TEORÍA DE LA INFORMACIÓN

Trabajo Práctico Integrador N°1

Grupo 9

Integrantes:

Ignacio Suárez Quilis

Lautaro Nahuel Bruses

Matías Angélico

E-mail de contacto:

ignaciogustavosuarez1234@gmail.com

lautarobruses@gmail.com

mnangelico@gmail.com

GitHub: https://github.com/MouGliXx/teoria-de-la-informacion

2. Índice

3. Resumen

El trabajo se enfocó en estudiar una fuente de información con tres símbolos {A, B, C}. La misma resultó ser de memoria no nula por lo que se determinó que es ergódica, estableciendo su vector estacionario. Para finalmente calcular su Entropía.

Posteriormente, se dividió la fuente en palabras de 3, 5 y 7 caracteres de extensión. Se identificaron las palabras código de cada alfabeto con su respectiva frecuencia, para luego obtener la cantidad de información que proporciona cada palabra, y en base a esta la entropía de cada código. Se clasificaron los códigos en base a sus características, resultando todos códigos instantáneos, debido a la naturaleza del problema. Se establecieron las inecuaciones de Kraft y McMillan y además, se calculó la longitud media del código. A partir de los datos hallados previamente, se analizó el rendimiento y la redundancia de cada código. Para finalizar, se trabajaron los símbolos de los códigos a través de la codificación de Huffman.

4. Introducción

La Teoría de la Información actúa como una respuesta a los problemas técnicos del proceso de comunicación. En esta teoría, predomina un elemento clave que es la fuente de información, la cual no es más que un conjunto finito de mensajes. El objetivo de este informe será analizar en detalle una fuente de 10.000 caracteres aleatorios prevista por la cátedra.

En la primer parte del informe se calcularán las probabilidades condicionales de una fuente, para determinar la clasificación de la misma según la relación entre sus símbolos. Esta clasificación,

determinará si se generará la extensión de orden 20 de la fuente o se determinará si la misma es ergódica, para establecer su vector estacionario. En ambas posibilidades se calculará la entropía.

Dentro de la segunda parte del informe, se dividirá la fuente de información en cadenas, de igual longitud, que representen palabras de un código. Se identificarán las palabras código y en base a ellas su frecuencia. Calcularemos la cantidad mínima de bits que se necesitan para representar esa cada palabra y la cantidad de bits necesarios para representar el código. Luego clasificaremos los códigos según sus características. Posteriormente, utilizamos la ecuación de Kraft-McMillan para aplicar sus enunciados y calcularemos la longitud media del código para concluir si los códigos trabajados son compactos o no. Continuaremos calculando la redundancia y el rendimiento de las cadenas para finalizar aplicando el algoritmo de Huffman que nos permitirá comparar los resultados obtenidos con aquellos obtenidos a partir del archivo dado.

5. Desarrollo

5.1 Primera parte

En esta primera etapa, el trabajo consistió en la lectura y el análisis de un archivo con 10.000 caracteres aleatorios, provisto por la cátedra, el cual simulaba una fuente de información.

5.1.1 Recolección de símbolos

Se inició leyendo el archivo y asignando sus valores a una variable de tipo String. Luego se recorrió caracter a caracter esta variable para verificar la cantidad y los símbolos de la fuente, dándole a cada uno de estos un valor mediante una tabla Hash Ordenada (*TreeMap* en Java), el cual corresponde con su posición en la matriz de probabilidades.

```
símbolos -> Tabla Hash
generaHashSimbolos( datos) {
    c=0
   for (t = 1 to largo (datos) )
      letra = datos en pos t
      if símbolos NO contiene (letra)
         poner en símbolos (llave ->letra, valor -> c);
         c += 1;
}
```

5.1.2 Probabilidades condicionales

Una vez concluido esto, se calcularon las probabilidades condicionales (que un símbolo se de si se dio otro), acumulando la aparición de cada uno en la matriz de la siguiente forma: el valor de la tabla Hash de posición actual refiere a la columna mientras que el valor de la posición anterior nos da el valor de la fila. Por último se dividieron los valores de la columna por la suma total de cada una de ellas.

```
n -> dimensión(matriz)
símbolos -> Tabla Hash
generarMatrizProbabilidades(datos) {
   probabilidades = matriz [n][n];
   totales = vector [n];
   for (i = 1 to largo (datos))
```

```
      columna = obtener de simbolos (datos (i));
      fila = obtener de simbolos (datos (i-1));
      probabilidades [fila][columna] = + 1;
      totales[columna] = + 1;

   for (i = 0 to n)
      for (j = 0 to n)
        probabilidades [i][j] = probabilidades [i][j] / totales[j];
   retornar probabilidades
}
```

