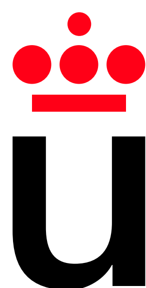


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Matemáticas

Curso 2022-2023

Trabajo Fin de Grado

ESTUDIO COMPARATIVO DE ALGORITMOS DE BÚSQUEDA DE VECINOS CERCANOS EN ESPACIOS MÉTRICOS

Autor: Daniel Calonge Jiménez
Tutora: María Jesús Algar Díaz

Índice de contenidos

1. Introducción **5**
- 1.1. Minería de Datos 5
  - 1.1.1. Aprendizaje supervisado 6
  - 1.1.2. Aprendizaje no supervisado 6
- 1.2. Espacios métricos 6
- 1.3. Tipos de búsqueda en espacios métricos 7
  - 1.3.1. Búsqueda por rango 7
  - 1.3.2. Vecino más cercano 7
  - 1.3.3. K vecinos más cercanos 7

2. Estado del Arte **8**
- 2.1. Búsquedas en espacios métricos y sus aplicaciones 8
  - 2.1.1. Consultas de contenido multimedia 10
  - 2.1.2. Consultas de fragmentos de texto 11
  - 2.1.3. Biología computacional 11
- 2.2. El algoritmo KNN 11
  - 2.2.1. Ejemplos reales de aplicaciones de KNN en sistemas de recomendaciones 14
  - 2.2.2. Ejemplos reales de aplicaciones de KNN en salud 15
  - 2.2.3. Ejemplos reales de aplicaciones de KNN en análisis de datos financieros 15
- 2.3. VP-Tree 17
- 2.4. Balltree 19

3. Estudio comparativo y resultados **23**
- 3.1. Implementación de los algoritmos 23
  - 3.1.1. KNN básico (fuerza bruta) 23
  - 3.1.2. KNN basado en VP-Tree 24
  - 3.1.3. KNN basado en Balltree 25
- 3.2. Conjuntos de datos de prueba 25
  - 3.2.1. Seis nubes de puntos sin solape 26
  - 3.2.2. Seis nubes de puntos con solape 27
  - 3.2.3. Base de datos GloVe 28

3.2.4. Base de datos MNIST 29
3.3. Criterios para la comparación de los algoritmos 30
3.3.1. Eficiencia computacional 31
3.3.2. Métricas de evaluación 31
3.4. Resultados de los test 32
3.4.1. Pruebas en las seis nubes de puntos sin solape 33
3.4.2. Resultados en las seis nubes de puntos con solape 35
3.4.3. Resultados en GloVe 37
3.4.4. Resultados en MNIST 38

4. Conclusiones **42**

5. Trabajos futuros **44**

Bibliografía **46**

Apéndices **49**

A. Códigos **51**
A.1. Seis nubes sin solape 51
A.1.1. Generar conjunto y manipulación de los datos 51
A.1.2. Visualización de los datos 52
A.1.3. KNN por fuerza bruta 52
A.1.4. KNN con VP-Tree 53
A.1.5. KNN con Balltree 54
A.1.6. Gráfica de tiempos 54
A.2. Seis nubes con solape 54
A.2.1. Generar conjunto y manipulación de los datos 54
A.2.2. Visualización de los datos 55
A.2.3. KNN por fuerza bruta 56
A.2.4. KNN con VP-Tree 56
A.2.5. KNN con Balltree 57
A.2.6. Gráfica de tiempos 57
A.2.7. Gráfica del accuracy 58
A.3. GloVe 58
A.3.1. Cargar y manipular datos 58
A.3.2. KNN por fuerza bruta 59
A.3.3. KNN con VP-Tree 59
A.3.4. KNN con Balltree 59
A.3.5. Gráfica de tiempos 60
A.4. MNIST 60
A.4.1. Cargamos los datos, manipulamos los datos y hacemos comprobaciones 60

A.4.2. Visualización de los datos 61
A.4.3. KNN por fuerza bruta 62
A.4.4. KNN con VP-Tree 62
A.4.5. KNN con Balltree 62
A.4.6. Gráfica de tiempos 63
A.4.7. Gráfica del accuracy 63

1
Introducción

En este proyecto vamos a analizar una serie de algoritmos de búsqueda de vecinos más cercanos con el objetivo de llevar a cabo un estudio comparativo entre ellos, por lo que el objeto de este trabajo se enmarca dentro de la *Minería de Datos*.

1.1. Minería de Datos

Debido a los recientes avances en informática, telecomunicaciones y, sobre todo, la aparición y democratización de internet, se están generando y almacenando una gran cantidad de datos digitales. La ciencia de datos es el área del conocimiento encargada de extraer información útil y tendencias de forma automática (o semiautomática) de estas bases de datos con mucha información. La ciencia de datos se aplica prácticamente en cualquier sector e industria que estén digitalizados actualmente, ya que su correcto uso da una ventaja competitiva enorme y, por tanto, hay un incentivo económico en aplicar y desarrollar este campo de la informática. En la minería de datos se aplican otras áreas del conocimiento como la inteligencia artificial, estadística, aprendizaje automático y sistemas de bases de datos.

En la minería de datos se aplican muchas técnicas de clasificación (árboles de decisión, k-vecinos más cercanos, redes neuronales), técnicas de clustering (k-medias). Para a la aplicación exitosa de la minería de datos, se requiere un preprocesamiento de datos (reducción de dimensionalidad, limpieza de ruido, eliminación de valores atípicos), pos-procesamiento (como de bien se entiende, resumen, presentación) y un conocimiento adecuado del problema en cuestión [24] [20].

1.1.1. Aprendizaje supervisado

En el aprendizaje supervisado, se obtiene una función que permite hacer predicciones sobre cualquier elemento de entrada. Esta función, se obtiene a partir de unos datos de entrenamiento. Es el tipo de aprendizaje más común en problemas de clasificación, regresión y clasificación. Un ejemplo de aprendizaje supervisado es el algoritmo KNN [14].

1.1.2. Aprendizaje no supervisado

Se diferencia del aprendizaje supervisado en que no tenemos datos de entrenamiento. En el aprendizaje no supervisado, se buscan patrones para hacer predicciones sobre los nuevos puntos. Un ejemplo del aprendizaje no supervisado es el clustering, en el que se clasifican en ”grupos”los nuevos datos en base a una serie de variables [14].

1.2. Espacios métricos

En los algoritmos de búsqueda de vecinos más cercanos es de vital importancia el concepto matemático de *Espacio métrico*. Podemos decir que un espacio métrico es un conjunto que tiene una función de distancia asociada. Para entender que es un espacio métrico, bastaría con dar su definición, pues es un concepto fácil de entender.

Un espacio métrico, es el par ordenado (M,d), donde M es un conjunto (normalmente llamado base de datos) y d es una función tal que $d : M \times M \rightarrow \mathbb{R}$ [5].

Que además cumple las siguientes propiedades:

- $\forall x \in M, d(x, x) = 0$
- $\forall x, y \in M$, si $x \neq y, d(x, y) > 0$
- $\forall x, y \in M,\ d(x, y) = d(y, x)$
- $\forall x, y, z \in M, d(x, z) \leq d(x, y) + d(y, z)$

Esta función nos devuelve la *proximidad* entre objetos, a menor valor más similares son. La familia de funciones de distancia más importante son las *Distancias de Minkowski*, que esta definida de la siguiente forma [1]:

$$L_p[(x_1, ..., x_n), (y_1, ..., y_n)] = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|} \tag{1.1}$$

Con el valor $p = 2$ en la ecuacion 1.1, tenemos la distancia euclídea.

1.3. Tipos de búsqueda en espacios métricos

La búsqueda en espacios métricos se usa para obtener la información de manera eficiente de una base de datos, pero en estas enormes bases de datos obtener exactamente lo que se busca es imposible. Por tanto, para realizar una búsqueda, se buscaran los objetos más 'próximos' usando una función de distancia. En esta sección vamos a explicar de manera muy breve los tipos existentes de búsqueda en espacios métricos.

1.3.1. Búsqueda por rango

La búsqueda por rango es el tipo de búsqueda más útil y por tanto el más común. Para esta búsqueda es necesario especificar un objeto q y un radio de búsqueda r y se nos devuelven todos los objetos que estén a una distancia menor o igual que r de q. Para este tipo de búsqueda existen varias técnicas y algoritmos que veremos más tarde [16].

1.3.2. Vecino más cercano

Dado un objeto q, este tipo de búsqueda encuentra el objeto más cercano a q. En la actualidad, los métodos para las búsquedas de vecino más cercano se basan en técnicas de búsqueda por rango [16].

1.3.3. K vecinos más cercanos

La búsqueda de los k vecinos más cercanos es una generalización de la búsqueda del vecino más cercano. Dado un objeto q y un número natural k, este tipo de búsqueda nos devuelve los k objetos más cercanos al objeto q. Este tipo de búsqueda es más fácil de entender que la búsqueda por rango, pues no se necesita saber como funciona la función de distancia, basta con saber el numero de objetos similares a buscar [16].

2

Estado del Arte

En la actualidad, para el problema de la búsqueda del vecino más cercano se utilizan una serie de algoritmos, siendo el principal el algoritmo KNN que se puede usar tanto para clasificación como para regresión. También hay una serie de familias de algoritmos útiles como los bounding sphere methods, los de indexación basados en distancia y los métodos de indexación multidimensional. En este trabajo nos vamos a centrar en los algoritmos de KNN, VP-Tree (algoritmo de indexación basados en distancia) y Balltree (bounding sphere method).

2.1. Búsquedas en espacios métricos y sus aplicaciones

Tradicionalmente las búsquedas se han realizado en "datos estructurados", en los que bastaría con buscar directamente el término exacto que deseamos, dado que todo dato archivado tiene una etiqueta única que lo identifica, pero esto no es asi en las bases de datos no estructuradas.

Con la rápida evolución en las TIC (Tecnologías de la Información y las Comunicaciones) en los últimos años, ha surgido la necesidad de usar bases de datos no estructuradas, en las que a pesar de que los objetos almacenados no tengan orden, pueden ser buscados y categorizados mediante distintos algoritmos de búsqueda en espacios métricos. Este tipo de bases de datos son útiles para almacenar cadenas de texto, imágenes, audio y vídeos, pues no pueden almacenarse en las bases de datos estructuradas tradicionales.

Para realizar una búsqueda en una base de datos no estructurada se usa el con-

cepto de proximidad en un espacio métrico, es decir, dada una consulta, buscamos los objetos más próximos a esta según una función de distancia que cumple todas las propiedades vistas previamente. Para este problema de búsqueda en espacios métricos han aparecido soluciones en diferentes áreas del conocimiento como por ejemplo en estadística, informática geometría funcional, inteligencia artificial, bases de datos, biología computacional y minería de datos [16].

Dado que todo espacio métrico es un espacio vectorial, podemos calcular las distancias entre los objetos del espacio métrico. El problema es que calcular las distancias entre todos los objetos de la base de datos es muy costoso computacionalmente, por lo que el objetivo actual es reducir el numero de veces que se calcula la distancia entre dos objetos y reducir el tiempo de ejecución del algoritmo.

La primera solución general a este problema, para funciones de distancia discretas, fue dada por Burkhard and Keller [1973] donde proponen una estructura de árbol llamado BKT (Burkhard Keller Tree). Sea $\mathbb{U}$ nuestro espacio métrico, elegimos un $p \in \mathbb{U}$ arbitrario como raíz de nuestro árbol. Ahora, para cada distancia $i > 0$(hay un numero finito pues la función de distancia es discreta) construimos $\mathbb{U}_i = \{u \in \mathbb{U} \mid d(u,p) = i\}$, osea el conjunto de todos los punto a distancia i de p. Para cada $\mathbb{U}_i$ no vacio definimos un nuevo $p \in \mathbb{U}_i$ y construimos recursivamente otro árbol para $\mathbb{U}_i$. Podemos repetir este proceso hasta donde deseemos o hasta que en cada hoja del árbol haya un solo elemento. La figura 2.1 es un ejemplo extraído de [16] donde se elige u_{11} como pivote.

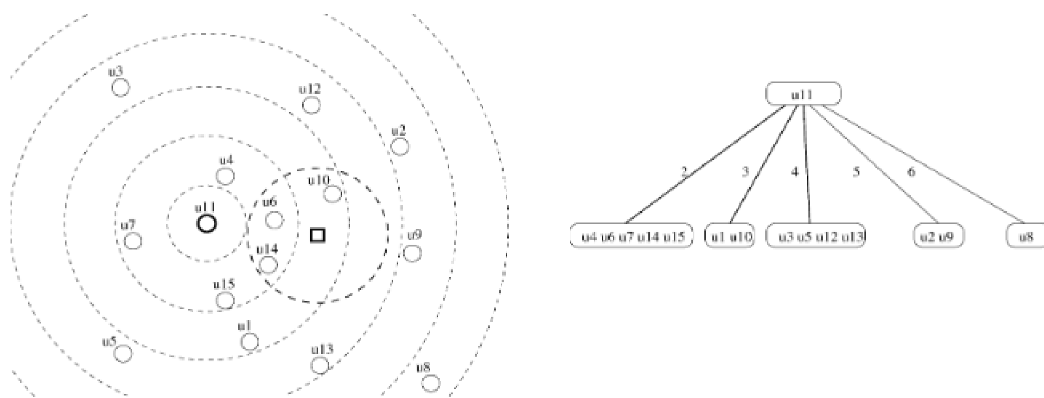


Figura 2.1: Ejemplo para aplicar BKT

Sobre la idea del BKT, se desarrollan otros algoritmos de búsqueda en espacios métricos (como por ejemplo, el FQT, FQHT, FQA), tanto para funciones de distancia discretas como continuas.

El FQT (*Fixed Query Tree*) es una solución de búsqueda en espacios métricos propuesta en [3]. Un FQT es básicamente un BKT donde todas las ramas

del mismo nivel, tienen el mismo pivote (por supuesto que no necesariamente pertenecen al conjunto guardado en el subárbol). Los elementos restantes están guardados en las hojas. La ventaja de tal construcción es que las comparaciones entre la consulta y los nodos (pivotes) solo se necesitan una vez y estas ayudan en el recorrido del árbol.

En [3] los autores proponen una variante llamada FHQT (*Fixed Height FQT*). Un FHQT es un FQT en el que todas las hojas están a la misma profundidad

El FQA (*Fixed Queries Array*) fue presentado en [4]. Aunque esta estructura no es un árbol, no es más que una representación compacta del FHQT. Dado un FHQT de una altura h, Si recorremos las hojas del árbol de izquierda a derecha y ponemos los elementos en un array, obtenemos un FQA [16].

La figura 2.2 es un ejemplo de BKT, FQT, FHQT y FQA extraido de [16]

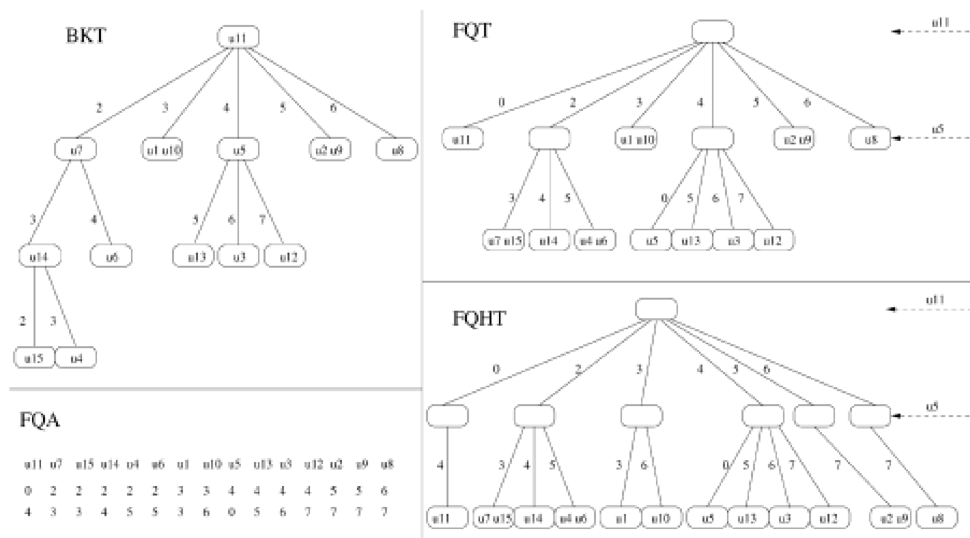


Figura 2.2: Ejemplo de BKT, FQT, FHQT y FQA para el conjuno descrito en la figura 2.1

2.1.1. Consultas de contenido multimedia

El contenido multimedia (imágenes, vídeos y audio) no puede ser ordenado de forma habitual, por lo que es almacenado en bases de datos no estructuradas. Dado que buscar un fragmento exacto de audio, de un vídeo o de una imagen no tiene sentido, para realizar consultas se usa la búsqueda por proximidad [16].

