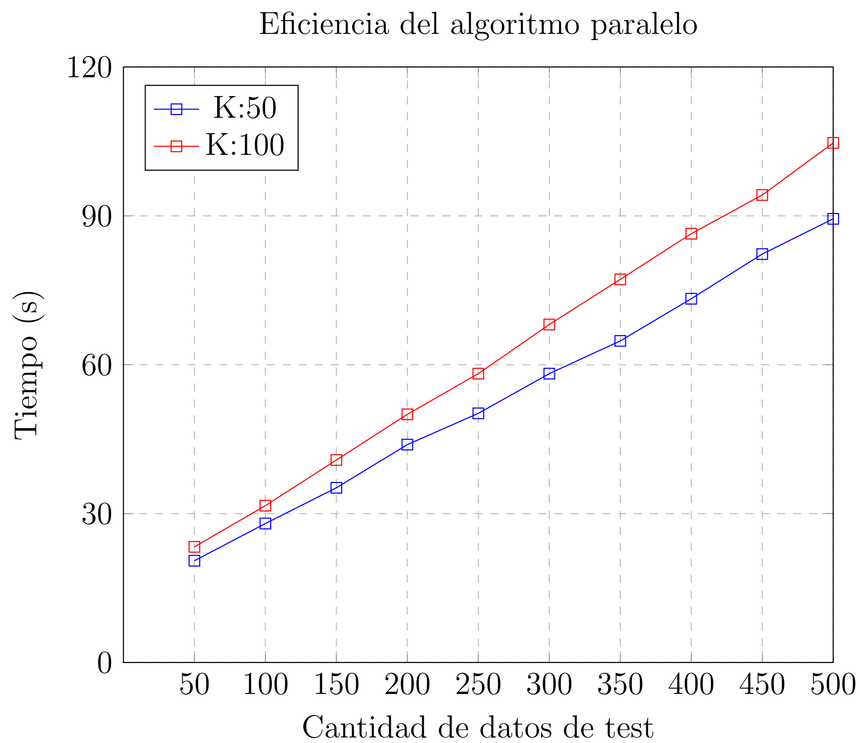

Figura 5.3: Tiempo de ejecución en segundos para el algoritmo paralelo en función de la cantidad de datos de prueba

En la figura 5.3 se observa la eficiencia del algoritmo paralelo para dos valores distintos de K. Se puede ver como al aumentar la cantidad de datos de prueba a ejecutar los tiempos de ejecución aumentan de forma lineal.

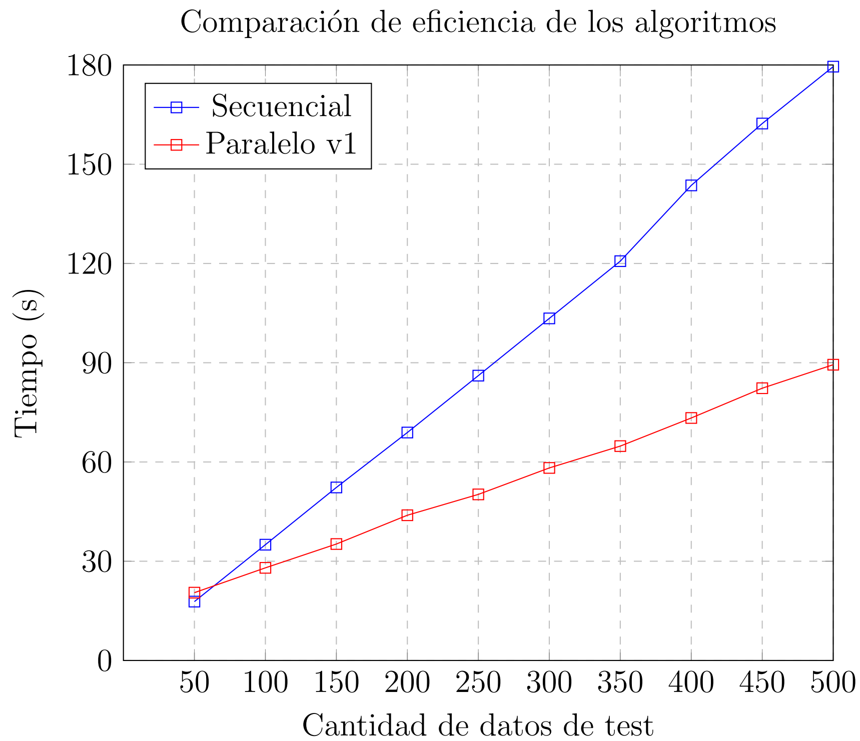

Figura 5.4: Tiempo de ejecución en segundos para los algoritmos realizados en función de la cantidad de datos de prueba. (K=50)

En la figura 5.4 se aprecia la comparación de ambos algoritmos (el paralelo y el secuencial). Se puede observar que los tiempos de ejecución de la versión paralela son inferiores y se alejan cada vez más de su contraparte secuencial a medida que aumenta el tamaño del conjunto de datos a clasificar.

6 Abstracción de la solución

En este capítulo se comentarán las alternativas y opciones escogidas para la realización de una abstracción del algoritmo paralelo desarrollado. El objetivo clave es que el algoritmo sea reutilizable para resolver distintos problemas en los cuales KNN es una solución viable.

6.1. MapReduce

Durante la abstracción del algoritmo se identificó la utilización del modelo de programación *map-reduce*. Se planteó realizar la generalización de este *map-reduce* en forma de una librería para que pudiera ser utilizado en otras implementaciones de soluciones a problemas con características similares al del presente trabajo, además de poder utilizarlo como parte de la abstracción que se estaba buscando.

6.1.1. Construcción

La librería implementa un patrón Productor-Consumidor que funciona como *pipeline* entre una colección de elementos a un resultado esperado. Mediante una función de mapeo, se paraleliza varios productores y aplican dicha función a los elementos de la colección recibida. De forma concurrente, van depositando los elementos resultantes de la función de mapeo en una estructura concurrente. De dicha estructura concurrente, un consumidor toma los elementos y aplicando una función de reducción, en base a los resultados obtenidos hasta el momento y el elemento que seleccionó, decide si es parte de los resultados a devolver, modificando de ser necesario el conjunto de elementos a retornar.

Decisiones tomadas

Se tomaron algunas decisiones al momento de diseñar y construir la librería en cuestión, dichas decisiones son comentadas y justificadas a continuación.

- La estructura intermedia para guardar los elementos procesados por los productores (aquellos que aplican la función de mapeo sobre el conjunto de elementos recibido por parámetro) es una ConcurrentBag ofrecida por .NET. La misma fue seleccionada por sobre otras opciones como por ejemplo ConcurrentQueue, luego de demostrar ser más eficiente para la utilización que se le dió.

- Debido a la característica del consumidor (va realizando la reducción en base a lo reducido hasta el momento), es que éste es único, para evitar *overhead* en la sincronización de la concurrencia sobre el conjunto de resultados que se van a devolver.

- El consumidor obtiene datos de la estructura intermedia, y no puede procesar mientras dicha estructura se encuentre vacía. Existen dos formas de implementar la espera: *busy-waiting* o bloqueo. No hay comentarios para hacer sobre la implementación de *busy-waiting*, simplemente se realizó mediante un *while*, el cual constantemente consulta la estructura. En cuanto a la implementación bloqueante, se realizó haciendo uso de BlockingCollections. Esta clase hace uso de cualquier colección concurrente que implemente la interfaz necesaria y permite bloquear al consumidor cuando no hay elementos para quitar, o bloquear al productor cuando la colección está llena.

  Sorprendentemente el caso más eficiente es el de *busy-waiting*. Se realizaron diversas pruebas con BlockingCollections, utilizando distintas estructuras pero de todas formas, la implementación con *busy-waiting* obtuvo mejores resultados. Para ver los resultados obtenidos, ver el cuadro 12.3 del Anexo A (Pruebas de eficiencia).

- Actualmente la librería admite ejecutarse de cualquiera de las dos formas, se puede especificar de forma paramétrica el modo en el que se va a ejecutar (bloqueante o no bloqueante). Se decidió dejar ambas implementaciones debido a que la mejor eficiencia obtenida haciendo uso de *busy-waiting* puede deberse a las características del problema que se esta resolviendo.

- La librería se realizó en F# como una librería portable, esta puede ser utilizada en programas que estén escritos en cualquier lenguaje soportado por el *framework* .NET.

6.1.2. Pruebas de eficiencia

Se modificó la versión 1 del algoritmo paralelo para que utilice nuestra librería MapReduce, esta nueva versión del algoritmo es referida en este documento como "algoritmo paralelo versión 2".