Lo que resultó en la siguiente matriz de probabilidades condicionales:

$$M_{ij} = \begin{pmatrix} 0.31949 & 0.30200 & 0.29000 \\ 0.50774 & 0.28396 & 0.32658 \\ 0.17277 & 0.41404 & 0.38342 \end{pmatrix}$$

5.1.3 Clasificación de la fuente

Con la matriz de probabilidades completa, se verificó si la fuente es o no de memoria nula, corroborando que las probabilidades de “llegada” a un símbolo tengan el mismo valor.

```
memoria (matriz) {
  memoriaNula = falso
  while (columna < nro columnas y memoriaNula = falso)
        while (fila < nro filas -1 y memoriaNula = falso)
              if (matriz[fila][columna] != matriz[fila][columna +1])
                    memoriaNula = verdadero;
              c +1
        f+1
  retornar memoriaNula
}
```

Como en nuestro caso, el método anterior retornó que la fuente es de **memoria no nula**, se procedió a verificar si la misma era ergódica. Para ello, se recorrió la matriz de probabilidades por filas chequeando que no todas las filas, salvo la misma de la entrada, sean iguales a cero, lo que no permitiría “salir” de ese estado.

```
ergódica(matriz) {
    ergódica = verdadero
     while (ergódica y columna < nro de columnas)
          suma = 0;
          for (fila = 0 to fila < nro de filas)
                if (fila != columna)
                      suma = suma + matriz[fila][columna]
          if (suma = 0)
                ergódica = falso
          c + 1
  retornar ergódica
```

```
}
```

5.1.4 Vector estacionario

Ya que la fuente a analizar resultó **ergódica** se calculó su vector estacionario resolviendo el siguiente sistema:

$$(M - I) * V = 0$$

Se comenzó restando a la matriz de probabilidades, la matriz identidad de orden n. Luego se creó una nueva matriz de orden (n, n+1), tomando los valores de la resta de matrices y colocando en la nueva columna los ceros correspondientes a la solución del sistema, a la cual se llamó: matriz Ampliada.

Posteriormente se triángulo esta matriz por el método de Gauss y se impuso en la última fila, la cual se había hecho cero por la triangulación, la condición :

$$V1 + V2 + V3 + ... + Vn = 1$$

Paso seguido se volvió a triangular la matriz resultante por Gauss, y se hizo una sustitución regresiva obteniendo los valores del Vector estacionario.

```
triangulacionGauss(mAmpliada) {
  for (i = 0 to nro de Filas) {
    for (t = i + 1 to nro de Filas + 1)
       mAmpliada[i][t] = mAmpliada[i][t] / mAmpliada[i][i];
    mAmpliada[i][i] = 1;
    for (j = i+1 to nro de Columnas )
      for (r = i + 1 to nro de Columnas +1)
        mAmpliada[j][r]= mAmpliada[j][r]-mAmpliada[i][r]*mAmpliada[j][i];
      matrizAmpliada[j][i] = 0;
  retornar mAmpliada
}

n-> nro Filas
m-> nro Col
sustitucionRegresiva(mAmpliada) {
  vEstacionario = vector [nro Filas];
  vEstacionario[n] = mAmpliada[n][m] / mAmpliada[n][m];
  for (f = n - 1 down to 0) {
     suma = 0;
     for (c = f+1 to nro columnas)
        suma += mAmpliada[f][c] * vEstacionario[c];
     vEstacionario[f]= (mAmpliada[f][n] - suma) / mAmpliada[f][f]
  retornar vEstacionario
}
```

Con lo cual el vector resultó ser el siguiente:

$$V_{estacionario} = (\ 0.30333\ ;\ 0.36593\ ;\ 0.33073\ )$$

5.1.5 Entropía

Para concluir con la primera parte se calculó la entropía de la fuente. Se obtuvo la cantidad de información de cada símbolo (previamente se verificó que la probabilidad no sea igual a cero, para no tener una indeterminación) y se la multiplicó por su probabilidad asociada, acumulando estos valores en una variable. Por último, se le fue sumando los productos de cada fila por el valor correspondiente del vector estacionario.

$$H = \sum_{i=1}^{m} V_i \cdot \sum_{j=1}^{n} P_{ij} \cdot \log\left(\frac{1}{P_{ij}}\right)$$

Se obtuvo el siguiente resultado:

H = 0.97 bits/símbolo

5.2 Segunda Parte

Inicialmente, en base a la variable en la que se almacenaron los valores del archivo, se implementó un algoritmo para separar la cadena de 10.000 caracteres en palabras de *N* caracteres.