Para ello, se desarrolla una función de distancia que tiene que ser diseñada explícitamente para un formato multimedia concreto por un experto en el área. Normalmente, se suelen extraer un numero k de características de cada objeto y a

este objeto se le asigna un punto en un espacio vectorial de k dimensiones. En [8] podemos encontrar un ejemplo. Concretamente en este trabajo se desarrolla un sistema que combina datos imágenes y de texto en un solo vector con el fin de hacer búsquedas en una base de datoos de imagenes de la WWW(*World Wide Web*).

2.1.2. Consultas de fragmentos de texto

La búsqueda de una determinada cadena de texto dentro de un documento, presenta los mismos problemas que la búsqueda de contenido multimedia, pues buscar de manera exacta un determinado fragmento de texto dentro de un documento suele ser poco efectivo. Esto se debe a el uso de sinónimos o que distintas cadenas de texto pueden tener el mismo significado pese a ser distintas tanto en su tamaño como en su contenido [16].

Recientemente, una serie de investigadores [Salton y McGill 1983; Frakes y Baeza-Yates 1992; Baeza-Yates y Ribeiro-Neto 1999] han desarrollado una serie de soluciones a este problema. Se ha logrado que dada una consulta (una cadena de texto o un documento), se nos devuelve un documento que contenga una cadena de texto similar a la consulta [16].

2.1.3. Biología computacional

El ácido desoxirribonucleico,abreviado como ADN, es un ácido nucleico que contiene las instrucciones genéticas usadas en el desarrollo y funcionamiento de todos los organismos vivos. La estructura química del ADN es de una cadena de nucleótidos. Cada nucleótido está formado por un azúcar (la desoxirribosa), un grupo fosfato y una base nitrogenada (puede ser adenina, timina, citosina o guanina). Dado que lo único que distingue un nucleótido de otro es su base nitrogenada, una secuencia de ADN se puede especificar solo nombrando su base nitrogenada (hay 4 distintas). La disposición secuencial de estas bases a lo largo de la cadena es lo que codifica la información genética [11].

Una coincidencia exacta dentro de una cadena de ADN con otra.Por lo que podemos hacer una búsqueda de una cadena similar codificándola como una cadena de texto y usando las técnicas consultas de fragmentos de texto que antes se han mencionado.[16]

2.2. El algoritmo KNN

El algoritmo de búsqueda de vecinos más cercanos, también conocido como KNN, es un clasificador de aprendizaje supervisado que usa la proximidad, por tanto utiliza una función de distancia en un espacio métrico, para clasificar o hacer predicciones sobre uno o varios puntos de datos determinados. Aunque se

puede utilizar para problemas de regresión o clasificación, se suele utilizar como algoritmo de clasificación, dado que se pueden encontrar puntos similares muy próximos entre sí. Es utilizado en el campo de la inteligencia artificial y el aprendizaje automático.

En este algoritmo, se utiliza un conjunto de datos de entrenamiento para clasificar nuevos datos en una o más categorías. Para esto se mira la categoría de los k datos más cercanos y se coge la categoría mayoritaria, en caso de que k sea par haya empate se elige l categoría del vecino más cercano. (en caso de aplicar la regla de la mayoría). Por ejemplo, si queremos clasificar un nuevo dato en la categoría A o en la categoría B, si cerca de este nuevo dato hay más datos de la categoría A que de categoría B, se clasificaría este nuevo dato como categoría A. Si estamos delante de un problema de regresión, podemos usar KNN tomando la media de los k vecinos más cercanos a este nuevo dato para así hacer una predicción.

A parte de la regla de la mayoría, existen otras reglas de clasificación como la regla del consenso y la regla aleatoria. Para la regla del consenso, al nuevo elemento solo se le asignara una categoría si todos sus k vecinos más cercanos pertenecen a esta categoría. En la regla aleatoria es igual que la regla mayoritaria para k impar, pero si k es par y hay empate, se selecciona una categoría aleatoria para desempatar.

Para ver como funciona, vamos a ver un sencillo ejemplo. Dados la altura, el peso y el sexo de cinco individuos, nos dan la altura y el peso de un sexto, pero no su sexo. Para clasificarlo, bastaría con ver que individuos tiene más cerca en una tabla que relacione la altura con el peso (o viceversa), como se puede ver en la Figura 2.3:

En este caso, los individuos que son hombres están marcados en rojo, las mujeres en azul y el nuevo dato (del que tenemos la altura y el peso, pero no el sexo) está en negro. Si tomamos k=3, cerca de este nuevo individuo hay dos hombres y una mujer, luego clasificamos a el nuevo individuo como hombre. Y si tomamos k=1, cerca de este nuevo individuo hay un hombre, luego clasificamos a el nuevo individuo como hombre[6].

Los algoritmos de búsqueda de vecinos más cercanos es una técnica simple pero efectiva para clasificar datos y es adecuado para problemas con un pequeño número de características o variables explicativas.

Si bien se suelen emplear para clasificar nuevos datos, estos algoritmos tienen otras importantes aplicaciones:

- **Sistemas de recomendaciones**: El algoritmo KNN se puede utilizar para identificar elementos similares en función de los hábitos y preferencias de los usuarios. Por ejemplo, si un usuario ha visto muchas series de misterio, un sistema de recomendación basado en KNN puede recomendarle otras

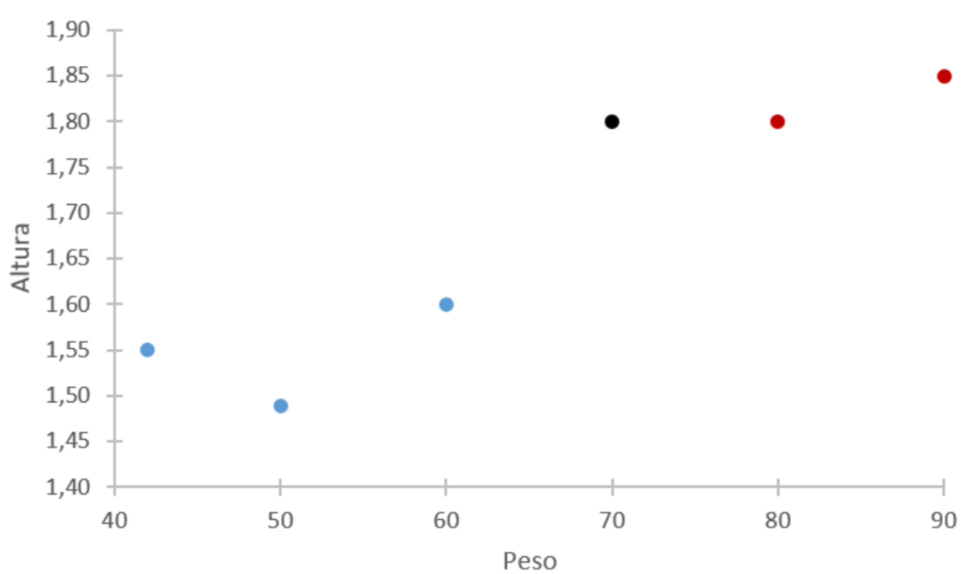


Figura 2.3: Ejemplo de aplicación del algoritmo KNN a un conjunto de 5 individuos.

series de misterio parecidas.

- **Análisis de imágenes:** Estos algoritmos se pueden utilizar para identificar objetos y patrones en imágenes. Por ejemplo, se puede entrenar un modelo de KNN para identificar árboles en una imagen en función de sus características, como el color, la forma y el tamaño.

- **Procesamiento del lenguaje natural:** También puede ser utilizado en el procesamiento del lenguaje natural para clasificar texto en diferentes categorías. Por ejemplo, se puede entrenar un modelo de KNN para clasificar correos electrónicos como spam o no spam en función de su contenido y características.

- **Análisis de datos financieros:** El algoritmo se puede utilizar para predecir la evolución de las acciones. Por ejemplo, se puede entrenar un modelo de KNN para predecir si una acción aumentará o disminuirá su valor en función de las características de la empresa y del estado actual del mercado o también se podría utilizar para valorar el riesgo que supone otorgar un préstamo a una determinada empresa o individuo.

- **Salud:** Este algoritmo también ha sido usado exitosamente en el área de la salud para predecir el riesgo de tener ataques al corazón o de tener distintos tipos de cáncer.

- **Reconocimiento de patrones:** Los algoritmos de búsqueda de vecinos más cercanos también se pueden utilizar para identificar números y letras escritos a mano.

Las principales ventajas del algoritmo KNN son que es muy fácil de implementar, que se adapta fácilmente a nuevos datos y que solo se necesita introducir el valor de la k y la métrica de distancia para que este funcione. La principales desventajas son que no funciona bien con datos de dimensiones altas y que tiende a sobreajustar [1].

2.2.1. Ejemplos reales de aplicaciones de KNN en sistemas de recomendaciones

Con el auge de internet en los últimos años, han aparecido una serie de servicios ofreciendo productos, contenido multimedia, etc. de forma personalizada a cada usuario. Para ello, se hace uso de sistemas de recomendación, que pueden ser implementados usando el algoritmo KNN. Vamos a ver un ejemplo de como KNN se puede usar para recomendar películas y series de televisión a usuarios [2]. Para ello el autor tiene en cuenta las preferencias de usuarios similares al usuario objetivo y que tipo de contenido le gusta a este usuario, lo que se llama un sistema de recomendación híbrido. Usando una implementación en Python de KNN y la base de datos *TMDB*, obtiene un sistema de recomendación que dada una película o una serie, le da una lista de 10 recomendaciones junto con una puntuación de su predicción. Si, por ejemplo, introduce la película *Notting Hill* obtenemos las siguientes 10 recomendaciones con su respectiva puntuación, como se puede ver en la Figura 2.4.

```
Enter a movie title: Notting Hill
Selected Movie:  Notting Hill

Recommended Movies:

Morning Glory | Genres: 'Comedy','Drama','Romance' | Rating: 6.1
Forrest Gump | Genres: 'Comedy','Drama','Romance' | Rating: 8.2
Cyrus | Genres: 'Comedy','Drama','Romance' | Rating: 6.1
Love Actually | Genres: 'Comedy','Drama','Romance' | Rating: 7.0
Larry Crowne | Genres: 'Comedy','Drama','Romance' | Rating: 5.7
In Good Company | Genres: 'Comedy','Drama','Romance' | Rating: 5.9
Molly | Genres: 'Comedy','Drama','Romance' | Rating: 5.5
When Harry Met Sally... | Genres: 'Comedy','Drama','Romance' | Rating: 7.3
Youth in Revolt | Genres: 'Comedy','Drama','Romance' | Rating: 5.9
And So It Goes | Genres: 'Comedy','Drama','Romance' | Rating: 5.7


The predicted rating for Notting Hill is: 6.340000
The actual rating for Notting Hill is 7.000000
```

Figura 2.4: Ejemplo de recomendación de películas usando KNN

2.2.2. Ejemplos reales de aplicaciones de KNN en salud

Otra aplicación del algoritmo de KNN es en el ámbito de la salud. Se puede usar para predecir el riesgo de cáncer o para predecir una diagnosis de los distintos tipo de cáncer.

En el artículo [13] vemos un ejemplo concreto en el que se KNN usa para predecir la diagnosis del cáncer de mama usando la base de datos *WBCD (Wisconsin Breast Cancer Database)*. Los autores del artículo, usan el algoritmo KNN con diferentes funciones de distancia (Euclidia, Manhattan, distancia del coseno y Correlación de la distancia) y con diferentes reglas de clasificación (mayoría, consenso y aleatorio; vistas en el aprtado 2.2 del trabajo.) en función del parámetro k que se va variando. En la Figura 2.5 vemos los resultados que se obtienen:

Nearest Rule				
Value of k	Euclidean	Cityblock	Cosine	Correlation
$k = 1$	98, 70	98, 48	95, 67	94, 69
$k = 10$	95, 41	95, 48	95, 41	95, 35
$k = 25$	95, 13	95, 16	95, 11	95, 07
$k = 50$	94, 15	94, 46	94, 43	94, 40
Consensus Rule				
Value of k	Euclidean	Cityblock	Cosine	Correlation
$k = 1$	98, 70	98, 48	95, 67	94, 69
$k = 10$	83, 35	83, 37	95, 12	95, 03
$k = 25$	73, 64	73, 67	94, 86	94, 83
$k = 50$	64, 36	64, 39	94, 27	94, 24
Random Rule				
Value of k	Euclidean	Cityblock	Cosine	Correlation
$k = 1$	98, 70	98, 48	95, 67	94, 69
$k = 10$	95, 12	95, 19	95, 12	95, 03
$k = 25$	94, 90	94, 91	94, 86	94, 83
$k = 50$	94, 29	94, 30	94, 27	94, 24

Figura 2.5: Resultados de KNN para predecir el riesgo de cáncer

2.2.3. Ejemplos reales de aplicaciones de KNN en análisis de datos financieros

El algoritmo KNN puede ser usado para predecir la evolución de los precios de una acción o para analizar el riesgo de dar un préstamo.

En el artículo [22], los autores escogen cuatro acciones de la bolsa de Estocolmo

y se compara el valor predicho por KNN con el valor real de las acciones durante un periodo de unos 2 años. En concreto la 4 acciones elegidas son: *Investor*, una compañía sueca de inversiones; *H&M*, una cadena multinacional sueca de ropa; *Astrazeneca*, una compañía farmacéutica sueca y británica; y *ABB*, una corporación multinacional tecnológica sueco-suiza. Se toma el valor k=4 y como datos de entrenamiento se toma la serie histórica de los valores de las acciones. Para ver como de efectivo es KNN en este caso vamos a ver en la Figura 2.6 los valores reales de la acción de Astrazeneca, los valores predichos y la diferencia entre estos por meses.

Astrazeneca	KNN			
Date	Actual Price	Predicted price	Deviation	Percentage Error %
1/2/12	315.80	324.10	8.30	2.63
1/3/12	324.40	323.35	1.05	0.32
1/4/12	324.70	322.68	2.02	0.62
1/5/12	321.40	322.68	1.28	0.40
1/6/12	321.40	321.43	0.03	0.01
1/9/12	321.60	324.13	2.52	0.79
1/10/12	323.10	323.38	0.27	0.09
1/11/12	323.70	326.20	2.50	0.77
1/12/12	325.60	324.73	0.88	0.27
1/13/12	325.00	328.03	3.02	0.93
1/16/12	328.20	329.88	1.68	0.51
1/17/12	328.90	327.63	1.27	0.39
1/18/12	328.60	322.35	6.25	1.90
1/19/12	321.10	317.00	4.10	1.28
1/20/12	319.90	322.30	2.40	0.75
1/23/12	322.00	322.35	0.35	0.11
1/24/12	321.10	321.43	0.32	0.10
1/25/12	321.60	324.13	2.52	0.79
1/26/12	323.00	322.45	0.55	0.17
1/27/12	322.40	321.30	1.10	0.34
1/30/12	323.80	326.28	2.47	0.76

Figura 2.6: Tabla de el valor de la acción de Astrazeneca

Podemos ver los resultados de la tabla 2.6 de forma gráfica en la figura 2.7, siendo esta figura la gráfica de los valores reales (azul) y los valores predichos (rojo) a lo lo largo del tiempo:

Las predicciones para las otras tres acciones tienen una precisión similar a las predicciones para Astrazeneca que, como se ha podido ver en la tabla y en el gráfico, son unas predicciones muy buenas.

