1. Introducción

En esta práctica he implementado en python el algoritmo de codificación de Huffman, uno de decodificación y funciones auxiliares para comprobar el Primer Teorema de Shannon.

2. Método

He implementado los algoritmos mediante programación orientada a objetos para aprovechar ciertas comodidades de python como la sobrecarga de operadores. Gracias a esto tambien se obtiene un código bastante modular y reutilizable.

Además he implementado un método `graph` para visualizar los árboles de huffman (he incluido ejemplos de visualizaciones en los anexos).

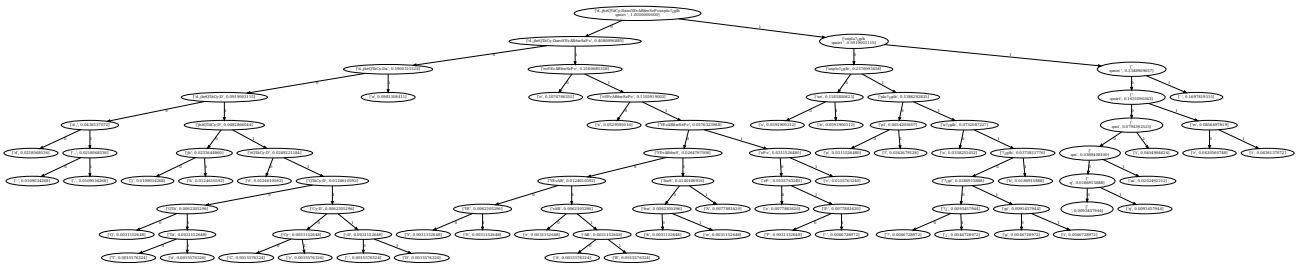
Para comprobar el Primer Teorema de Shannon he utilizado la siguiente definición de entropía, para un sistema de estados no equiprobables:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

donde X es una variable aleatoria discreta con $x_1, \ldots, x_n$ posibles estados que ocurren con probabilidades $P(x_1), \ldots, P(x_n)$. En nuestro caso $C = X$ es el conjunto de caracteres de un idioma y las probabilidades son las obtenidas a partir de las frecuencias de las muestras (los textos anexos).

3. Resultados

En la siguiente gráfica (includa como vectorial para poder visualizarse ampliando el documento) se puede ver el árbol de Huffman correspodiente al texto de referencia en Castellano.



3.1. Comprobación del Primer Teorema de Shannon

El Primer Teorema de Shannon nos da la siguiente cota para la longitud esperada de la codificación de Huffman:

$$H(C) \leqslant L(C) < H(C) + 1$$

donde $H(C)$ es la entropía del sistema y $L(C)$ la longitud esperada.

Los valores obtenidos para las muestras de textos en Español e Inglés son:

- Español: $H(C) = 4.4074$ y $L(C) = 4.4439$ con lo que
$$H(C) = 4.4074 \leqslant L(C) = 4.4439 < H(C) + 1 = 5.4074.$$
- Inglés: $H(C) = 4.1309$ y $L(C) = 4.1726$ con lo que
$$H(C) = 4.1309 \leqslant L(C) = 4.1726 < H(C) + 1 = 5.1309.$$

3.2. Codificación de la palabra 'medieval'

Una posible codificación binaria no comprimida para la palabra 'medieval' es la dada por la codificación *ASCII*, que codifica cada caracter mediante un byte (ocho bits). Por tanto la longitud de la codificación 'medieval' es $8 \cdot \text{long('medieval')} = 8 \cdot 8 = 64$.

Utilizando una codificación de Huffman se consigue utilizar códigos más cortos para los caracteres de mayor frecuencia, con la finalidad de reducir la longitud final de la codificación. Estas son las codificaciones que he obtenido para la palabra medieval, con sus longitudes correspodientes:

- Español: El código obtenido es 11000101000000110010100111001000110101, de longitud 38.
- Inglés: El código obtenido es 010011101110111110011000111010100000111001110101, de longitud 48.

Con lo que se comprueba que en cierto sentido se ha conseguido el objetivo de reducir las codificaciones. El punto delicado es que las frecuencias esperadas de caracteres con las que se construyen los árboles de Huffman se correspondan realmente con las frecuencias de los textos que queremos codificar. De no ser así podríamos generar codificaciones incluso menos óptimas que la *ASCII*.