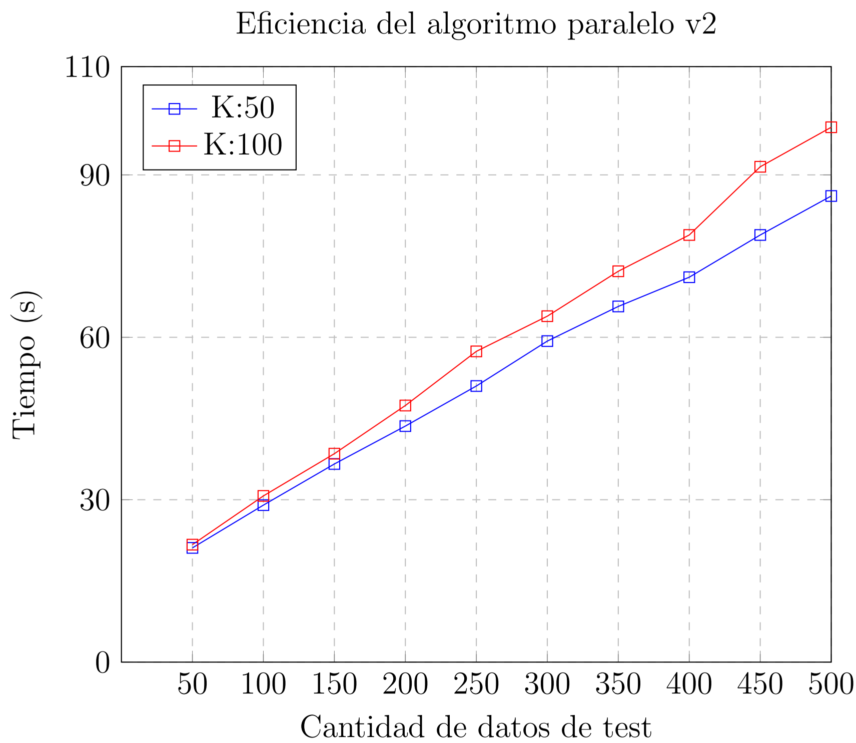

Figura 6.1: Tiempo de ejecución en segundos para el algoritmo paralelo versión 2 en función de la cantidad de datos de prueba

En la figura 6.1 se puede ver los tiempos de ejecución del algoritmo con valores de K iguales a 50 y 100 en función de la cantidad de datos de test clasificados. Para ver la tabla de resultados, ver el cuadro 12.9 en el Anexo A (Pruebas de eficiencia).

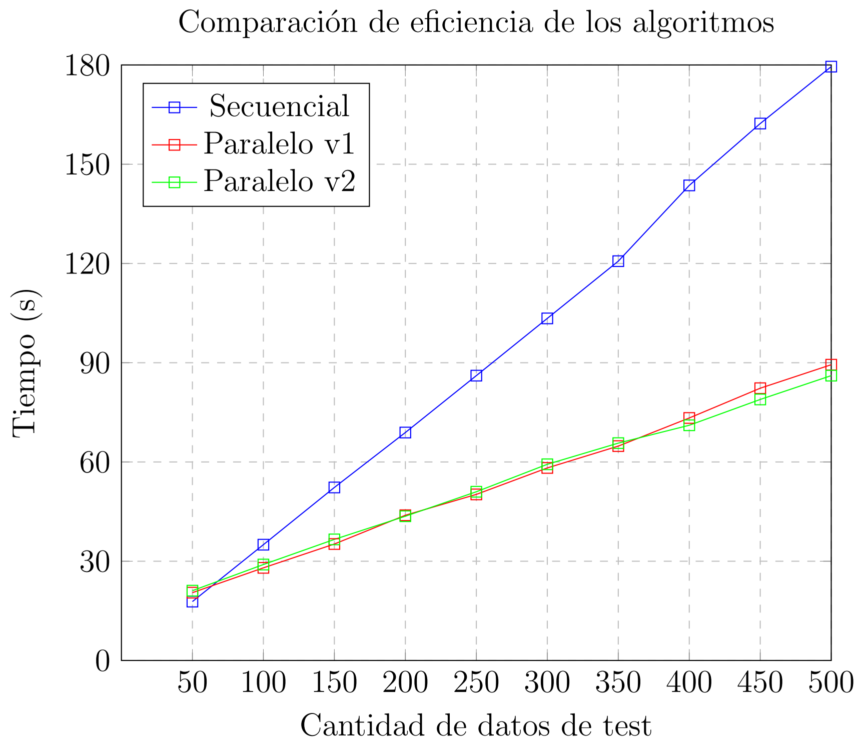

Figura 6.2: Tiempo de ejecución en segundos para los algoritmos realizados en función de la cantidad de datos de prueba (K=50)

En la figura 6.2 se puede observar la comparación de los tres algoritmos. Se puede observar que prácticamente no hay diferencia en los tiempos de ejecución para las versiones 1 y 2 del algoritmo paralelo. Ambas versiones representan una mejora equivalente al algoritmo secuencial.

A pesar de que la llamada a MapReduce agrega un nivel de indirección, pues es una librería externa precompilada, no se observó un detrimento en los tiempos de ejecución lo cual es algo positivo en términos de eficiencia.

6.2. KNN

El segundo paso realizado en busca de la abstracción deseada del algoritmo fue diseñar y construir una librería que implemente el algoritmo de KNN. Esta tarea se realizó buscando dar la misma generalidad que la librería anterior por los mismos motivos. La librería hace uso de la de MapReduce para funcionar.

6.2.1. Construcción

La librería implementa el algoritmo de KNN y permite aplicar dicho algoritmo a un conjunto de datos de prueba, devolviendo el veredicto de clasificación para cada uno de ellos.

Se tuvieron que implementar tipos de datos genéricos para garantizar la mayor abstracción posible, permitiendo así una mayor reutilización en una amplia variedad de situaciones. Es así que se crearon dos tipos de datos que establecen el formato de los datos de entrada al algoritmo:

TestData representa un dato de prueba y esta compuesto por un atributo denominado DistanceAttribute que posibilita el calculo de la distancia entre dicho dato de prueba y el conjunto de entrenamiento.

TrainingData posee el atributo DistanceAttribute y además tiene un atributo denominado Label, el cual sirve para identificar la clase a la cual pertenece el dato de entrenamiento.

6.2.2. Pruebas de eficiencia

Se realizó un *refactoring* de la versión 2 del algoritmo paralelo para mejorar la claridad del código. Esta modificación implicó cambios en funciones y porciones de código, a su vez el código fue traducido por completo al idioma inglés. La nueva versión del algoritmo es referida en el documento como "algoritmo paralelo versión 3".

La versión antes mencionada se ejecutó con el fin de comprobar que no hubiera cambios en los tiempos de ejecución, los resultados pueden verse en el cuadro 12.10 en el Anexo A (Pruebas de eficiencia).

Para generar la librería portable se modificó la versión 3 del algoritmo paralelo, esta nueva versión del algoritmo es referida en el documento como "algoritmo paralelo versión 4".

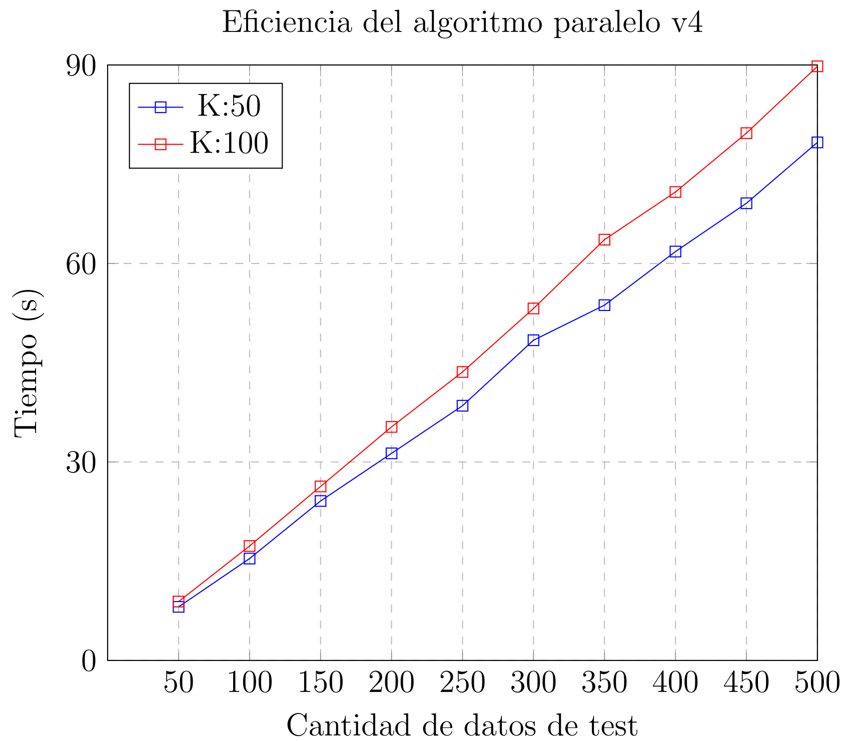

Figura 6.3: Tiempo de ejecución en segundos para el algoritmo paralelo versión 4 en función de la cantidad de datos de prueba

En la figura 6.3 se ven los tiempos de ejecución del algoritmo con valores de K iguales a 50 y 100 en función de la cantidad de datos de test clasificados. Para ver la tabla con los resultados, ver el cuadro 12.11 en el Anexo A (Pruebas de eficiencia).

