Trabajo Práctico Nº1

Nombre, apellido y legajo de los integrantes: Mateo Etchepare (15093), Gregorio Firmani (15051), Franco Sardi([DNI]40831214).
Emails de contacto correspondientes: mateoetchepare@gmail.com , gregoriofirmani@gmail.com, fraansardi@gmail.com .
Número de grupo: 11
Universidad: Universidad Nacional de Mar del Plata.
URL GitHub: https://github.com/Greg1704/Teoria-de-la-Informacion/

Índice

Resumen **3**

Introducción **3**

Desarrollo **4**

Primera Parte 4

Segunda Parte 6

Apéndice **9**

Conclusión **10**

Resumen

La Teoría de la Información es una propuesta teórica originada con un artículo de Claude E. Shannon, y trata de investigar y medir la información, su almacenamiento y comunicación. En el núcleo de ésta teoría están los emisores y receptores de
Información: en el proceso de comunicación el mensaje va del emisor o fuente hacia el receptor.
AUXILIAR: El principal objetivo de esta teoría es el de proporcionar una definición rigurosa de la noción de información que permita cuantificarla.

En este trabajo, se analizará un archivo de texto, tomándolo como fuente de información y aportando datos como su entropía, qué tipo de fuente es y características propias de ser nula o no nula. Luego, para la segunda parte se considerará la cadena de caracteres del archivo como palabras de un código de distintas dimensiones, y se analizarán dichos códigos desde el punto de vista de la Teoría de la Información.

Introducción

Para plantear la resolución del problema en cuestión, utilizamos el lenguaje de programación **Java**, ya que tiene la potencia necesaria para resolver cálculos matemáticos complejos (como se detallará más adelante). Se irá recolectando cada símbolo del archivo y se procesará de forma de obtener los números necesarios para analizar y clasificar la fuente de información.

Usando un archivo de texto provisto por la cátedra, se tomó como fuente de información una cadena de 10.000 caracteres, que consta de 3 símbolos diferentes. Para la primera parte se determinó la independencia estadística entre símbolos (fuente de memoria nula o de Markov) y la cantidad media de información suministrada por símbolo de manera cuantitativa (entropía). Luego, en la segunda parte, se tomó la cadena representando códigos de 3, 5 y 7 caracteres, en base a las distintas palabras clave y su frecuencia se calculó la información que provee, su entropía, qué tipo de códigos son, si son compactos, su rendimiento y redundancia y se codificaron según

Desarrollo

Primera Parte

Para comenzar con el problema, se escribió un código que permita **leer** el archivo y **procesarlo** de manera que se acumule la ocurrencia absoluta de cada símbolo en un **vector** y en una **matriz** la ocurrencia de cada símbolo S_i luego de la ocurrencia de cada símbolo S_j, siendo **i** las filas y **j** las columnas.. Luego, con esto, se calculó la **matriz de probabilidades condicionales**, la cuál contiene la probabilidad de que cada símbolo S_i ocurra luego de cada símbolo S_j con la fórmula siguiente:

FUNCIÓN EN PSEUDOCÓDIGO PARA OBTENER **MATRIZ DE PASAJE**
PARA i = 0 HASTA i = N CON PASO 1 HACER (N: TAMAÑO DE MATRIZ): PARA j = 0 HASTA j = N CON PASO 1 HACER (N: TAMAÑO DE MATRIZ): MATRIZ DE PASAJE [I][J] = MATRIZ DE OCURRENCIAS[I][J] / OCURRENCIA DEL SÍMBOLO J

Siendo N, en este caso, N = 3.

Con los resultados obtenidos en **(1)**, analizamos cada fila y sus respectivas probabilidades, de modo que si en alguna de ellas se encuentra que la diferencia entre el valor mayor y el menor es muy chica entonces consideramos que la fuente es de **memoria nula,** ya que no hay cambio significativo en las probabilidades de que salga ese símbolo dados los distintos símbolos anteriores posibles. De otra forma, la fuente sería considerada una **fuente de Markov**. En éste caso, tomamos que si la diferencia entre la mayor probabilidad y la menor probabilidad, es mayor a 0.03 entonces es lo suficientemente significativa para ser considerada una fuente de Markov, caso contrario es una fuente de memoria nula.

En esta parte del código, se implementaron ambos casos, para hacer el programa lo más genérico posible, es decir, que sirva para cualquier fuente, de N cantidad de símbolos.

