Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructuras de Datos y Algoritmos

Informe Tarea 1

28 de abril de 2019

1º semestre 2019 - Profesor Yadran Eterovic

Paul Heinsohn Manetti - 1562305J

Análisis teórico

En primer lugar, se abordará la complejidad para armar el *KDTree* y luego la de generar un *max heap* para los vecinos más cercanos para realizar una búsqueda en nuestra estructura. El *KDTree* implementado se arma en base al siguiente procedimiento:

1. QuickSort de *data train* en base a la dimensión de los xs para guardar los índices de la posición de los vectores.
2. QuickSort de *data train* en base a la dimensión de los ys para guardar los índices de la posición de los vectores.
3. Generar 4 arreglos resultantes de la separación de la mediana de 1 o 2 (acorde a la conveniencia) para que sean entregados a los hijos de la raiz.
4. Si el tamaño de los arreglos es mayor al deseado por hoja volver a 3, en caso contrario destinar nodo actual como nodo hoja.

Si bien 1 y 2 en su peor caso son de complejidad $O(|data\ train|^2)$ cada uno, se tiene que para su mejor caso y promedio la complejidad es bastante buena: $O(|data\ train|)$, y debido al tamaño (grande) del set de entrenamiento, más dificil será obtener el peor o mejor caso, por lo que se asumirá la ocurrencia del caso promedio. Para 3 se tiene que la cantidad de datos a procesar disminuye a la mitad cada vez hasta llegar a $O(log_2(|data\ train|))$ niveles de recursión aproximadamente. Por lo tanto, se tiene que la complejidad total de generar el árbol es de $O(|dim|\cdot|data\ train|\cdot\log(|data\ train|)+|data\ train|)$, lo que equivale a $O(dim\cdot n\cdot\log(n))$, siendo *dim* la cantidad de dimensiones de los datos, en este caso 2.

Para realizar la búsqueda, se tiene que cada punto a clasificar debe generar un *max heap* para almacenar los k vecinos más cercanos, en donde el nodo raíz almacena la distancia más grande de estos hasta el momento, por lo que decidí dividir en 2 esta parte: llenar el *heap* con elementos presentes en la misma caja que el punto a clasificar y de las cercanas, y luego revisar las otras cajas aledañas que puedan contener puntos más cercanos. Ambas partes requieren realizar búsqueda binaria para llegar a los nodos hojas, pero la diferencia es que el primero se detiene al llenar el *heap* con k elementos, mientras que la búsqueda se debe realizar revisando todo el árbol, por lo que la complejidad que aporta el primero es de $O(|data\ test| \cdot log(k))$ mientras que el segundo, es de $O(|data\ test| \cdot log(|data\ train|))$

Dado lo anterior, solo basándonos en el algoritmo *KNN*, es decir, sin tomar en cuenta la construcción del *KDTree*, tendremos que la complejidad es

Análisis Empírico

Los siguientes cuadros muestran diferentes valores para la cantidad de vecinos y el tamaño de cada set con el fin de realizar un análisis comparativo para medir el impacto que tiene cada uno en el algoritmo *KNN* empleado.

Cuadro 1: "Tiempo ejecución con k variable"

Variable	Valores Originales	Valores Nuevos
Cantidad de Vecinos	50	30
Set de Entrenamiento	60000	60000
Set de Prueba	10000	10000
Precisión	0.969	0.970
Tiempo Ejecución	8.780	5.514

Cuadro 2: "Tiempo ejecución con set de entrenamiento variable"

Variable	Valores Originales	Valores Nuevos
Cantidad de Vecinos	50	50
Set de Entrenamiento	60000	30000
Set de Prueba	10000	10000
Precisión	0.969	0.969
Tiempo Ejecución	8.780	8.580

Cuadro 3: "Tiempo ejecución con set de test variable"

Variable	Valores Originales	Valores Nuevos
Cantidad de Vecinos	50	50
Set de Entrenamiento	60000	60000
Set de Prueba	10000	5000
Precisión	0.969	0.977
Tiempo Ejecución	8.780	8.382

Dado lo anterior, es posible apreciar que el valor del *input* menos influyente respecto al tiempo de ejecución fue el del set de entrenamiento, lo que era esperado ya que la obtención de los *KNN* se ve poco influenciada por el tamaño de dicho set, sin embargo, lo inesperado ocurrió en la precisión obtenida, ya que al eliminar la mitad de los datos uno esperaría que esta disminuyera y no se mantuviera. Esto refleja que la mitad de los datos son suficientes para poder clasificar bastante bien este set de prueba, es más, incrementarlos levemente podría conllevar al *overfitting*. También, es posible destacar que disminuir la cantidad de elementos en el set de prueba no genera mucha diferencia en cuanto al tiempo, naturalmente éste desciende, pero se dió el caso en que la precisión aumentó, y debido a que solo eliminé los últimos 5000 elementos, se puede afirmar que en los datos eliminados había un promedio de fallas más elevado que en la primera mitad. Por otra parte, la disminución de vecinos disminuye el tiempo mientras mantiene la precisión, lo que tiene sentido, ya que el procedimiento es el mismo con los mismos datos, solo que se buscan menos elementos para llenar el *heap* y se realizan menos movimientos al hacer *heapify*.