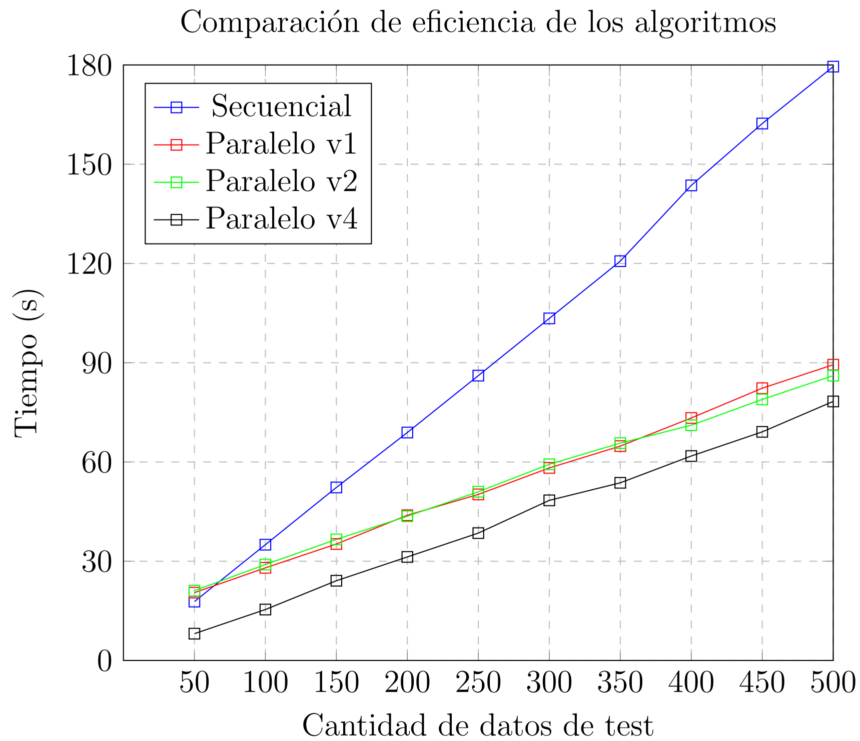

Figura 6.4: Tiempo de ejecución en segundos para los algoritmos realizados en función de la cantidad de datos de prueba (K=50)

En la figura 6.4 se puede observar la comparación de los cuatro algoritmos (secuencial y versiones 1, 2 y 4 del paralelo). Se puede ver que los tiempos de ejecución para la versión 4 representan una mejora a las versiones 1 y 2 de dicho algoritmo. Esta mejora de la *performance* en el algoritmo versión 4 se debe a que es una librería precompilada, a diferencia de los *scripts* que se compilan a medida que el interpretador lo necesita.

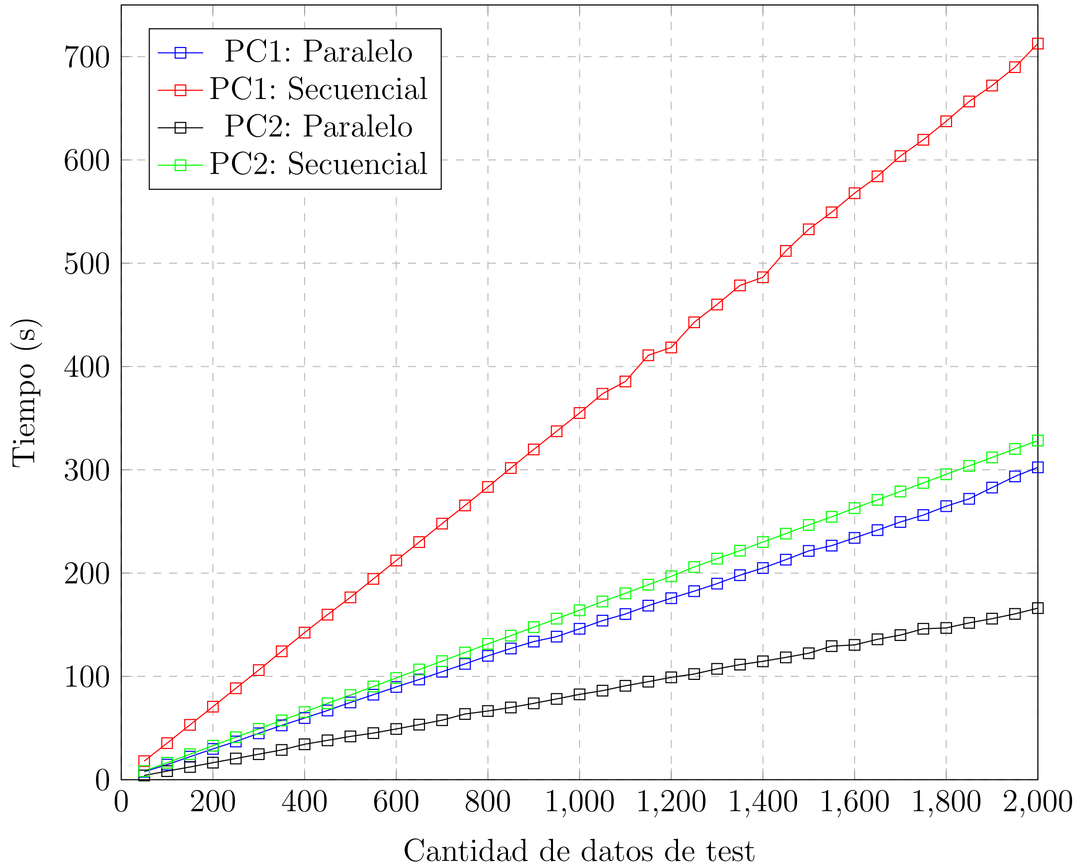

Figura 6.5: Tiempo de ejecución en segundos en función de la cantidad de datos de prueba (K=50)

En la figura 6.5 se puede observar el crecimiento de los tiempos de ejecución del algoritmo con distintas cantidades de datos de prueba. De esta forma se puede ver como se comporta el algoritmo paralelo, comparándolo con el secuencial, cuando se le solicita que procese grandes cantidades de datos. Se puede concluir que los tiempos de ejecución para distintas cantidades de información a procesar se aproximan a una función lineal en ambos algoritmos. Para ver las características de las computadoras utilizadas, visitar el Anexo A (Pruebas de eficiencia) específicamente la sección 4.1.1. Para ver la tabla con los resultados ver los cuadros 12.12, 12.13, 12.14, 12.15.

Resulta interesante también observar el *speedup* obtenido en los distintos contextos de hardware (PC1 y PC2). En la medida que no se cuenta con los porcentajes de código paralelizables y no paralelizables, se calculará el *speedup* general u *overall speedup*.

Como dice la ley de Amdahl [13]:

$$\textit{overall speedup} = \frac{T}{T_{(s)}}$$

Siendo T el tiempo de ejecución del código sin uso de paralelismo (por ejemplo el Algoritmo Secuencial) y $T_{(s)}$ el tiempo de ejecución del código utilizando paralelismo (por ejemplo el algoritmo paralelo versión 4).

Los resultados de *speedup* se obtuvieron de calcular el promedio para las pruebas de la figura 6.5.

- *speedup* PC1: 2.4
- *speedup* PC2: 2.0

La diferencia en los resultados se debe a las características de hardware de las máquinas, la PC2 cuenta con mayor cantidad de procesadores para realizar cómputo paralelo pero sin embargo no logra un *speedup* mucho mayor que PC1, esto se debe probablemente a que la porción de código paralelizable no es tan grande.

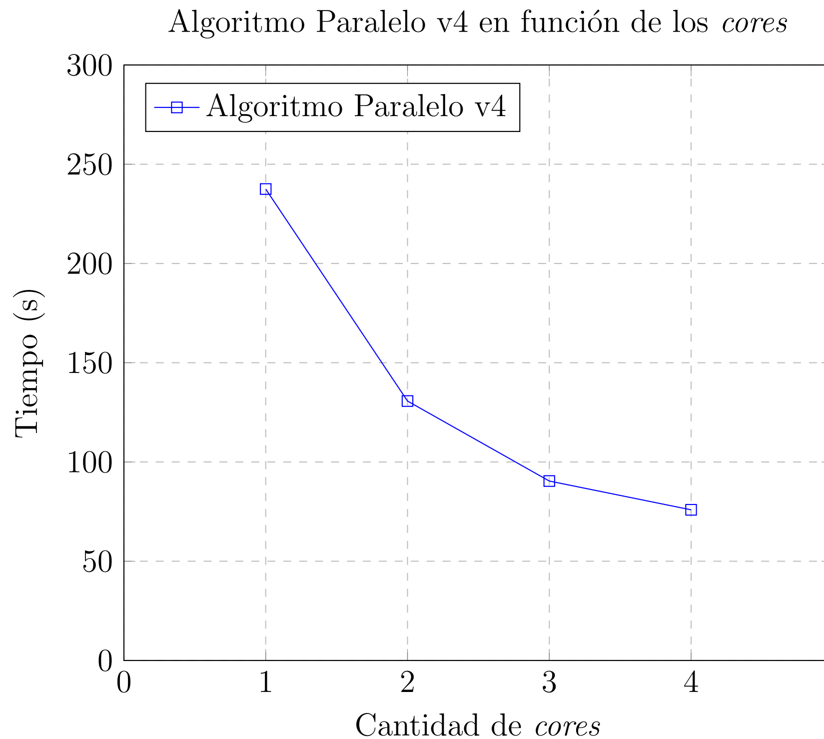

Figura 6.6: Tiempo de ejecución en segundos para los algoritmos realizados en función de la cantidad de *cores* disponibles (K=50, Datos de prueba=500)

En la figura 6.6 se puede observar el tiempo de ejecución del algoritmo paralelo versión 4 en función del número de *cores* disponibles para la ejecución. Para ver la tabla con los resultados, ver el cuadro 12.16 en el Anexo A (Pruebas de eficiencia).

