Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
2019- 1

Tarea 1

Fecha de entrega código: Miércoles 24 de abril
Fecha de entrega informe: Viernes 26 de abril

Objetivos

- Investigar e implementar una estructura de datos nueva
- Aprender de algoritmos de clustering y clasificación
- Hacer un análisis los parámetros de tu programa

Introducción

Para esta tarea tendrán la oportunidad de optimizar un algoritmo de clasificación utilizado el llamado *KNN* o *k nearest neighbours*. Un problema de clasificación consiste en asignar una etiqueta de un conjunto de etiquetas a un objeto dadas su propiedades. En este caso trabajaremos con vectores de 2 dimensiones y los clasificaremos con una etiqueta numérica que indica la clase a la que pertenecen. Para tener cierta noción de la etiqueta de un vector se utiliza una base de datos de vectores ya etiquetados. A esta base de datos se le llama "set de entrenamiento".

KNN

El objetivo de este algoritmo consiste en elegir la etiqueta de un vector v. Para esto se buscan los k vectores más cercanos a v en el set de entrenamiento, los cuales se denominan la **vecindad** de v. Luego se determina la etiqueta ponderando las etiquetas de la vecindad según su distancia.

Por ejemplo en la Figura 1 se muestra el punto a etiquetar de color negro y se busca su etiqueta con una vecindad de tamaño 4 representada por el círculo negro. Dado que tiene más vecinos azules y más cercanos, se clasifica el vector como de color azul.

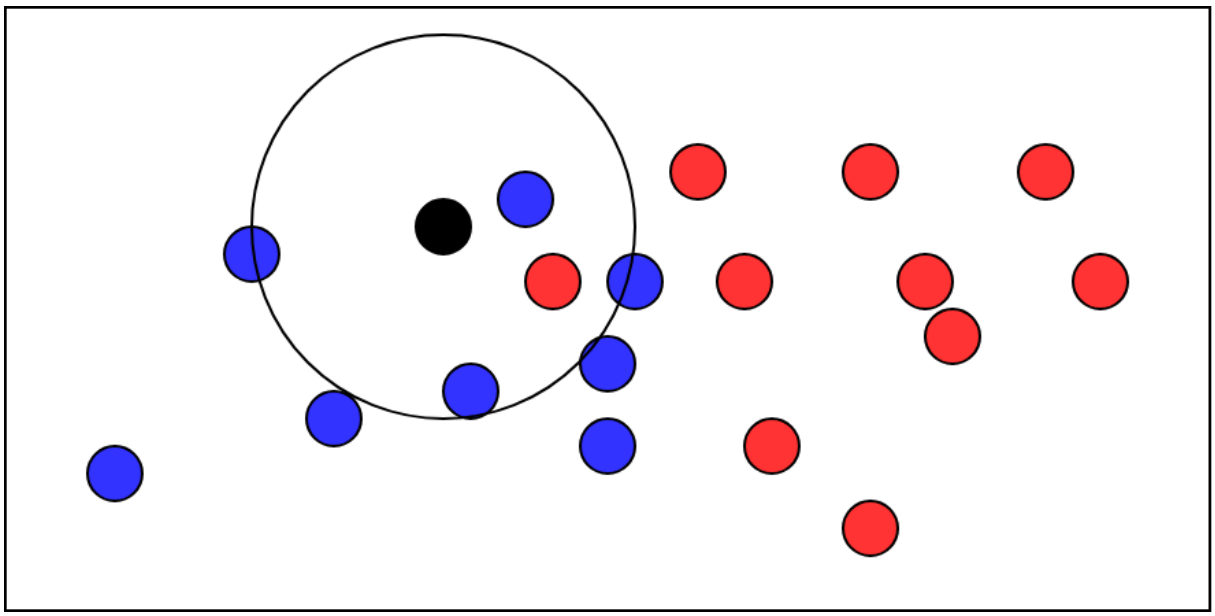

Figura 1: Vecindad del punto negro

Implementación

La implementación básica de KNN recorre todo el set de entrenamiento para encontrar los k vecinos más cercanos del vector a consultar. En general se busca clasificar un set completo de vectores llamado set de test por lo que la complejidad completa del algoritmo es $O(|train| \cdot |test| \cdot k)$. En el uso realista del algoritmo se usan en el orden cientos de miles de datos de entrenamiento y de test y se prueba el valor de k de manera empírica por lo que se ejecuta muchas veces con distintos valores de k.