En el segmento de análisis para una fuente de **memoria nula**, se diseñó un algoritmo que genera la extensión de orden 20, de manera recursiva, sin almacenarse en ninguna estructura ya que para almacenar, en este caso, 3^{20} (fórmula de **variación con repetición)** casos posibles, se necesitaría muchísima memoria. Ya que lo único que se pide para la extensión de orden 20, es calcular la entropía, se puede ir calculando a medida que se va generando cada símbolo con su palabra código posible. Los resultados se muestran en el ítem **(2)** del apéndice.

FUNCIÓN EN PSEUDOCÓDIGO PARA OBTENER LA ENTROPÍA DE ORDEN 20:

Aclaración: El código sirve para obtener la entropía de cualquier orden que se quiera, en el código, dejamos de orden 3 para que puedan correr el código rápidamente, ya que para orden 20 le toma alrededor de 90 minutos.

Definición variables:
-input: Variable local del tipo String que cuenta con todos los caracteres posibles, en este caso "ABC".
-orden: Variable local del tipo Integer que lleva la cuenta del largo restante para que se pueda calcular la entropía de la cadena.
-auxentrop: Variable local del tipo Double que contiene la multiplicación de las probabilidades de aparición de cada símbolo, la cual luego se usa para calcular la entropía de la fuente.
-VProb: Variable local del tipo ArrayList que contiene las probabilidades de aparición de los símbolos que se encuentran en input.
-orden20entrop: Variable global del tipo Double que contiene la entropía total de orden 20.

```
Funcion void calculoEntropiaOrden20(input,orden,auxentrop,VProb[])

SI orden = 0 ENTONCES
   Cálculo de la entropía en la variable auxentrop
   Sumo auxentrop en la variable global orden20entrop
SINO
   PARA i HASTA input.length CON PASO 1 HACER
      auxentrop = auxentrop * (probabilidad de alguno de los char de input)
      calculoEntropiaOrden20(input,orden-1,auxentrop,VProb[])
      auxentrop = auxentrop /  (probabilidad de alguno de los char de input)
```

Por otro lado, si se tratase de una **fuente de Markov**, se usó un algoritmo muy conocido: el **algoritmo de Warshall**. Éste comprueba la existencia de caminos entre todos los pares de vértices de un grafo (en este caso, el grafo estaría constituído por los símbolos que puede emitir la fuente). El algoritmo, en nuestra implementación, guardó en cada índice de una matriz un "0" si **NO** se encuentra un camino entre vértices y un "1" **SI** hay algún camino. Por lo que luego de terminado el algoritmo, el programa chequea la existencia de algún "0", de manera que si lo hay, la fuente **no** será ergódica.

Si la fuente fuese ergódica, se debe obtener el valor del **vector estacionario**. Para obtener el valor del vector estacionario, se debe usar la fórmula (M-I).V* = 0, siendo M la matriz de pasaje, I la matriz de identidad, y V el vector estacionario mencionado previamente. Esto resulta en un sistema de ecuaciones, que para su resolución se usó la librería **la4j (7)**. Aquí es donde se necesita

la potencia del lenguaje de programación **Java**, ya que tiene una gran cantidad de librerías disponible para su uso, y en este caso en particular, había que resolver un sistema de ecuaciones lineales de pocas incógnitas, pero podrían ser muchas.

Segunda Parte

a) Para calcular la cantidad de información de cada palabra del código, obtuvimos primero su cantidad de ocurrencias y con eso sacamos la probabilidad de ocurrencia del símbolo, usando la fórmula simple de
$Cantidad\ de\ ocurrencias$ / (10000 / $tamaño\ de\ cada\ símbolo$ (3, 5 *o* 7)).

El denominador representa la cantidad de símbolos que entran en el archivo. Dada la probabilidad de ocurrencia de un suceso, calculamos la información que transmite con la fórmula: I(E)=log(1 / P(E)), en este caso usando logaritmo de base 3 ya que el alfabeto código tiene 3 caracteres.

Para calcular la entropía de cada fuente, se hace la sumatoria de la probabilidad de cada símbolo por su cantidad de información. Los resultados se muestran en el ítem **(5)** del apéndice.

