Un algoritmo aleatorizado para calcular la mediana

Erwin Talla Chumpitaz*, Carlos Aznarán Laos†, Miller Silva Menejes‡ y Jesús Jáuregui Alvarado§
Facultad de Ciencias, Universidad Nacional de Ingeniería
Av. Túpac Amaru 210, Rímac, Lima 25, Perú
Email: *erwinleo_98@hotmail.com, †caznaranl@uni.pe, ‡miller_silva_96@hotmail.com, §jjaureguia@uni.pe

***Resumen*—En el ámbito de la estadística y probabilidad existe controversia sobre cuándo utilizar la mediana como medida de tendencia central. El objetivo de este artículo es valorar mediante un programa en R su utilidad.**

Para poder llevar a cabo esta investigación, se han revisado artículos científicos similares y consultado la base de datos de "data.worldbank.org" World Bank Open Data, ScienceDirect y otros de donde extrajimos la muestra para la experimentación.

Después de haber probado diferentes métodos para el cálculo aleatorizado de la mediana, llamamos "randomquicksort" a nuestro programa más eficiente.

***Index Terms*—tendencia central, mediana, algoritmo aleatorizado, randomquicksort**

I. Introducción

No hay persona en el mundo que no se haya valido en la Estadística y Probabilidad en algún momento de su vida, desde calcular el tiempo promedio al salir de su casa a esperar el bus para dirigirse al trabajo, pasando por juegos de azar, hasta realizar complejos cálculos para pronosticar resultados futuros para invertir en la bolsa. En el mundo de la estadística existen distintas medidas de tendencia central, entre las más comunes se encuentran; la media, la mediana y la moda, en este reporte nos enfocaremos en la Mediana, repasando desde su origen, importancia y sus aplicaciones en la vida, incluso encontrar el algoritmo aleatorizado más eficiente para el cálculo de la misma, entonces al finalizar de leer este reporte usted sabrá como y cuando utilizar la mediana como medida de tendencia central, así mismo le mostraremos como implementar un código en R que realice este procedimiento de la manera más eficientemente posible.

Definición I-.1 (**Percentil).** Es una medida de posición, donde Percentil i es aquel valor P_i que deja a su izquierda el i y el resto por encima de los valores de la muestra ordenada de menor a mayor, de esta forma el valor de la mediana es representada como el Percentil 50, P_{50}.

Definición I-.2 (**Cálculo de percentiles (Caso Continuo)).** Se realizan los siguientes pasos:

1. Construir la tabla de porcentajes acumulados P_l.
2. Ubicar el 50 % tal que:

$$P_{i-1} \leq l \leq P_i$$

.

3. L_{i-1} y L_i son los límites inferior y superior del intervalo correspondientes a P_{i-1} y P_i respectivamente.

Entonces el percentil p_l se calcula mediante la siguiente fórmula:

$$p_l = L_{i-1} + \frac{l - P_{i-1}}{P_i + P_{i-1}}(L_i - L_{i-1}).$$

Definición I-.3 (**Mediana de una distribución).** Es un valor x_m de $\mathbb{X}$ tal que el 50 % de los posibles valores de X están por debajo de x_m y el 50 % de los posibles valores de $\mathbb{X}$ están por arriba de x_m.

Definición I-.4 (**Algoritmos de búsqueda).** Un algoritmo de búsqueda se encarga de, a partir de una serie de criterios, encontrar un determinado elemento dentro de un conjunto de datos. Dentro de los más conocidos y más utilizados están:

- *Búsqueda secuencial*: Para encontrar un elemento en un conjunto estructurado, se analiza elemento por elemento hasta encontrar con aquel elemento solicitado. Tiene un tiempo, en el peor caso, de $O(n)$.
- *Búsqueda binaria*: En este caso, el conjunto de elementos debe estar ordenado de modo que se compara con el elemento medio si es menor(izquierda), mayor(derecha) e igual (devuelve el valor), y se repite el proceso reduciendo la cantidad de datos. Tiene un tiempo, en el peor caso, de $O(log2n)$.

Definición I-.5 (**Algoritmo de ordenamiento).** Un algoritmo de ordenamiento se encarga de comparar todos los elementos y colocarlos en un orden, mediante algún método (intercambio, mezcla, partición, etc.). Los algoritmos más utilizados son:

- *Ordenamiento de burbuja (BUBBLESORT)*: Este algoritmo funciona comparando cada elemento de la lista con el siguiente e intercambiando si es que el elemento comparado es menor. Tiene como complejidad $O(n^2)$.
- *Ordenamiento por mezcla (MERGESORT)*: Este algoritmo funciona dividiendo en la mitad recursivamente en una cantidad de elementos, cuando son pequeñas cantidades se procede a ordenarlos y agruparlos, de manera progresiva hasta su totalidad. Tiene complejidad $O(n \log_2 n)$.
- *Ordenamiento rápido (QUICKSORT)*: Este algoritmo funciona utilizando un pivote, el cual tiene la función de comparar y particionar el conjunto de datos entre menores y mayores que el pivote, este procedimiento se repite en las dos particiones y así sucesivamente. Tiene complejidad $O(n \log_2 n)$. Es uno de los algoritmos más rápidos, por tal motivo se tiene pensado utilizar este algoritmo para lograr el objetivo de este proyecto.

