Trabajo Práctico Integrador N°1

Grupo N° 8

UNMdP – Facultad de Ingeniería

Materia: Teoría de la Información

Profesores: Massa, Stella Maris; Spinelli, Adolfo Tomás; Lanzillota, Franco

Autores:

Battista, Franco(franco.battista99@outlook.com)

Boolls, Nicolas (nicolasboolls@gmail.com)

Mujica, Juan Manuel (mujicajuanm@gmail.com)

Fecha de entrega: 20 de octubre de 2022

Github: https://github.com/JuanMaMujica/TeoriaDeLaInformacion

Índice

Resumen.....3

Introducción.....3

Desarrollo

Ejercicio 1

a).....3

b).....4

c).....5

Ejercicio 2

a).....6

b).....7

c).....8

d).....9

e).....10

Conclusión.....11

Resumen

El trabajo trata temas referentes al manejo de las fuentes de memoria no nula, las fuentes de Markov, las propiedades de los códigos y la codificación de las fuentes. Para el estudio y la comprensión de cada uno de estos puntos se desarrollan conceptos particulares, o en el caso de, por ejemplo, la entropía, se calcula de manera diferente en función del tema en cuestión. A continuación se detallan los mismos:

- Fuentes de memoria nula: Cantidad de información; Entropía.
- Fuente de Markov: Cantidad de información; Vector estacionario; Entropía.
- Propiedades de los códigos y codificación: Cantidad de información; Entropía; Longitud media; Cumplimiento de la inecuación de Kraft y McMillan; Código es instantáneo; Código es compacto.
- Codificación de fuentes de información: Primer Teorema de Shannon; rendimiento y redundancia de un código; códigos de Huffman y Shannon-Fano.

Introducción

Para el desarrollo del trabajo práctico, los dos aspectos principales a considerar fueron:

la resolución de los ejercicios, cada uno con su propia complejidad respecto de los temas
tratados en las clases teóricas y prácticas; además del desarrollo de programas para cumplir los objetivos y obtención de resultados.

En cuanto a la elección del lenguaje de programación, se optó por utilizar Java, ya que además de que todos los integrantes son conocedores del mismo, este lenguaje brinda librerías que facilitan todo tipo de operaciones con archivos (crear, abrir, modificar, leer), lo cual es fundamental dado que conlleva tanto una facilidad en el manejo de los mismos como también la implementación a rasgos generales.

Por último, para la implementación de los programas se reutilizó código y se compactaron diferentes incisos del trabajo en un mismo programa, ya que al realizar algunos cálculos, luego los mismos servían para poder cumplimentar la información que otros procedimientos necesitaban.

Desarrollo

PRIMERA PARTE

a) Para este inciso, se pedía calcular las probabilidades condicionales y en base a ello determinar si es una fuente de memoria nula o no nula. Como nuestro sistema es ternario, decidimos utilizar un arreglo de 9 elementos donde en cada uno de ellos almacenamos las ocurrencias condicionales. Nuestro criterio fue: si el carácter anterior fue, por ejemplo, una ‘A’, entonces se incrementa un contador en alguno de los primeros tres índices del arreglo dependiendo del carácter leído actual.

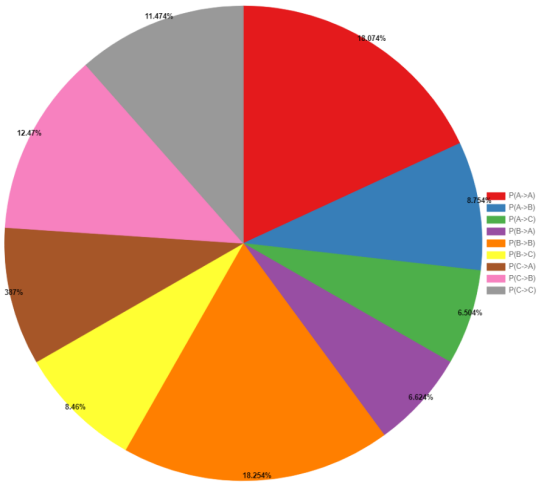
Si fue una ‘B’, se incrementa en los índices 4, 5 y 6 y en caso de ser ‘C’, en los 7, 8 y 9.
Una vez hecho esto, calculamos las probabilidades condicionales dividiendo las ocurrencias por la cantidad de caracteres totales del archivo, y las almacenamos en una matriz de transición de estados para más comodidad. Esto fue lo que obtuvimos:

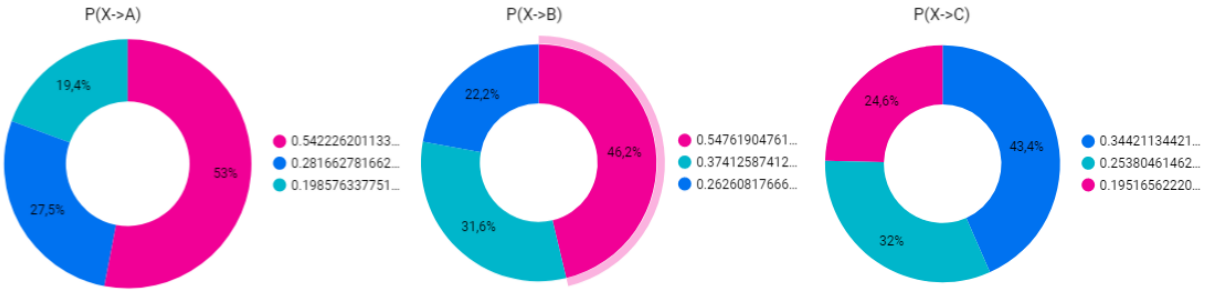
```
[0.5422262011339899]  [0.19857633775159547]  [0.28166278166278164]
[0.26260817666368247]  [0.5476190476190477]  [0.3741258741258741]
[0.19516562220232767]  [0.2538046146293569]  [0.3442113442113442]
```

Los elementos por columna representan las probabilidades de “salida” de cada símbolo. Es decir, la probabilidad de que un símbolo se dé, dado el que estamos analizando.

Por el contrario, los elementos por fila representan las probabilidades de “llegada” de cada símbolo, lo cual simboliza la probabilidad de que el símbolo que estamos analizando se dé, dado otro.
Como es de esperarse, se verifica que la suma de los elementos por columna es igual a 1. Esto nos dice que siempre se dará un símbolo dado otro, es decir, siempre habrá un cambio de estado.

Por otro lado, al observar las probabilidades por fila, vemos que son muy variables (con variaciones de más del 10% en todos los casos, véase el gráfico adjuntado a continuación), lo cual representa que la probabilidad de “alcanzar” dicho estado desde cualquiera de los otros (o el mismo estado) dista mucho de la equiprobabilidad. En base a esto concluimos que la aparición de cada símbolo es estadísticamente independiente y por lo tanto estamos trabajando con una fuente de memoria **no nula.**





c) Para determinar si una fuente es ergódica, se debe observar si durante un tiempo suficientemente largo, se emite con probabilidad 1, una secuencia “típica de símbolos”. Lo que esta propiedad nos dice es que básicamente para que nuestra fuente sea ergódica, debe existir el vector estacionario. Por lo tanto, utilizamos la fórmula ***V*= M.V**** y fuimos iterando sobre ella una cantidad variable de veces. Si la cantidad es la suficiente (con más de 10 iteraciones bastó en nuestro caso), los resultados en todos los casos que nos dio el vector estacionario fueron los siguientes:

```
V[0]: 0.3351335133513353
V[1]: 0.4074407440744076
V[2]: 0.25742574257425754
```

El pseudocódigo utilizado para el cálculo del vector estacionario es el siguiente:

```
definir i,j,k,p como entero
definir aux como real

para (k=0 hasta 249) hacer
        para (i=0 hasta 2) hacer
                para (j=0 hasta 2) hacer
                        aux= aux + vectorEstacionario[j]*matriz[i][j]
                fin para
        vectorAuxiliar[i]=aux
        aux=0
        fin para
        para (p desde 0 hasta 2) hacer
                vectorEstacionario[p]=vectorAuxiliar[p]
        fin para
fin para
```

