Kodowanie, Kompresja, Kryptografia

konspekt III, 27.III.2023

Zależne ciągi znaków — entropia języka angielskiego

Dane — tekst w języku angielskim zbudowany z 26 liter + spacji.

$$H_n(A_1, A_2, ..., A_n) \; < \; k\, H_1(A_1)$$

Oznaczmy $h_n = H_n/n$

	h_0	h_1	h_2	h_3	h_4	h_5	h_w
27 znaków	4.76	4.03	3.32	3.1	2.5	1.9	2.14

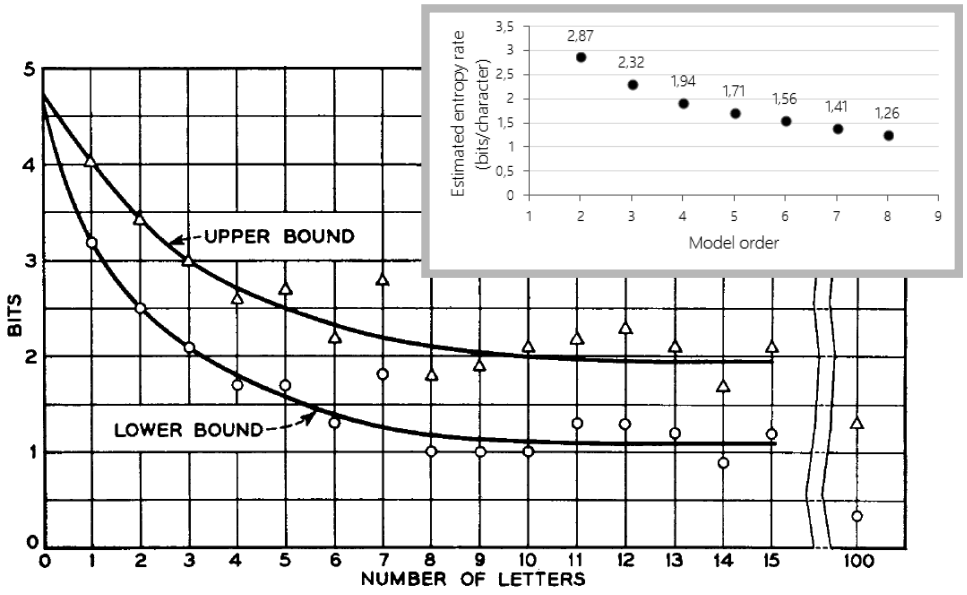


Fig. 4—Upper and lower experimental bounds for the entropy of 27-letter English.

Trudności z wyznaczaniem entropii n-gramów dla większych n:

$$27^n \quad \rightarrow \quad 27, \quad 729, \quad 19\,683, \quad 531\,441, \quad 14\,348\,907, \quad ...$$

Większości potencjalnych 3-, 4-gramów w ogóle nie ma w języku!

Inne metody: czytamy tekst i tworzymy słownik napotkanych n-gramów rejestrując częstość ich występowania.
Shannon: eksperymenty z ludźmi odgadującymi kolejne zasłonięte litery próbnego tekstu. Ocenił, że asymptotycznie (duże n) entropia na 1 literę w angielskim tekście wynosi około 1.3 bita. Czyli potencjalnie możliwy stopień kompresji to

$$1 - \frac{h_\infty}{h_0} \approx 0.727$$

Kompresja LZW

A. Lempel & J. Ziv (1978), ulepszona wersja T. Welch (1984).

Algorytm buduje słownik powtarzających się w tekście sekwencji liter i przyporządkowuje im numeryczne kody. Słownik budowany jest "w locie" — czytając kolejne znaki na wejściu, algorytm szuka *najdłuższego* zgodnego z nimi łańcucha α zapamiętanego już w słowniku. Wyprowadzany jest odpowiadający mu kod, a do słownika dopisywany jest nowy łańcuch postaci α + kolejny znak.

Zakładając, że początkowa zawartość słownika to pojedyncze litery, nie trzeba go zapamiętywać, można go systematyczne odbudować w procesie odkodowywania, odwracając kolejność kroków kodowania.

Słownik `Dict[k]` — kody pojedynczych liter, `Dict[0]='A', Dict[1]='B',...`

KODOWANIE:

```
s = getchar();
while ( !EOF(input) ) {
    c = getchar();
    if ( s+c in Dict ) { s = s+c; }
    else {
        output( code[s] );
        s+c --> Dict;
        s = c;
    }
}
output( code[s] );
```

DEKODOWANIE (I):

```
pk = nextcode();
output( Dict[pk] );
while ( !EOF(input) ){
    k = nextcode();
    s =  Dict[k];
    output( s );
    Dict[pk] + s[0] --> Dict;
    pk = k;
}
```

DEKODOWANIE (II):

```
pk = nextcode();
output( Dict[pk] );
c = Dict[pk][0];
while ( !EOF(input) ){
    k = nextcode();
    if ( k !in Dict ){ s = Dict[pk] + c; }
    else { s =  Dict[k]; }
    output( s );
    c = s[0];
    Dict[pk] + c --> Dict;
    pk = k;
}
```

Oryginalny algorytm kompresji Welcha z 1984 r. koduje sekwencje 8-bitowych danych jako 12-bitowe kody o stałej długości. Kody od 0 do 255 reprezentują sekwencje 1-znakowe składające się z odpowiedniego znaku 8-bitowego, a kody od 256 do 4095 są tworzone w słowniku dla sekwencji znaków dołączanych do słownika podczas ich kodowania.