Para responder al inciso b), determinamos que el tipo de código es instantáneo, ya que la consigna establece una longitud fija de las palabras, por lo que esto ya fija su condición de **código bloque**. Luego, nosotros almacenamos las ocurrencias de cada palabra, siendo la cantidad de palabras distintas posibles:
$\#símbolosDelAlfabetoCódigo^{longitudPedida}$ $(variación\ con\ repetición)$. Por lo que esto afirma su condición de **código no singular**. Al cumplir estas dos condiciones simultáneamente, es un **código unívocamente decodificable**. Por último, quedaba chequear si se trataba de un código instantáneo. Al ser todos las palabras código de **misma longitud**, **todas diferentes** unas de las otras, es **imposible** que una sea el prefijo de la otra.

En el inciso c), la inecuación de Kraft nos dió los resultados indicados en **(3)**, por lo que para los tres tipos de fuente planteados por la consigna, la inecuación de Kraft se cumple, ya que el resultado es menor o igual a 1. Debido a que la fórmula de MacMillan es la misma que la planteada por Kraft, los resultados son los mismos.

La inecuación de Kraft es sólo una medida cuantitativa basada en las longitudes de los códigos, pero **no** asegura que el código armado sea instantáneo. De todos modos, por lo explicado en el inciso anterior, el código es instantáneo.

La longitud media del código obtenido y mostrado en **(4)** es totalmente lógico, ya que las longitudes pedidas por la consigna son fijas: **NO** hay palabras código que sean de diferente longitud.

Para analizar la compacidad del código, primero se debe analizar si es **unívoco**. Como se mencionó anteriormente, efectivamente lo es. Para que un código sea considerado compacto, se debe cumplir que la entropía de la fuente sea menor o igual a la longitud media del código, pero esto **no** asegura su compacidad, ya que habría que verificar la longitud media de todos los códigos unívocos que pueden aplicarse a la misma fuente y mismo alfabeto.. De todas maneras, para la fuente de 3 y de 5 caracteres, la entropía difiere mínimamente de la longitud media del código (<0.1), por lo que se podría asegurar que es compacto. Con la fuente de 7 caracteres, el valor de la entropía sigue siendo muy cercano al valor de la longitud media, sin embargo, con una diferencia más notable en comparación a las dos fuentes anteriores. Esto se debe principalmente a lo explicado más adelante, en el inciso d), con respecto a las ocurrencias de cada símbolo en el archivo.

d) Para explicar el concepto de rendimiento, se debe tener en cuenta la entropía de la fuente. Cuanto más información aporte una fuente, su rendimiento será mayor. Los resultados a analizar se encuentran en **(6)**.

Como se puede observar, el rendimiento de la fuente de símbolos de longitud 3 y de longitud 5, el rendimiento es 0,98, muy cercano a 1, mientras que en la fuente de símbolos de longitud 7, se reduce a 0,88. Esto se debe a que, en la secuencia de 10.000 caracteres brindada por el archivo, no caben el total de símbolos diferentes que se pueden generar con una longitud de 7 caracteres, por lo que hay símbolos que **NO** aparecen, mientras que los que sí se encuentran en el archivo tienen una o múltiples apariciones. Para contrarrestar que no quepan todos los símbolos posibles, lo deseado sería que **TODOS** (no la mayoría) de los símbolos aparezcan por lo menos **UNA** vez, así se obtiene mayor cantidad de símbolos diferentes, y que **NO** se repita ningún símbolo, y si lo hacen, lo menos posible.

Todo esto hace que la entropía no sea igual a la longitud media del código, pero de todas formas, es **casi** igual.

En cambio, la redundancia, es lo opuesto al rendimiento. Cuánto menos información aporte una fuente, su redundancia será mayor. Por lo que la redundancia de la fuente de símbolos de 7 caracteres, es mayor a la de las fuentes de símbolos de 3 y de 5 caracteres. Esto es lógico, ya que rendimiento y redundancia tienen que sumar 1.

e)En el inciso e), decidimos aplicar el Algoritmo de Huffman. Éste algoritmo está diseñado para codificar los símbolos de una fuente con alfabeto r-ario a un alfabeto binario. El propósito principal de la codificación a alfabeto binario es comprimir el archivo original.

El archivo brindado por la cátedra tiene un tamaño de ≈ 10 kB. En tanto que los archivos obtenidos por el algoritmo de Huffman dieron como resultado un tamaño de:

Fuente de 3 caracteres: ≈ 3 kB
Fuente de 5 caracteres: ≈ 2 kB
Fuente de 7 caracteres: ≈ 1 kB

Por lo que para el primer caso, se obtuvo una compresión de un 70%, para el segundo caso, una compresión de un 80%, y para el último caso, un 90%.

