Entropia de Shannon aplicada à compressão de dados

May 17, 2021

Edelson Luis Pinheiro Sezerotto Júnior, 288739

0.1 Introdução

Qualquer tipo de informação - seja um texto, um filme, uma música, etc - pode ser representada através de um conjunto de números. Por exemplo, para representarmos uma palavra, podemos associar cada letra à sua posição no alfabeto, de forma que a letra A é associada ao número 0 (se quisermos começar contando pelo 0), B ao 1, e assim por diante. Como temos 26 possíveis letras, é conveniente que escrevamos os números de forma que cada um tenha 2 algarismos, ou seja, A seria associado a 00, B a 01, até chegar ao 26, que seria o Z. Assim, a palavra DADO, por exemplo, seria associada ao conjunto de números 03000314, que, lidos aos pares, nos informam a palavra codificada.

Devido à forma como um computador armazena informações, através de transistores que podem assumir dois estados (permitindo ou barrando a passagem de elétrons), é conveniente armazenar informações nesses aparelhos utilizando números no formato binário. Assim, cada número é uma lista de zeros e uns, onde o 0 pode ser associado a um transistor barrando corrente elétrica e 1 pode ser associado a ele conduzindo corrente. Nesse formato, cada dígito 0 ou 1 é chamado de "bit".

Acontece que ao representarmos uma certa informação, ela pode conter redundâncias que fazem com que precisemos usar mais bits do que seriam necessários. Podemos pensar em um exemplo simples: considere a sequência de letras "AAAAABBBBB". Ao invés de representarmos cada letra dessa sequência com um número binário, podemos usar um número para representar todo o conjunto "AAAAA" e outro para o conjunto "BBBBB", usando 2 números binários ao invés de 10. Ao reduzirmos os bits, dizemos que estamos fazendo uma "compressão" nos dados.

A Teoria da Informação estuda as propriedades das informações, incluindo como calcular suas redundâncias e nos fornecendo métodos para evitá-las. Neste trabalho iremos explicar o conceito da Entropia de Shannon, ver como ele se relaciona com a termodinâmica e explorar sua utilidade na compressão de dados. Serão feitas ainda algumas simulações em Python para mostrar na prática a utilidade das ferramentas que desenvolveremos.

0.2 Conceitos teóricos

0.2.1 Entropia termodinâmica

De acordo com a mecânica estatística, a entropia S_T de um sistema termodinâmico é uma medida da quantidade Ω de microestados que ele pode assumir, enquanto sujeito a um conjunto de parâmetros externos, e é dada pela expressão

$$S_T = k_B \ln \Omega \tag{1}$$

onde k_B é é a constante de Boltzmann. Usando um exemplo simples, vamos motivar uma outra definição para S_T. Consideremos um sistema de dois spins, onde cada um pode assumir dois estados, que podemos representar por 0 e 1. As combinações de spins possíveis para esse sistema são, portanto, {00, 01, 10, 11}. Vamos supor que todas as combinações são igualmente possíveis. Temos então $\Omega = 4$ e notando que a probabilidade P_i de encontrarmos o sistema em cada macrostado é $P_i = \frac{1}{\Omega}$, podemos calcular a entropia como sendo

$$S_T = k_B \ln \Omega = -k_B \ln \frac{1}{\Omega} = -k_B \Omega \frac{1}{\Omega} \ln \frac{1}{\Omega} = -k_B \sum_{\Omega} \frac{1}{\Omega} \ln \frac{1}{\Omega} = -k_B \sum_{\Omega} P_i \ln P_i$$

ou seja,

$$S_T = -k_B \sum_{i} P_i \ln P_i \tag{2}$$

Acima vimos um exemplo particular, mas a equação (2) pode ser usada como uma forma geral de se definir a entropia de um sistema, conforme explicado na referência [1]. Agora vamos definir um conceito análogo que nos ajudará a medir a informação contida em um sistema (a explicação será baseada na referência [2]).

0.2.2 Entropia de Shannon

Vamos considerar três informaçõs diferentes, que podem ser representadas, respectivamente, pelas sequências de bits 0000, 0001 e 0011. Agora suponha que vamos sortear uma sequência de números aleatórios de cada uma dessas sequências. Qual a probabilidade P de essa sequência de números selecionados ser igual à sequência original? Para a primeira sequência, a probabilidade um número selecionado ser 0 é 100% = 1, portanto P = 1x1x1x1 = 1; já para a segunda, a probabilidade de ser 0 é 0.75, e de ser 1 é 0.25, portanto P = 0.75x0.75x0.75x0.25 = 0.105, e a mesma lógica se aplica para a terceira sequência. As probabilidades são listadas abaixo:

Sequência	P
0000	1 x 1 x 1 x 1 = 1
0001	0.75 x 0.75 x 0.75 x 0.25 = 0.105
0011	0.5 x 0.5 x 0.5 x 0.5 = 0.0625

Percebemos que P é uma medida da nossa incerteza em relação às informações que vamos obter ao sortear números de uma certa sequência: quanto menor for P, maior nossa incerteza e maior é a quantidade de informações que ganhamos ao olharmos para a sequência que foi sorteada (para a sequência 0000, antes de olharmos o resultado do nosso sorteio, já sabíamos de antemão qual seria o resultado, enquanto que para a sequência 0011 é a que tínhamos a maior incerteza sobre o resultado).