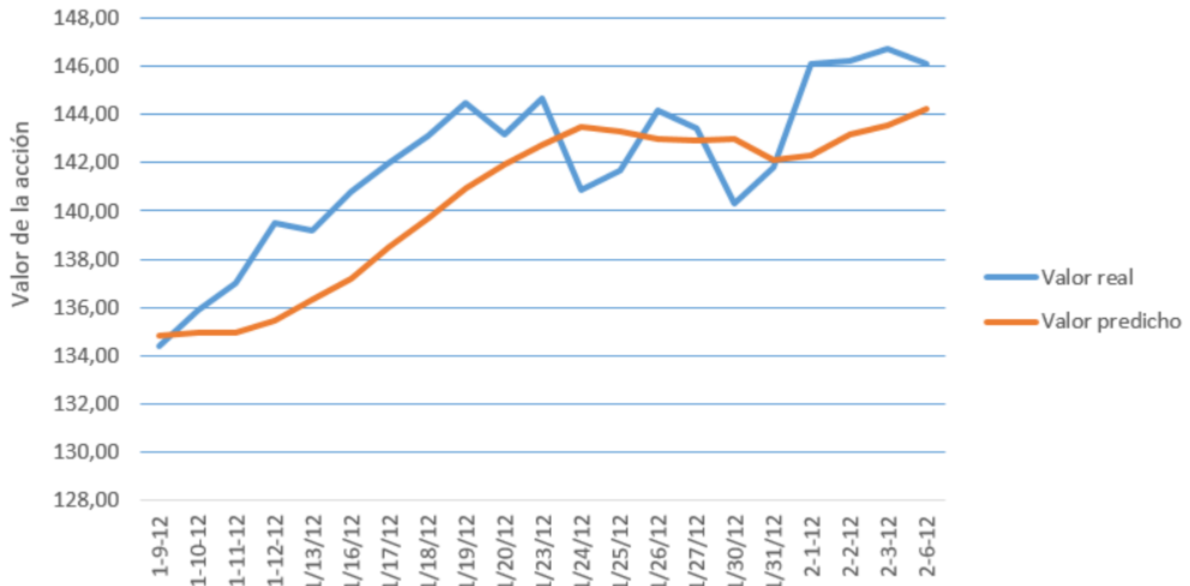


Figura 2.7: Gráfica de el valor de la acción de Astrazeneca frente al tiempo

2.3. VP-Tree

El VP-Tree (Vantage-point tree) es una estructura de datos en forma de árbol que tiene como función indexar espacios métricos con funciones de distancia continuas. Esta estructura fue descubierta independientemente por Peter Yianilos [25] y por Jeffrey Uhlmann [21], aunque el trabajo de Uhlman fue publicado antes. El nombre de *VP-Tree* lo dio Yianilos, mientras que Uhlman lo nombró *metric tree*.

Dado un espqcio métrico $(\mathbb{U}, d)$, se elige arbitrariamente un punto $p \in \mathbb{U}$ y se toma la mediana de las distancias $M = \{d(p, u) | u \in \mathbb{U}\}$. Luego, guardamos los puntos $u \in \mathbb{U}$ tales que $d(u, p) \leq M$ en el sub-árbol de la izquierda y los puntos tales que $d(u, p) > M$ en el de la derecha. A partir de estos dos nuevos sub-árboles repetimos los pasos anteriores hasta que en las hojas del árbol tengamos solo un punto de $\mathbb{U}$, entonces ya tendríamos el VP-Tree construido[16].

En la Figura 2.8 hay un ejemplo extraído de un artículo[16] en donde se muestra un VP-Tree con raíz u_{11} (la mediana M para la raíz es dibujada como una circunferencia rayada).

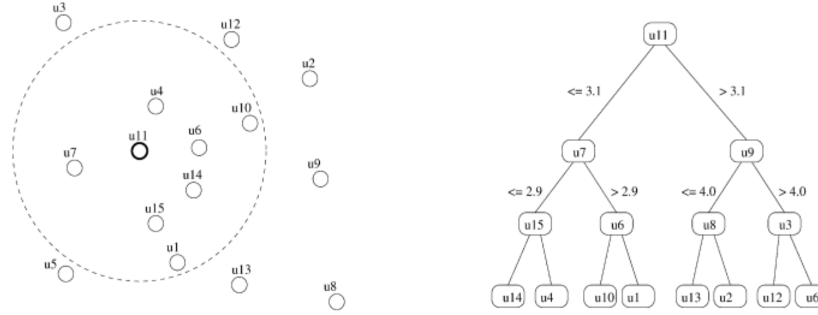


Figura 2.8: Ejemplo de VP-Tree

Usando la estructura de VP-Tree nos puede servir para hacer una búsqueda de los k-vecinos más cercanos. En el artículo de Yianilos [25] se describe un algoritmo en pseudocódigo (que podemos ver en la figura 2.9) que tiene como entrada un conjunto S sobre el que se va a construir el árbol. También se hace uso de una función, llamada $Seleccionar_vp(S)$, para elegir el mejor *vantage point* de S evaluando cada uno de los candidatos y escogiendo el mejor.

Algorithm 1 VP tree

1: **procedure** $Construir_vp_tree(S)$
2:     **if** $S = \emptyset$ **then return** $\emptyset$
3:     nuevo(nodo);
4:     $p \leftarrow Seleccionar_vp(S)$;
5:     m $\leftarrow Mediana_{s \in S} d(p,s)$;
6:     $L \leftarrow \{s \in S \setminus \{p\} | d(p,s) < m\}$;
7:     $R \leftarrow \{s \in S \setminus \{p\} | d(p,s) \geq m\}$;
8:     Izquierda $\leftarrow Construir_vp_tree(L)$;
9:     Derecha $\leftarrow Construir_vp_tree(R)$;
10:     **return** b
11: **procedure** $Selecionar_vp(S)$
12:     $P \leftarrow$ Muestra aleatoria de S;
13:     $Mejor_varianza \leftarrow 0$;
14:     **for** $p \in P$ **do**
15:         $D \leftarrow$ Muestra aleatoria de S;
16:         $m \leftarrow Mediana_{s \in S} d(p,s)$;
17:         $Varianza \leftarrow Var(d(p,d) - m)$;
18:         **if** $Varianza > Mejor_varianza$ **then**
19:             $Mejor_varianza \leftarrow Varianza$;
20:             $Mejor_p \leftarrow p$;
    return $Mejor_p$;

Figura 2.9: Pseudocódigo del algortimo de construcción de un VP-Tree

Para construir el árbol la complejidad temporal es de *O(n log(n))* y para rea-

lizar una búsqueda en el árbol la complejidad temporal es de *O(logn)*[25].

Existen múltiples aplicaciones para el VP-Tree. Por ejemplo, en un artículo por I.Markov [12], se desarrolla un método para indexar bases de datos de imágenes utilizando el VP-Tree. Para ello usan el hecho de que toda imagen es un array bidimensional de píxeles para extraer una serie de características de la imagen. Un ejemplo de estas características es la distribución de color de la imagen usando su histograma. Una característica de una imagen toma un valor escalar y, por tanto, dadas K características, cada imagen puede ser representada en un espacio K-dimensional. Una vez las características de todas las imágenes de la base de datos son extraídas, se puede aplicar el VP-Tree para hacer búsquedas de vecinos más cercanos.

Otra aplicación del VP-Tree es dada por S.Lee en [9]. En este artículo, se usa el VP-Tree para indexar y hacer búsquedas en bases de datos de vídeos empleando las características de un vídeo de forma similar a lo que se vio en el párrafo anterior con imágenes. Se utiliza un VP-Tree mejorado en el que se escoge como pivote el punto que nos da mayor varianza. Todo esto se aplica a una base de datos de 10000 vídeos y se usa el histograma HSI par obtener las características, se llega a la conclusión de que el VP-Tree realiza las búsquedas entre un 5 % y un 12 % más rápido que el algoritmo AESA.

El VP-Tree también es útil para realizar búsquedas en bases de datos de melodías, como Michael Skalak, Jinyu Han y Bryan Pardo explican en su artículo [19]. Para indexar una melodía, está se codifica en la base de datos como una tabla de la frecuencia con respecto del tiempo y se extraen caracteristicas como el ritmo o el tono de la melodía.

2.4. Balltree

Los Balltree son estructuras de datos similares a los VP-Tree, pues ambas son árboles y ambas particionan el espacio. Debido a sus características, el Balltree tiene entre sus aplicaciones la búsqueda del vecino más cercano, al igual que el VP-Tree.

En [15] tenemos como se construye este árbol. En el balltree, la raíz del árbol es todo el dataset. Un nodo del árbol puede ser una hoja o no. Si el nodo es una hoja, contiene una lista de los puntos que representa. Un nodo que no es una hoja tiene dos nodos hijos, que podemos llamar *Hijo1* e *Hijo2*, disjuntos entre si y cuya unión es todos los puntos del nodo. Cada nodo tiene un *Pivote* que, según la implementación, puede ser un punto del nodo o el centroide de todos los

puntos del nodo. Cada nodo tiene asociada una bola con un radio tal que:

$$Radio = \max_{x \in Puntos.Nodo} |Pivote - x| \tag{2.1}$$

A medida que bajamos en el árbol, los nodos van teniendo menor radio. Conseguimos esto haciendo que:

$$x \in Puntos.Hijo1 \implies |x - Pivote.Hijo1| \leq |x - PivotedeHijo2| \tag{2.2}$$

$$x \in Puntos.Hijo2 \implies |x - Pivote.Hijo2| \leq |x - PivotedeHijo1| \tag{2.3}$$

Dadas las ecuaciones 2.2 y 2.3 y dado que toda función de distancia cumple la desigualdad triangular, tenemos que dado un punto nuevo q, podemos acotar la distancia de q a cualquier punto en cualquier nodo dado que si $x \in Puntos.Nodo$:

$$|x - q| \geq |q - Pivote| - Radio \tag{2.4}$$

$$|x - q| \leq |q - Pivote| + Radio \tag{2.5}$$

En resumen, cada nodo del árbol tiene asociado una bola, por lo que cada nodo del árbol parte el espacio en dos subconjuntos disjuntos (el interior y el exterior de la bola). Tomamos en cada nodo una bola tal que el interior de cada bola es el menor que contiene las bolas de sus nodos hijos. Es decir, cada nodo del árbol define la bola más pequeña que contiene todos los puntos de su subárbol. Las hojas del árbol son los puntos del conjunto y la raíz es el conjunto completo.[17]

A. Moore et al. muestran de forma visual en su trabajo [15] un ejemplo muy sencillo el proceso de creación de un Balltree, que se reproduce en la Figura 2.10.

El algoritmo más simple de construcción del Balltree es el *Algoritmo de construcción k-d* descrito en [17]. El árbol se construye de arriba hacia abajo particionando recursivamente el conjunto de puntos en dos conjuntos. Para esta partición se selecciona la mediana de la dimensión con mayor varianza y los puntos con mayor valor que la mediana en esa dimensión pertenecen a un subconjunto y los de menor valor a otro, por lo que la intersección de ambos conjuntos es vacía y su unión es el conjunto total. La función toma como *input* el conjunto S que es el conjunto de datos. Dado que encontrar la mediana para cada nivel tiene una complejidad de *O(n)* y hay *log(n)* niveles, el algoritmo de construcción del árbol tiene orden *O(n log(n)))*. En la figura 2.11 vemos el pseudocódigo del algoritmo.

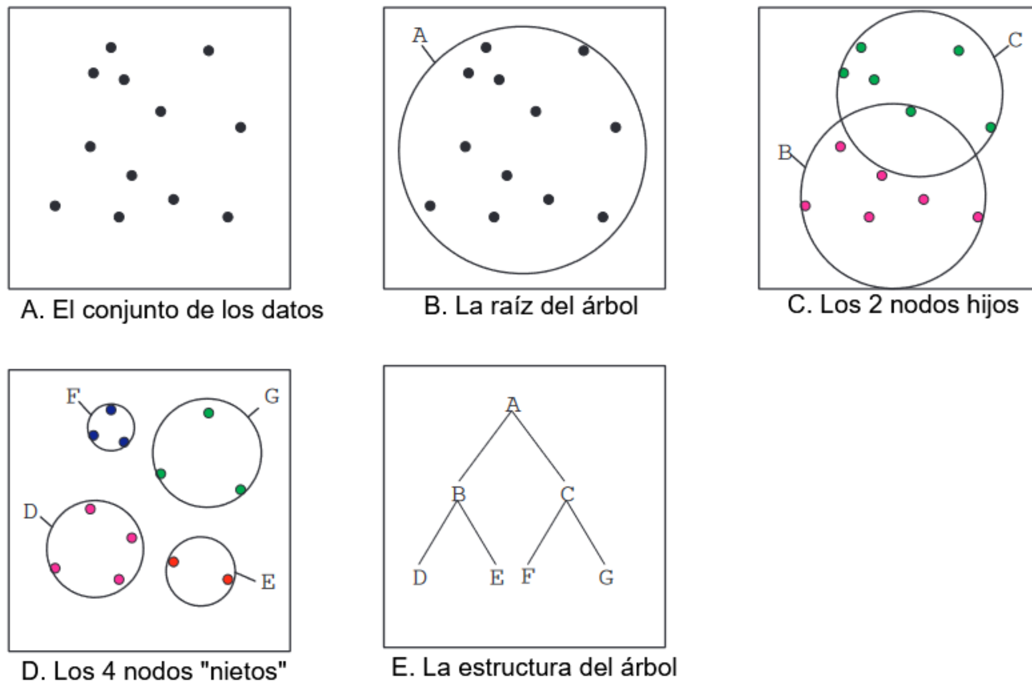


Figura 2.10: Ejemplo de Balltree

Algorithm 2 Balltree

1: **procedure** *construir_Balltree*(S)
2:     **if** S solo tiene un punto **then**
3:         Crear un hoja B que contenga el único punto en S;
4:         **return** B
5:     Sea d la dimensión de mayor varianza;
6:     Sea p el punto central considerando la dimensión d;
7:     Sea L El conjunto de los puntos a la izquierda de la mediana en la dimensión d;
8:     Sea R El conjunto de los puntos a la derecha de la mediana en la dimensión d;
9:     Crear B con dos subárboles hijos;
10:     $B.pivote \leftarrow p$;
11:     $B.hijo1 \leftarrow$ construir_Balltree(L);
12:     $B.hijo2 \leftarrow$ construir_Balltree(R);
13:     Sea $B.radio$ la mayor distancia hasta p entre los dos hijos de B;
14:     **return** B

Figura 2.11: Pseudocódigo del algortimo de construcción de un VP-Tree

Una aplicación muy interesante del Balltree es dada por Weiguo Wan y Hyo Jong Lee en un artículo [23] en donde usan el Balltree para el reconocimiento de imágenes con el fin de asociar un sketch de una cara con la foto real de la respectiva cara. Para ello se requiere de una base da datos con las fotos de las cara y otra con los sketch. En el preprocesamiento de los datos primero se normalizan

las imágenes de las caras y los sketch fijando los ojos en el centro, haciendo uso de la librería *Dlib*. El ultimo paso del preprocesamiento de los datos que hacen es la extracción de características mediante la red neuronal *VGG-face*. Una vez preprocesados los datos ya podemos expresar las fotos y los sketch como vectores y hacer uso del Balltree. Para ver la precisión del método que proponen, lo aplican a dos *datasets* distintos obteniendo una precisión de entre el 70 % y el 100 % usando la distancia Chi-cuadrado, que es con la que mejor rendimiento obtenían.

En el artículo [10] encontramos otra aplicación del Balltree. Con la explosión de internet, la cantidad de datos que las compañías recogen de sus usuarios ha crecido exponencialmente, incluidas las imágenes, y muchas veces los servidores no son del todo seguros. Los autores de este artículo proponen un un esquema de recuperación de imágenes escalable y que preserva la privacidad mediante un Balltree para resolver este problema. Para ello, primero se emplea el modelo preentrenado *Convolutional Neural Network* (CNN) para extraer las características de las imágenes. Luego, se construye un Balltree encriptado mediante un algoritmo KNN seguro basado en *Learning With Errors*(LWE). Tras una serie de experimentos con conjuntos de datos del mundo real, se llega ala conclusión de que su sistema es práctico en términos de seguridad, precisión y eficacia.