Estos resultados son los encontrados si no ingresamos en el mismo binario la tabla con las traducciones de binario a alfabeto código, si agregamos las tablas en los archivos binarios, los pesos no cambian de manera notable. Para lograr ingresar la tabla sin comprometer la compresión del algoritmo transformamos las palabras código del alfabeto r-ario en un Integer que equivale a la suma de los caracteres de la palabra código en valor ASCII.

A continuación, adjuntamos un pseudocódigo del Algoritmo de Huffman utilizado en el código fuente:

PSEUDOCÓDIGO DE FUNCIÓN PRINCIPAL MÉTODO HUFFMAN

```
Función void MetodoHuffman(map){ //Funcion recursiva
   //inicialización de variables locales
   if(map tiene más de 2 elementos){
      Búsqueda de los 2 elementos de menor probabilidad
      Remuevo ambos elementos del map
      Agregó un nuevo elemento con la probabilidad de los dos eliminados sumadas
      MetodoHuffman(map); //Llamado recursivo
      Se recuperan los 2 elementos  anteriores
      Se reemplaza sus identificadores por el identificador padre más 0 o 1.
      Se remueve el elemento padre
      Se agrega nuevamente los 2 elementos que se encontraban anteriormente
   }else{
      Reemplazo el identificador del primer elemento restante por 0.
      Reemplazo el identificador del segundo elemento restante por 1.
   }
}
```

Apéndice

(1):

```
0.27149832248687744  0.26289990544319153  0.26610806584358215
0.3803980350494385  0.3890608847141266  0.39150533080101013
0.3481036424636841  0.3480392098426819  0.342097669839859
```

Símbolo	Probabilidad condicional más alta	Probabilidad condicional más baja	Diferencia
A	0.271	0.262	0.009
B	0.391	0.380	0.011
C	0.348	0.342	0.006

(2):

```
Entropia inicial = 0.9893678631344445
Entropia Orden 20 = 19.78735726268889
```

(3):

```
la inecuacion de kraft para el primer caso es 0.9999999999999993
la inecuacion de kraft para el segundo caso es 0.9999999999999968
la inecuacion de kraft para el tercer caso es 0.4563328760859707
```

(4):

```
Longitud media de codigo de 3 caracteres es = 3.0
Longitud media de codigo de 5 caracteres es = 5.0
Longitud media de codigo de 7 caracteres es = 6.999999999999977
```

(5):

```
Entropia de fuente de 3 caracteres es = 2.9655102414935146
Entropia de fuente de 5 caracteres es = 4.900100409558381
Entropia de fuente de 7 caracteres es = 6.186111818549913
```

(6):

```
El rendimiento de 3 caracteres es 0.9885034138311716
El rendimiento de 5 caracteres es 0.9800200819116762
El rendimiento de 7 caracteres es 0.8837302597928476
La redundancia de 3 caracteres es 0.011496586168828427
La redundancia de 5 caracteres es 0.019979918088323778
La redundancia de 7 caracteres es 0.11626974020715242
```

(7): Tuvimos problemas con la librería al hacer un deploy del proyecto en Jdeveloper, donde el .jar resultante no reconocía las clases que aportaba la librería. Por esto, los ejecutables de los ejercicios tienen una versión ligeramente del código, sin usar la librería. Lo cual solo afecta a una función del ejercicio 1, la cuál por la condición del inciso no fue necesaria resolverla.

Conclusión

Para finalizar, podemos mencionar los aspectos más importantes de los códigos analizados:

- Se determinó que la primera fuente era una fuente de memoria nula. Es decir, emite símbolos de manera aleatoria, por lo que cada símbolo tiene la misma probabilidad de ocurrencia sin importar el símbolo anterior.
- Al separar los códigos en longitudes diferentes, se constató que eran códigos instantáneos, por lo que se pudo aplicar el Algoritmo de Huffman para traducirlo a lenguaje binario y así comprimir el tamaño de cada archivo respecto del archivo original.
- Todo lo anterior mencionado, siempre fue justificado con cada propiedad cuantificada, como por ejemplo, la entropía, la longitud media del código, etcétera.