Para sequências muito grandes, o procedimento acima pode gerar valores muito pequenos de P (porque estamos multiplicando muitos números positivos menores que 1 entre si), e para evitar esse

problema podemos tomar logaritmos de P. Qualquer base usada vai resolver o problema dos números pequenos, mas vamos usar aqui uma base 2, por motivos que logo serão explicados. Além disso, vamos tomar os módulos dos logaritmos para não termos que trabalhar com números negativos:

Sequência	P	-$\log_2$ P
0000	1 x 1 x 1 x 1 = 1	0 + 0 + 0 + 0
0001	0.75 x 0.75 x 0.75 x 0.25 = 0.105	0.415 + 0.415 + 0.415 + 0.415 + 2
0011	0.5 x 0.5 x 0.5 x 0.5 = 0.0625	1 + 1 + 1 + 1

Finalmente, vamos definir a entropia de Shannon S associada a cada sequência como sendo a média de $-\log_2 P$:

Sequência	S
0000	0
0001	0.81
0011	1

De forma mais geral, se temos um sistema de afirmações i, onde cada afirmação tem uma probabilidade P_i de ser verdadeira, dizemos que a entropia de Shannon para esse conjunto de afirmações é

$$S_{k,b} = -k \sum_i P_i \log_b P_i \tag{3}$$

Repare que quanto maior for $S_{k,b}$, maior é a informação contida nesse sistema. Note também que $S_{k,b}$ é uma função de k e b, e nesse trabalho estamos considerando $S \equiv S_{1,2}$. Vamos agora analisar mais a fundo o significado de S para um certo sistema.

Interpretação de S Consideremos uma caixa com moedas dentro. Cada moeda tem suas faces pintadas em combinações das cores azul e verde (algumas moedas estão com ambas as faces pintadas de azul; outras estão com o lado "cara" pintado de azul e o "coroa" de verde; e assim por diante). Podemos representar a cor azul pelo número 0 e verde por 1. Assim, cada moeda possui dois números associados a ela, que podem assumir quatro possíveis combinações: {00, 01, 10, 11} (o dígito à direita pode indicar a cor do lado "cara", e o da esquerda o "coroa"). Imagine agora que vamos tirando moedas de dentro da caixa e anotando on números associados a elas. No final, teremos algo do tipo: 00 00 10 00 01 01 11 00. Vamos agora associar cada combinação a uma letra, conforme a convenção abaixo:

Letra	Combinação
A	00
B	01
C	10
D	11

Supondo que vamos passar o código binário obtido para um computador interpretar e decodificar as letras, então para cada letra ele precisará ler 2 bits de informação. Assim, o valor médio de bits que precisarão ser lidos por letra é 2. Agora, calculamos as probabilidades associadas a encontrar cada letra na sequência de bits. Para encontrar uma letra "A", precisamos encontrar dois 0's seguidos; para um "B" precisamos de um 0 seguido de um 1; e assim por diante. Na sequência indicada acima, temos 16 bits, dos quais 11 são 0 e 5 são 1. Ou seja, as probabilidades de encontrarmos um 0 e um 1, são, respectivamente, $P_0 = \frac{11}{16}$ e $P_1 = \frac{5}{16}$. A partir disso calculamos as probabilidades associadas a cada letra, exibidas abaixo:

Letra	Combinação	Probabilidade
A	00	P_0^2
B	01	P_0P_1
C	10	P_0P_1
D	11	P_1^2

E agora, vamos associar cada letra acima a um código (a maneira como esses códigos foram obtidos será explicada mais a frente), conforme a tabela abaixo:

Letra	Combinação	Probabilidade	Código
A	00	0.47	0
B	01	0.21	10
C	10	0.21	110
D	11	0.11	111

Esses códigos irão substituir as combinações obtidas anteriormente. Ou seja, cada vez que eu tiver a sequência 00, eu vou trocá-la por 0, e assim por diante. Agora façamos o computador ler a nova sequência composta por esses códigos. Em média, quantos bits ele precisará ler? O primeiro código possui um único bit e aparece 47% das vezes; o segundo possui 2 bits e aparece 21% das vezes; e assim por diante. Assim, a quantidade média M^* de bits que serão lidos será

$$M^* = \sum_i P_i L(i) = 1.85$$

onde $L(i)$ é a quantidade de bits de cada código. O valor de M^* indica que a cada 100 letras, precisaremos ler apenas 185 bits ao invés de 200, como era feito anteriormente. Ou seja, nosso código foi comprimido. Se ao invés de quatro possibilidades tivéssemos apenas duas (por exemplo, cada moeda está totalmente pintada ou de verde, ou de azul), e imaginando que tivéssemos a mesma sequência de antes: 00 00 10 00 01 01 11 00 - nesse caso, precisaríamos ler um único bit de cada vez, e usando a nomenclatura de que cada combinação é lida em um "experimento", então a média de bits lidos por experimento seria 1. Usando o método de compressão descrito, a média de bits lidos por experimento passa a ser $M = \frac{M^*}{n} = 0.925$, onde $n = 2$ é a quantidade de bits de cada combinação. Ou seja, a cada 1000 experimentos, precisaríamos ler apenas 925 bits ao invés de 1000.

