Teoría de la Información
Trabajo Práctico Especial

Integrantes

- Mateo Marcos (marcosmateoiae@gmail.com - 249202)
- Emiliano Escudero (emi.escudero1998@gmail.com - 249199)
- Lisandro Ledesma (ledesmalisandro22@gmail.com - 249265)

Introducción

En este informe se detallan los procedimiento utilizados para la resolución de los ejercicios planteados en el Trabajo Práctico Especial de Teoría de la Información. El trabajo consiste en utilizar los conceptos y metodologías enseñados por la cátedra para analizar el comportamiento del valor de dos criptomonedas (Etherium y Bitcoin) en diferentes instantes de tiempo, en este caso, cada 4 horas. Estos valores se nos dieron en 2 archivos de texto, y en base a estos los analizamos.

Desarrollo y Análisis

1 - Fuentes de Información y Muestreo Computacional

a. Pseudocódigo de la Matriz de Pasaje

El cálculo de la matriz de pasaje de cada moneda se realiza cargando todos los valores del archivo de la moneda específica en un arreglo llamado moneda. Luego, se recorreran todos sus valores registrando en cada iteración la incidencia (Si subió con respecto al valor anterior, se mantuvo o bajó) para al final devolver una matriz de 3 filas y 3 columnas con las probabilidades obtenidas al dividir cada combinación de fila y columna por el total de incidencias en por la columna en cuestión.

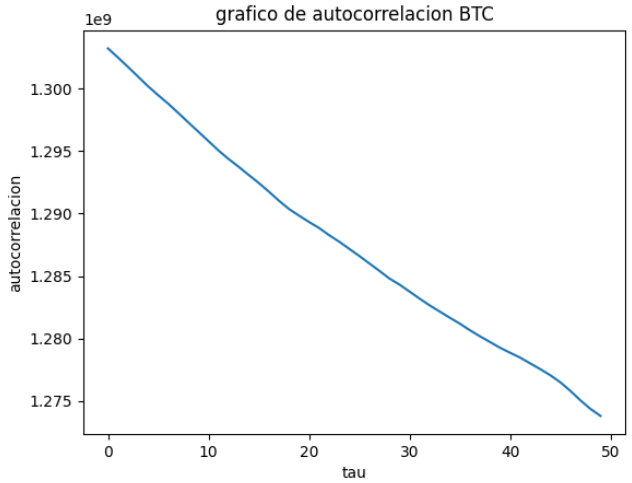
```
    def calc_pasaje(moneda):
        retorno = [[0,0,0],[0,0,0],[0,0,0]]
        tactual, ultretorno, ultretornocolumna, s = 0,0,0,0
        retorno, totalretoronos = [0,0,0], [0,0,0]
        while not converge or t_actual <1000:
             s = moneda[tactual]
             tactual += 1
             totalretornos[ultretornocolumna] += 1

            retorno[X][ultretornocolumna] +=1 #Si el nuevo retorno es mayor,
igual o menor al ultimo retorno
            ultretornocolumna = X
            ultretorno = s
            #recorrer retorno con dos indices
            retorno[i][j] = retorno[i][j]/totalretornos[j]
            return retorno
```

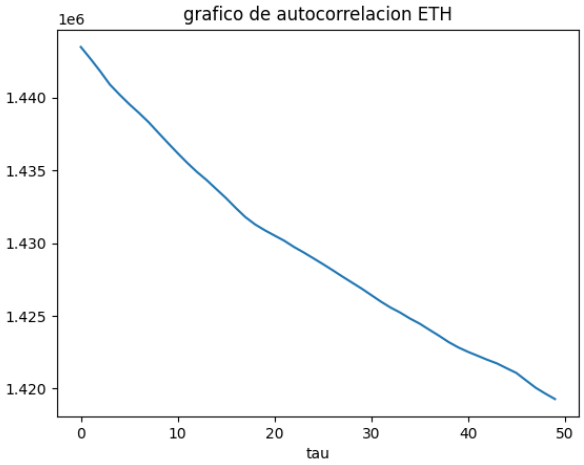
b. Pseudocódigo de la Autocorrelación

Para el cálculo mediante muestreo computacional de la autocorrelación primero seteamos todas las variables que se van a usar, en este caso, un vector $autocorrelacion$ de tamaño 50, una $autocorrelacion\_simple$ que va a ir haciendo la suma de la autocorrelación en cierto valor del *tau*; se setea una variable $tau = 1$ que va a ser el primer valor del *tau* y una variable t_actual que lleve la cuenta de la posición actual en el vector de valores de la moneda.