Otra aplicación del Balltree se propone en [7]. Aquí los autores presentan un sistema de recomendaciones basado en Balltree, que llaman *Ball-sim*, con una nueva métrica de similaridad. Los escenarios experimentales del trabajo describen los pasos para construir un balltree e identificar a los usuarios cercanos basándose en la estructura del árbol. Además, el trabajo también evalúa el sistema de recomendación implementado comparando los resultados del sistema de recomendación. Los datos utilizados en los experimentos son el conjunto de datos *Movielens*, un sistema de recomendación de películas basado en la web.En Este conjunto de datos encontramos 100.000 muestras, incluyendo valoraciones de 943 usuarios para 1.664 películas. Los resultados muestran que el sistema de recomendación con una métrica de similitud basada en Balltree puede mejorar la precisión en comparación con una métrica de uso común como la métrica del coseno.

3

Estudio comparativo y resultados

En este capítulo del trabajo vamos a describir la metodología que se ha usado en las distintas pruebas para comparar KNN, VP-Tree y Balltree, así como los resultados obtenidos en estas. Para realizar las pruebas se ha elegido el lenguaje de programación Python 3.9.12. Todos los casos han sido simulados en un Lenovo IdeaPad 5 Gen 7 con el sistema operativo Fedora. Este equipo cuenta con 16 GB de memoria RAM, una CPU AMD Ryzen 7 5700U y el kernel Linux está en la versión 6.3.4.

3.1. Implementación de los algoritmos

3.1.1. KNN básico (fuerza bruta)

El modelo KNN básico se crea utilizando la biblioteca scikit-learn. Esta librería contiene una serie de herramientas para Python muy eficientes y útiles para el análisis de datos, por lo que la vamos a usar bastante en este trabajo. Dentro de esta librería, tenemos la función *sklearn.neighbors.KNeighborsClassifier*. La función recibe como parámetros el valor de k que elegimos, el método usado (para la versión "básica"de KNN elegimos el método 'brute', ya lo que denominamos como versión básica no es más que un algoritmo de fuerza bruta) y la métrica, que en este caso usaremos siempre la euclídea. Para indexar nuestro conjunto de puntos, empleamos la función *fit* de la misma librería, que sirve para 'entrenar' al modelo para poder hacer predicciones después. Una vez indexado el conjunto de puntos, hacemos la búsqueda de los k vecinos más cercanos mediante la función *predict* de esta librería, que nos devuelve la categoría asignada según KNN.

En el siguiente código se puede ver un ejemplo muy simple de como se pueden

usar estas funciones para la búsqueda de los k vecinos más cercanos:

```
from sklearn.neighbors import KNeighborsClassifier

X = [[0], [1], [2], [3]]
y = [0, 0, 1, 1]

knn = KNeighborsClassifier(n_neighbors=3, algorithm='brute',
    metric='euclidean')
knn.fit(X, y)

knn.predict([[1.1]])
```

En este ejemplo el conjunto de datos son 4 puntos $\{0, 1, 2, 3\}$ con $\{0, 0, 1, 1\}$ como sus respectivas categorías. Si tomamos k=3, el algoritmo knn nos clasifica al punto 1.1 en la categoría 0. Esta clasificación tiene sentido, pues los puntos más cercanos a 1.1 son $\{0, 1, 2\}$ con categorías $\{0, 0, 1\}$ respectivamente. Dado que hay dos 0 y un 1, se clasifica a 1.1 como 0.

3.1.2. KNN basado en VP-Tree

Para crear el modelo KNN basado en VP-Tree, vamos a utilizar la librería NMSLIB. Esta librería contiene varios métodos de búsqueda por similitud (entre ellos KNN), aparte, está librería funciona tanto en Python como en C++. Dentro de esta, tenemos la función *nmslib.init* que recibe como parámetros el método, en este caso elegimos el método 'vptree', y para usar la distancia euclídea usamos el parámetro $space =' l2$. Luego, creamos el árbol con el conjunto de datos mediante la función *addDataPoinBach* e introducimos el conjunto de datos como argumento. Para crear el índice usamos la función createIndex (sin parámetros).

Una vez creado el índice, realizamos la búsqueda de los k vecinos más cercanos con la función *knnQueryBatch* que recibe como argumentos el valor de la k y el conjunto de puntos del que queremos saber sus k vecinos más cercanos. A diferencia de la implementación de KNN básico, esta función nos devuelve los índices y las distancias de los vecinos, pero no nos devuelve la categoría que se le asignaría, por lo que tenemos que realizar las votaciones con funciones de la librería NumPy, es decir, le asignaremos la etiqueta mayoritaria de los k vecinos.

El siguiente código es un ejemplo usando los mismos datos que en el apartado anterior que puede ayudar a ver mejor como funciona esta implementación.

```
import nmslib
import numpy as np

X = [[0], [1], [2], [3]]
y = [0, 0, 1, 1]

#genereamos el arbol con los datos:
index = nmslib.init(method='vptree', space='l2')
```

```
9 index.addDataPointBatch(X)  de X
10 index.createIndex()
11
12 #busqueda de los 3 vecinos mas cercanos a 1.1:
13 vp_tree_indices = index.knnQueryBatch([[1.1]], k=3)
14
15 #categorizamos a 1.1 segun sus 3 vecinos mas cercanos:
16 y_pred = []
17 for x in vp_tree_indices:
18     unique_labels, counts = np.unique(y[x[0]],
19         return_counts=True)
20     pred_label = unique_labels[np.argmax(counts)]
21     y_pred.append(pred_label)
22
23 print(y_pred)
```

Como hemos usado los mismos datos y hemos realizado la misma búsqueda que en el apartado anterior, obtenemos el mismos resultado, es decir, la categoría 0.

3.1.3. KNN basado en Balltree

El caso de KNN usando un Balltree es bastante similar al caso de KNN básico, pues en ambos uso la librería scikit-learn. La única diferencia es que se usa la función sklearn.neighbors.BallTree. Vemos otra vez el mismo ejemplo que antes, pero para el caso del Balltree:

```
1 from sklearn.neighbors import KNeighborsClassifier
2
3 X = [[0], [1], [2], [3]]
4 y = [0, 0, 1, 1]
5
6 knn = BallTree(X, metric='euclidean')
7 dist, ind = knn.query([[1.1]], k=1) )
```

Y obtenemos que al punto 1.1 se le asigna la categoría 0, al igual que en el caso de KNN básico y KNN con VP-Tree.

En este algoritmo el parámetro *leaf_size* es importante para reducir el tiempo de búsqueda. Este parámetro controla el número de puntos en los que se cambia a fuerza bruta. Cambiar el *leaf_size* no afectará a los resultados de una consulta. La cantidad de memoria necesaria para almacenar el árbol es aproximadamente $\frac{n_muestras}{leaf_size}$. Para un *leaf_size* especifico, se garantiza que una hoja satisface $leaf_size \leq n_puntos \leq 2 * leaf_size$, excepto en el caso de que $n_muestras < leaf_size$ [18].

3.2. Conjuntos de datos de prueba

Para hacer las distintas pruebas, vamos a aplicar las implementaciones de los algoritmos previamente descritas a cuatro conjuntos de datos distintos:

1. 6 nubes de puntos sin solape entre ellas
2. 6 nubes de puntos solape entre ellas
3. El dataset GloVe
4. El dataset MNIST

3.2.1. Seis nubes de puntos sin solape

Para generar este conjunto de datos usamos la función *make_blobs* de la librería scikit-learn. Esta función nos permite generar un número de nubes de datos, elegir cuantos puntos generar, fijar el centro que queramos para cada nube y las dimensiones del conjunto, entre otras cosas. Para estas pruebas se generan 6 nubes en dos dimensiones con 1000 puntos cada una y fijamos los centros de tal forma que la separación entre nubes sea suficiente para que no se solapen.

Para que este test sea reproducible usamos 42 como semilla. El código para esto es el siguiente:

```
from sklearn.datasets import make_blobs

np.random.seed(42)
n_samples = 1000  # N mero de puntos por grupo
n_groups = 6  # N mero de grupos
n_features = 2  # N mero de dimensiones
centers = np.array([[-5, 10], [5, 10], [-5, -10],
    [5, -10], [-5, 0], [5, 0]])

X, y = make_blobs(n_samples=n_samples * n_groups,
    n_features=n_features, centers=centers)
```

La función devuelve dos vectores 'X' e 'y'. 'X' es el vector que contiene el par de coordenadas de cada punto y el vector 'y' contiene las categorías asignadas a cada punto, es decir, *y(i)* es la categoría de *X(i)*.

Luego, se divide el conjunto en un conjunto de entrenamiento y otro de prueba. Esto nos será útil a la hora de hacer las comparaciones más tarde. Para esto me valgo de la función *train_test_split* de scikit-learn. Dividimos el conjunto de tal forma que el 1200 de los puntos se vayan al conjunto de pruebas y el resto al de entrenamiento. El código que se emplea para esto es el siguiente:

```
from sklearn.model_selection import tran_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=1200)
```

Podemos visualizar los datos generados en la figura 3.1 mediante la la librería Pyplot con el siguiente código:

```
import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1], c=y)
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Dataset con 6 grupos sin solapamiento')
plt.show()
```

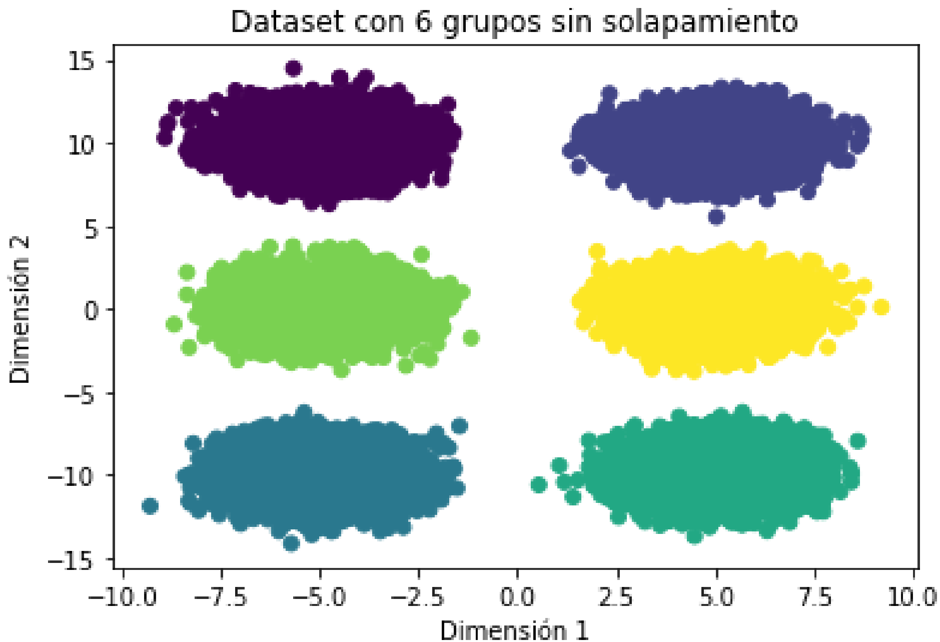


Figura 3.1: Gráfica de las 6 nubes sin solape

3.2.2. Seis nubes de puntos con solape

Para obtener seis nubes de puntos con solape entre si, usamos un código muy parecido al caso en el que no había solape. La diferencia es que no se especifican los centros en la función. Para estos datos se generan 50000 puntos por nube para ver como se comportan los algoritmos con más puntos. *make_blobs*. Para que los resultados sean reproducibles, tomamos 0 como semilla. Por lo que el código es el siguiente:

```
from sklearn.datasets import make_blobs

np.random.seed(42)
n_samples = 50000  # Numero de puntos por grupo
n_groups = 6  # Numero de grupos
n_features = 2  # Numero de dimensiones
X, y = make_blobs(n_samples=n_samples * n_groups,
    n_features=n_features, centers=n_groups)
```

Luego se separan los datos en conjunto de entrenamiento y de prueba de la misma forma:

```
from sklearn.model_selection import tran_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=1200)
```

En la figura 3.2 se muestra la gráfica del conjunto de datos que se obtiene:

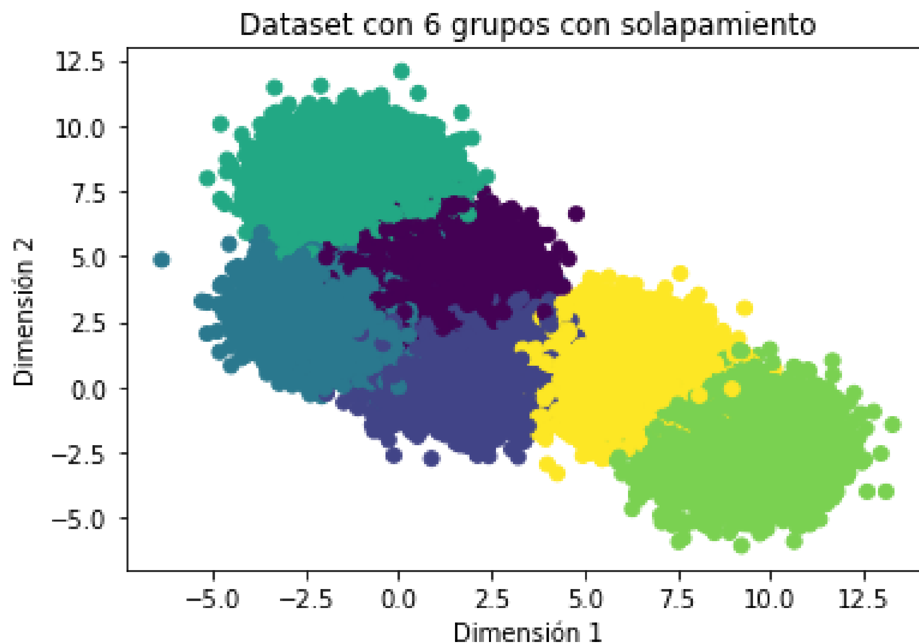


Figura 3.2: Gráfica de las 6 nubes con solape

3.2.3. Base de datos GloVe

Esta base de datos contiene 400000 palabras que son expresadas como vectores de $\mathbb{R}^{100}$. Por tanto, esta base de datos consta de 400000 puntos en un espacio de 100 dimensiones. Solo vamos a centrarnos en encontrar los k vecinos más cercanos, es decir, encontrar las k palabras más similares a una dada. Como cada palabra del conjunto no tiene una categoría asignada, el objetivo de KNN en este caso es encontrar las k palabras más parecidas.

La base de datos se descarga en forma de txt desde este enlace. Para cargar el txt podemos usar la función *read_table* de pandas, donde hay que indicar que se usa el espacio en blanco para separar los datos, que el índice es la columna 0 y que no hay cabecera. El siguiente código muestra como se implementa:

```
import pandas as pd

```

```
words=pd.read_table('glove.6B.100d.txt', sep=' ',
    index_col=0, header=None)
```

Debido a la gran cantidad de puntos que hay en el dataset, el tiempo que se tarda en el equipo de pruebas en realizar los tests no es práctico para analizar los algoritmos correctamente, por lo que sería conveniente tomar un muestra aleatoria de 120000 puntos del conjunto de datos. Con esta muestra aleatoria, podemos realizar los tests sin comprometer la calidad de las pruebas. Para coger la muestra aleatoria de 120000 palabras, vamos a emplear la función *sample* de pandas. El código, donde *random_state=1* es la semilla, para esto es:

```
samp=120000
words=words.sample(n=samp, random_state=1)
```

Los datos leídos están en un formato de DataFrame en el que el índice de las filas son las palabras y cada fila es el vector de $\mathbb{R}^{100}$ que corresponde a su palabra. Vamos a guardar la lista de palabras en la variable y con sus respectivos vectores en la variable X:

```
y=list(words.index) #tomamos solo los indices y los convertimos
    en una lista
X=words.to_numpy()
```

Por último, se separan los datos en conjunto de entrenamiento y de prueba de la misma manera que los anteriores conjuntos de datos, como se puede observar en el siguiente fragmento de código:

```
from sklearn.model_selection import tran_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=1200, random_state=42)
```

3.2.4. Base de datos MNIST

MNIST(Modified National Institute of Standards and Technology database) es una base de datos que contiene los dígitos del 0 al 9 escritos a mano. EL conjunto de entrenamiento está compuesto por 60000 ejemplos y el conjunto de prueba por 10000 ejemplos. El MNIST es un subconjunto de la base de datos NIST con los dígitos normalizados y centrados en la imagen. Cada dato es una imagen en escala de grises con dimensiones 28x28.