Por lo que concluimos que nuestra fuente es ergódica y el vector estacionario es el dado. Para el cálculo de la entropía, utilizamos la fórmula vista en clase y el resultado que nos dio fue de **1.4769973604775612 bits/símbolo.**

Pseudocódigo utilizado para el cálculo de la entropía:

```
definir i,j como entero
definir entropia como real
entropia=0
para (j=0 hasta 2) hacer
        para (j=0 hasta 2) hacer
                entropia=entropia+vectorEstacionario[j]*matrizDePasajes[i][j]*log2(1/matrizDe
        Pasajes[i][j])
        fin para
fin para
```

SEGUNDA PARTE

a)

Fuente con palabras de 3 dígitos:
Cantidad de información total de la fuente 84.66 bits
Entropía total de la fuente: 2.85 bits.

Fuente con palabras de 5 dígitos:
Cantidad de información total de la fuente 1263.89 bits
Entropía total de la fuente: 4.66 bits.

Fuente con palabras de 7 dígitos:
Cantidad de información de la fuente 5380.04 bits
Entropía total de la fuente: 5.96 bits.

En los 3 distintos escenarios, tanto la cantidad de información como la entropía de la fuente se incrementa a medida que las palabras son de mayor longitud. Esto se debe a que al ser mayor las longitudes también hay una cantidad mayor de combinaciones para codificar una palabra:

Para el escenario de 3 dígitos pueden existir como máximo 8 palabras distintas (2^3 combinaciones). Luego, para 5 dígitos pueden existir como máximo 32 palabras distintas (2^5 combinaciones), y por último para los 7 dígitos, 128 (2^7 combinaciones)

En conclusión, como la fuente emitirá cada vez más cantidades de palabras distintas,
la cantidad de información será mayor. Por lo tanto, como la información emitida es mayor, esto afecta directamente a la entropía por definición: cantidad media de información de una fuente.

$$H(S)=\sum_{S} P(S_i)\log\frac{1}{P(S_i)} \qquad I(s_i)=\log\frac{1}{P(s_i)}$$

Se adjuntan en la imagen, las dos fórmulas utilizadas para calcular la entropía, y la cantidad de información, respectivamente

PSEUDOCÓDIGO CANTIDAD INFORMACIÓN Y ENTROPÍA

```
public static double info(Matriz mapa)
        double aux = 0;
        double size= 0;
        int longitud = 0;
        para i = 0 hasta mapa.longitud con pasos de 1 hacer
                size+=mapa[i][0];
                longitud++;
        fin para
        int i = 0
        mientras (i < longitud)
              String clave = mapa[i][1];
              aux+=Math.log(1/(mapa[clave]/size))/Math.log(3);
       fin mientras
        return aux;
fin funcion

public static double Entropia(Matriz mapa) {
        double aux = 0;
        double size= 0;
        int longitud = 0;
        para i = 0 hasta mapa.longitud con pasos de 1 hacer
                size+=mapa[i][0];
                longitud++;
        fin para

        mientras (i < longitud)
              String clave = mapa[i][1];
              aux+=(mapa[clave]/size)* Math.log(1/((mapa[clave]/size))/Math.log(3);
       fin mientras
        return aux;
fin funcion
```

b)

	Bloque	Singular	Unívocamente Decodificable	Instantáneo
3 dígitos	**SI**	**NO**	**SI**	**SI**
5 dígitos	**SI**	**NO**	**SI**	**SI**
7 dígitos	**SI**	**NO**	**SI**	**SI**

Todos son códigos bloque, ya que a cada símbolo (8,32, 128 cantidades respectivas) le pertenece una palabra de código que permanece constante en el tiempo (durante la ejecución del programa, ninguna palabra es reasignada a otro símbolo, causando alguna variación).

Todos los códigos son no singulares dado que a todos sus símbolos les corresponde una palabra código diferente. Esto se puede deducir, ya que si una palabra se repite a la hora de leerla, únicamente aumenta la frecuencia de ésta en la fuente, pero no significa que esa misma palabra pertenezca a otro símbolo.