Una vez seteadas todas las variables, se comienza a calcular la autocorrelación. Para ello usando un while para calcular cada autocorrelación de cada *tau* (en este caso, de 1 a 50). Dentro de este while se tiene otro while, que hace el cálculo de la autocorrelación para un *tau* específico. Dentro del while de la autocorrelación se obtienen 2 valores de la monedas, el del tiempo actual ($s1 = moneda[t\_actual]$) y el del desplazamiento del *tau* ($s2 = moneda[t\_actual + tau]$). Una vez obtenidos estos 2 valores, se multiplican y el valor obtenido de esa multiplicación se suma a la variable $autocorrelacion\_simple$ que lleva el valor de todas las multiplicaciones hechas para calcular la autocorrelación.
Una vez que se calcula la autocorrelación para un *tau* específico, en el vector $autocorrelacion$ en la posición tau le asigno el valor de la división entre $autocorrelacion\_simple$ y t_actual, obteniendo así la autocorrelación para un *tau* específico. Luego vuelvo a setear t_actual en 0 y a tau le sumo 1, para que en la próxima iteración del while, calcule la autocorrelación del *tau* siguiente.
Al finalizar el cálculo de todas las autocorrelaciones necesarias, devuelvo el vector $autocorrelacion$.

```
calcular_autocorrelacion(moneda){
    autocorrelacion = [0,0,...,0,0] //50 ceros ya que se van a obtener
                                    //50 autocorrelaciones!
    autocorrelacion_simple = 0
    tau = 1
    t_actual = 0
    while(tau < 50){
        while(tau + t_actual < moneda.size){
            s1 = moneda[t_actual]
            s2 = moneda[t_actual+tau]
            t_actual++
            autocorrelacion_simple = autocorrelacion_simple + s1*s2
        }
        autocorrelacion[tau] =  autocorrelacion_simple/t_actual
        t_actual = 0
        tau++
    }
    return autocorrelacion
}
```

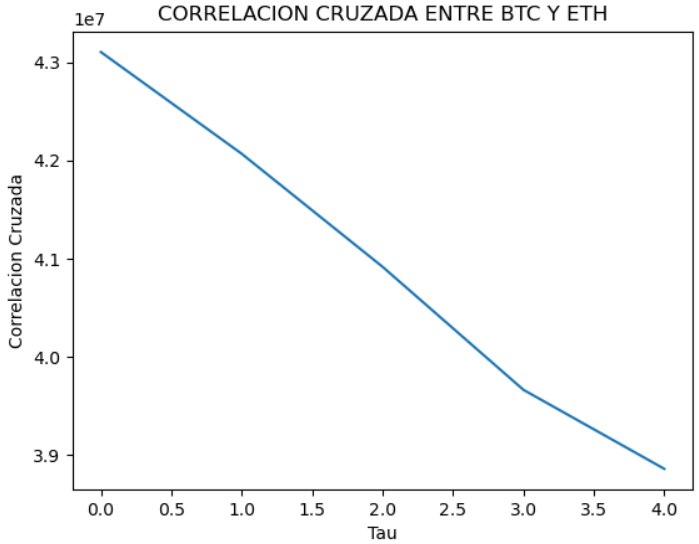




c. Pseudocódigo de la Correlación Cruzada

Para calcular la correlación cruzada debemos inicializar las variables que se utilizaran para los cálculos, un vector $arreglo$ de tamaño 5 con dos posiciones donde se almacenarán el valor de $max\_correlacion$ (la correlación entre las monedas) y el tau utilizado.