Para poder trabajar con MNIST en Python, primero tenemos que importarlo desde la librería Keras de TensorFlow, que podemos hacer con esta linea de código:

```
from tensorflow.keras.datasets import mnist
```

Ahora guardamos el conjunto de entrenamiento (60000 datos) y el de prueba (10000 datos) con sus respectivas etiquetas en variables distintas:

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Ahora bien si vemos las dimensiones de *X_train*, son de (60000,28,28), es decir cada dato se almacena como una matriz 28x28. Esto no es ideal para hacer KNN, pues no existe una métrica cómoda para matrices. Por lo que sería conveniente un procesamiento previo de los datos.

Para poder trabajar correctamente con estos datos, tenemos que aplanar las matrices (convertir una matriz $n \times n$ en un vector de dimensión n^2) mediante la función *reshape* de la librería NumPy en el conjunto de entrenamiento y de pruebas de la siguiente forma:

```
X_train=X_train.reshape(60000,-1)
X_test=X_test.reshape(10000,-1)
```

Ahora nuestros datos deberían de tener dimensiones (60000,784) en el caso de los datos de entrenamiento y (10000,784) en el caso de los datos de prueba. Para comprobarlo hacemos uso de la función *reshape* de la siguiente forma:

```
print('X_train: ' + str(X_train.shape))
print('X_test:  '  + str(X_test.shape))
```

Y, efectivamente, comprobamos que las dimensiones de cada punto son las deseadas:

```
X_train: (60000, 784)
X_test:  (10000, 784)
```

Como vemos, el conjunto de pruebas no tiene la misma cantidad de puntos que en el resto de datasets (1200 puntos). Para separar el conjunto de la misma manera que el resto, primero juntaremos estos conjuntos de entrenamiento y de prueba de la siguiente manera:

```
X=np.concatenate((X_train,X_test))
y=np.concatenate((y_train,y_test))
```

Y ahora ya podemos separar el conjunto para obtener un conjunto de pruebas con 1200 puntos con la función *train_test_split*:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=1200, random_state=42)
```

3.3. Criterios para la comparación de los algoritmos

Para comparar los diferentes algoritmos de KNN, hay varios factores que puede tener en cuenta. El primero de estos factores, son las métricas de evaluación, que explicaremos que son en este apartado, y el coste computacional de los distintos algoritmos en los conjuntos de datos previamente descritos.

3.3.1. Eficiencia computacional

Para medir la eficiencia de cada algoritmo me nos centraremos en el tiempo que tarda cada uno. Para ello usamos la función *time* del módulo time. Se medirá el tiempo que tarda cada uno de la siguiente forma:

```
import time
start_time=time.time()

    (Mi codigo)

print("%s tiempo que tarda en ejecutarse mi codigo"
    %(time.time()-start_time))
```

3.3.2. Métricas de evaluación

En una tarea de clasificación, nuestra labor principal es predecir la variable objetivo que está en forma de valores discretos (categorías). Para evaluar el rendimiento de un modelo de este tipo existen una serie de métricas como accuracy, precisión, recall o F1 score que voy a explicar a continuación.

El accuracy es la proporción de predicciones correctas entre el número total de casos examinados. La fórmula es muy simple:

$$Accuracy = \frac{clasificaciones\ correctas}{todas\ las\ clasificaciones\ realizadas} \tag{3.1}$$

Podemos implementar en Python esta métrica gracias a la función *accuracy_score* del módulo sklearn.metrics. Para importar esta función bastaría con una simple linea de código:

```
from sklearn.metrics import accuracy_score
```

La función recibe como argumentos el conjunto de etiquetas reales y el conjunto de etiquetas predichas (ambos en forma de array).

La precisión es una métrica de evaluación binaria, solo aplicable cuando queremos clasificar un dato en dos categorías distintas, no más. Podemos definir la precisión como número de positivos verdaderos (el número de objetos clasificados correctamente en la categoría positiva) dividido por el número total de positivos. La fórmula es la siguiente:

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

Donde TP es el número de positivos verdaderos y FP el número de positivos no clasificados correctamente. Podemos usar esta métrica en clasificaciones multiclase si calculamos la precisión globalmente contando el total de verdaderos positivos, falsos negativos y falsos positivos. La función *precision_score* del

módulo sklearn.metrics implementa esta formula en Python. Para importar esta función bastaría con una simple linea de código:

```
1 from sklearn.metrics import precision_score
```

La función recibe como argumentos el conjunto de etiquetas reales, el conjunto de etiquetas predichas (ambos en forma de array) y, en caso de que el problema no sea binario, el tipo de promediado de los datos. Para este último parámetro, *average="micro"* se comporta de la forma que se ha descrito antes.

El recall también es una métrica de evaluación binaria. Esta métrica es la proporción de positivos reales que se identificaron de forma correcta. La fórmula del recall es:

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

Donde TP es el número de positivos verdaderos y FN el número de negativos no clasificados correctamente. Podemos usar esta métrica en clasificaciones multiclase de la misma forma que la precisión. Podemos implementar en Python esta métrica gracias a la función *recall_score* del módulo sklearn.metrics. Se puede importar esta función mediente la siguiente linea de código:

```
1 from sklearn.metrics import recall_score
```

La función recibe como argumentos el conjunto de etiquetas reales, el conjunto de etiquetas predichas (ambos en forma de array) y, en caso de que el problema no sea binario, el tipo de promediado de los datos. Para este último parámetro, *average="micro"* se comporta de la forma que se ha descrito antes.

Por último, tenemos al F1 score. Esta métrica se puede interpretar como la media harmónica de la precisión y el recall. La fórmula matemática del F1 score es:

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{3.4}$$

Para usar el F1 score en Python, podemos emplear la función *f1_score* de sklearn.metrics, para importarla habría que escribir la siguiente linea de código:

```
1 from sklearn.metrics import f1_score
```

3.4. Resultados de los test

Vamos a analizar los resultados obtenidos de forma independiente en cada dataset y haremos un análisis de estos para cada conjunto de datos. Vamos a buscar los k puntos más cercanos a cada punto de un conjunto de 'test', llamado *X_test*, en cada conjunto de prueba. Para las comparaciones vamos a usar distintos valores de *leaf_size* y distintos valores de k, es decir, vamos a encontrar un número variable de vecinos más cercanos.

3.4.1. Pruebas en las seis nubes de puntos sin solape

Primero, comparamos los tiempos tomando como valores de k los números naturales impares entre 3 y 100. Para visualizar los resultados podemos ver la figura 3.3 que es la gráfica del tiempo respecto al valor de k de cada algoritmo.

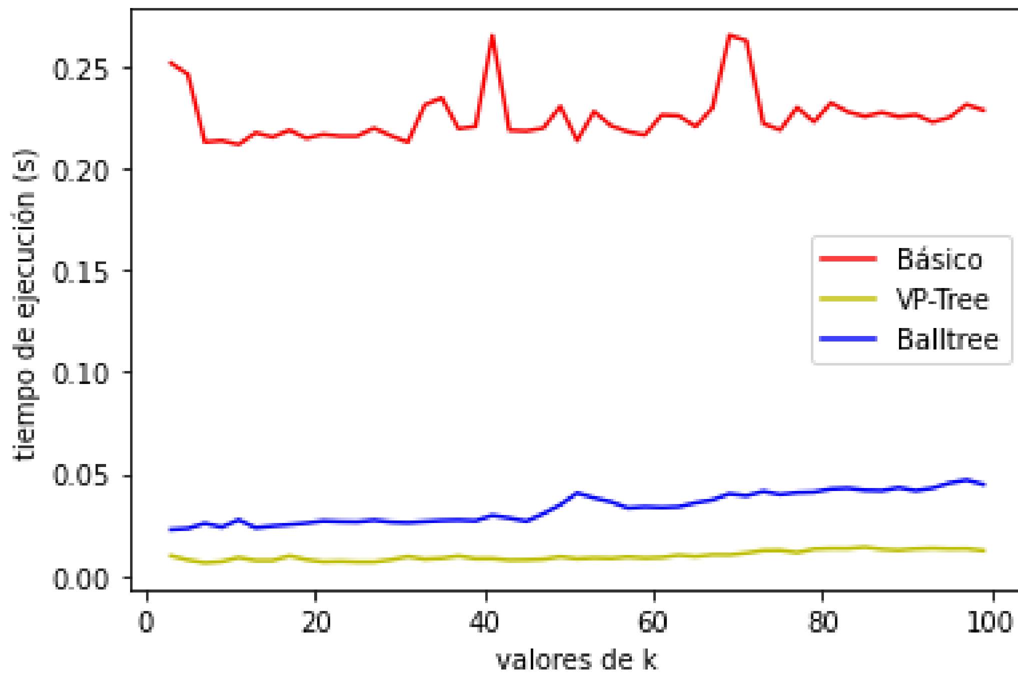


Figura 3.3: Gráfica del tiempo de ejecución (segundos) frente a k.

En Balltree se puede tomar el valor de *leaf_size* por defecto, pues como no hay muchos datos en este conjunto no merece la pena hacer un estudio del tiempo según el valor de *leaf_size*.

Como podemos ver en la gráfica 3.3, el algoritmo que mejor ha rendido para este conjunto (6000 puntos de 2 dimensiones) ha sido el VP-Tree, seguido del Ball Tree y el último ha sido el KNN básico. Por fuerza bruta se suele tardar décimas de segundo, con Balltree centésimas y con VP-Tree milésimas. Por tanto, el coste computacional del algoritmo por fuerza bruta comparado con los otros dos algoritmos es el más alto para estos datos y VP-Tree es el de menor coste.

Ahora, vamos a ver los resultados para las métricas de evaluación de cada algoritmo:

Métricas de evaluación				
Algoritmo	Accuracy	Precisión	Recall	F1 score
KNN Básico	1.00	1.00	1.00	1.00
VP-Tree	1.00	1.00	1.00	1.00
Balltree	1.00	1.00	1.00	1.00

Tabla 3.1: Tabla de comparación de las métricas de evaluación.

Para el Ball Tree volvemos a usar el valor por defecto de *leaf_size*, ya que este parámetro solo afectaría al tiempo de búsqueda del algoritmo no a su precisión, según la guía de la función Balltree.

Los resultados de la tabla 3.1 han sido obtenidos con k=5, pero con valores de k entre 1 y 100 obtenemos los mismos resultados. Podemos ver que todos han hecho las predicciones perfectas. Esto tiene sentido dado que hay una separación en las nubes y un punto cualquiera 'cae' en una nube o en otra pues no hay solape entre ellas.

Para ver que estos algoritmos encuentran los vecinos más cercanos, vamos a graficar el conjunto de datos junto a los cinco vecinos más cercanos obtenidos. Para ello, vamos a encontrar, por ejemplo, los vecinos del punto *X_test[1]* y vemos si son los más próximos al punto en la figura 3.4. El punto *X_test[1]* es representado como una estrella roja y los vecinos son representados como diamantes rojos. Como hemos visto con las métricas de evaluación, todos los algoritmos hacen las mismas predicciones, por lo que no importa que algoritmo usemos para encontrar los vecinos.

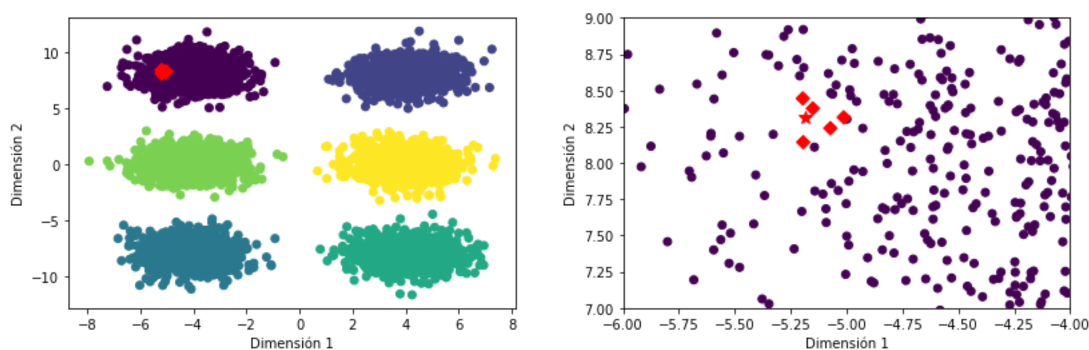


Figura 3.4: Cinco vecinos más cercanos de *X_test[1]*

Efectivamente, se puede ver como los diamantes rojos son los puntos más cercanos a la estrella roja.

3.4.2. Resultados en las seis nubes de puntos con solape

Para el estudio del coste computacional de cada algoritmo, vamos a tomar como valores de k los números naturales impares entre 3 y 100. Los resultados de esta prueba se muestran en la figura 3.5. En el Balltree vamos tomar el valor de leaf_size por defecto, ya que como tenemos pocos datos en el dataset, no merece la pena hacer un estudio del tiempo según el valor de leaf_size.

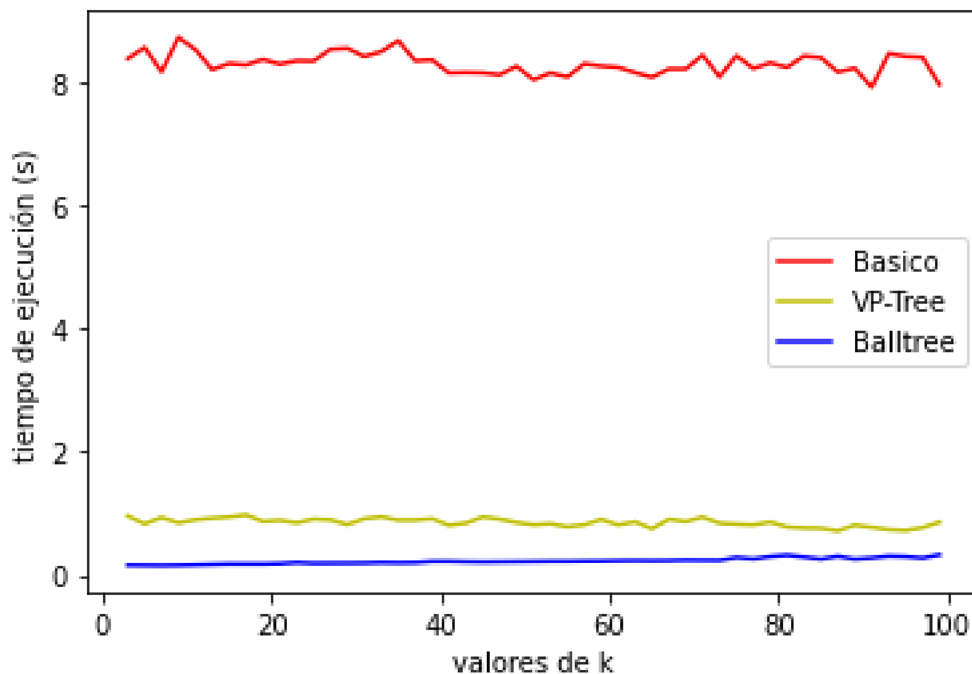


Figura 3.5: Gráfica del tiempo de ejecución (segundos) en función de la k

En la figura 3.5, podemos ver que el que menos tiempo tarda es el Balltree y el que más es el algoritmo básico. Tanto el Balltree como el VP-Tree suelen tardar menos de un segundo, mientras que el algoritmo por fuerza bruta tarda alrededor de 8 segundos. Por tanto, en este caso (300000 puntos en dos dimensiones) el algoritmo básico tarda significativamente más que VP-Tree y Balltree, mientras que el que menos tarda es el Balltree.

Para la comparación de las métricas de evaluación, encontramos el mejor valor de k para el accuracy probando valores de k impares entre 3 y 100. Si realizamos la gráfica del accuracy frente al tiempo, obtenemos la figura 3.6, en la que se puede ver que todos los algoritmos obtienen los mismos resultados.

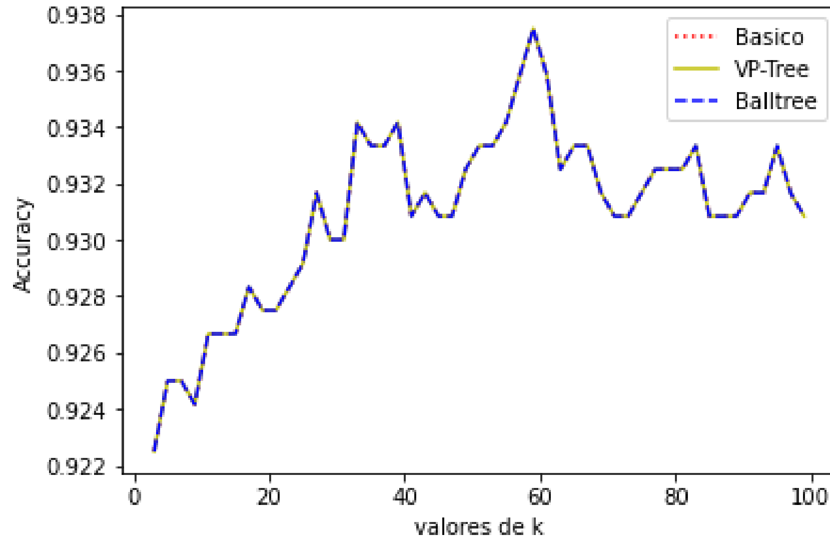


Figura 3.6: Accuracy frente al valor de k

En la figura 3.6 las curvas de los tres algoritmos están superpuestas, por lo que nos dan exactamente los mismos resultados. El valor máximo del accuracy es 0.9375 que se llega con k=59 en todos los algoritmos. También hay que tener en cuenta que en la figura 3.6 el eje Y está acotado, por lo que los picos parecen más pronunciados de lo que serían si el eje Y no estuviese acotado.

De la misma forma que en el anterior conjunto, podemos ver que vecinos más cercanos del punto *X_test[1]* (se llaman igual pero es otro punto distinto al anterior) se encuentran para estos datos en la figura 3.7. Dado que las métricas de evaluación arrojan los mismos resultados para los tres algoritmos, es indiferente el algoritmo que usemos para este ejemplo pues los tres dan el mismo resultado.El punto *X_test[1]* aparece representado en la gráfica como una estrella roja. Los cinco vecinos más cercanos son representados como diamantes rojos.

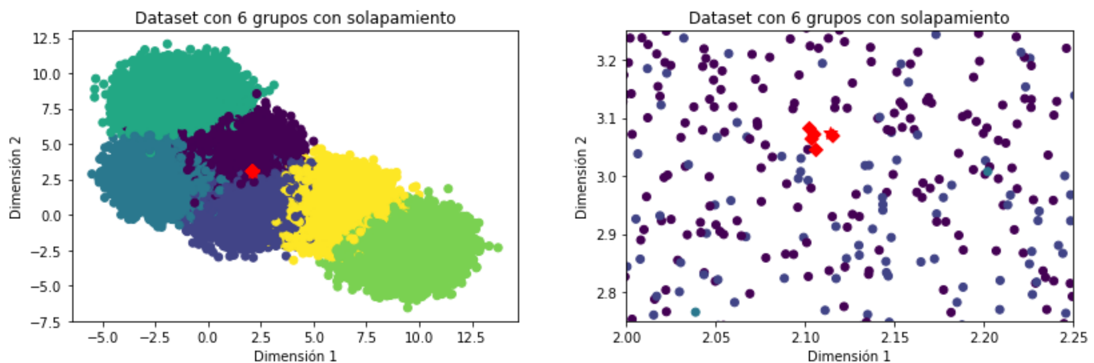


Figura 3.7: Cinco vecinos más cercanos de *X_test[1]*

En la figura 3.7 se ve como los diamantes son los puntos más cercanos a la estrella, por lo que si se encuentran los cinco vecinos más cercanos.

3.4.3. Resultados en GloVe

Para la comparación de los algoritmos en este dataset, solo vamos a tener en cuenta el tiempo de búsqueda de cada uno de ellos. Dado que en este conjunto de datos ya hay una gran cantidad de puntos, nos interesa hacer un estudio de como se comporta el tiempo de ejecución según el valor del *leaf_size* en Balltree. Si probamos valores de *leaf_size* desde 100 hasta 120000 (pues para un valor mayor de 120000 se construiría un árbol con una hoja y luego aplicaría fuerza bruta) con saltos de 5000, obtenemos los resultados de la gráfica 3.8.

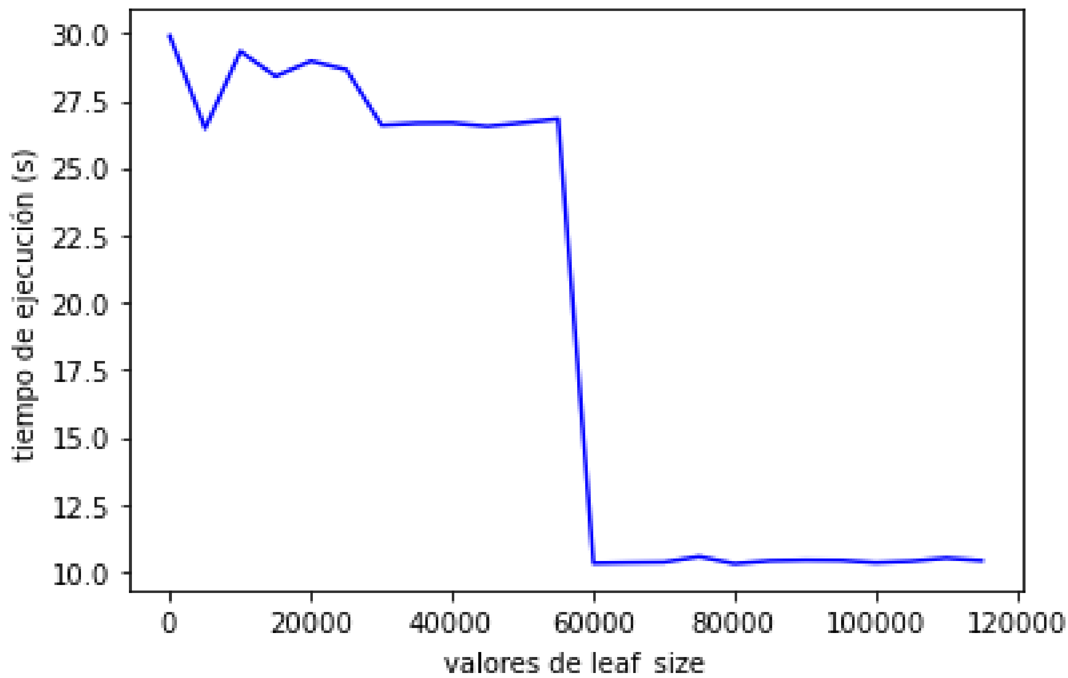


Figura 3.8: tiempo de ejecución en segundos frente al leaf_size

Como vemos en la gráfica 3.8, a partir de 50000 el tiempo se reduce considerablemente. Como vimos en el apartado 3.1.3, el número de elementos en las hojas cumple que $leaf_size \leq n_puntos \leq 2 * leaf_size$. Por lo que para este caso, tenemos que $50000 \leq n_puntos \leq 100000$. Es decir, habría solo dos hojas en el árbol. Se obtiene que el mejor valor es 80100. En la figura 3.8 también vemos que a más puntos en las hojas haya, y por tanto menos nodos, se tarda menos.

Ahora, probamos el tiempo de ejecución de cada algoritmo tomando como valores de k los números naturales impares entre 3 y 100 y 80100 como valor de *leaf_size*

para el Balltree. Los resultados obtenidos en esta prueba se muestran en la figura 3.9.

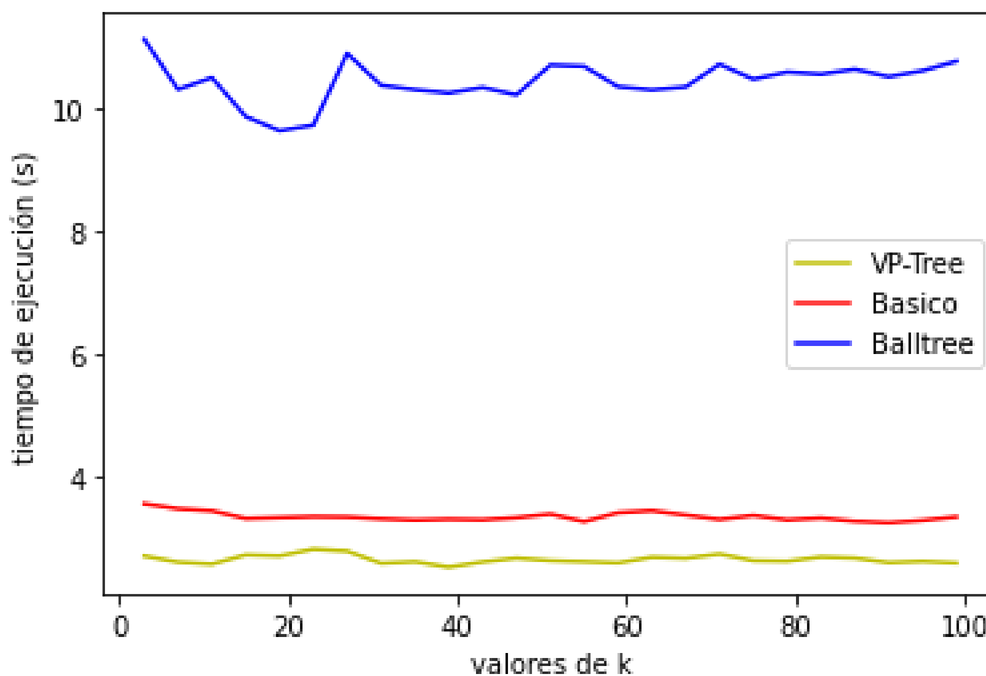


Figura 3.9: tiempo de ejecución segundos frente al valor de k

Como vemos en la Figura 3.9, en el dataset GloVe el algoritmo que más tiempo tarda con diferencia es el Balltree. En este caso el mejor va a ser el VP-Tree, aunque el rendimiento del algoritmo básico también es bueno.

3.4.4. Resultados en MNIST

Dado que en este conjunto de datos tenemos un gran cantidad de puntos (70000 puntos en total), sería interesante hacer un estudio de como se comporta el tiempo de ejecución del Balltree si variamos el valor del *leaf_size*. Para estos vamos a estudiar valores de *leaf_size* de 100 a 70000 con saltos de 5000. Se fija como valor máximo 70000 porque a partir de este valor el árbol tendría una única hoja y simplemente se aplicaría fuerza bruta. En la figura 3.10 se muestran los resultados de esta prueba.

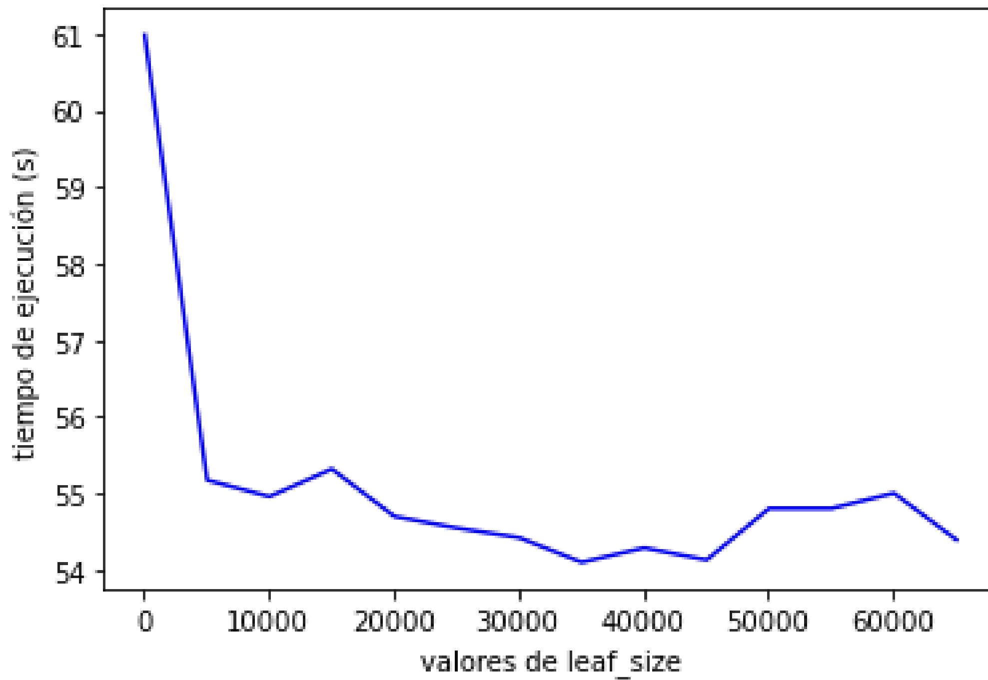


Figura 3.10: Tiempo de ejecución en segundos frente al leaf_size

En esta prueba obtenemos que el menor tiempo se alcanza en 35100, siendo este tiempo 54.099 segundos.

Una vez hallado el valor del *leaf_size* óptimo para este conjunto, ya podemos probar el tiempo de ejecución de cada algoritmo. Para ello vamos a medir el tiempo que tarda si k toma como valores los números naturales impares entre 3 y 100 y 35100 como valor de *leaf_size* para el Balltree. Los resultados obtenidos en esta prueba se muestran en la figura 3.11.

Como se puede ver en la figura 3.11, el Balltree es el que peor rinde de los tres con bastante diferencia. El mejor algoritmo en este caso es el algoritmo básico, pero el rendimiento del VP-Tree no está nada mal comparado al Balltree. En este conjunto de datos vemos que los algoritmos que usan una estructura de árbol tardan más que el algoritmo por fuerza bruta. Esto se puede deber a que al haber un número muy superior de dimensiones comparado con el resto de dataset, el tiempo de construcción del árbol tarda significativamente más.

Para la comparación de las métricas de evaluación, podemos encontrar el mejor valor de k para el accuracy probando valores de k impares entre 3 y 100. Si realizamos la gráfica del accuracy frente al tiempo, obtenemos la figura 3.12, en la que se puede ver que todos los algoritmos obtienen los mismos resultados.

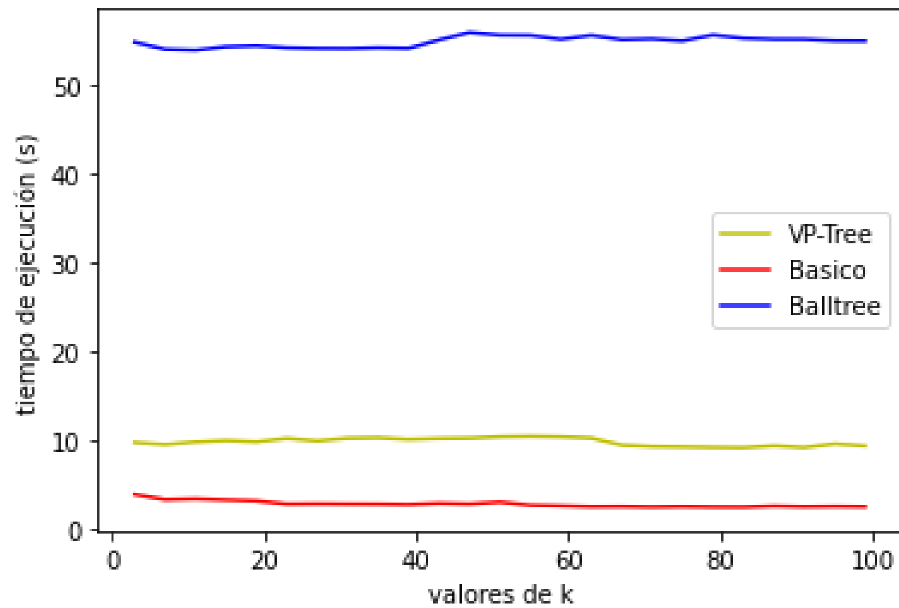


Figura 3.11: Tiempo de ejecución de cada algoritmo según la k

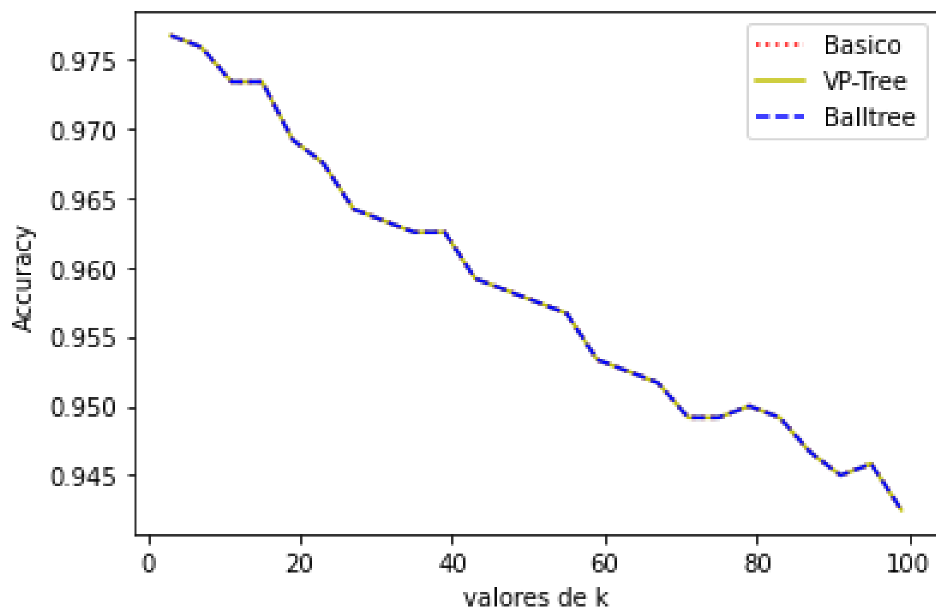


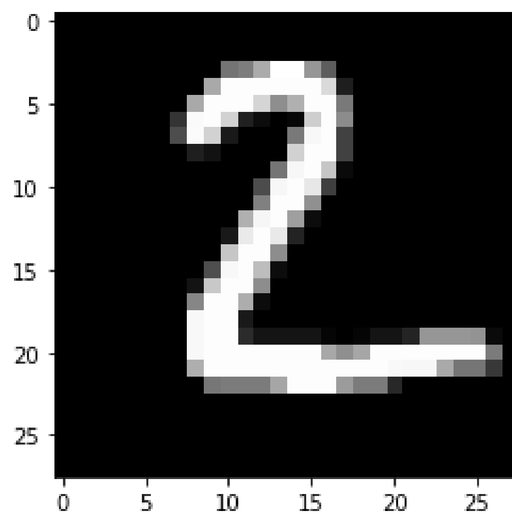
Figura 3.12: Accuracy frente al valor de k

El valor máximo de accuracy es 0.977 y se obtiene en k=3 en todos los algoritmos por igual, ya que, como se puede ver en la figura 3.12, los tres nos arrojan los mismos resultados. Hay que tener en cuenta que en la figura 3.12 el eje Y está acotado, por lo que la pendiente no sería tan pronunciada si no estuviera acotado el eje Y.

Para ver si KNN encuentra los puntos más cercanos a un punto dado del conjunto de forma efectiva, podemos pasar este punto de un vector a una imagen definiendo esta función que hace uso de la librería PyPlot:

```
def image_show(i, data):
    x = data[i] #guardar el punto en la variable x
    x = x.reshape((28,28)) #transfromar el vector en una matriz
   de 28x28
    plt.imshow(x, cmap='gray') #graficar con pyplot
```

Ahora, si tomamos, por ejemplo, el punto *X_test[1]* y lo graficamos obtenemos el siguiente dibujo:



Como podemos ver es un '2' dibujado a mano. Si encontramos los 3 vecinos mas cercanos, obtenemos que ocupan las posiciones 28882, 49160 y 24612 en *X_train* y los 3 tienen de etiqueta 2. Por tanto KNN entrenado con estos datos es capaz de reconocer dígitos escritos a mano.

4
Conclusiones

En este Trabajo de Fin de Grado se ha tratado de hacer una comparación entre tres tipos de algoritmos de búsqueda de vecinos más cercanos: el algoritmo básico (por 'fuerza bruta'), el algoritmo basado en un VP-Tree y el algoritmo basado en un Balltree.

El objetivo de esta comparación es que, dado un conjunto de datos, se pueda saber cual es la mejor opción entre estos tres algoritmos de búsqueda de vecinos más cercanos según la naturaleza del conjunto con el que se quiera trabajar.

Para ello, hemos realizado una serie de pruebas con conjuntos de pocas dimensiones (2 dimensiones) y conjuntos de más de 100 dimensiones para ver como se comporta cada algoritmo según la dimensionalidad de nuestros datos. Dado que cada conjunto tiene distinto número de datos, también hemos podido estudiar como se comporta cada algoritmo según el número de puntos. Estas pruebas se han realizado sobre cuatro conjuntos de datos de varios orígenes, desde conjuntos sintéticos hasta conjuntos de dígitos dibujados a mano.

Lo primero que hemos observado es que en cuanto a las métricas de evaluación de precisión, es que los tres algoritmos nos arrojan los mismos resultados. Esto es debido a que los algoritmos obtienen exactamente los mismos vecinos más cercanos a un punto dado. Ahora bien, no sabemos si tomando una distancia que no sea la habitual, es decir, la distancia euclídea, vamos a obtener los mismos resultados o va a haber diferencias entre los algoritmos. También hay que tener en cuenta que según la calidad de la implementación de cada algoritmo podríamos obtener distintos resultados, por lo que sería conveniente emplear implementaciones que sabemos con seguridad que funcionan como deberían.

Donde si hemos visto una diferencia notable es en la eficiencia computacional de cada algoritmo, es decir, en sus tiempos de ejecución. En primer lugar, vemos que si tenemos un número muy alto de dimensiones, unas 700 dimensiones en este caso, el algoritmo con el Balltree tarda demasiado comparado con los otros dos, por lo que lo recomendable en este caso es descartar por completo emplear este algoritmo. También vemos que para un número muy alto de dimensiones el algoritmo básico es el mejor. Esto puede ser porque el construir un árbol es muy costoso computacionalmente en muy altas dimensiones. Si tenemos alrededor de 100 dimensiones el mejor rendimiento nos lo va a dar el VP-Tree, aunque el algoritmo básico no lo hace tan mal comparado con el Balltree. En dimensiones bajas, vemos que el algoritmo por fuerza bruta es el peor. Respecto a los otros dos algoritmos, si tenemos muchos puntos el Balltree es mejor, pero con pocos puntos es mejor VP-Tree, pero no hay mucha diferencia.

En resumen, si nuestro conjunto tiene pocas dimensiones el algoritmo básico sería el peor en cuanto a eficiencia computacional y según el número de puntos podemos favorecer el VP-Tree o el Balltree, aunque no hay una diferencia significativa entre ellos. Si tenemos alrededor de 100 dimensiones el mejor rendimiento lo brinda el VP-Tree y el peor es con diferencia el Balltree aun con un valor de *leaf_size* adecuado. Si nuestro conjunto tiene un número muy alto de dimensiones, unas 700, el mejor es el algoritmo por fuerza bruta.

Algo que también hay que tener en cuenta, es que para estas implementaciones de Python, para el VP-Tree y el Balltree no podemos construir el árbol y buscar en este de forma independiente. Por tanto, si usamos cualquiera de estas dos estructuras de árbol en Python la complejidad va a ser de $O(n\ log(n))$. El algoritmo por fuerza bruta calcula todas las distancias de todos los puntos a un punto dado, $O(n)$, y luego las tiene que ordenar para encontrar las k distancias más pequeñas, esto es $O(n\ log(n))$ usando QuickSort o cualquier otro algoritmo de ordenación con complejidad $O(n\ log(n))$. Por tanto el algoritmo de KNN por fuerza bruta va a tener complejidad de $O(n\ log(n))$, al igual que los otros dos algoritmos. Es decir, en cuanto a complejidad algorítmica según el número de puntos, los tres algoritmos se comportan igual, al menos si se usan estas implementaciones de Python.

En este trabajo también se han visto algunos ejemplos de la importancia y las aplicaciones prácticas de este tipo de algoritmos y saber cual es el mejor algoritmo a emplear en cada caso nos ayudaría a usar este clasificador KNN tan potente de la manera más óptima.

5
Trabajos futuros

Un trabajo de investigación sobre algoritmos de clasificación supervisada podría ser muy extenso, pues es una área muy compleja y de gran interés. En este trabajo nos hemos centrado únicamente en tres algoritmos de búsqueda de vecinos más cercanos, por lo que sería interesante que en un futuro se estudiaran otros algoritmos de KNN. Por ejemplo, es posible buscar los vecinos más cercanos de un punto dado usando una estructura de árbol llamada KD Tree. En este árbol cada nodo que no sea una hoja tiene un hiper-plano asociado que particiona el espacio, de tal forma que los puntos a la izquierda de este hiper-plano pertenecen a un subárbol hijo y el resto al otro. Sería interesante hacer una comparación de este algoritmo con el resto pues la construcción del árbol tiene una complejidad algorítmica de $O(n\ log(n))$ y buscar dentro de este tiene una complejidad de $O(log(n))$. Además, ya existe una implementación para Python de este algoritmo en la librería *scikit-learn*. Teniendo en cuenta esto, podríamos esperar de este algoritmo un rendimiento similar al del VP-Tree. Otra ventaja de este algoritmo es que, a diferencia del Balltree, no tiene que hacer cálculos extra cuyo tiempo de ejecución dependa del número de dimensiones y lastren al algoritmo si trabajamos en altas dimensiones.

Otro estudio de interés sería el comparar estos algoritmos de clasificación supervisada con otros que no sean necesariamente de vecinos más cercanos, como SVMs, regresión logística, regresión lineal, árboles de decisión o redes neuronales.

También considero que las conclusiones de este trabajo se podrían afinar si dejamos de lado los conjuntos de datos 'reales' y usamos datos sintéticos. Aunque emplear datos del mundo real es ciertamente importante, pero encontrar un conjunto de datos con las dimensiones y número de puntos que queramos es muy

difícil y en muchos casos imposible. En cambio, con datos sintéticos podemos generar exactamente el conjunto que queremos.

Bibliografía

[1] Algoritmo de k vecinos más cercanos. https://www.ibm.com/es-es/topics/knn. Accedido: 05/05/2023.

[2] Movie recommendation and rating prediction using k-nearest neighbors. https://www.analyticsvidhya.com/blog/2020/08/recommendation-system-k-nearest-neighbors/, 2020. Accedido: 05/05/2023.

[3] Ricardo Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In *CPM*, volume 94, pages 198–212. Citeseer, 1994.

[4] Edgar Chávez, José L Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14:113–135, 2001.

[5] Vlastislav Dohnal. Indexing structures for searching in metric spaces. *Ph. D Thesis, Masaryk University*, 2004.

[6] Roberto Díaz. Algoritmo knn – cómo funciona y ejemplos en python. https://www.themachinelearners.com/algoritmo-knn/. Accedido: 05/05/2023.

[7] Hiep Xuan Huynh, Nhung Cam Thi Mai, and Hai Thanh Nguyen. Balltree similarity: A novel space partition approach for collaborative recommender systems. In Cong Vinh Phan and Thanh Dung Nguyen, editors, *Context-Aware Systems and Applications*, pages 112–128, Cham, 2023. Springer Nature Switzerland.

[8] Marco La Cascia, Saratendu Sethi, and Stan Sclaroff. Combining textual and visual cues for content-based image retrieval on the world wide web. In *Proceedings. ieee workshop on content-based access of image and video libraries (cat. no. 98ex173)*, pages 24–28. IEEE, 1998.

[9] Samuel Sangkon Lee, Masami Shishibori, and Chia Y Han. An improvement video search method for vp-tree by using a trigonometric inequality. *Journal of Information Processing Systems*, 9(2):315–332, 2013.

[10] Xianxian Li, Jie Lei, Zhenkui Shi, and Feng Yu. An efficient and accurate encrypted image retrieval scheme via ball tree. In *2022 8th International Conference on Big Data Computing and Communications (BigCom)*, pages 365–371, 2022.

[11] Antonio Alcalá Malavé. *Genética de la emoción: El origen de la enfermedad*. Ediciones B, 08009 Barcelona (España), 2015.

[12] Il'ya Markov. Vp-tree: Content-based image indexing. In *Proceedings of the Spring Young Researcher's*, volume 7, page 00268a, 2007.

[13] Seyyid Ahmed Medjahed, Tamazouzt Ait Saadi, and Abdelkader Benyettou. Breast cancer diagnosis by using k-nearest neighbor with different distances and classification rules. *International Journal of Computer Applications*, 62(1), 2013.

[14] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[15] Andrew Moore, Alexander Gray, and Ting liu. New algorithms for efficient high dimensional non-parametric classification. *Advances in Neural Information Processing Systems*, 16, 2003.

[16] Gonzalo Navarro, Ricardo Baeza-Yates, and J Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[17] Stephen M Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] Michael Skalak, Jinyu Han, and Bryan Pardo. Speeding melody search with vantage point trees. In *ISMIR*, pages 95–100, 2008.

[20] Sai Sumathi and SN Sivanandam. *Introduction to data mining and its applications*, volume 29. Springer, 2006.

[21] Jeffrey K. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[22] Ida Vainionpää and Sophie Davidsson. Stock market prediction using the k nearestneighbours algorithm and a comparison withthe moving average formula, 2014.

[23] Weiguo Wan and Hyo Jong Lee. Deep feature representation and ball-tree for face sketch recognition. *International Journal of System Assurance Engineering and Management*, 11:818–823, 2020.

[24] Ian H Witten, Eibe Frank, Mark A Hall, Christopher J Pal, and MINING DATA. Practical machine learning tools and techniques. In *Data Mining*, volume 2, 2005.

[25] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, page 311–321, USA, 1993. Society for Industrial and Applied Mathematics.

Apéndices

A

Códigos

A.1. Seis nubes sin solape

A.1.1. Generar conjunto y manipulación de los datos

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score,
    recall_score,f1_score
from sklearn.datasets import make_blobs
import time
import nmslib

# Generar el dataset sin solapamiento
np.random.seed(42)

n_samples = 1000  # Numero de puntos por grupo
n_groups = 6  # Numero de grupos
n_features = 2  # Numero de dimensiones
centers = np.array([[-4, 8], [4, 8], [-4, -8], [4, -8], [-4, 0],
    [4, 0]])

X, y = make_blobs(n_samples=n_samples * n_groups, n_features=
    n_features, centers=centers)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=1200 random_state=0)
```

A.1.2. Visualización de los datos

Ver todos los puntos

```
# Dibujar el dataset
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.xlabel('Dimensi n 1')
plt.ylabel('Dimensi n 2')
plt.title('Dataset con 6 grupos sin solapamiento')
plt.show()
```

Ver el conjunto con un punto y sus vecinos resaltados

```
index = nmslib.init(method='vptree', space='l2') #buscamos los vecinos de X_test[1x]
index.addDataPointBatch(X_train)
index.createIndex()
indices = index.knnQueryBatch([X_test[1]], k=5)
Xg=[]
Yg=[]
for i in range(5):
    Xg.append(X_train[indices[0][0]][i][0])
    Yg.append(X_train[indices[0][0]][i][1])
print(indices[0][0])


s=50
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, marker='o')
plt.plot(X_test[1][0],X_test[1][1], marker='*', color='red', markersize=10)
plt.scatter(Xg,Yg, marker='D', color='red',sizes=[s,s,s,s,s])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Dataset con 6 grupos sin solapamiento')
plt.show()
```

Para ampliar en la zona de interés:

```
s=50
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, marker='o')
plt.plot(X_test[1][0],X_test[1][1], marker='*', color='red', markersize=10)
plt.scatter(Xg,Yg, marker='D', color='red',sizes=[s,s,s,s,s])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Dataset con 6 grupos sin solapamiento')
plt.axis([-6, -4, 7, 9])
plt.show()
```

A.1.3. KNN por fuerza bruta

```
tiempos_basico=[]
accuracy_basico=[]
k=range(3,100,2)

for i in k:
    start_time = time.time()
    knn = KNeighborsClassifier(n_neighbors=i, algorithm='brute',
   metric='euclidean')

    # Entrenar el clasificador KNN con el conjunto de
   entrenamiento
    knn.fit(X_train, y_train)


    # Realizar las predicciones para el conjunto de prueba
    y_pred = knn.predict(X_test)
    tiempos_basico.append(time.time() - start_time)
    accuracy_basico.append(accuracy_score(y_test, y_pred))

print(k[np.argmax(accuracy_basico)])
print(max(accuracy_basico))
```