Es notorio como el tiempo de ejecución disminuye a medida que aumentan los *cores* disponibles, el *speedup* apreciable en la gráfica es mucho mayor al que se ve al comparar el algoritmo paralelo versión 4 con el algoritmo secuencial. Esto se debe a que el algoritmo secuencial no tiene el *overhead* que tiene la versión paralela ejecutando en un solo núcleo, puesto que es una versión optimizada para ejecutar de manera secuencial.

7 Exploración del uso de las librerías

El fin de realizar una librería es garantizar un nivel de abstracción y encapsulamiento tal que permita el reuso en situaciones variadas. Con esto en mente y con el objetivo de explorar la integración de otras estrategias de paralelización se exploró la opción de realizar ejecuciones en GPU o de forma distribuida.

7.1. GPU con librería KNN

Dentro de las exploraciones realizadas estuvo la utilización de la GPU como un recurso de hardware distinto del CPU para computar el algoritmo, ofreciendo gran potencia al momento de realizar los cómputos en paralelo. Para ello y con el fin de probar la genericidad de nuestra librería se implementó una función de cálculo de distancias que corría en GPU utilizando la librería Alea GPU [23].

Lamentablemente se vió que este enfoque no es el adecuado para trabajar con la unidad de procesamiento gráfico puesto que los tiempos resultantes al ejecutar el algoritmo fueron muy poco eficientes como se puede observar en la sección 7.1.1. La razón de estos tiempos es el modo que trabaja nuestra librería MapReduce, que es llamada por KNN. El *mapping* implica llamar de manera concurrente varias operaciones de distancia a realizarse a la vez, en este caso en GPU, generando problemas de contención a la vez de demoras por no ser óptimo el manejo de los datos de prueba a clasificar, ya que el dato de prueba se ve replicado en GPU cada vez que se desea calcular la distancia.

Otros autores han logrado buenos resultados al ejecutar KNN en GPU [24], sin embargo estos diseñaron todo su algoritmo para que sea ejecutado en dicha unidad de procesamiento. En nuestro caso hacer eso implicaría diseñar la librería de KNN desde cero, y debido a que esto se aleja del foco del presente trabajo, no se continuó investigando al respecto.

7.1.1. Pruebas de eficiencia

Como parte de la exploración se hicieron pruebas de eficiencia. Estas pruebas específicamente consistieron en la ejecución del cálculo de distancias entre un dato de prueba y un dato de entrenamiento, y luego se fueron incrementando la cantidad de datos de entrenamiento. Se tomaron sus vectores con todas las dimensiones que contienen, y se realizaron el cálculo de la distancia.

El cálculo de la distancia se hizo en CPU y luego se realizó en GPU para poder comparar los resultados. Y como se adelantó anteriormente, los resultados de CPU fueron los mejores en términos de eficiencia, por ende y al ser el cálculo de distancia la única variación probada en el algoritmo, esto resulta en una peor eficiencia al ejecutar el algoritmo entero. Se pueden ver los resultados obtenidos en el cuadro 12.5 en el Anexo A (Pruebas de eficiencia).

7.2. Librería KNN con MapReduce jerárquico

Pensando en la opción de distribuir cómputo en una red de nodos, se desarrolló un prototipo para evaluar la factibilidad de realizar *chunking* del conjunto de entrenamiento. Con este fin se implementó una nueva librería de KNN que hace uso de nuestra abstracción de MapReduce en dos niveles.

El algoritmo se encarga de hacer *chunking* sobre los datos de entrenamiento (obteniendo un conjunto de *chunks*), para luego realizar MapReduce sobre estos. La función a mapear es un nuevo MapReduce (el mismo que realiza KNN), recibiendo el mismo dato de prueba. La diferencia es que se llamará sobre cada uno de los *chunks*, y el reductor de más alto nivel se encargará de tomar los K vecinos más cercanos dentro de cada *chunk*. Ver el diagrama 7.1 para comprender el funcionamiento.

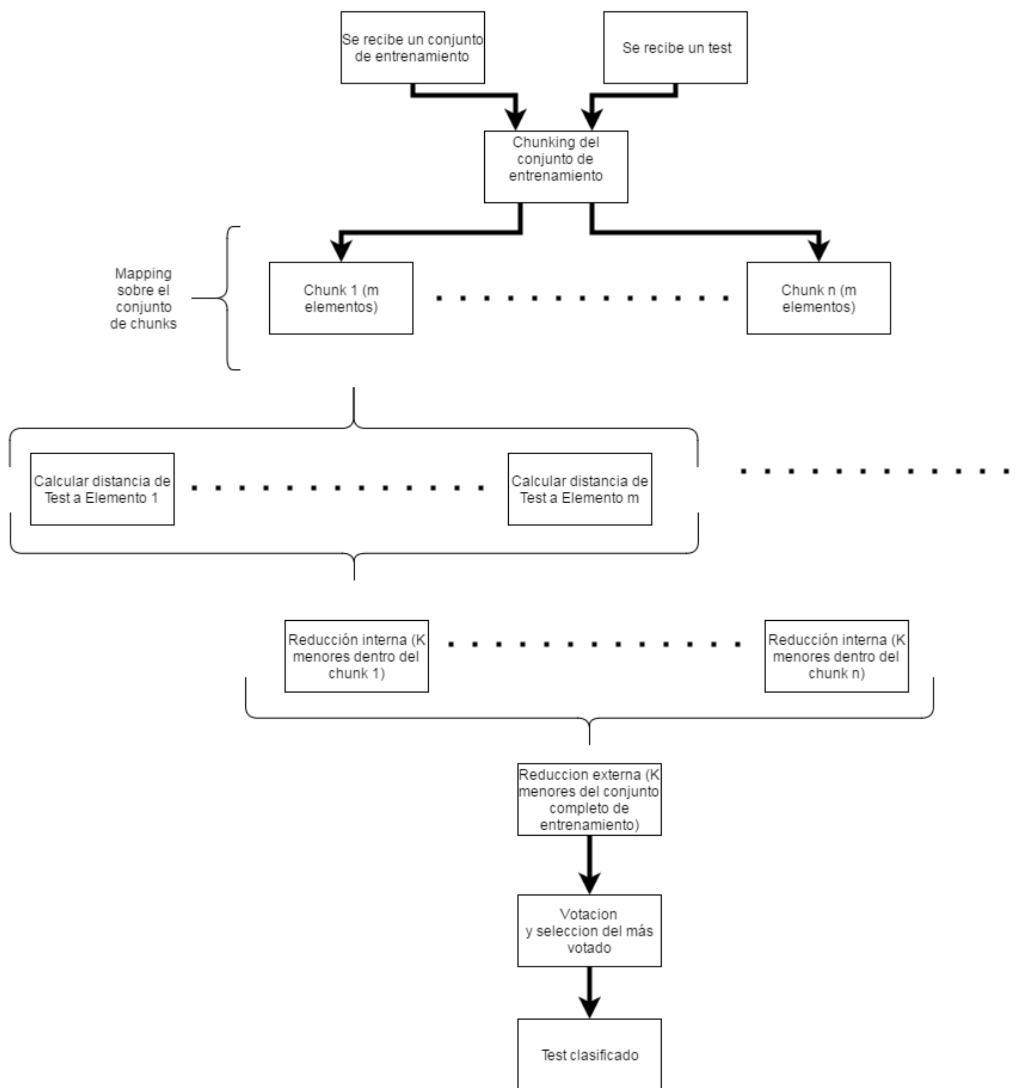

Figura 7.1: Diagrama de MapReduce jerárquico

Este enfoque podría distribuirse, enviando cada *chunk* a un nodo distinto, para luego de finalizar el procesamiento de datos recibir su conjunto reducido y evaluar cuales elementos corresponden al conjunto de los K vecinos más cercanos.