Para calcular la correlación cruzada primero comenzamos recorriendo un while por cada tau que se desea analizar (0, 50, 100, 150, 200), dentro de dicho while se encuentra otro encargado de calcular la correlación entre ambas monedas. Se obtienen 2 valores de la monedas, el del tiempo actual ($d1 = moneda1[t\_actual]$) y el del desplazamiento del *tau* ($d2 = moneda2[t\_actual + tau]$). Una vez obtenidos estos 2 valores, se multiplican y el valor obtenido de esa multiplicación se suma a la variable acc que lleva el valor de todas las multiplicaciones hechas para calcular la autocorrelación, siendo posteriormente dividida entre la cantidad de iteraciones totales, obteniendo así el valor de la correlación cruzada en un tau específico, los cuales son almacenados en el vector $arreglo$. Luego se actualiza nuevamente t_actual en 0 y se repite el while con el siguiente tau hasta finalizar, devolviendo el vector $arreglo$ con sus respectivos valores.



En el eje x (*tau*), el rango es de 0 a 4, pero cada valor entero representa los valores de *tau* que se usan para el ejercicio, por ejemplo, cuando el eje x vale 2, está representando al valor *tau* = 100.

```
def calcular_corr_cruz(moneda1, moneda2):
    arreglo = [[0,0],[0,0],[0,0],[0,0],[0,0]] /primera posicion valor de la
                                                corralacion, segunda posicion
                                                valor del tau utilizado
    max_correlacion = 0
    tau = 0
    i = 0
    while (tau <= 200):         /itera por cada valor que tome tau
       acc = 0
       max_correlacion = 0
       t_actual = 0
       while (t_actual + tau < moneda.size):    /correlacion entre las monedas
            d1 = moneda1[t_actual]
            d2 = moneda2[(t_actual + tau)
            acc = acc + (d1 * d2)
            j++
        max_correlacion = acc/t_actual
        arreglo[i] = max_correlacion, tau
        tau = tau + 50
        i++
    return arreglo
```

d. Análisis de los Indicadores

Autocorrelación:
La autocorrelación es una herramienta estadística utilizada frecuentemente en el procesamiento de señales.
Como el gráfico generado por el código de autocorrelación en ambas monedas es descendente, podemos decir que a medida que se mueve el *tau* hay menos relación entre los símbolos de la fuente. Por lo tanto a mayor *tau* menor relación tienen los símbolos, esto quiere decir que la acción (ya sea subir, bajar o mantener su valor) que tomará la fuente para el siguiente símbolo está muy relacionada con el valor inmediatamente anterior, mientras que los valores antecesores a este no influyen en la acción que toma. Esto tiene sentido, ya que los valores que toma una criptomoneda no dependen de los valores anteriores, si no que de la demanda actual de la misma.

Correlación Cruzada:
La correlación cruzada mide el grado de relación lineal entre dos señales. Cuando se obtiene un valor, se cuantifica la semejanza entre estas formas de onda. A mayor valor de correlación cruzada (Normalizada sería máximo 1), el ajuste tiende a ser perfecto. Es decir, las señales emitidas son muy similares (Ambas funciones crecen, se mantienen o decrecen).

Viendo el gráfico generado por el código de la correlación cruzada, se deduce algo similar a lo sucedido con la *autocorrelación*, que a mayor *tau* hay menor relación entre las señales de las fuentes BTC y ETH, por lo que podemos decir que no hay una relación fuerte entre ambas señales ya que mientras más se desplace una fuente de otra, el valor de la correlación cruzada va disminuyendo.