5.2.1 Codificación de la Información

En nuestro caso, se separó en 3 *ArrayList*, los códigos correspondientes a palabras de 3, 5 y 7 caracteres. Existen casos en donde N no es divisor de 10.000, esto genera que el último elemento de la estructura contenga una cantidad de caracteres menor a N. Para evitar esto, se incluyó una condición final que verifica que el resto de la división entre ambos números sea 0, que en caso de no cumplirse, se elimina el último elemento de la estructura, logrando así que todas las palabras del código final posean la misma cantidad de caracteres.

```
diferenciaPalabras (datos, cantCaracteres) {
  código = new ArrayList
  i = 0
  while (i < longitud(datos)) {
    aux = “”
    for (k = 0 to cantCaracteres)
      if (i + k < longitud(datos))
          aux = aux + datos[i + k]
    add aux in código
    i = i + cantCaracteres
  }
  if (longitud(datos) % cantCaracteres != 0)
    remove último elemento in código
return codigo
}
```

Para cada código, se identificó sus respectivas palabras, filtrando las que se encontrasen repetidas. El resultado de este procedimiento se almacenó en 3 *ArrayList*. Luego, se calcularon las frecuencias de cada palabra, almacenándolas en un *HashMap*, en donde la *llave* de la estructura es la palabra, y su *valor* asociado es la frecuencia correspondiente a esa palabra.

```
calculaFrecuencias(datos) {
  frecuencias = new HashMap
  for (palabra in datos)
    int j = frecuencia of palabra
    if (j == null)
      put (palabra, 1) in frecuencias
    else
      put (palabra, j + 1) in frecuencias
 return frecuencias
}
```

5.2.2 Información y entropía

A continuación, se calculó la cantidad de información que contiene cada palabra. Para esto, a partir de las frecuencias halladas anteriormente y el número total de palabras código (incluyendo las repetidas) se obtuvieron sus probabilidades absolutas. Luego, almacenamos en un *HashMap* la palabra código como *llave* y la cantidad mínima de bits que se necesitan para representar esa información como *valor* asociado. Cabe mencionar, que al no estar presente en el lenguaje, la posibilidad de trabajar directamente con el logaritmo en base “r”, se programó un método que lo resuelve, gracias a la propiedad de los logaritmos de cambio de base. En donde “r” tomará el valor del número de símbolos diferentes que constituyen el alfabeto código, en nuestro caso 3.

```
calculaInformacion(frecuencias, total) {
  informacion = new HashMap
  for (palabra, frecuencia) in frecuencias {
    probabilidad = frecuencia / total
    inf= logOrden(1 / probabilidad)
    put (palabra, inf) in informacion }
return informacion
}
```

Con el resultado anterior, y la probabilidad de cada palabra, se calculó la cantidad de unidades de información necesarias para representar el mensaje que se desea transmitir, es decir la entropía de la fuente. En este caso, las fuentes son los distintos códigos que se obtuvieron anteriormente a partir de los datos. Para calcular la entropía se acumuló el resultado del producto entre la probabilidad y la información de cada palabra contenida en el código en una variable.

```
calculaEntropia(código, informacion, frecuencias, total) {
  resultado = 0
  for palabra in código
    probabilidad = (valor of palabra in frecuencias) / total
return resultado
}
```

5.2.3 Clasificación de los códigos

Posteriormente, se pasó a clasificar los códigos obtenidos. Para cada clasificación se utilizó un método para determinar si un código pertenece o no a dicha clasificación.

En primer lugar, para determinar si un código es bloque o no bloque, se verificó que cada carácter de las palabras código esté contenido en el alfabeto {A, B, C} y que todos los símbolos del código tengan palabras.

```
esCodigoBloque(simbolos, codigo) {
  for i = 0 to size(codigo)
    palabra = codigo[i]
    if (palabra is empty)
      return false
    for i to longitud(palabra)
      if (simbolos not contains palabra[i])
        return false
  return true
}
```

Después, para determinar si un código es singular o no singular, se verificó que ninguna palabra esté repetida, es decir que no tenga una frecuencia mayor a 1.