7.2.1. Pruebas de eficiencia

Cabe aclarar que estas pruebas representan una primera aproximación, la librería podría ser modificada para optimizar su rendimiento, existen variables a analizar como lo son el tamaño de los *chunks* (que en estas pruebas es de 10.000 datos de entrenamiento cada *chunk*) y la configuración de la llamada a la librería MapReduce para hacerla bloqueante en el nivel más externo. Para ver la tabla con lo resultados expresados en la gráfica, ver el cuadro 12.6 en Anexo A (Pruebas de eficiencia).

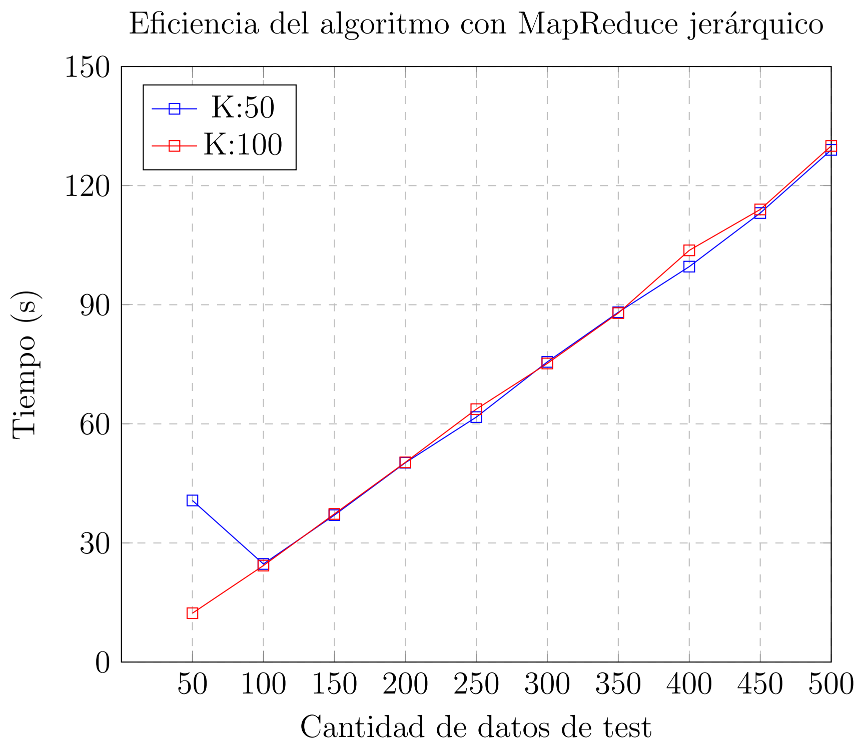

Figura 7.2: Tiempo de ejecución en segundos para el algoritmo paralelo que implementa el MapReduce jerárquico en función de la cantidad de datos de prueba

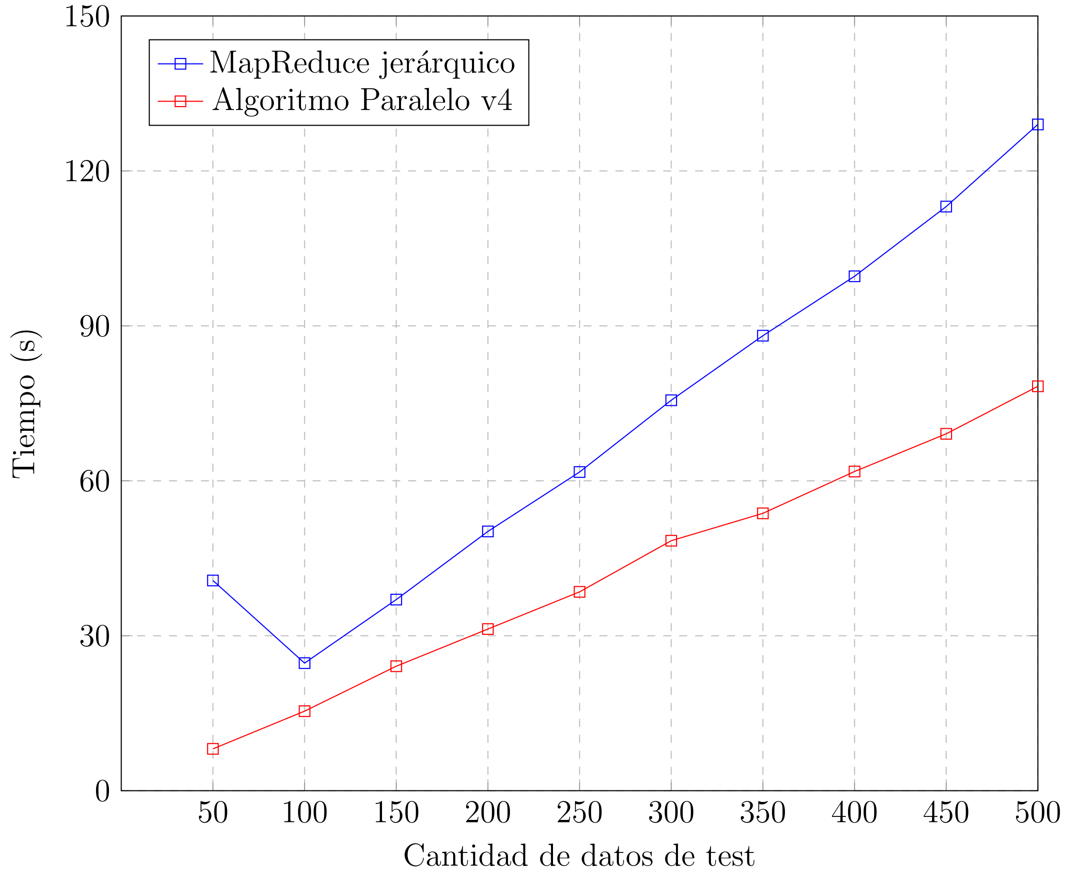

Figura 7.3: Tiempo de ejecución en segundos en función de la cantidad de datos de prueba (K=50)

Como la gráfica 7.3 muestra no existe una mejora de eficiencia en la realización del MapReduce jerárquico. Pero esto puede deberse a que es simplemente una aproximación, y como se adelantaba al comienzo de esta sección, puede existir la necesidad de ajustes en el tamaño de los *chunks* o la utilización de la versión bloqueante para el MapReduce del nivel más externo. De todas formas, si los tiempos de ejecución no pueden mejorarse, se debería evaluar si es una alternativa viable en caso de que se haga una distribución de la carga de cómputos en una red de nodos distribuida, ya que la ganancia de eficiencia en dicho caso puede ser mayor a la pérdida.

8 Pruebas de eficacia

En este capítulo se hará foco en discutir la eficacia de los algoritmos implementados. Si bien el objetivo de este trabajo no es el de mejorar la precisión en la clasificación, es necesario confirmar que no hubo una pérdida de la misma al cambiar las implementaciones. Esto se debe a que el objetivo del trabajo es hacer el algoritmo más eficiente sin ir en contra de la precisión del mismo.

8.1. Contexto de las pruebas

8.1.1. Ambiente de ejecución

El ambiente de pruebas utilizado para todas las pruebas de eficacia del presente documento es el siguiente:

PC3 Computadora corriendo Windows 10 Pro en un microprocesador Intel Core i5-5200U de 2.20 Ghz con dos *cores* de dos *threads* o subprocesos cada uno, junto con una memoria RAM de 6 GB.

8.1.2. Problema utilizado

Para realizar las pruebas se utilizó una versión del conjunto de datos del MNIST [21] ("Modified National Institute of Standards and Technology"), ampliamente utilizado por la comunidad de *machine learning*, en formato ".CSV".

El problema consiste en el reconocimiento de dígitos escritos a mano, estos dígitos están definidos por imágenes de 28 píxeles de alto por 28 de ancho, resultando en un conjunto de 784 píxeles con valores enteros de 0 a 255 representando el color.

Número de datos de entrenamiento: 60,000. Número de datos de test: 10,000.

8.2. Conclusiones de las pruebas

Como se puede observar en Anexo B (Pruebas de eficacia), los porcentajes de clasificaciones correctas para un mismo K a lo largo de las tres versiones de algoritmos comparadas (versión secuencial, y versiones paralelas v4 y MapReduce jerárquico) son muy similares pero no idénticas. Este resultado es interesante, pues algoritmos que clasifican de igual modo deberían arrojar los mismos resultados.

La causa de estas pequeñas diferencias al clasificar esta en el orden en que las distancias se van calculando y evaluando, los algoritmos de reducción no tienen preferencia por una clase concreta, de modo tal que ante la presencia de dos datos de entrenamiento equidistantes al dato a clasificar, el algoritmo se quedará con el dato de entrenamiento que haya llegado primero. En casos donde no haya una clase ampliamente dominante dentro de los K seleccionados, esta variabilidad del algoritmo puede hacer que dos ejecuciones arrojen una clasificación distinta del dato. En otras palabras, inclusive para dos ejecuciones consecutivas del mismo algoritmo pueden resultar en dos clasificaciones distintas para ciertos datos.