Definición I-.6 (Algoritmos aleatorios). Los algoritmos aleatorios son aquellos que utilizan en alguna parte de su estructura números aleatorios, con el fin de lograr las mismas posibilidades frente a cualquier caso.

II. Estado del arte

- El primer algoritmo aleatorizado fue un método desarrollado por Michael O. Rabin para el problema de par más cercano en la geometría computacional:
  "Dados n puntos en el espacio métrico , encuentre un par de puntos con la menor distancia entre ellos"
- Un algoritmo ingenuo para encontrar distancias entre todos los pares de puntos en un espacio de dimensión d y seleccionar el mínimo requiere $O(n^2)$ tiempo y se puede resolver en $O(n \log n)$ tiempo en un espacio euclidiano. En el modelo computacional que asume que la función de piso es computable en tiempo constante, el problema se puede resolver en el tiempo $O(n \log n \log n)$. Si permitimos que la aleatorización se use junto con la función de piso, el problema se puede resolver en el tiempo $O(n)$.
- Michael O. Rabin demostró que la prueba de primalidad de Miller de 1976 se puede convertir en un algoritmo aleatorizado. En ese momento, no se conocía ningún algoritmo determinístico práctico para la primalidad.
- Actualmente existen varios algoritmos aleatorios para encontrar la mediana, cada algoritmo se diferencia en su procedimiento para ordenar los datos aleatorios. Los algoritmos más usados por su eficacia y eficiencia para ordenar datos son merge sort, heapsort y quicksort.

III. Diseño del experimento

A partir de lo señalado y enfocado a nuestro proyecto, es necesario utilizar la aleatoriedad para obtener una mejor probabilidad en caso cualquiera. Por lo expuesto, se procede a elaborar un seudocódigo para hallar la mediana de manera aleatoria con la idea de poder transcribirlo a cualquier lenguaje de programación, en nuestro caso en lenguaje R. Para hallar la mediana de un conjunto de datos, se deben de utilizar un algoritmo de ordenamiento y de búsqueda, ya que no se sabe si el conjunto de datos está ordenado. Para ello se analiza primero los algoritmos de búsqueda.
Aplicado a nuestro objetivo sería ineficiente buscar secuencialmente la mediana, ya que no necesitamos recorrer toda la cadena para hallarlo, aun peor cuando no está ordenado. Igualmente pasaría con un algoritmo de búsqueda binaria, ya que de estar ordenado tan solo se necesitaría devolver. Entonces, es necesario implementar algún algoritmo de ordenamiento.

III-A. Uso de los algoritmos de ordenamiento

Como se presentó anteriormente, existen diferentes tipos de algoritmos de ordenamiento. Para escoger el más indicado se debe tomar en cuenta el tiempo de ejecución, es decir cuánto tiempo demora en ordenar todo el conjunto de datos. Para ello se utiliza la gran O como determinante para el análisis de los tiempos. Se tiene algoritmos de ordenamiento en $O(n^2)$, $O(n \log_2 n)$ y hasta $O(n)$.
Sin embargo, no es necesario ordenarlo todo para hallarlo. Para ello se podría ordenar hasta $n/2$ de forma secuencial sin problema alguno y el resto dejarlo tal como estaba. Podría ser una solución buena, pero se puede mejorar aún más. Si analizamos la estructura del Quicksort, nos puede ayudar en la búsqueda de la mediana, el cual cuenta con dos secciones:

- La función **PARTICION** tiene como entrada A, p y r. A es una cadena de elementos de tamaño n, p es la posición inicial relativa y r es una posición final relativa. La función se encarga de seleccionar un pivote, el cual es el último elemento en la cadena, en donde se una separación de los números en comparación con el pivote, donde los menores irán a la izquierda y los mayores a la derecha. Por último, el pivote intercambia de posición con el primer mayor encontrado, y devuelve esa posición.