Calculando a entropia de Shannon para o sistema acima, teríamos

$$S = -[\frac{11}{16}\log_2(\frac{11}{16}) + \frac{5}{16}\log_2(\frac{5}{16})] \simeq 0.896$$

De acordo com o Teorema de Codificação da Fonte (ver referência [1]), o valor de S representa o menor valor possível de se obter para M, independente do método de compressão utilizado. Se utilizarmos algum método de codificação no qual $M < S$, teremos perda de informação e a mensagem original não poderá ser recuperada. Note que com o sistema de códigos que usamos, obtivemos $M > S$. Utilizando uma codificação diferente, possivelmente obteríamos valores menores para M; a única limitação é de que jamais seremos capazes de inventar um método, por mais elaborado que seja, que produza um M menor do que o S do sistema. Adicionalmente, vamos verificar qual a interpretação do caso geral $S_{k,b}$, o que nos permitirá entender o significado de S_T do ponto de vista da Teoria da Informação (notando que $S_T = S_{k_B,e}$, sendo e o número de euler).

Interpretação de $S_{k,b}$ A ideia aqui é nos darmos conta de que $S_{k,b}$ pode ser escrito em termos de S. Usando a equação (2), $S_{k,b}$ pode ser escrito no seguinte formato:

$$S_{k,b} = -k\sum_i P_i \log_b P_i = -k\sum_i \log_b P_i^{P_i} = -k\log_b(\prod_i P_i^{P_i}) = \gamma S = -\gamma\log_2(\prod_i P_i^{P_i})$$

Isolando o fator de proporcionalidade γ e fazendo $\prod_i P_i^{P_i} \equiv x$, temos:

$$\gamma = k\frac{\log_b x}{\log_2 x} = k\frac{\log_b x}{\frac{\log_b x}{\log_b 2}} = k\log_b 2$$

Finalmente,

$$S_{k,b} = k\log_b(2)S \tag{4}$$

Com a relação acima, podemos usar o Teorema de Codificação da Fonte para interpretar o significado da entropia de Shannon para quaisquer valores (k, b): o $S_{k,b}$ de qualquer sistema equivale à menor quantidade média de bits possível necessária para representar as informações desse sistema, multiplicada por um fator $k\log_b(2)$. Em particular, notamos que

$$S = \frac{S_T}{k_B \ln 2} \tag{5}$$

Assim, se tivermos um sistema termodinâmico para o qual $S_T = 10^{22}$, se quisermos representar toda sua informação usando bits de forma comprimida, a menor quantidade média de bits necessária será

$$S = \frac{10^{22}}{k_B \ln 2} \simeq 0.105$$

onde se usou $k_B = (1.38)10^{23}$. Agora, veremos um algoritmo que pode ser usado para codificar informações.

0.2.3 Codificação de Shannon-Fano

Anteriormente, foi gerado um código para representar duplas de bits. Esses códigos foram gerados utilizando um algoritmo chamado de Codificação de Shannon-Fano. Iremos explicar e implementar um código que o efetue, com base nas referências [3] e [4].

Consideremos uma lista de bits de tamanho arbitrário. Vamos separar essa lista em uma certa quantidade C de combinações, onde cada combinação será representado por um número binário, começando pelo 0. Usando $C = 4$ e supondo que as probabilidades de encontrar um 0 e um 1 nessa lista são, respectivamente, $P_0 = \frac{11}{16}$ e $P_1 = \frac{5}{16}$ (identicamente ao que foi feito em um exemplo anterior), recuperamos a tabela abaixo:

Letra	Combinação	Probabilidade
A	00	P_0^2
B	01	P_0P_1
C	10	P_0P_1
D	11	P_1^2

O próximo passo agora é ordenar a tabela da maior para a menor probabilidade. Por acaso, ela já está nessa ordem. Abaixo exibimos a tabela com os valores explícitos das probabilidades e omitindo as combinações:

Letra	Probabilidade
A	0.47
B	0.21
C	0.21
D	0.11

Agora, dividimos nossa tabela em dois blocos, usando a seguinte lógica: para o primeiro bloco, pegamos a primeira linha e vamos adicinando mais linhas para formar o bloco de cima - as demais linhas compõem o bloco de baixo - até que as somas das probabilidades de cada bloco estejam o mais próximo possível entre si. Colocamos uma nova coluna, na qual preenchemos com 0 as linhas do primeiro bloco, e com 1 as linhas do segundo bloco (nesse trabalho vamos nos referir a essa coluna como "primeira camada"; a próxima será a segunda camada, e assim por diante). A tabela vai ficar como abaixo:

Letra	Probabilidade	1
A	0.47	0
B	0.21	1
C	0.21	1
D	0.11	1

Note que a soma das probabilidades do primeiro bloco (que contém apenas a letra A) vale 0.47, enquanto que a soma do bloco 2 (que contém as letras B, C e D) vale 0.53, nos dando uma diferença, em módulo, igual a 0.06 - repare ainda que caso considerássemos que o primeiro bloco é formado

por A e B e o segundo por C e D, as somas das probabilidades seriam, respectivamente, 0.68 e 0.32, caso no qual a diferença em módulo seria 0.36, sendo maior do que a divisão que foi feita. Quando um dos blocos atinge um tamanho de apenas uma linha, paramos de gerar números para essa linha e vamos repetindo o procedimento para os outros blocos:

Letra	Probabilidade	1	2
A	0.47	0	x
B	0.21	1	0
C	0.21	1	1
D	0.11	1	1

Na tabela acima, o "x" indica que não estamos mais gerando números para aquela linha. Repare que o procedimento foi repetido com o segundo bloco (referente aos valores 1). Quando um bloco atinge duas linhas, a de cima ganha mais um valor 0, e a de baixo mais um valor 1, e paramos de gerar números para ambas. A tabela agora fica assim:

Letra	Probabilidade	1	2	3
A	0.47	0	x	x
B	0.21	1	0	x
C	0.21	1	1	0
D	0.11	1	1	1

Fazemos um último passo para finalizar a tabela:

Letra	Probabilidade	1	2	3	4
A	0.47	0	x	x	x
B	0.21	1	0	x	x
C	0.21	1	1	0	x
D	0.11	1	1	1	x

Note que o procedimento termina quando todos os valores da coluna mais à direita são "x". Os códigos são dados pelas sequências de digitos formados em cada linha:

Letra	Probabilidade	Código
A	0.47	0
B	0.21	10
C	0.21	110
D	0.11	111

Geramos ainda uma nova coluna contendo os valores de L (a quantidade de bits de cada código):

Letra	Probabilidade	Código	L
A	0.47	0	1
B	0.21	10	2
C	0.21	110	3
D	0.11	111	3

Para terminar a tabela, geramos uma coluna na qual cada linha contém a multipicação da probabilidade associada àquela linha, multiplicada por L (vamos identificar essa coluna pela letra "e", de "entropia", porque ela está associada a S):

Letra	Combinação	Probabilidade	Código	L	e
A	00	0.47	0	1	0.47
B	01	0.21	10	2	0.42
C	10	0.21	110	3	0.63
D	11	0.11	111	3	0.33

(a tabela foi colocada no mesmo formato da simulação da próxima seção). A soma dos elementos da coluna e dividida pela quantidade de bits n de uma das combinações (nesse caso $n = 2$) é a quantidade média M de bits necessários para representar as informações do sistema. Quanto maior for n, menor será M, até o momento em que teremos um n_j para o qual $M_j < S$. Quando isso acontecer significa que n_{j-1} é o maior valor de letras possível, ou seja, aquele que gera o menor valor permitido para M, usando esse método de codificação (note que em princípio é possível obter valores ainda menores para M utilizando outros métodos, desde que tenhamos $M \geq S$). Na próxima seção veremos como desenvolver um código que implemente a Codificação de Shannon-Fano, nos permitindo descobrir a maior compressão possível de se obter com ela para um dado conjunto de bits.

0.3 Implementação e resultados

Vamos imaginar que temos um certo conjunto de informações que está sendo representado através de uma lista de bits. Essa lista possui n_0 elementos 0 e n_1 elementos 1, portanto a probabilidade de encontrarmos o bit 0 é $P_0 = \frac{n_0}{n_0+n_1}$, e a de encontrarmos um bit 1 é $P_1 = 1 - P_0$. Nós queremos comprimir essa lista usando a Codificação de Shannon-Fano através de uma quantidade C de letras, de acordo com o esquema explicado na seção anterior - assim, se $C = 6$, por exemplo, teremos as letras {A, B, C, D, E, F} associadas às combinações {000, 001, 010, 011, 100, 101}, que são os grupos de bits em que iremos dividir nossos dados.

Nossa intenção nessa seção é a seguinte: queremos desenvolver um código que tome como inputs P_0 e C e nos retorne $R = \frac{M}{S}$, ou seja, a quantidade média de bits relativa à entropia. Se $R \geq 1$, a compressão é possível, e quanto maior for R, mais os dados serão comprimidos. Começamos importando algumas bibliotecas e definindo os valores de entrada:

[77]:
```
import math
import pandas as pd
import numpy as np
import warnings
```

```
warnings.filterwarnings('ignore')
import matplotlib.pyplot as plt

P = 11/16 #probabilidade de ser 0
qL = 8 #qtd de letras
```

math e numpy servem para automatizar alguns cálculos; a biblioteca pandas ajuda a lidar com conjuntos de dados; warnings ajuda a lidar com alguns problemas de incompatibilidade entre a versão original do programa (escrita no software Spyder) e a versão desenvolvida aqui; e o matplotlib serve para fazermos gráficos. Agora vamos gerar as combinações no formato de números binários:

[78]:
```
p = 1-P
lNd = [i for i in range(qL)] #lista de numeros decimais que serao convertidos␣
 ↪para binario
qmd = len(str(int(bin(lNd[-1])[2:]))) #qtd maxima de digitos dos meus nros␣
 ↪binarios
lNb = [str(int(bin(lNd[i])[2:])) for i in range(qL)]
for i in range(qL):
    while len(lNb[i]) < qmd: lNb[i] = '0' + lNb[i]
print(lNb)
```

```
['000', '001', '010', '011', '100', '101', '110', '111']
```

Como esperado, temos oito números binários porque definimos qL = 8. Geramos agora um data frame - um objeto do pandas que funciona como uma tabela associando um cojunto de dados -, o qual conterá as informações que mostramos na tabela da seção anterior:

[79]:
```
lp = [] #lista de probabilidades
for i in lNb:
    l = []
    for c in i:
        if c == '0': l.append(P)
        else: l.append(p)
    lp.append(math.prod(l))
df = pd.DataFrame(lNd,columns=['i'])
df['b'] = lNb
df['p'] = lp
df = df.sort_values('p', ascending=False)
dfo = df.copy() #data frame original
ldf = [] #lista de data frames
lcf = [] #lista de codigos finais
c = 0 #contador
df
```

[79]:

	i	b	p
0	0	000	0.324951
1	1	001	0.147705
2	2	010	0.147705

```
4  4  100  0.147705
3  3  011  0.067139
5  5  101  0.067139
6  6  110  0.067139
7  7  111  0.030518
```

No data frame acima, i é um número análogo às letras da tabela (0 é análogo a A, 1 análogo a B, e assim por diante. Claro, não precisamos necessariamente usar letras para representar as combinações binárias e nesse caso números são mais convenientes); b é a combinação binária; e p é a probabilidade associada a cada uma (aqui cabe lembrar: a probabilidade associada à combinação 010, por exemplo, vale $P_0^2 P_1$, porque temos dois 0's e um 1; e assim por diante). Vamos agora adicionar a primeira camada, na qual separamos as linhas em dois blocos, conforme explicado anteriormente:

[80]:
```
#gerando os codigos comprimidos - primeira camada
lcc = ['']*len(df) #lista de codigos comprimidos
c += 1 #contador
lp = df['p'].tolist()
ldsp = [] #lista das diferencas das somas das probabilidades
for i in range(1,len(lp)):
    lpd = [lp[:i],lp[i:]] #lista das probabilidades divididas
    ldsp.append(abs(sum(lpd[0])-sum(lpd[1])))

index = ldsp.index(min(ldsp))
for i in range(index+1): lcc[i] = lcc[i]+'0'
for i in range(index+1,len(df)): lcc[i] = lcc[i]+'1'
dfo[c] = lcc
dfo
```

[80]:
```
   i    b         p  1
0  0  000  0.324951  0
1  1  001  0.147705  0
2  2  010  0.147705  1
4  4  100  0.147705  1
3  3  011  0.067139  1
5  5  101  0.067139  1
6  6  110  0.067139  1
7  7  111  0.030518  1
```

Após gerar a primeira camada, as outras serão geradas através de um loop while. Para explicar o código de maneira mais detalhada, vamos gerar explicitamente a segunda camada, e depois montar o loop que montará todas de forma automática. Primeiramente, precisamos saber onde devemos separar os blocos criados acima (note que eles deverão ser separados na segunda linha):

[81]:
```
l = dfo[c].tolist()
lv = [] #lista verdade - determina onde separar os data frames
for i in range(len(l)-1):
    if (l[i+1] != l[i]) or (l[i] == 'x'): lv.append(True)
```

```
    else: lv.append(False)
lv.append(True)
lv
```

[81]:
```
[False, True, False, False, False, False, False, True]
```

Na lista acima, sempre que tivermos o valor False em uma posição j, significa que um determinado bloco termina nessa posição j; o próximo começa em $j+1$ e se extende até a posição relativa ao próximo True, e assim por diante. Vamos agora criar um data frame para cada bloco e colocá-los na lista ldf:

[82]:
```
colunas = []
for col in dfo.columns: colunas.append(col)
dfl = dfo.values.tolist()
ldf = []
li = []
for i in range(len(lv)):
    if lv[i] == False: li.append(dfl[i])
    else:
        li.append(dfl[i])
        ldf.append(pd.DataFrame(li,columns=colunas))
        li = []
ldf[0]
```

[82]:

	i	b	p	1
0	0	000	0.324951	0
1	1	001	0.147705	0

Colocamos dois data frames em ldf, sendo que o primeiro deles - relativo ao bloco com valores 0 - foi mostrado acima. Agora preenchemos a segunda camada de cada data frame seguindo a lógica explicada na seção anterior:

[83]:
```
c += 1
for j in range(len(ldf)):
    df = pd.DataFrame(ldf[j],columns=colunas)
    if len(df) == 1: lcc = 'x'
    if len(df) == 2: lcc = ['0','1']
    elif (len(df)) > 2:
        lcc = ['']*len(df) #lista de codigos comprimidos
        lp = df['p'].tolist()
        ldsp = [] #lista das diferencas das somas das probabilidades
        for i in range(1,len(lp)):
            lpd = [lp[:i],lp[i:]] #lista das probabilidades divididas
            ldsp.append(abs(sum(lpd[0])-sum(lpd[1])))

        index = ldsp.index(min(ldsp))
        for i in range(index+1): lcc[i] = lcc[i]+'0'
        for i in range(index+1,len(df)): lcc[i] = lcc[i]+'1'
```

```
    ldf[j][c] = lcc
ldf[0]
```

[83]:

	i	b	p	1	2
0	0	000	0.324951	0	0
1	1	001	0.147705	0	1

Acima foi mostrado o primeiro data frame de ldf com a segunda camada adicionada. Repare que usamos o critério de que o bloco continha duas linhas, portanto a primeira linha da segunda camada recebeu um 0 e a segunda recebeu um 1. Repare ainda que mais à frente esse bloco será dividido em dois, gerando duas linhas individuais, e como cada novo bloco será composto de apenas uma linha, os códigos serão completados - ou seja, na prática, já temos os códigos para os símbolos i 0 e 1, que são, respectivamente, 00 e 01. Agora juntamos todos os data frames de ldf em um só:

[84]:
```
ldfi = []
for i in range(len(ldf)): ldfi.append(ldf[i])

dfo = pd.concat([ldfi[0],ldfi[1]]) #data frame final, contendo todos os votos␣
 ↪de todas as roll calls
for i in range(2,len(ldfi)): dfo = pd.concat([dfo,ldfi[i]]) #adiciono os␣
 ↪valores em dff
dfo = dfo.reset_index(drop=True)
dfo
```

[84]:

	i	b	p	1	2
0	0	000	0.324951	0	0
1	1	001	0.147705	0	1
2	2	010	0.147705	1	0
3	4	100	0.147705	1	0
4	3	011	0.067139	1	1
5	5	101	0.067139	1	1
6	6	110	0.067139	1	1
7	7	111	0.030518	1	1

Para ilustrar todo o procedimento, vamos considerar que o processo termina aqui (mais à frente vamos reutilizar os códigos mostrados para gerar um loop que obtém os códigos completos). Nós agora geramos um data frame contendo apenas as colunas referentes às camadas dos códigos:

[85]:
```
dfo_copy = dfo.copy()
dfo = dfo.drop('i', axis=1)
dfo = dfo.drop('b', axis=1)
dfo = dfo.drop('p', axis=1)
dfo
```

[85]:

	1	2
0	0	0
1	0	1
2	1	0

```
3  1  0
4  1  1
5  1  1
6  1  1
7  1  1
```

Contabilizamos as quantidades de bits de cada código:

[86]:
```
lcci = dfo.values.tolist()
lcc = ['']*len(lcci)
for i in range(len(lcci)):
    for j in lcci[i]:
        if j != 'x': lcc[i] += j
lB = [len(i) for i in lcc] #quantidades de bits de cada codigo comprimido
lB
```

[86]:
```
[2, 2, 2, 2, 2, 2, 2, 2]
```

Como esperado, todos os códigos possuem 2 bits porque interrompemos o processo depois de duas iterações. Agora montamos o data frame final, contendo as informações descritas na tabela final da seção anterior:

[87]:
```
dfo = dfo_copy[['i', 'b', 'p']]
dfo['cc'] = lcc
dfo['qb'] = lB
dfo = dfo.sort_values('i', ascending=True)
dfo['e'] = dfo['p']*dfo['qb'] #entropia associada a cada estado
dfo = dfo.reset_index(drop=True) #dfo[estado, representacao binaria,␣
↪probabilidade, codigo comprimido, quantidade de bits]
dfo
```

[87]:

	i	b	p	cc	qb	e
0	0	000	0.324951	00	2	0.649902
1	1	001	0.147705	01	2	0.295410
2	2	010	0.147705	10	2	0.295410
3	3	011	0.067139	11	2	0.134277
4	4	100	0.147705	10	2	0.295410
5	5	101	0.067139	11	2	0.134277
6	6	110	0.067139	11	2	0.134277
7	7	111	0.030518	11	2	0.061035

Para manter a clareza, vamos resumir o que cada coluna representa:

Coluna	Significado
i	Símbolo associado a uma combinação de bits
b	Combinação de n bits
p	Probabilidade de encontrarmos cada combinação na lista de bits original
cc	Código comprimido pelo qual cada combinação será substituída

Coluna	Significado
qb	Quantidade de bits de cada código
e	Probabilidade x quantidade de bits

Por fim, calculamos M, S e R:

[88]:
```
S = -(P*math.log(P,2)+p*math.log(p,2)) #entropia
M = np.sum(dfo['e'])/len(dfo['b'][0]) #valor medio de bits
R = M/S #qtd media de bits relativa a S
print('R = '+str(round(R,3)))
```

```
R = 0.744
```