De lo anterior se desprende la diferencia de precisiones. Esta diferencia es muy pequeña y no atribuible a un cambio en la funcionalidad o precisión.

9 Conclusiones

Luego de realizada la investigación, especificación, diseño e implementación con todas las pruebas que la acompañaron, se pueden extraer las siguientes conclusiones.

Conclusiones sobre la eficiencia y eficacia:

- Las gráficas comparativas entre el algoritmo secuencial y el paralelo muestran claramente que existe una mejora en los tiempos de ejecución al implementar de forma correcta las oportunidades de paralelización identificadas en la etapa de diseño del algoritmo.
- Al comparar los tiempos de ejecución del algoritmo paralelo en las dos PCs (PC1 y PC2), se ve que aunque se duplique la cantidad de núcleos (en realidad lo que se duplica es la cantidad de subprocesos o "núcleos virtuales") la mejora de *speedup* no es proporcional, esto puede deberse a que el algoritmo presenta gran cantidad de consultas a datos en memoria RAM generando así un cuello de botella, esto queda para investigar a futuro donde se deberán correr pruebas de consumo de memoria RAM. Otra razón es la antes explicada Ley de Amdahl, puesto que el código no debe tener porcentajes muy grandes de secciones paralelizables.
- Existe un equilibrio entre el *overhead* en tiempo de ejecución que agrega la paralelización de un cómputo y el beneficio de *performance* que este trae. Esto se ve de forma clara en el intento de paralelización del cálculo de distancias entre dos datos, tal como se comenta en la sección 5.2.1 que detalla la implementación del algoritmo paralelo.
- Tal como se comprueba en la gráfica 6.5, los tiempos de ejecución del algoritmo paralelo construido y del algoritmo secuencial se aproximan a una función lineal.

- A pesar de que se logró un nivel de abstracción bastante grande, se pudo comprobar que, al igual que lo dice la literatura, para poder implementar un algoritmo como KNN se debe tener en cuenta el problema específico a solucionar, para poder realizar las modificaciones necesarias en busca de una buena eficiencia y eficacia. Por ello es que se deja libertad de implementación mediante parametrización de ciertas secciones del código, como el cálculo de distancia, el mapeo y reducción realizados internamente, etc.
- Al realizar las modificaciones en busca de una mejor eficiencia del algoritmo, no se afecto la eficacia del mismo. Así lo demuestran las pruebas de eficacia realizada sobre las versiones implementadas (ver Anexo B (Pruebas de eficacia)). Existen pequeñas variaciones en los porcentajes de precisión calculados, pero éstos se deben a como ésta implementado el algoritmo, y el orden en que se realizan los cálculos de distancia en cada ejecución, para ver más información leer el capítulo 8.

Sobre las librerías realizadas:

- Al agregar la librería de MapReduce en el algoritmo, para poder aumentar la abstracción de la implementación, se observó que la eficiencia no se afectó a pesar de agregar un nivel de indirección (ver gráfica 6.2). Se puede concluir que esto se debe a que la librería en cuestión esta precompilada y evita tener que realizar la compilación en el momento de ejecución.
- Al igual que lo ocurrido con la librería MapReduce, con la de KNN también se observó que la *performance* no se vió afectada, sino que ocurrió lo contrario (ver la gráfica 6.4). Se observan pequeñas mejoras con respecto a versiones anteriores del algoritmo que no utilizan la librería, esto se atribuye nuevamente a la precomplilación que se da al utilizar la librería, evitando que se tengan que interpretar y compilar los *scripts* en tiempo de ejecución.

Sobre el lenguaje y su ecosistema:

- La herramienta de paralelización que mejores resultados arrojó fue PLINQ. Esto se observó al implementar la paralelización con dicha tecnología y compararla con los resultados de implementar paralelismo con otras opciones.

- Se comprobó parte de la capacidad de toma de decisiones de PLINQ en cuanto a la paralelización de cómputo dependiendo del contexto de ejecución en el que se encuentra el algoritmo ejecutando. PLINQ es capaz de determinar si es conveniente o no la ejecución en paralelo de una operación, dependiendo de las capacidades de la computadora en la que se está ejecutando.

Se concluye que F# permite hacer uso del cómputo en paralelo en busca de la optimización de algoritmos, ofreciendo una amplia gama de opciones de paralelización disponibles para el desarrollador. Específicamente se comprobó para la optimización del algoritmo de KNN, dónde se obtuvo un *speedup* de 2 respecto a la versión secuencial de la implementación en el contexto de pruebas PC1.

El uso de un lenguaje funcional permitió una implementación clara y concisa, en lo que a código respecta, de una solución a un problema relativamente complejo.

En el proceso se crearon dos librerías, una que implementa el algoritmo KNN y otra el patrón *map-reduce*, ambas utilizables en aplicaciones escritas en distintos lenguajes pertenecientes al ecosistema de .NET.

10 Trabajo a futuro

Existen varias vías a seguir como trabajo futuro. Las centrales son el continuar con las exploraciones iniciadas, tanto en el uso de GPU como en la distribución del cómputo.

- Para el uso de GPU será necesario reescribir la librería KNN para que esta sea eficiente al usar dicha unidad de procesamiento, esto implica que se debe de re-analizar la especificación del problema en FXML evaluando los puntos fuertes donde este tipo de unidades de procesamiento resulten de mayor provecho, y en función de esto diseñar e implementar dicha librería.
- En el enfoque de distribución se deben analizar mecanismos de comunicación entre nodos (.NET provee varias alternativas). Un enfoque lógico basado en el primer paso realizado, pensando en la utilización del MapReduce jerárquico, sería buscar que la función de MapReduce de más alto nivel se encargue de enviar los *chunks* a los respectivos nodos.
  Se deberá evaluar el potencial de la librería y optimizarla en dicho contexto de ejecución.
- Otra área de exploración relacionada con las tecnologías emergentes sería lograr que el cómputo se realice en la nube, buscando dar aún más independencia del contexto de ejecución concreto en donde sea invocada la librería.

Antes de continuar por las vías de exploración mencionadas se deben realizar pruebas de consumo de memoria RAM para comprobar si el aparente límite en el *speedup* conseguido al mejorar el contexto de hardware es debido a un cuello de botella en el acceso a datos.

11 Bibliografía

[1] S. Gallagher, "The future is the internet of things—deal with it." [Online]. Available: http://arstechnica.com/unite/2015/10/the-future-is-the-internet-of-things-deal-with-it/

[2] A. Amster, "The role of machine learning in big data." [Online]. Available: https://www.qubole.com/blog/big-data/big-data-machine-learning/

[3] "F# software foundation." [Online]. Available: http://fsharp.org/

[4] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation, First Edition.* Morgan Kaufmann, 2012.

[5] T. G. Mitchell, *Machine Learning.* McGraw-Hill Education, 1997.

[6] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition (Springer Series in Statistics).* Springer, 2011.

[7] S. Marsland, *Machine Learning: An Algorithmic Perspective.* Chapman and Hall/CRC, 2011.

[8] M. Mohri, *Foundations of Machine Learning.* The MIT Press, 2012.

[9] S. Thirumuruganathan, "A detailed introduction to k-nearest neighbor (knn) algorithm." [Online]. Available: https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/

[10] E. Fix and J. L. Hodges, Jr, "Discriminatory analysis - nonparametric discrimination: Consistency properties," Universidad de California, Berkeley, Tech. Rep., 1951.

[11] B. Barney, "Introduction to parallel computing." [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

[12] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing, Second Edition.* Addison Wesley, 2003.

[13] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities." [Online]. Available: http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

[14] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters." [Online]. Available: http://static.googleusercontent.com/media/research.google.com/es//archive/mapreduce-osdi04.pdf

[15] S. Yovine, "Fxml/jahuel: A formal framework for software synthesis."

[16] M. Alexander and W. Gardner, *Process Algebra for Parallel and Distributed Processing.* Chapman and Hall/CRC, 2008.

[17] G. R. Rivadera, "La programación funcional: Un poderoso paradigma," 2008. [Online]. Available: http://www.ucasal.edu.ar/htm/ingenieria/cuadernos/archivos/3-p63-Rivadera.pdf

[18] B. C. Ruiz, F. Gutiérrez, P. Guerrero, and J. E. Gallardo, *Razonando con Haskell: Un curso sobre programación funcional.* Thompson, 2004.