3.3. Decodificación de un código dado

He hecho una implementación del algoritmo de Huffman inspirada en la implementación clásica, utilizando montículos. Debido a esto el árbol resultante es distinto al de la plantilla, puesto que el algoritmo de ordenación de montículos no tiene la misma estabilidad que el sort de numpy.

Por tanto con mi implementación no es posible decodificar el código proporcionado en el enunciado de la entrega. En lugar de esto he incluido en el código comprobaciones de que los códigos de la palabra 'medieval' obtenidos en el apartado anterior se decodifican correctamente en la palabra 'medieval'.

4. Código

El siguiente código con la implementación también está adjunto en la entrega y disponible, junto con esta memoria, en un repositorio git en el siguiente enlace: github.com/haztecaso/gcomp22.

```
"""
Autor: Adrián Lattes Grassi
"""

from decimal import Decimal
from functools import reduce
from heapq import heappop, heappush
from os import remove as remove_file
from typing import Dict, List, Optional
from math import log2


def frecuencias(texto:str)-> Dict[str, Decimal]:
    """
    Dado un texto calcula las frecuencias de cada caracter del texto.
    """
    letras = dict()
    for char in texto:
        if char not in letras:
            letras[char] = Decimal(1)
        else:
            letras[char] += 1
    n = len(texto)
    return {char:frec/n for (char, frec) in letras.items()}


class Codigo():
    """
    Clase para códigos binarios.
    """
    def __init__(self, codigo:List[bool] = None):
        self._codigo = codigo if codigo is not None else []

    def pre(self, valor:bool):
        """
        Añade un valor al inicio del código
        """
        return Codigo([valor]+self._codigo)