Por definición de código unívocamente decodificable: un código bloque no singular en el que todas las palabras código tienen la misma longitud es un código unívocamente decodificable. Por lo tanto, en todos los escenarios las respectivas longitudes de las palabras son las mismas, y además son códigos no singulares, lo enunciado anteriormente se cumple, y en consecuencia, los códigos son UD.

Al ser todas las palabras código de la misma longitud en cada escenario, la única forma de que alguna palabra sea prefijo de otra es que sean iguales, y al ser los códigos mencionados no singulares, esto no es posible. Lo que sí es posible en definitiva, es poder decodificar las palabras de una secuencia sin precisar el conocimiento de los símbolos que las suceden, es decir, los códigos son instantáneos.

c)

Escenario/ Valores	Inecuación Kraft	Inecuación McMillan	Longitud Media	Compacto (posibles)
Palabra 3 caracteres	**1**	**1**	**3**	**SI**
Palabras 5 caracteres	**1**	**1**	**5**	**SI**
Palabras 7 caracteres	**1**	**1**	**7**	**SI**

Inecuación de Kraft/Mcmillan: $$\sum_{i=1}^{q} r^{-l_i} \leq 1$$

Como observamos en la tabla, para los 3 escenarios planteados se cumple la inecuación de Kraft. Esto nos indica que existe un código instantáneo para las palabras con

las longitudes planteadas , debido a que también se cumple, como se mencionó en el inciso anterior, que los códigos son no singulares.

Por otra parte, podemos afirmar que la inecuación de Kraft se cumple por la demostración de McMillan, donde afirma que si existe un código unívoco, entonces se verifica la inecuación de Kraft. Esto se debe a la conclusión llegada con anterioridad en donde se indica que los códigos son unívocos por ende se cumple Kraft.

Dado que los tres códigos cumplen con la inecuación de Kraft, se puede asegurar la existencia de un código instantáneo con esas longitudes para cada uno, y al ser las palabras de cada código de la misma longitud y los códigos no singulares, todos los escenarios presentan un código instantáneo. Además, McMillan demostró que si existe un código unívoco, entonces este debe verificar la inecuación de Kraft. Como los tres códigos son unívocamente decodificables, se confirma que los tres cumplen con la inecuación. Si bien se desarrolló un código para el cálculo, al ser las palabras de los códigos de la misma longitud, se puede realizar el cálculo rápidamente:

Ecuación de la Longitud Media de un Código: $$L = \sum_{i=1}^{q} p_i * l_i$$

Con respecto a la longitud media es muy fácil afirmar que serán 3, 5 y 7. Esto se debe a que ya tenemos las longitudes de las palabras códigos desde un principio, por lo que si hacemos los cálculos pertinentes nos darán dichos valores.

Debido a que desconocemos si hay algún código unívoco que tenga menor longitud media que los códigos de longitud 3, 5 y 7, aplicables a la misma fuente y al mismo alfabeto, no podemos afirmar si alguno de los códigos obtenidos es compacto o no.
Sí podemos decir que son candidatos a ser compactos ya que partimos de la premisa *“con un código instantáneo y una fuente de memoria nula, L debe ser igual o mayor que la entropía”*

$$H_r(S) \leq L$$

d)

	Rendimiento	Redundancia
3 dígitos	**0.95064**	**0.04936**
5 dígitos	**0.93217**	**0.06783**
7 dígitos	**0.85251**	**0.14749**

Tanto el rendimiento como la redundancia únicamente pueden ser obtenidas si se cumple que la longitud media del código es mayor o igual a la entropía del mismo. Como todos los códigos cumplían esa condición se obtuvieron los valores de la tabla adjuntada anteriormente.

Es una característica común de los 3 escenarios el hecho de ser códigos con un alto rendimiento, muy cerca del máximo (de valor 1) lo que significa, en consecuencia, que presentan una baja redundancia. Mayor redundancia implica menor información.