[19] Microsoft, "Visual f#." [Online]. Available: https://msdn.microsoft.com/es-es/library/dd233154.aspx

[20] C. Campbell, R. Johnson, A. Miller, and S. Toub, "Parallel programming with microsoft .net." [Online]. Available: https://msdn.microsoft.com/en-us/library/ff963557.aspx

[21] Y. LeCun, C. Cortes, and C. J. C. Burges, "The mnist database of handwritten digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[22] Microsoft, "Parallel linq (plinq)." [Online]. Available: https://msdn.microsoft.com/es-es/library/dd460688(v=vs.110).aspx

[23] QuantAlea AG, "Alea gpu." [Online]. Available: http://www.quantalea.com/static/app/tutorial/index.html

[24] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu." [Online]. Available: http://nichol.as/papers/Garcia/Fast%20k%20nearest%20neighbor%20search%20using.pdf

[25] Microsoft, "Task parallelism (task parallel library)." [Online]. Available: https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx

[26] ——, "Asynchronous workflows (f#)." [Online]. Available: https://msdn.microsoft.com/en-us/library/dd233250.aspx

[27] ——, "Thread-safe collections." [Online]. Available: https://msdn.microsoft.com/en-us/library/dd997305(v=vs.110).aspx

12 Anexo A: Cuadros de pruebas de eficiencia

12.1. Pruebas Intermedias

Cuadro 12.1: Pruebas de eficiencia de funciones

ID	Función	Parámetros		Tiempo (ms)
		K	Cant. Tests	
1	ObtenerDistancias	-	1	837.02
2			1	172.08
3			1	70.14
4	ObtenerKMenores	50	1	129.36
5		100	1	246.72
6		200	1	582.56
7	ObtenerDistanciasYKMenores	50	1	112.40
8		100	1	157.76
9		200	1	217.36

El cuadro 12.1 muestra los tiempos de ejecución de tres funciones pertenecientes al algoritmo paralelo. Estos tiempos corresponden a distintas implementaciones que se probaron con el fin de seleccionar la más óptima.

Lo probado en ObtenerDistancias fue:

- ID 1: Mapeo sobre el conjunto de entrenamiento mediante uso de consulta de PLINQ, distancia calculada con Array.Parallel.map.
- ID 2: Mapeo sobre el conjunto de entrenamiento mediante uso de consulta de PLINQ, distancia calculada con Array.map.

- ID 3: Mapeo sobre el conjunto de entrenamiento mediante uso de Array.Parallel.map, distancia calculada con Array.map.

Lo probado en ObtenerKMenores (ID 4, 5 y 6) fue recorrer el resultado generado en ObtenerDistancias de manera iterativa del mismo modo que la versión secuencial. No hubo un cambio pero los tiempos sirven de referencia para la prueba siguiente.

Lo probado en ObtenerDistanciasYKMenores (ID 7, 8 y 9) fue ir obteniendo los K menores a medida que las distancias se iban procesando, sin esperar que todos los cómputos estén completos. Esta prueba es la que finalmente se utiliza en el algoritmo y que luego se abstrae en la librería MapReduce. Los tiempos de ejecución son menores que la suma de ObtenerKMenores y ObtenerDistancias con un mismo K.

Cuadro 12.2: Pruebas de eficiencia Algoritmo Paralelo para evaluar alternativas al ejecutar KNN para múltiples datos

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	20	71.4
2		40	146.3
3	50	20	17.0
4		40	34.2

En el cuadro 12.2 se pueden apreciar las pruebas realizadas para evaluar que opción de paralelización utilizar al ejecutar KNN sobre un conjunto de datos de prueba. Las pruebas con ID 1 y 2 corresponden al uso de map paralelo ofrecido por F# y las pruebas con ID 3 y 4 al uso de AsParallel de PLINQ.

Cuadro 12.3: Pruebas de eficiencia Algoritmo Paralelo para evaluar alternativas de colección concurrente

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	100	100	29.6
2		200	43.7
3		300	57.1
4	100	100	41.8
5		200	69.5
6		300	99.6
7	100	100	39.6
8		200	63.2
9		300	90.9

En el cuadro 12.3 se encuentran los resultados de la ejecución del algoritmo paralelo versión 3 con distintas versiones del MapReduce, los detalles de cada prueba se pueden ver a continuación:

- ID 1,2,3: Pruebas con MapReduce que utiliza *busy-waiting*.
- ID 4,5,6: Pruebas con MapReduce que utiliza una implementación bloqueante haciendo uso de BlockingCollections y la colección por defecto: ConcurrentQueue.
- ID 7,8,9: Pruebas con MapReduce que utiliza una implementación bloqueante haciendo uso de BlockingCollections y la colección ConcurrentBag.

Cuadro 12.4: Pruebas de eficiencia Algoritmo Paralelo para evaluar “inteligencia”de PLINQ

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	100	100	29.6
2		200	43.7
3		300	57.1
4		400	73.2
5	100	100	29.3
6		200	43.9
7		300	56.5
8		400	71.3

En el cuadro 12.4 se comprueba la capacidad de PLINQ para reconocer el contexto de ejecución en el que se encuentra y tomar decisiones de paralelismo.

Se obtuvieron resultados similares de eficiencia ya sea para cuando se solicita que PLINQ haga la paralelización como para cuando se realiza el cómputo secuencialmente. Esto se debe a que PLINQ identifica que dado el contexto de ejecución y el consumo de recursos realizado, no es conveniente realizar la paralelización, aunque haya sido solicitado al momento de programar el algoritmo.

A continuación se detallan algunos comentarios sobre las pruebas:

- ID 1,2,3,4: Pruebas realizadas con el algoritmo paralelo versión 3 y el AsParallel de PLINQ en la función CalcularKNNParaDatosPrueba.
- ID 5,6,7,8: Pruebas realizadas con el algoritmo paralelo versión 3 con una versión secuencial de la función CalcularKNNParaDatosPrueba que no utiliza AsParallel.

12.2. Pruebas Exploratorias

Cuadro 12.5: Pruebas de eficiencia distancia en GPU vs distancia en CPU

ID	Parámetros		Tiempo (ms)
	Cant. Tests	Cant. Train	
1	1	1	1.33
2		10	10.42
3		100	89.28
4		1000	840.27
5	1	1	0.13
6		10	0.18
7		100	0.81
8		1000	8.35
9	1	1	1.38
10		10	9.11
11		100	70.46
12		1000	680.10
13	1	1	2.54
14		10	10.72
15		100	93.38
16		1000	813.60

Los resultados de las pruebas del cuadro 12.5, consisten en pruebas del cálculo de distancia entre vectores de dimensiones y se agrupan de la siguiente forma:

- ID 1,2,3,4: Se realizan los cálculos en GPU.
- ID 5,6,7,8: Las operaciones se hacen en la CPU.
- ID 9,10,11,12: Se vuelven a realizar las operaciones en GPU, pero con una configuración de hilos y bloques de ejecución más óptima.
- ID 13,14,15,16: Cálculos en GPU con otra configuración de hilos y bloques de ejecución.

Cuadro 12.6: Pruebas de eficiencia Algoritmo Paralelo con MapReduce Jerárquico

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	17.8
2		100	35.0
3		150	52.3
4		200	68.9
5		250	86.1
6		300	103.4
7		350	120.7
8		400	143.6
9		450	162.3
10		500	179.5
11	100	50	20.2
12		100	39.5
13		150	59.3
14		200	78.6
15		250	97.0
16		300	116.5
17		350	135.9
18		400	155.2
19		450	174.5
20		500	193.9

12.3. Pruebas Completas

Cuadro 12.7: Pruebas de eficiencia Algoritmo Secuencial

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	17.8
2		100	35.0
3		150	52.3
4		200	68.9
5		250	86.1
6		300	103.4
7		350	120.7
8		400	143.6
9		450	162.3
10		500	179.5
11	100	50	20.2
12		100	39.5
13		150	59.3
14		200	78.6
15		250	97.0
16		300	116.5
17		350	135.9
18		400	155.2
19		450	174.5
20		500	193.9

Cuadro 12.8: Pruebas de eficiencia Algoritmo Paralelo v1

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	20.5
2		100	28.0
3		150	35.2
4		200	43.9
5		250	50.2
6		300	58.2
7		350	64.8
8		400	73.3
9		450	82.3
10		500	89.4
11	100	50	23.3
12		100	31.6
13		150	40.8
14		200	50.0
15		250	58.2
16		300	68.1
17		350	77.2
18		400	86.4
19		450	94.2
20		500	104.7

Cuadro 12.9: Pruebas de eficiencia Algoritmo Paralelo v2

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	21.1
2		100	29.0
3		150	36.6
4		200	43.6
5		250	51.0
6		300	59.3
7		350	65.7
8		400	71.1
9		450	78.9
10		500	86.1
11	100	50	21.7
12		100	30.7
13		150	38.5
14		200	47.4
15		250	57.4
16		300	63.9
17		350	72.2
18		400	78.9
19		450	91.5
20		500	98.8

Cuadro 12.10: Pruebas de eficiencia Algoritmo Paralelo v3

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	20.5
2		100	27.6
3		150	34.9
4		200	43.9
5		250	50.1
6		300	57.5
7		350	62.9
8		400	71.0
9		450	80.5
10		500	84.9
11	100	50	22.9
12		100	30.3
13		150	38.6
14		200	46.2
15		250	56.6
16		300	65.2
17		350	73.6
18		400	79.7
19		450	92.2
20		500	105.5