Es por esto que tu trabajo para esta tarea es optimizar este algoritmo utilizando estructuras de datos. Para esto debes investigar sobre la estructura de datos **KD-Tree** para almacenar el set de entrenamiento de manera de acelerar la búsqueda de los vecinos cercanos. También es de utilidad el uso de un heap para almacenar la vecindad de un vector durante la búsqueda.

KD-Tree

Para hacer una solución eficiente es necesario considerar que la complejidad esta mayoritariamente dominada por la cantidad de muestras de entrenamiento, ya que normalmente $|train| > |test|$. Esto significa que podemos invertir tiempo preprocesando los datos de entrenamiento al iniciar el programa para poder hacer más eficiente las operaciones de búsqueda de los k vecinos más cercanos.

Una estructura de datos que permite hacer eficientemente esto es un **KD-Tree** o **K-Dimentional Tree**. Este es un árbol binario que agrupa los datos según posiciones en un espacio de **K** dimensiones (no confundir K del número de dimensiones con k el número de vecinos). En este caso nos interesa dividir los núcleos en un espacio de 2 dimensiones por lo que nuestra estructura será un **2D-Tree**.

Cada nodo del **KD-Tree** representa una caja **K**-dimensional que contiene distintos vectores de entrenamiento. Los hijos de un nodo también son cajas que están separadas por un plano divisor y están contenidas dentro del nodo padre y contienen a todos los núcleos del padre.

Por ejemplo en la Figura 2 se muestra un **KD-Tree** con 14 puntos en el cual el nodo A es la caja de mayor tamaño y que contiene a todos los puntos. La caja A está dividida por un plano que separa la caja B de la caja C. Es importante notar que las cajas B y C están contenidas en la caja A y no se traslapan entre si.

Finalmente las hojas del árbol representan las cajas más pequeñas y son las que almacenan realmente los vectores.

Al igual que en un árbol de búsqueda binaria, es posible buscar un punto que pertenece al árbol recorriendo los nodos según a que lado del plano divisor está el punto que buscamos.

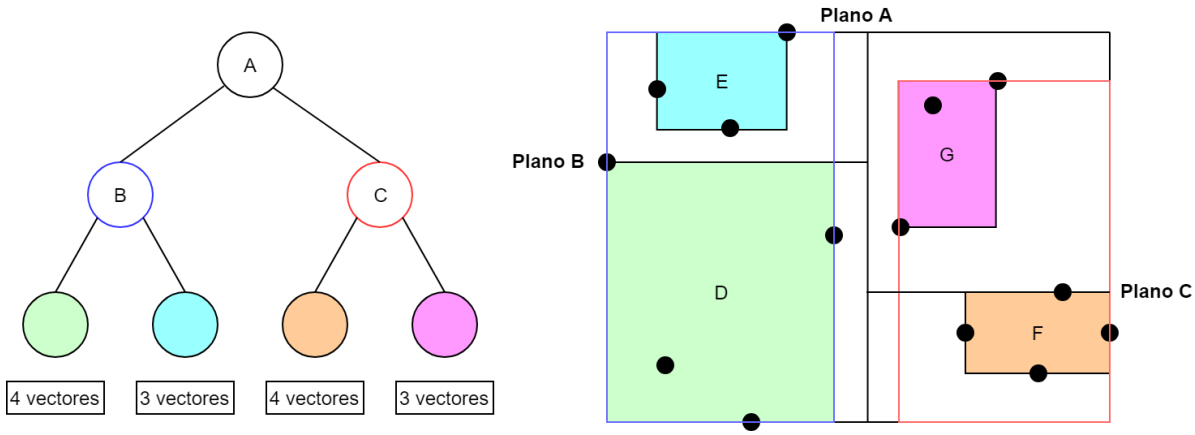

Figura 2: **KD-Tree** de 2 dimensiones y su representación en el plano

También es posible buscar los vectores más cercanos a un vector nuevo sin tener que recorrer todo el árbol. Esto permite disminuir la complejidad del algoritmo KNN considerablemente.

Tu deber es investigar esta estructura e implementarla para hacer el proceso mas eficiente. Se recomienda buscar sobre su uso en búsqueda de vecinos mas cercanos.

Programa

El código base que se les da en esta tarea está hecho de manera de que el **KDTree** y el algoritmo **KNN** sean una librería del programa principal. Esto quiere decir que tu debes implementar las funciones de esta librería para que **KNN** funcione correctamente y eficientemente.