A.1.4. KNN con VP-Tree

```
import nmslib

accuracy_vp=[]
tiempos_vp=[]
k=range(3,100,2)
for i in k:
    start_time=time.time()
    index = nmslib.init(method='vptree', space='l2')
    index.addDataPointBatch(X_train)
    index.createIndex()
    # Query the VPTree for nearest neighbors

    vp_tree_indices = index.knnQueryBatch(X_test, k=i) # Only
   retrieve the indices
    tiempos_vp.append(time.time()-start_time)

    y_pred = []
    for x in vp_tree_indices:
        unique_labels, counts = np.unique(y_train[x[0]],
   return_counts=True)
        pred_label = unique_labels[np.argmax(counts)]
        y_pred.append(pred_label)
    accuracy_vp.append(accuracy_score(y_test, y_pred))

print(k[np.argmax(accuracy_vp)])
print(max(accuracy_vp))
```

A.1.5. KNN con Balltree

```
import nmslib
import numpy as np

accuracy_balltree=[]
tiempos_balltree=[]
k=range(3,100,2)
for i in k:
    start_time=time.time()
    knn = KNeighborsClassifier(n_neighbors=i, algorithm='ball_tree', metric='euclidean')

    # Entrenar el clasificador KNN con el conjunto de entrenamiento
    knn.fit(X_train, y_train)

    # Realizar las predicciones para el conjunto de prueba
    y_pred = knn.predict(X_test)

    tiempos_balltree.append(time.time()-start_time)
    accuracy_balltree.append(accuracy_score(y_test, y_pred))

print(k[np.argmax(accuracy_balltree)])
print(max(accuracy_balltree))
```

A.1.6. Gráfica de tiempos

```
plt.plot(k, tiempos_basico, color='r', label='B sico')
plt.plot(k, tiempos_vp, color='y', label='VP-Tree')
plt.plot(k, tiempos_balltree, color='b', label='Balltree')



# nombramos el eje x
plt.xlabel('valores de k')
# nombremos el eje y
plt.ylabel('tiempo de ejecucion (s)')
#ponemos la leyenda
plt.legend()
#mostramos la grafica
plt.show()
```

A.2. Seis nubes con solape

A.2.1. Generar conjunto y manipulación de los datos

```
import numpy as np
import matplotlib.pyplot as plt
```

```python
from sklearn.neighbors import KNeighborsClassifier, BallTree,
    DistanceMetric

from sklearn.model_selection import train_test_split
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
import time


# Generar datos de ejemplo
np.random.seed(0)
n_samples = 50000  # N mero de puntos por grupo
n_groups = 6  # N mero de grupos
n_features = 2  # N mero de dimensiones
X,y = make_blobs(n_samples=n_samples * n_groups, n_features=
    n_features, centers=n_groups)

# Dividir los datos en conjunto de entrenamiento y conjunto de
    prueba
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=1200 random_state=0)
```

A.2.2. Visualización de los datos

Ver todos los puntos

```python
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Dataset con 6 grupos con solapamiento')
plt.show()
```

Ver el conjunto con un punto y sus vecinos resaltados

```python
index = nmslib.init(method='vptree', space='l2')
index.addDataPointBatch(X_train)
index.createIndex()
indices = index.knnQueryBatch([X_test[1]], k=5)
Xg=[]
Yg=[]
for i in range(5):
    Xg.append(X_train[indices[0][0]][i][0])
    Yg.append(X_train[indices[0][0]][i][1])

s=50
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, marker='o')
plt.plot(X_test[1][0],X_test[1][1], marker='*', color='red',
    markersize=10)
plt.scatter(Xg,Yg, marker='D', color='red',sizes=[s,s,s,s,s])
plt.xlabel('Dimension 1')
```

```
plt.ylabel('Dimension 2')
plt.title('Dataset con 6 grupos con solapamiento')
plt.show()
```

Para ampliar en la zona de interés:

```
s=50
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, marker='o')
plt.plot(X_test[1][0],X_test[1][1], marker='*', color='red',
    markersize=10)
plt.scatter(Xg,Yg, marker='D', color='red',sizes=[s,s,s,s,s])
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.title('Dataset con 6 grupos con solapamiento')
plt.axis([2,2.2,3,3.2])

plt.show()
```

A.2.3. KNN por fuerza bruta

```
tiempos_basico=[]
accuracy_basico=[]
k=range(3,100,2)

for i in k:
    start_time = time.time()
    knn = KNeighborsClassifier(n_neighbors=i, algorithm='brute',
    metric='euclidean')

    # Entrenar el clasificador KNN con el conjunto de
    entrenamiento
    knn.fit(X_train, y_train)

    # Realizar las predicciones para el conjunto de prueba
    y_pred = knn.predict(X_test)
    tiempos_basico.append(time.time() - start_time)
    accuracy_basico.append(accuracy_score(y_test, y_pred))

print(k[np.argmax(accuracy_basico)])
print(max(accuracy_basico))
```

A.2.4. KNN con VP-Tree

```
import nmslib

accuracy_vp=[]
tiempos_vp=[]
k=range(3,100,2)
for i in k:
    start_time=time.time()
    index = nmslib.init(method='vptree', space='l2')
```

```python
    index.addDataPointBatch(X_train)
    index.createIndex()
    # Query the VPTree for nearest neighbors

    vp_tree_indices = index.knnQueryBatch(X_test, k=i) # Only retrieve the indices
    tiempos_vp.append(time.time()-start_time)

    y_pred = []
    for x in vp_tree_indices:
        unique_labels, counts = np.unique(y_train[x[0]], return_counts=True)
        pred_label = unique_labels[np.argmax(counts)]
        y_pred.append(pred_label)
    accuracy_vp.append(accuracy_score(y_test, y_pred))

print(k[np.argmax(accuracy_vp)])
print(max(accuracy_vp))
```

A.2.5. KNN con Balltree

```python
import nmslib
import numpy as np

accuracy_balltree=[]
tiempos_balltree=[]
k=range(3,100,2)
for i in k:
    start_time=time.time()
    knn = KNeighborsClassifier(n_neighbors=i, algorithm='ball_tree', metric='euclidean')

    # Entrenar el clasificador KNN con el conjunto de entrenamiento
    knn.fit(X_train, y_train)

    # Realizar las predicciones para el conjunto de prueba
    y_pred = knn.predict(X_test)

    tiempos_balltree.append(time.time()-start_time)
    accuracy_balltree.append(accuracy_score(y_test, y_pred))

print(k[np.argmax(accuracy_balltree)])
print(max(accuracy_balltree))
```

A.2.6. Gráfica de tiempos

```python
plt.plot(k, tiempos_basico, color='r', label='B sico')
plt.plot(k, tiempos_vp, color='y', label='VP-Tree')
plt.plot(k, tiempos_balltree, color='b', label='Balltree')
```

```
# nombramos el eje x
plt.xlabel('valores de k')
# nombremos el eje y
plt.ylabel('tiempo de ejecucion (s)')
#ponemos la leyenda
plt.legend()
#mostramos la grafica
plt.show()
```

A.2.7. Gráfica del accuracy

```
plt.plot(k, accuracy_basico, color='r',linestyle='dotted', label=
    'Basico')
plt.plot(k, accuracy_vp, color='y', label='VP-Tree')
plt.plot(k, accuracy_balltree, linestyle='dashed', color='b',
    label='Balltree')



# nombramos el eje x
plt.xlabel('valores de k')
# nombremos el eje y
plt.ylabel('Accuracy')
#ponemos la leyenda
plt.legend()
#mostramos la grafica
plt.show()
```

A.3. GloVe

A.3.1. Cargar y manipular datos

```
import pandas as pd
import csv
from sklearn.neighbors import BallTree, DistanceMetric
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score,
    recall_score
from sklearn.neighbors import NearestNeighbors
import time
import matplotlib.pyplot as plt
from random import sample

words = pd.read_table('glove.6B.100d.txt', sep=" ", index_col=0,
    header=None, quoting=csv.QUOTE_NONE)

# list(data_top) or
```

```
samp=120000
words=words.sample(n=samp, random_state=1)
y=list(words.index)
X=words.to_numpy()
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=1200, random_state=42)
```

A.3.2. KNN por fuerza bruta

```
tiempos_basico=[]
k=range(3,100,2)

for i in k:
    start_time = time.time()
    nbrs = NearestNeighbors(n_neighbors=i, algorithm='brute').fit
    (X_train)

    # Realizar las predicciones para el conjunto de prueba
    distances, indices = nbrs.kneighbors(X_test, n_neighbors=i)
    tiempos_basico.append(time.time() - start_time)
```

A.3.3. KNN con VP-Tree

```
import nmslib
import numpy as np

tiempos_vp=[]
k=range(3,100,2)
for i in k:
    start_time = time.time()
    index = nmslib.init(method='vptree', space='l2')
    index.addDataPointBatch(X_train)
    index.createIndex()

    # Query the VPTree for nearest neighbors
    vp_tree_indices = index.knnQueryBatch(X_test, k=i) # Only
    retrieve the indices
    tiempos_vp.append(time.time() - start_time)
```

A.3.4. KNN con Balltree

Estudio según el leaf_size

```
tiempos_ball_ls=[]
ls=range(100,120001,5000)
from sklearn.neighbors import BallTree

for i in ls:
    start_time = time.time()
```

```
    nbrs=BallTree(X_train, leaf_size=i, metric='euclidean')

    # Realizar las predicciones para el conjunto de prueba
    indices = nbrs.query(X_test, k=1)
    tiempos_ball_ls.append(time.time() - start_time)

plt.plot(ls, tiempos_ball_ls, color='b')
plt.xlabel('valores de leaf_size')
plt.ylabel('tiempo de ejecuci n (s)')
plt.show()
print(ls[np.argmin(tiempos_ball_ls)])
print(min(tiempos_ball_ls))
```

KNN

```
tiempos_balltree=[]
k=range(3,100,2)
for i in k:
    start_time = time.time()
    nbrs=BallTree(X_train, leaf_size=80100, metric='euclidean')

    # Realizar las predicciones para el conjunto de prueba
    indices = nbrs.query(X_test, k=i)
    tiempos_balltree.append(time.time() - start_time)
```

A.3.5. Gráfica de tiempos

```
plt.plot(k, tiempos_basico, color='r', label='B sico')
plt.plot(k, tiempos_vp, color='y', label='VP-Tree')
plt.plot(k, tiempos_balltree, color='b', label='Balltree')

# nombramos el eje x
plt.xlabel('valores de k')
# nombremos el eje y
plt.ylabel('tiempo de ejecucion (s)')
#ponemos la leyenda
plt.legend()
#mostramos la grafica
plt.show()
```

A.4. MNIST

A.4.1. Cargamos los datos, manipulamos los datos y hacemos comprobaciones

```
from tensorflow.keras.datasets import mnist
import numpy as np
```

```
3 from sklearn.neighbors import KNeighborsClassifier
4 from sklearn.metrics import accuracy_score, precision_score,
    recall_score, f1_score
5 import time
6 import matplotlib.pyplot as plt
7 from sklearn.neighbors import BallTree
8
9 #cargar el dataset
10 (X_train, y_train), (X_test, y_test) = mnist.load_data()
11
12 #Comprobamos dimensiones de los datos
13 print('X_train: ' + str(X_train.shape))
14 print('Y_train: ' + str(y_train.shape))
15 print('X_test:  '  + str(X_test.shape))
16 print('Y_test:  '  + str(y_test.shape))
17 #comprobamos que estan hecha la conversion corectamente
18 X_train=X_train.reshape(60000,-1)
19 X_test=X_test.reshape(10000,-1)
20 print('X_train: ' + str(X_train.shape))
21 print('X_test:  '  + str(X_test.shape))
22
23 #Separamos correctamente train y test
24 X=np.concatenate((X_train,X_test))
25 y=np.concatenate((y_train,y_test))
26 X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=1200, random_state=42)
```

A.4.2. Visualización de los datos

```
1 import nmslib
2 import numpy as np
3
4 def image_show(i, data): #definimos la funci n para ver los
    datos
5     x = data[i] # obtener el vector
6     x = x.reshape((28,28)) # convertirlo a matriz 28x28
7     plt.imshow(x, cmap='gray')
8
9 image_show(1,X_test)
10
11 index = nmslib.init(method='vptree', space='l2')
12 index.addDataPointBatch(X_train)
13 index.createIndex()
14
15 # Busqueda de los vecinos m s cercanos
16 vp_tree_indices = index.knnQueryBatch([X_test[1]], k=3) #Obtener
    solo los indices
17 print(vp_tree_indices[0][0])
18 print(y_train[vp_tree_indices[0][0]])
```

A.4.3. KNN por fuerza bruta

```
accuracy_basico=[]
tiempos_basico=[]
k=range(3,100,2)

for i in k:
    start_time=time.time()
    knn = KNeighborsClassifier(n_neighbors=i, algorithm='brute', metric='euclidean')
    # Entrenar el clasificador KNN con el conjunto de entrenamiento
    knn.fit(X_train, y_train)
    # Realizar las predicciones para el conjunto de prueba
    y_pred = knn.predict(X_test)
    tiempos_basico.append(time.time() - start_time)
    accuracy_basico.append(accuracy_score(y_test, y_pred))
```

A.4.4. KNN con VP-Tree

```
import nmslib

tiempos_vp=[]
accuracy_vp=[]
k=range(3,100,2)

for i in k:
    start_time = time.time()
    index = nmslib.init(method='vptree', space='l2')
    index.addDataPointBatch(X_train)
    index.createIndex()

    # Busqueda de los vecinos mas cercanos con VP-Tree
    vp_tree_indices = index.knnQueryBatch(X_test, k=i) #obtener solo los indices
    tiempos_vp.append(time.time() - start_time)

    y_pred = []
    for x in vp_tree_indices:
        unique_labels, counts = np.unique(y_train[x[0]], return_counts=True)
        pred_label = unique_labels[np.argmax(counts)]
        y_pred.append(pred_label)
    accuracy_vp.append(accuracy_score(y_test, y_pred))
```

A.4.5. KNN con Balltree

Estudio según el leaf size

```
tiempos_ball_ls=[]
```

```
ls=range(100,70001,5000)
from sklearn.neighbors import BallTree

for i in ls:
    start_time = time.time()
    nbrs=BallTree(X_train, leaf_size=i, metric='euclidean')

    # Realizar las predicciones para el conjunto de prueba
    indices = nbrs.query(X_test, k=1)
    tiempos_ball_ls.append(time.time() - start_time)

plt.plot(ls, tiempos_ball_ls, color='b')
plt.xlabel('valores de leaf_size')
plt.ylabel('tiempo de ejecuci n (s)')
plt.show()
print(ls[np.argmin(tiempos_ball_ls)])
print(min(tiempos_ball_ls))
```

KNN

```
tiempos_balltree=[]
accuracy_balltree=[]
k=range(3,100,2)

for i in k:
    start_time = time.time()
    nbrs=BallTree(X_train, leaf_size=35100, metric='euclidean')

    # Realizar las predicciones para el conjunto de prueba
    indices = nbrs.query(X_test, k=i)
    tiempos_balltree.append(time.time() - start_time)
    accuracy_balltree.append(accuracy_score(y_test, y_pred))
```

A.4.6. Gráfica de tiempos

```
plt.plot(k, tiempos_basico, color='r', label='B sico')
plt.plot(k, tiempos_vp, color='y', label='VP-Tree')
plt.plot(k, tiempos_balltree, color='b', label='Balltree')

# nombramos el eje x
plt.xlabel('valores de k')
# nombremos el eje y
plt.ylabel('tiempo de ejecucion (s)')
#ponemos la leyenda
plt.legend()
#mostramos la grafica
plt.show()
```

A.4.7. Gráfica del accuracy

```
plt.plot(k, accuracy_basico, color='r',linestyle='dotted', label='Basico')
plt.plot(k, accuracy_vp, color='y', label='VP-Tree')
plt.plot(k, accuracy_balltree, linestyle='dashed', color='b', label='Balltree')


# nombramos el eje x
plt.xlabel('valores de k')
# nombremos el eje y
plt.ylabel('Accuracy')
#ponemos la leyenda
plt.legend()
#mostramos la grafica
plt.show()
```