Cuadro 12.11: Pruebas de eficiencia Algoritmo Paralelo v4

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	8.1
2		100	15.4
3		150	24.1
4		200	31.3
5		250	38.5
6		300	48.4
7		350	53.7
8		400	61.8
9		450	69.1
10		500	78.3
11	100	50	8.9
12		100	17.3
13		150	26.3
14		200	35.3
15		250	43.6
16		300	53.2
17		350	63.6
18		400	70.8
19		450	79.7
20		500	89.8

Cuadro 12.12: Pruebas de eficiencia Algoritmo Secuencial en PC1

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	18.0
2		100	35.4
3		150	53.1
4		200	70.7
5		250	88.5
6		300	106.1
7		350	124.3
8		400	142.4
9		450	159.8
10		500	176.5
11		550	194.4
12		600	212.2
13		650	230.0
14		700	247.9
15		750	265.5
16		800	283.4
17		850	301.7
18		900	319.7
19		950	337.3
20		1000	355.0
21		1050	373.6
22		1100	385.5
23		1150	410.9
24		1200	418.4
25		1250	442.8
26		1300	460.0
27		1350	478.5
28		1400	486.4
29		1450	511.9
30		1500	532.8
31		1550	549.3
32		1600	567.8
33		1650	584.1
34		1700	603.7
35		1750	619.5
36		1800	637.4
37		1850	656.6
38		1900	672.0
39		1950	689.8
40		2000	712.6

Cuadro 12.13: Pruebas de eficiencia Algoritmo Paralelo Versión 4 en PC1

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	7.7
2		100	14.6
3		150	22.3
4		200	29.8
5		250	37.0
6		300	44.9
7		350	52.6
8		400	59.7
9		450	67.0
10		500	74.8
11		550	82.3
12		600	89.8
13		650	97.0
14		700	104.5
15		750	112.1
16		800	119.9
17		850	127.0
18		900	133.7
19		950	138.5
20		1000	146.1
21		1050	154.0
22		1100	160.4
23		1150	168.5
24		1200	175.7
25		1250	182.5
26		1300	189.8
27		1350	198.0
28		1400	204.9
29		1450	213.0
30		1500	221.5
31		1550	226.6
32		1600	234.1
33		1650	241.6
34		1700	249.5
35		1750	256.2
36		1800	264.8
37		1850	271.9
38		1900	282.8
39		1950	293.6
40		2000	302.4

Cuadro 12.14: Pruebas de eficiencia Algoritmo Secuencial en PC2

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	8.3
2		100	16.4
3		150	24.7
4		200	32.9
5		250	41.1
6		300	49.2
7		350	57.4
8		400	65.5
9		450	73.9
10		500	82.0
11		550	90.2
12		600	98.5
13		650	106.7
14		700	114.8
15		750	123.1
16		800	131.5
17		850	139.4
18		900	147.7
19		950	155.9
20		1000	164.0
21		1050	172.5
22		1100	180.4
23		1150	188.8
24		1200	196.9
25		1250	205.9
26		1300	214.0
27		1350	221.7
28		1400	230.1
29		1450	238.2
30		1500	246.6
31		1550	254.5
32		1600	262.9
33		1650	270.9
34		1700	279.0
35		1750	287.3
36		1800	295.7
37		1850	303.8
38		1900	312.0
39		1950	320.2
40		2000	328.3

Cuadro 12.15: Pruebas de eficiencia Algoritmo Paralelo Versión 4 en PC2

ID	Parámetros		Tiempo (s)
	K	Cant. Tests	
1	50	50	4.1
2		100	8.3
3		150	12.3
4		200	16.5
5		250	20.5
6		300	24.7
7		350	28.8
8		400	34.3
9		450	38.1
10		500	41.9
11		550	45.1
12		600	49.1
13		650	53.3
14		700	57.6
15		750	63.6
16		800	66.5
17		850	69.9
18		900	73.9
19		950	78.2
20		1000	82.6
21		1050	86.2
22		1100	90.9
23		1150	94.9
24		1200	99.1
25		1250	102.3
26		1300	107.3
27		1350	111.4
28		1400	114.6
29		1450	118.4
30		1500	122.4
31		1550	129.3
32		1600	130.5
33		1650	135.9
34		1700	140.0
35		1750	146.1
36		1800	146.9
37		1850	151.8
38		1900	155.9
39		1950	160.5
40		2000	166.1

Cuadro 12.16: Pruebas de eficiencia Algoritmo Paralelo Versión 4 - cambios en número de *cores*

ID	Parámetros			Tiempo (s)
	K	Cant. Tests	Cant. Cores	
1	50	500	1	237.5
2			2	130.7
3			3	90.4
4			4	75.9

13 Anexo B: Cuadros de pruebas de eficacia

Cuadro 13.1: Pruebas de eficacia Algoritmo Secuencial

K	Precisión (%)
10	96.71
20	96.24
50	95.37
100	94.38

Cuadro 13.2: Pruebas de eficacia Algoritmo Paralelo v4

K	Precisión (%)
10	96.73
20	96.22
50	95.38
100	94.43

Cuadro 13.3: Pruebas de eficacia Algoritmo Paralelo con MapReduce Jerárquico

K	Precisión (%)
10	96.65
20	96.22
50	95.37
100	94.40

14 Anexo C: Guías de uso de las librerías

14.1. MapReduce

14.1.1. Tipos de datos

Genéricos

‘T : Tipo de dato de los elementos que se reciben.

‘U : Tipo de dato de los elementos una vez aplicada la función de mapeo

14.1.2. Funciones

La librería cuenta con una única función homónima.

MapReduce

```
1 MapReduce (data: 'T[]) (mapFunction: 'T -> 'U) (reduceFunction: 'U[]
      -> 'U -> unit) (initialValue: 'U) (numberOfItemsToReturn: int) (
      blocking: bool)
```

Código 14.1: Función MapReduce

La función MapReduce cuenta con los siguientes parámetros:

data Arreglo de tipo genérico $<`T>$ sobre el que se realizará el mapeo.

mapFunction Función a computar sobre los datos pertenecientes al parámetro *data*.

reduceFunction Función de reducción que se computará sobre el arreglo de resultados del mapeo.

initialValue Valor inicial de un elemento de tipo $< `U >$ que utilizará la función de reducción para procesar los datos.

numberOfItemsToReturn Tamaño del arreglo que retornará la función de reducción.

blocking Booleano que define si la librería utilizará acceso bloqueante a los datos mapeados o *busy-waiting* (la ultima opción es más eficiente en el contexto de ejecución de KNN para reconocimiento de dígitos).

14.2. KNN

14.2.1. Tipos de datos

Genéricos

‘L : Tipo de dato de la *label* utilizada para identificar los datos de entrenamiento, y que se espera recibir para cada dato de prueba una vez ejecutado el algoritmo.

‘D : Tipo de dato que representa el atributo que definirá la distancia entre cada elemento.

No Genéricos

$TrainingData <' L,' D >= \{Label :' L; DistanceAttribute :' D\}$

$TestData <' D >= \{DistanceAttribute :' D\}$

14.2.2. Funciones

La librería cuenta con dos funciones KNN y Main.

KNN

```
1 KNN (testData: TestData<'D>) (k: int) (trainingData: TrainingData<'L,'
     D>[]) (distanceFunction: 'D -> 'D -> int)
```

Código 14.2: Función KNN

Función para calcular KNN sobre un dato de prueba. Recibe el dato de prueba (testData), el valor de K, el conjunto de datos de entrenamiento (trainingData) y una función para calcular la distancia entre dos atributos de distancia (distanceFunction).

La función KNN cuenta con los siguientes parámetros:

testData Elemento de tipo genérico $TestData <' D >$ que el algoritmo clasificará.

k Entero que selecciona la cantidad de vecinos más cercanos a tener en cuenta para clasificar el dato.

trainingData Arreglo de tipo genérico $TrainingData <' L,' D >$ que contiene los datos de entrenamiento.

distanceFunction Función que define cómo se calcula la distancia entre el dato a clasificar y un dato de entrenamiento.

Main

```
1 Main (tests: TestData<'D>[]) (trainingData: TrainingData<'L,'D>[]) (K:
      int) (distanceFunction: 'D -> 'D -> int)
```

Código 14.3: Función Main

Función para calcular KNN sobre un conjunto de datos de prueba. Recibe el conjunto de datos de prueba (tests), el conjunto de datos de entrenamiento (trainingData), el valor de K, y una función para calcular la distancia entre dos atributos de distancia.

La función Main cuenta con los siguientes parámetros:

tests Arreglo de tipo genérico $TestData < D >$ que cuenta con el conjunto de datos el algoritmo clasificará.

K Entero que selecciona la cantidad de vecinos más cercanos a tener en cuenta para clasificar los datos.

trainingData Arreglo de tipo genérico $TrainingData < L, D >$ que contiene los datos de entrenamiento.

distanceFunction Función que define cómo se calcula la distancia entre un dato a clasificar y un dato de entrenamiento.