Por otra parte, si se analiza caso por caso, el tercer código presenta el menor rendimiento de todos y el primer código presenta el mayor rendimiento. Se puede concluir que el 1° código emite más información que el resto y que, por el contrario, el 3° código emite menos información.

e)

Mediante el código de Shannon Fano se construyó un código instantáneo óptimo. Es un procedimiento que busca formar subconjuntos cuya diferencia entre la sumatoria de las probabilidades de los símbolos de dichos subconjuntos en valor absoluto sea mínima. El procedimiento por el cual se logró la codificación a través de Shannon Fano consiste en, partiendo de dos HashMaps ,uno en donde las claves son los símbolos leídos desde el “archivo.txt” y los valores son sus respectivas codificaciones de Shannon Fano que se van a ir formando a medida que se avance en el procedimiento y el otro que tendrá los subconjuntos superior o inferior.

Así es como se llevará a cabo el algoritmo de Shannon Fano:

```
procedimiento void iniciaShanonFano(Mapa<String,String> codificacion,
Mapa<String,Integer>  conjunto)
    si (conjunto.longitud > 1) entonces
        int suma=0
        int mitad=0
        int sumaAux=0
        double aux=0
        String simbolo=null
        String acumula=null
        paraCada Elemento de conjunto hacer
            suma -> suma + Elemento.valor
        fin paraCada
        paraCada Elemento  de conjunto hacer
            aux = Elemento.valor
            simbolo = Elemento.clave
            si simbolo != null entonces
                simbolo -> “”
            fin si
            acumula = codificacion.getLlave(simbolo)
            si sumaAux + aux < suma/2 entonces
                codificacion.agregar(simbolo, acumula+”0”)
                mitad->mitad + 1
                sumaAux-> sumaAux + aux
            sino
```

```
si aux!=0 Y ValorAbsoluto(suma/2 - sumaAux) >=ValorAbsoluto((suma/2) - (sumaAux+aux))
                                        codificacion.agregar(simbolo,acumula+"0")
                                        mitad->mitad+1
                                        sumaAux-> sumaAux+aux
                                sino
                                        codificacion.agregar(simbolo,acumula+"1")
                                fin si
                        fin si
                fin paraCada
        fin si
        si mitad != 0 entonces
                Mapa<String,Integer> superior -> nuevo Mapa
                Mapa <String,integer> inferior ->nuevo Mapa
                int i=0
                int j=0
                paraCada Elemento de conjunto hacer
                        si i<mitad entonces
                                superior.agregar(Elemento.llave, Elemento.valor)
                        fin si
                        i-> i+1
                fin paraCada
        iniciaShanonFano(codificación,superior)
        paraCada Elemento de conjunto hacer
                si j>=mitad Y j<conjunto.tamaño entonces
                        inferior.agregar(Elemento.llave, Elemento.valor)
                fin si
        fin paraCada
fin procedimiento
```

Una vez tenemos los códigos de Shannon Fano, lo que se hizo fue ir recorriendo el archivo de texto provisto por la cátedra leyendo las palabras de a 3, 5 y 7. Una vez tenemos las palabras buscamos en el HashMap en donde tenemos los códigos de Shannon Fano y lo pasamos a bits para luego cargarlos en un array de tipo byte y finalmente escribirlo en un archivo binario.

Como conclusión observamos que a pesar de ser un procedimiento subóptimo se logró construir un código instantáneo óptimo. Si observamos los tamaños de los archivos vemos que se comprimió visiblemente la información.

Conclusión

Realizados todos los cálculos y con los conceptos vistos hasta el momento, concluimos que los resultados fueron exitosos y coherentes.

El archivo “archivo.txt” fue leído e interpretado correctamente, además de compactado como se pedía en los enunciados, mediante las técnicas correspondientes.

Para lograr todo esto decidimos el uso del lenguaje Java, ya que es un lenguaje que dominamos casi al completo y con mucha variedad de estructuras para tratar los datos que se ingresan. Como entorno de desarrollo utilizamos JDeveloper ya que nos permite trabajar de manera fácil en el código, con una interfaz amigable e intuitiva. Nos permite construir el ejecutable de una manera rápida y sencilla.

Para finalizar, concluimos que el código quedó realizado de una forma prolija y con una buena base para continuar escalando con distintas técnicas a futuro.