Luego, para determinar si un código es unívocamente decodificable o no, se verificó que se cumpla la condición necesaria de la ecuación de McMillan, la cual retomaremos más adelante.

Por último, para determinar si un código es instantáneo o no instantáneo, se verificó que ninguna palabra coincide con el prefijo de otra, en otras palabras, que ninguna palabra empiece con otra del código.

```
esInstantaneo(ArrayList<String> codigo) {
  for palabraPrefijo in codigo
    for palabraIterada in codigo
      if (palabraPrefijo != palabraIterada )
        if (palabraIterada starts with palabraPrefijo)
          return false
  return true
}
```

5.2.4 Propiedades de los códigos

La inecuación de Kraft permite conocer, a partir de un conjunto de cadenas código con su longitud, la condición suficiente para que un código sea instantáneo. Siendo que los códigos unívocos incluyen a los instantáneos entonces se puede ver que si la inecuación se cumple entonces existe un código unívoco. Por su parte, McMillan demostró la recíproca, es decir que si existe un código unívoco entonces se cumple la inecuación. La ecuación previamente mencionada responde a la siguiente fórmula:

$$\sum_{i=1}^{q} r^{-l_i} \leq 1$$

Siendo **i** el número de palabras del código, **r** el número de símbolos diferentes que constituyen el alfabeto del código y, **l** la longitud de la palabra.

Por su parte, la longitud media de un codigo **L** estara dada por la siguiente inecuacion:

$$L = \sum_{i=1}^{q} \rho_i \cdot l_i$$

Siendo ***q*** el número de palabras del código, **p** la probabilidad de aparición, **l** la longitud de la palabra.

Un código unívoco será compacto si su longitud media es igual o menor que la longitud media de todos los códigos unívocos que pueden aplicarse a la misma fuente y al mismo alfabeto.

Cantidad caracteres	Inecuación Kraft-McMillan	L	H(S)
3	1.0	3	2.9375
5	0.99	5	4.8153
7	0.43	7	6.1082

Como se puede observar, la inecuación *Kraft-McMillan* se cumple en todos los escenarios. Este resultado resulta coherente, ya que al tratarse de cadenas de igual longitud, ninguna podrá ser prefijo de otra por lo que los códigos son unívocamente decodificables en todos los casos. Esto se puede reflejar en el enunciado de McMillan el cual nos dice que, en caso de ser unívocos cumpliran la inecuación.

Por su parte, un código será compacto respecto a S si el mismo es unívoco y será el de menor longitud. Es decir, su longitud media deberá ser igual o menor que la longitud media de todos los códigos unívocos.

Para nuestros 3 escenarios la codificación trabajada es unívocamente decodificable e instantánea. Al trabajar con cadenas de igual longitud para todos los casos, podremos afirmar que el código será el de menor longitud únicamente si la probabilidad de aparición de cada una de las cadenas posibles es la misma, es decir si las palabras código resultan equiprobables. Para el escenario con cadenas de 3 caracteres, las 27 cadenas posibles están presentes pero no en igual frecuencia. Por su parte, para aquellas de longitud de 5 y 7 caracteres con 243 y 2187 cadenas posibles respectivamente solo tienen 240 y 939 cadenas presentes por lo que su frecuencia de aparición no será la misma para todas las cadenas posibles, determinando así que los códigos trabajados no son compactos.

Respecto al código realizado, iteramos por todas ellas para saber si su longitud es menor o igual al resultado del cálculo de longitudes mínimas asociado a su frecuencia redondeado para arriba, en caso de cumplir la condición sabremos que será el de menor longitud asociada a su probabilidad y continuaremos iterando. En caso de encontrar una cadena que no cumpla dicha condición podremos afirmar que el código no será compacto.

5.2.5 Rendimiento y redundancia

La redundancia es la información superflua o innecesaria para interpretar el significado de los datos originales. Una mayor redundancia implica menor información.

Definiremos el rendimiento o eficiencia de un código como:

$$\eta = \frac{H_r(S)}{L}$$

Mientras que la redundancia, será 1-η.

Para los ejemplos trabajados se obtuvieron los siguientes resultados:

Cantidad caracteres	Rendimiento	Redundancia
3	0.9792	0.0208
5	0.9631	0.0369
7	0.8726	0.1274

Como podemos observar en los resultados obtenidos, a medida que aumentamos el largo de las cadenas de igual manera lo hace la redundancia. Esto se debe a que el rendimiento disminuye con cadenas de mayor longitud producto del aumento en la diferencia entre esta y la entropía.