Algoritmo 1 PARTITION

Input: A , p , r
Output: q posición del pivote

```
x ← A[r]
i ← p − 1
for j do p r - 1
    if A[j] ≤ x then
        i ← i + 1
        Cambio (A[i] , A[j])
        Cambio (A[i + 1] , A[r])
    end if
end for
Retorna i + 1
```

- La función **QUICKSORT** tiene las mismas entradas que la función anterior. Su trabajo es realizar el proceso anterior de manera recursiva, es decir realizando el mismo proceso una vez subdivida la cadena inicial por el pivote (parte menor y parte mayor).

Algoritmo 2 QUICKSORT

Input: A, p, r
Output: A array arreglado

```
if p < r then
    q ← PARTITION (A , p , r)
    QUICKSORT (A , p , q − 1)
    QUICKSORT (A , q + 1 , r)
end if
```

Lo importante a resaltar es que el algoritmo **QUICKSORT** tiene $O(nlogn)$ como tiempo promedio y que es el más rápido en comparación con los otros algoritmos de igual complejidad, pero tiene $O(n^2)$ en el peor de los casos. Esto hace de que tenga una gran desventaja.

III-B. Uso de la aleatorización

Para ello se utiliza la aleatorización con el propósito de evitar que se presente el peor caso. En ese sentido lo que se hará es seleccionar de manera aleatoria el pivote. Esto realiza con las siguientes funciones:

Algoritmo 3 RANDOM_PARTITION

Input: A , p , r
Output: q posición del pivote
1: $i \leftarrow$ Random(p , r)
2: Cambio ($A[r]$, $A[i]$)
3: Retorna PARTITION(A , p , r)

Algoritmo 4 RANDOM_QUICKSORT

Input: A , p , r
Output: A array arreglado
1: **if** $p < r$ **then**
2: $q \leftarrow$ RANDOM_PARTITION (A , p , r)
3: RANDOM_QUICKSORT (A , p , $q-1$)
4: RANDOM_QUICKSORT (A , $q+1$, r)
5: **end if**

III-C. Mejoras en el algoritmo

Como ya se había mencionado, se puede mejorar aún más el resultado. Por ello, se utiliza una variación del **QUICKSORT** el cual es **QUICKSELECTION**. El **QUICKSELECTION** tiene como entradas A,p,r e i, donde i es la posición que estamos buscando (mediana). Lo primero que hace es comparar si p y r son iguales, es decir el tamaño es 1, y procede a devolver único número. Luego utiliza la función **PARTITION** al igual que en el **QUICKSORT** y coloca en q la posición del pivote. Luego se crea una variable k, la cual es la posición central obtenida por **PARTITION**, y se compara con i, que en este caso es la mediana o $n/2$. Si lo es devuelve el resultado, de lo contrario hará una comparación. Si k es mayor a la posición central, realizará una recursión para el lado izquierdo, sino lo hará al lado derecho.

Algoritmo 5 QUICKSELECTION

Input: A , p , r , i
Output: $A[i]$ elemento del array semiarreglado
1: **if** $p = r$ **then**
2: Retorna $A[p]$
3: **end if**
4: $q \leftarrow$ PARTITION (A , p , r)
5: **if** $q = i$ **then**
6: Retorna $A[p]$
7: **else**
8: **if** $q > i$ **then**
9: QUICKSELECTION (A , p , $q-1$, i)
10: **else**
11: QUICKSELECTION (A , $q+1$, r , i)
12: **end if**
13: **end if**

Utilizando la aleatorización para mejorar el rendimiento del algoritmo:

Algoritmo 6 RANDOM_QUICKSELECTION

Input: A , p , r , i
Output: $A[i]$ elemento del array semiarreglado
1: **if** $p = r$ **then**
2: Retorna $A[p]$
3: **end if**
4: $q \leftarrow$ RANDOM_PARTITION (A , p , r)
5: **if** $q = i$ **then**
6: Retorna $A[p]$
7: **else**
8: **if** $q > i$ **then**
9: RANDOM_QUICKSELECTION (A , p , $q-1$, i)
10: **else**
11: RANDOM_QUICKSELECTION (A , $q+1$, r , i)
12: **end if**
13: **end if**

Frente a esto tenemos que el algoritmo es mucho más eficiente y rápido al tratar de hallar la mediana que anteriormente, llegando inclusive a una ejecución de tiempo $O(n)$, ya que no utiliza la otra división restante ni llega a completar todo el ordenamiento.

Referencias

[1] Thomas H Cormen y col. *Introduction to algorithms*. MIT press, 2009.

[2] Thomas Cover y Peter Hart. "Nearest neighbor pattern classification". En: *IEEE transactions on information theory* 13.1 (1967), págs. 21-27.

[3] Michael O Rabin. "Probabilistic algorithm for testing primality". En: *Journal of Number Theory* 12 (1 1980). DOI: 10.1016/0022-314x(80)90084-0.