En particular debes implementar 3 funciones que son llamadas por el programa principal:

- `kd_init(train)`: construye el kdtree a partir del set de entrenamiento.
- `knn(neighbours, kdtree, train, objective, k)`: Busca los k vecinos más cercanos al vector `objective` del set de entrenamiento usando la estructura `kdtree` y almacena los vectores en el arreglo `neighbours`.
- `kd_destroy(kdtree)`: Libera todos los recursos utilizados por el `kdtree` para que no hayan memory leaks.

Usando estas funciones el programa principal clasifica los vectores del set de test y calcula el porcentaje de vectores bien clasificados. OJO: usar el método ineficiente y el método con **KD-Tree** debería dar el mismo porcentaje de acierto pero en un menor tiempo.

También tienes disponibles ciertas funciones que podrías necesitar para la implementación del KDTree:

- `distance(v1, v2)`: Calcula la distancia entre los vectores $v1$ y $v2$.
- `collision(cx, cy, r, mx, my, Mx, My)`: Retorna True si la circunferencia centrada en (cx, cy) y de radio r colisiona con la caja rectangular definida por las esquinas (mx, my) y (Mx, My).

Input

El programa se ejecuta con el siguiente comando:

```
./classifier <train.txt> <test.txt> <k> <l>
```

Donde los archivos `train.txt` y `test.txt` contienen los set de entrenamiento y test respectivamente, k es el tamaño de la vecindad a buscar y l es el número de etiquetas distintas que existen en los datos.

Output

Si el programa funciona correctamente debería imprimir el porcentaje de aciertos obtenido con el algoritmo de clasificación.

Interfaz

Para ayudarte a debuguear tu programa se agregó al código una interfaz gráfica llamada *watcher* que permite visualizar puntos, líneas y círculos. Su uso es opcional pero podría ser de mucha ayuda. Revisa sus funciones en `src/watcher/watcher.h`

OJO: No debes subir tu código con funciones de la interfaz ya que podrían detener tu programa o hacerlo más lento. Esto perjudicaría la corrección automática del programa.

Informe

Deberás entregar un informe donde hagas un análisis teórico y empírico de tu solución, en particular, se espera lo siguiente:

Análisis teórico

Discuta teóricamente la complejidad en tiempo que debería tomar el algoritmo de KNN usando un **KD-Tree** con respecto a las variables $|train|$, $|test|$ y k.

Análisis empírico

Debes probar tu código con distintos valores de $|train|$, $|test|$, $|k|$ y discutir cómo se comporta el tiempo de ejecución con respecto a los parámetros. Para efectos prácticos puedes mantener 2 variables fijas mientras pruebas la complejidad de la tercera.

Debes justificar a partir de los resultados obtenidos si se logró cumplir con la complejidad estimada en el análisis teórico.

Evaluación

Tu código será probado con un set de entrenamiento y test distintos a los entregados pero del mismo tamaño. Luego se ejecutará un programa que usa tus funciones para guardar los arreglos de vectores más cercanos en un archivo. Una vez terminado tu programa o pasado 10 segundos se verificará el porcentaje de correctitud de los vecinos cercanos entregados.

Por ejemplo si se consultan 6000 vectores y tu programa solo alcanza a terminar 5000 en 10 segundos, de los cuales 4500 son correctos tu nota sería 5.5. En los próximos días se entregará un programa que les ayudará a precorregir su tarea.

La nota de tu tarea se descompone como se detalla a continuación:

- 70% A la nota de tu código
- 30% A la nota de tu informe

Entrega

- **Código:** GIT - Repositorio asignado.
  - En la raiz del repositorio debe encontrarse una makefile y la carpeta src.
  - Se recogerá el estado en la rama *master*.
  - Se espera que el código compile con `make` en la raiz del repositorio y genere un ejecutable de nombre `classifier` en esa misma carpeta.
- **Informe:** SIDING - En la encuesta correspondiente, en formato PDF
- **Hora Límite:** 1 minuto pasadas las 23:59 del día de la entrega.
- No se permitirán entregas atrasadas.

Bonus

Los bonus solo se harán efectivos si la nota correspondiente es superior a 3.9.

Buen uso del espacio y del formato (+5% a la nota de *Informe*)

La nota de tu informe aumentará en un 5% si tu informe, a criterio del corrector, está bien presentado y usa el espacio y formato a favor de entregar la información.

Manejo de memoria perfecto (+5% a la nota de *Código*)

Se aplicará este bonus si **valgrind** reporta en tu código 0 leaks y 0 errores de memoria, considerando que tu programa haga lo que tiene que hacer.