5.2.6 Codificación de la Información

Para finalizar, se decidió realizar la codificación de los códigos mediante el algoritmo de Huffman. Para trabajar con el árbol se debió crear una estructura de tipo nodo, la misma será la siguiente:

```
class Nodo {
  String cadena;
  Integer frecuencia;
  Nodo izq = null, der = null;
}
```

El algoritmo se basa en el uso de una cola de prioridad con un comparador que permite saber qué nodo tiene menor frecuencia. Se pasaron los datos por parámetro, que contendrán las cadenas con sus respectivas frecuencias y se conformó una cola con las cadenas ordenadas por frecuencia. Se iteró mientras la longitud de la cola sea mayor a 1, en cada ciclo se sacara los dos últimos nodos que serán los de menor frecuencia y se insertará un único nodo con la frecuencia resultado de la suma, teniendo como hijos a izquierda y derecha los nodos extraídos previamente de la cola. Luego, con la función encode se recorre el árbol concatenando dentro de un parámetro llamado "**str**": 0 o 1, al llegar a una hoja la recursión se detiene. Es así como se obtuvo una codificación que relaciona el árbol de Huffman con la original.

Se generaron dos archivos por cada uno de los 3 escenarios, uno de texto con las cadenas y sus prefijos asociados y otro binario con la reconstrucción del archivo original bajo la codificación de Huffman.

6. Conclusiones

Al estudiar el comportamiento de la fuente a partir de una secuencia de símbolos, pudimos determinar que la misma es de Memoria No nula o de Markov para orden igual a uno, o sea que, la presencia de un determinado símbolo depende de su precedente.

A partir de este dato, determinamos que la misma es ergódica, es decir que todos los estados del proceso son alcanzables desde otro estado, o lo que es lo mismo, que no hay estados o clases absorbentes.

De esta forma y teniendo en cuenta que el conjunto de datos es finito nos aseguramos que esta distribución, la cual recibe el nombre de distribución estacionaria del proceso ergódico de Markov, puede calcularse directamente a partir de las probabilidades condicionales, obteniendo así la distribución de probabilidades estabilizada, o sea, el vector estacionario.

Para concluir, la entropía calculada fue de aproximadamente 1,5 con lo que podemos afirmar que esa es la información media por símbolo para esta fuente.

Posteriormente, al considerar las cadenas como las palabras de un código de N caracteres, a simple vista se observa que a medida que aumentamos el valor de N, también lo hace considerablemente la cantidad total de palabras código. Esto debido a que la probabilidad de que haya una o más palabras iguales disminuye a medida que N aumenta.

Como consecuencia de este fenómeno, las frecuencias asociadas a cada palabra código, irán disminuyendo a medida que N aumente. Esto influirá directamente en la cantidad mínima de unidades de información que se necesitarán para representar cada código. Ya que, en base a que las palabras se tornen menos probables, a medida que N aumente, aportaran más información.

El mismo efecto ocurre con la entropía. La cantidad media de información por símbolo del código, siempre será menor o igual a la cantidad de caracteres que formen a cada palabra, en otras palabras, a N.

Al momento de clasificar los códigos, analiticamente se puede determinar a simple vista que todos los códigos serán instantáneos. Pues desde un principio para todos los códigos, ninguna palabra se repite, por lo que los códigos serán no singulares. Y al ser todas las palabras de la misma longitud, los códigos serán unívocamente decodificables. Y en consecuencia, ninguna palabra será prefijo de otra, es decir que los códigos serán instantáneos.

Al analizar la inecuación junto a los enunciados de Kraft-McMillan concluimos que existe un código instantáneo de las dimensiones dadas y además que las cadenas trabajadas resultan unívocamente decodificables para los 3 códigos. Además, pudimos concluir que en los 3 escenarios posibles los mismos cuentan con la propiedad de no ser compactos.

Luego, procedimos a calcular el rendimiento y la redundancia de cada escenario pudiendo observar que a mayor longitud de cadena el rendimiento iba disminuyendo. Esto resulta lógico ya que la diferencia entre el rendimiento y la longitud media aumenta a medida que lo hace la longitud de las cadenas y por ende el rendimiento disminuye.

Finalmente utilizamos la codificación de Huffman para poder comprimir nuestro archivo original para cada código posible mediante la utilización de prefijos, los cuales serán menores en aquellas cadenas más frecuentes.