Como já era esperado, obtivemos $R < 1$ porque afinal não geramos os códigos de forma completa - com os códigos incompletos que foram gerados, nossa compressão resultaria em perda de dados. Definindo agora uma função para obter R através dos códigos completos:

[89]:
```
def calculos(P,qL):
    #gerando os numeros binarios
    p = 1-P #probabilidade de ser 1
    lNd = [i for i in range(qL)] #lista de numeros decimais que serao␣
 ↪convertidos para binario
    qmd = len(str(int(bin(lNd[-1])[2:]))) #qtd maxima de digitos dos meus nros␣
 ↪binarios
    lNb = [str(int(bin(lNd[i])[2:])) for i in range(qL)]
    for i in range(qL):
        while len(lNb[i]) < qmd: lNb[i] = '0' + lNb[i]

    #gerando data frame com as probabilidades
    lp = [] #lista de probabilidades
    for i in lNb:
        l = []
        for c in i:
            if c == '0': l.append(P)
            else: l.append(p)
        lp.append(math.prod(l))
    df = pd.DataFrame(lNd,columns=['i'])
    df['b'] = lNb
    df['p'] = lp
    df = df.sort_values('p', ascending=False)
    dfo = df.copy() #data frame original
    ldf = [] #lista de data frames
    lcf = [] #lista de codigos finais
    c = 0 #contador

    #gerando os codigos comprimidos - primeira camada
    lcc = ['']*len(df) #lista de codigos comprimidos
```

```
    c += 1 #contador
    lp = df['p'].tolist()
    ldsp = [] #lista das diferencas das somas das probabilidades
    for i in range(1,len(lp)):
        lpd = [lp[:i],lp[i:]] #lista das probabilidades divididas
        ldsp.append(abs(sum(lpd[0])-sum(lpd[1])))

    index = ldsp.index(min(ldsp))
    for i in range(index+1): lcc[i] = lcc[i]+'0'
    for i in range(index+1,len(df)): lcc[i] = lcc[i]+'1'
    dfo[c] = lcc
    verificar = 1 #variavel que verifica quando encerrar o processo

    #gerando as outras camadas
    while verificar != 0:
        l = dfo[c].tolist()
        lv = [] #lista verdade - determina onde separar os data frames
        for i in range(len(l)-1):
            if (l[i+1] != l[i]) or (l[i] == 'x'): lv.append(True)
            else: lv.append(False)
        lv.append(True)

        colunas = []
        for col in dfo.columns: colunas.append(col)
        dfl = dfo.values.tolist()
        ldf = []
        li = []
        for i in range(len(lv)):
            if lv[i] == False: li.append(dfl[i])
            else:
                li.append(dfl[i])
                ldf.append(pd.DataFrame(li,columns=colunas))
                li = []

        c += 1
        for j in range(len(ldf)):
            df = pd.DataFrame(ldf[j],columns=colunas)
            if len(df) == 1: lcc = 'x'
            if len(df) == 2: lcc = ['0','1']
            elif (len(df)) > 2:
                lcc = ['']*len(df) #lista de codigos comprimidos
                lp = df['p'].tolist()
                ldsp = [] #lista das diferencas das somas das probabilidades
                for i in range(1,len(lp)):
                    lpd = [lp[:i],lp[i:]] #lista das probabilidades divididas
                    ldsp.append(abs(sum(lpd[0])-sum(lpd[1])))
```

```
                index = ldsp.index(min(ldsp))
                for i in range(index+1): lcc[i] = lcc[i]+'0'
                for i in range(index+1,len(df)): lcc[i] = lcc[i]+'1'
            ldf[j][c] = lcc

        ldfi = []
        for i in range(len(ldf)): ldfi.append(ldf[i])
        dfo = pd.concat([ldfi[0],ldfi[1]]) #data frame final, contendo todos os␣
↪votos de todas as roll calls
        for i in range(2,len(ldfi)): dfo = pd.concat([dfo,ldfi[i]]) #adiciono␣
↪os valores em dff
        dfo = dfo.reset_index(drop=True)

        verificar = 0
        uc = dfo.iloc[:,-1:][c].tolist()
        for i in uc:
            if i != 'x': verificar += 1

    #coletando os codigos comprimidos
    dfo_copy = dfo.copy()
    dfo = dfo.drop('i', axis=1)
    dfo = dfo.drop('b', axis=1)
    dfo = dfo.drop('p', axis=1)
    dfo = dfo.drop(c, axis=1)

    lcci = dfo.values.tolist()
    lcc = ['']*len(lcci)
    for i in range(len(lcci)):
        for j in lcci[i]:
            if j != 'x': lcc[i] += j
    lB = [len(i) for i in lcc] #quantidades de bits de cada codigo comprimido

    #del dfo
    dfo = dfo_copy[['i', 'b', 'p']]
    dfo['cc'] = lcc
    dfo['qb'] = lB
    dfo = dfo.sort_values('i', ascending=True)
    dfo['e'] = dfo['p']*dfo['qb'] #entropia associada a cada estado
    dfo = dfo.reset_index(drop=True) #dfo[estado, representacao binaria,␣
↪probabilidade, codigo comprimido, quantidade de bits]

    #calculando as entropias ideal e obtida
    S = -(P*math.log(P,2)+p*math.log(p,2)) #entropia
    M = np.sum(dfo['e'])/len(dfo['b'][0]) #valor medio de bits
    R = M/S #qtd media de bits relativa a S
    return R
```