2 - Codificación y Compresión

a. Distribución de Probabilidades

Para calcular la distribución de probabilidades de las cotizaciones de la moneda, simplemente utilizamos un diccionario (clave, valor) recorremos cada arreglo (Que contiene todos los valores de una moneda extraídos del archivo txt) y por cada iteración verificamos si hay reincidencia. Si la hay, se aumenta en 1 en valor y si no se crea una nueva entrada al diccionario con 1 como valor y el número como clave. Finalmente se divide cada elemento en el diccionario por el tamaño del arreglo, dando a lugar la distribución de probabilidades cuya sumatoria dará 1.

```
    def distribucion_probabilidad(moneda):
        dict = {}
        i = 0
        tam = len(moneda)
        while (i < tam):
            dato = int(moneda[i])
            if (dato in dict):
                dict[dato] += 1
            else:
                dict.update({dato: 1}) #Agregamos el dato como nuevo
            i += 1
        para cada dato, valor en dict:
            dct[dato] = valor/tam
        return dict
```

b. Codificación Huffman Semi-estático

Para hacer la codificación de cada fuente, nos basamos en el siguiente pseudocódigo:

```
def get_codification(self, fuente, prob):
        #armo el arbol y obtengo la raiz
        raiz =calcular_huffman_semiestatico(prob) //obtengo la raiz de arbol
        codigos = {}
```

```
        get_codigo(raiz,codigos) //genero el codigo para cada valor
        i = 0
        tam = len(fuente)
        retorno = ""
        #empiezo a generar el codigo de la fuente
        while(i < tam):
            valor =fuente[i]
            retorno = retorno + codigos[valor] //agrego a la variable a
//retornar el codigo del valor
            i += 1
        return retorno
```

```
calcular_huffman_semiestatico(prob):
        nodos = self.getNodos(prob) //obtengo una lista ordenada con todos los
      //nodos hijos del arbol
        val_nodo1 = nodos[0].getProb()
        val_nodo2 = nodos[1].getProb()
        while(val_nodo1 + val_nodo2 != 1):
            nodo1 = nodos.pop(0) //quito los nodos usados de la lista de nodos
            nodo2 = nodos.pop(0)
            suma =nodo1.getProb() + nodo2.getProb()
            new_nodo = Nodo(suma, nodo1, nodo2)//nuevo nodo del arbol
            nodos.insert_order(new_nodo) //inserto el nuevo nodo de fomra
      //ordenada
            val_nodo1 = nodos[0].getProb()
            val_nodo2 = nodos[1].getProb()
        nodo1 = nodos.pop(0)
        nodo2 = nodos.pop(0)
        raiz = Nodo((val_nodo1 + val_nodo2),nodo1,nodo2)
        return raiz
```

Lo primero que hace este pseudocódigo es generar los nodos hojas del árbol de huffman, para luego tenerlos en una lista ordenada de menor a maoyr en la cual voy a ir sacando siempre los 2 menores y de ahí hacer la suma de sus probabilidades, y generar un nuevo nodo con esas probabilidades y como hijos de ese nodo, los nodos que saqué de la lista anteriormente; luego se agrega ese nuevo nodo a la lista de forma ordenada y repite el proceso hasta que la suma de probabilidades de los nodos que se sacaron de la lista sea 1. Una vez que esto sucede, se genera un nuevo nodo, el nodo raíz, con probabilidad 1 y nodos hijos, estos dos últimos nodos que quedaron en la lista. Y se retorna el nodo raíz para poder hacer el seguimiento del árbol.

Una vez que obtengo la raíz se genera el código para cada nodo del árbol y se guardan en un diccionario, asociando el código generado a cada valor. Por último, el por cada valor de la fuente, se agrega su código a un string retorno, que al finalizar el while contendrá el código que se genera para la fuente dada.

c. Codificación RLC

d. Análisis de las tasas de compresión

Valores a tener en cuenta:

- Peso BTC.txt → 6830 B
- Peso ETH.txt → 5390 B

Mediante el algoritmo de huffman semi-estático se generaron 2 archivos binarios:

- Peso *codigoBTC.bin* → 1170 B
- Peso *codigoETH.bin* → 838 B

Entonces con *huffman semi-estático* se obtuvieron las siguientes tasas de compresión:

Tasa compresión BTC:

$Tasa_{BTC} = \frac{6830}{1170} \simeq 5.8$, por lo que se puede decir que tiene una compresión de 6:1

Tasa compresión ETH:

$Tasa_{ETH} = \frac{5390}{838} \simeq 6.4$, por lo que se puede decir que tiene una compresión de 6:1