```

```
    @property
    def vacio(self):
        return len(self._codigo) == 0

    def __add__(self, other):
        """
        Sobrecarga del operador + para concatenar códigos
        """
        return Codigo(self._codigo + other._codigo)

    def __len__(self):
        return len(self._codigo)

    def __iter__(self):
        yield from self._codigo

    def __repr__(self):
        return f"Codigo<{''.join(map(lambda b: '1' if b else '0', self._codigo))}>"


class ArbolHuffman():
    """
    Clase para árboles de Huffman.
    """
    def __init__(self, **kwargs):
        assert 'clave' in kwargs or ('iz' in kwargs and 'dr' in kwargs)
        if 'clave' in kwargs:
            self.hoja = True
            assert 'peso' in kwargs
            self.peso = kwargs['peso']
            self.clave = kwargs['clave']
        else:
            self.hoja = False
            self.iz = kwargs['iz']
            self.dr = kwargs['dr']
            self.peso = self.iz.peso + self.dr.peso
            self.clave = self.iz.clave + self.dr.clave
        self._tabla_codigos = None

    def __lt__(self, other):
        return self.peso < other.peso

    def __eq__(self, other):
        return self.peso == other.peso

    def __repr__(self):
        return f"['{self.clave}', {self.peso:.10f}]"

    @property
    def tabla_codigos(self):
        if self._tabla_codigos is None:
            if self.hoja:
                self._tabla_codigos = {self.clave: Codigo()}
            else:
                codigos_iz = { clave: codigo.pre(False) for (clave, codigo) in self.iz.tabla_codigos.items()}
                codigos_dr = { clave: codigo.pre(True) for (clave, codigo) in self.dr.tabla_codigos.items()}
                self._tabla_codigos:Optional[Dict[str,Codigo]] = {**codigos_iz, **codigos_dr}
        return self._tabla_codigos

    def codificar(self, data:str):
        """
        Codifica una cadena de caracteres
        """
        return reduce(lambda x,y: x+y,
                map(lambda e: self.tabla_codigos[e], data),
                Codigo())

    def decodificar(self, codigo:Codigo):
        """
        Decodifica un código (o secuencia de códigos) de Huffman.
        """
        actual = self
        codigo_actual = Codigo()
        result = ""
        for b in codigo:
            actual = actual.dr if b else actual.iz
            codigo_actual += Codigo([b])
            if actual.hoja:
                result += actual.clave
                actual = self
                codigo_actual = Codigo()
        assert codigo_actual.vacio, f"No se ha terminado de decodificar el código. Código restante: {codigo_actual}"
        return result

    def graph(self, dot = None, render:bool = True, title:str='arbol'):
        """
        Exporta un dibujo del árbol en pdf
        """
        try:
            from graphviz import Digraph
        except ModuleNotFoundError as e:
```

```
            print(f"ATENCIÓN: Para generar la gráfica de un ArbolHuffman es necesario instalar el paquete {e.name}.")
        else:
            if dot is None:
                dot = Digraph(comment = title)
            if self.hoja:
                dot.node(self.clave, repr(self))
            else:
                dot.node(self.clave, repr(self))
                self.iz.graph(dot, False)
                self.dr.graph(dot, False)
                dot.edge(self.clave, self.iz.clave, label="0")
                dot.edge(self.clave, self.dr.clave, label="1")
            if render:
                print(f"Árbol de Huffman guardado en {title}.pdf")
                dot.render(title)
                remove_file(title) # Borrando archivo extra que crea graphviz
            return dot


def huffman(frecs:Dict[str, Decimal]) -> ArbolHuffman:
    """
    Implementación del algoritmo de Huffman.
    """
    heap = []
    for clave, peso in frecs.items():
        heappush(heap, ArbolHuffman(clave = clave, peso = peso))
    while len(heap) > 1:
        iz = heappop(heap)
        dr = heappop(heap)
        heappush(heap, ArbolHuffman(iz = iz, dr = dr))
    return heap[0]


def longitud_media(frecuencias:Dict[str, Decimal], tabla_codigos:Dict[str,Codigo]) -> Decimal:
    """
    Calcula la longitud esperada de la codificación de Huffman de un caracter,
    dadas las frecuencias de los caracteres y la tabla de códigos de Huffman.
    """
    result = Decimal(0)
    for clave, peso in frecuencias.items():
        result += peso*len(tabla_codigos[clave])
    return result


def entropia(frecuencias:Dict[str, Decimal]):
    """
    Entropía de Shannon calculada según la definición de wikipedia:
    https://en.wikipedia.org/wiki/Entropy_(information_theory)
    """
    result = Decimal(0)
    for peso in frecuencias.values():
        result += peso*Decimal(log2(peso))
    return -result


def main():
    with open("GCOM2022_pract2_auxiliar_esp.txt") as f:
        texto = '\n'.join(f.readlines())
        frec_es = frecuencias(texto)
    with open("GCOM2022_pract2_auxiliar_eng.txt") as f:
        texto = '\n'.join(f.readlines())
        frec_en = frecuencias(texto)

    arbol_es = huffman(frec_es)
    arbol_en = huffman(frec_en)
    arbol_es.graph(title = 'arbol_es')
    arbol_en.graph(title = 'arbol_en')
    print()

    print(f"""i) Comprobación del primer teorema de Shannon:
    - Español: H(C) = {entropia(frec_es):.4f}, L(C) = {longitud_media(frec_es, arbol_es.tabla_codigos):.4f}
    - Inglés:  H(C) = {entropia(frec_en):.4f}, L(C) = {longitud_media(frec_en, arbol_en.tabla_codigos):.4f}
    """)

    medieval = "medieval"
    codigo_medieval_es = arbol_es.codificar(medieval)
    codigo_medieval_en = arbol_en.codificar(medieval)
    print(f"""ii) Codificación de la palabra '{medieval}' en los dos idiomas:
    - Español: {codigo_medieval_es} (longitud {len(codigo_medieval_es)})
    - Inglés:  {codigo_medieval_en} (longitud {len(codigo_medieval_en)})
    - Con la codificación ASCII se utiliza un número fijo de bytes igual a 8,
      con lo que la longitud de la palabra {medieval} sería {len(medieval)*8}.
    """)

    codigo = Codigo(list(map(lambda d:True if d == '1' else False, "10111101101110110111011111")))
    print(f"iii A) Decodificación del código {codigo}:")
    try:
        print(f"{codigo, arbol_en.decodificar(codigo) = }\n")
    except AssertionError:
        print(f"ERROR: No se ha podido decodificar el código {codigo}\n")
```

```
224
225     print("iii B) Comprobación de que la decodificación funciona correctamente:")
226     print(f"  - Español: {arbol_es.decodificar(codigo_medieval_es) = }")
227     print(f"  - Inglés:  {arbol_en.decodificar(codigo_medieval_en) = }")
228
229
230 if __name__ == "__main__":
231     main()
```