A função acima reutiliza praticamente todo o código desenvolvido ao longo da seção. Para utilizá-la, passamos valores P e qL, e ela nos retorna R. O trecho abaixo determina R para os valores já definidos de P e qL:

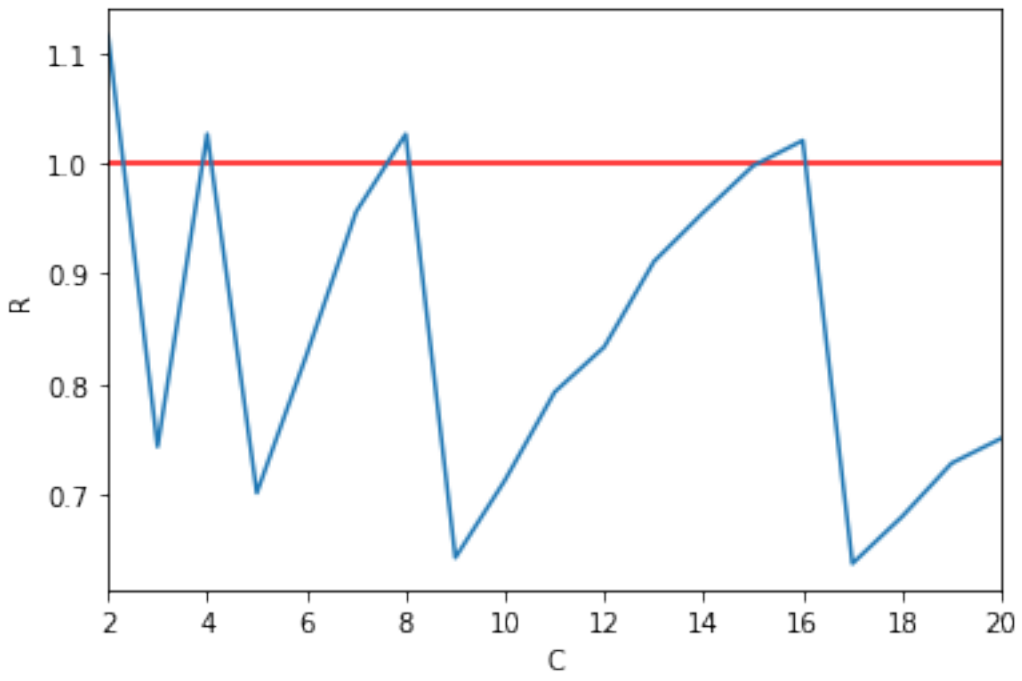
[90]:
```
R = calculos(P,qL)

print('R = '+str(round(R,3)))
```

```
R = 1.026
```

Percebemos então que usando os códigos completos, obtemos $R > 1$, e portanto a compressão é possível, de forma que M vai representar 1.026 do menor valor possível de ser atingido. Para visualizarmos a evolução de R conforme variamos C (a quantidade de símbolos, ou de grupos em que separamos os dados), usamos o trecho a seguir:

[91]:
```
qLmax = 20
lR = []
for qL in range(2, qLmax+1):
    R = calculos(P, qL)
    lR.append(R)
plt.xlabel('C')
plt.ylabel('R')
plt.xlim([2,qLmax])
x = np.arange(2,qLmax+1)
plt.axhline(y=1, color='r', linestyle='-')
plt.plot(x,lR)
```

[91]:
```
[<matplotlib.lines.Line2D at 0x7f2d62cb8d60>]
```



Notamos então que R não diminui de forma monótona, mais sim vai oscilando. Para o intervalo de valores testados para C (indo de 2 a 20), existem 4 valores para os quais a compressão seria possível, sendo que eles vão ficando progressimente menores, como podemos verificar com o trecho abaixo:

[92]:
```
lvp = [] #lista de valores possiveis
for i in lR:
    if i >= 1: lvp.append(i)
lvp
```

[92]:
```
[1.1160238075683149,
 1.0266547136028834,
 1.0264730691029538,
 1.0208009357731165]
```

Acima temos a sequência de valores permitidos de R, que de fato diminuem progressivamente. A pincípio, existe algum valor n para o qual R atinge seu menor valor, e aí teremos atingido a melhor compressão permitida por esse método.

0.4 Referências

[1] Concepts in Thermal Physics, Stephen J. Blundell and Katherine M. Blundell, segunda edição

[2] https://medium.com/udacity/shannon-entropy-information-gain-and-picking-balls-from-buckets-5810d35d54b4

[3] https://www.youtube.com/watch?v=dJCck1OgsIA

[4] https://www.geeksforgeeks.org/shannon-fano-algorithm-for-data-compression/