Okazało się jednak, że kod stałej długości nie był zbyt efektywny, oferując słabą kompresję, szczególnie gdy liczba różnych znaków pliku wejściowego była niewielka. Pojawił się wówczas pomysł kodu o zmiennej długości: kody słownikowe zwykle liczą o jeden bit więcej niż kodowane symbole, a kiedy dana długość kodu zostaje wykorzystana, długość kodu zwiększa się o 1 bit, aż do ustalonego maksimum (zwykle 12 bitów). Po osiągnięciu maksymalnej wartości kodu w słowniku kodowanie jest kontynuowane przy użyciu istniejącej tabeli, ale nowe kody nie są już do niej dodawane.

Kolejne udoskonalenia dodawały do słownika specjalny, zarezerwowany kod *"clear table"*, zwykle o 1 większy od największego kodu litery alfabetu. Jego wystąpienie wskazuje, że tablica kodów powinna zostać wyczyszczona i przywrócona do stanu początkowego. Ponadto potrzebny jest jeszcze kod końca danych, *"stop code"*, o 1 większy od kody *"clear table"*.

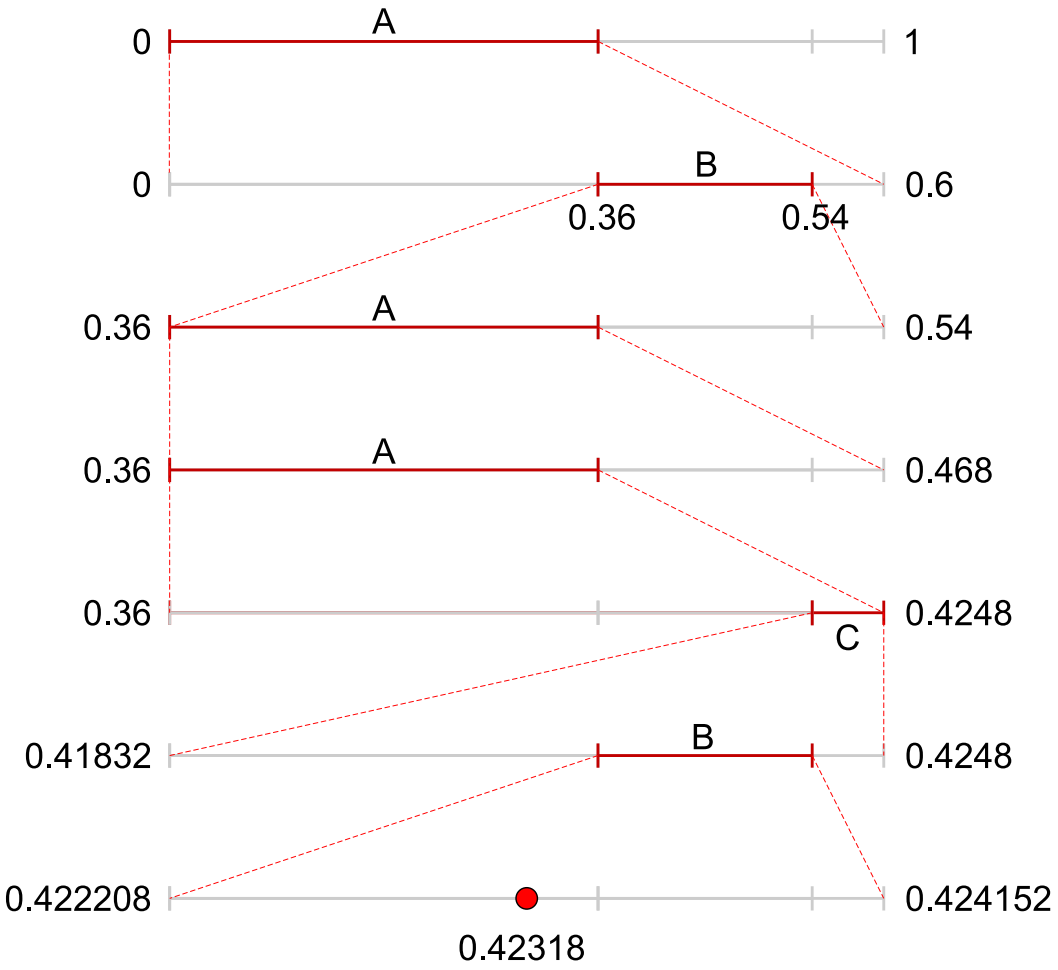
Kod *"clear table"* umożliwia ponowne zainicjowanie tabeli po jej zapełnieniu, co pozwala dostosować kodowanie do zmieniających się wzorców w danych wejściowych. Inteligentne warianty algortymu mogą monitorować wydajność kompresji i czyścić tabelę, gdy ta wydajność wyraźnie spada, co zwykle oznacza, że istniejąca tabela nie pasuje już do danych wejściowych.

Ponieważ porządek dodawania kodów określony jest jednoznacznie przez dane, dekoder rekonstruuje słownik naśladując kroki kodera na podstawie wynikowych kodów. Istotne jest aby skompresowany plik zawierał w części nagłowkowej wszelkie specyfikacje co do wersji algorytmu: rozmiar alfabetu, maksymalny rozmiar słownika (max długość kodu), czy kodowanie wykorzystuje słowa o stałej czy też zmiennej długości, wartości kodów *"clear"* i *"stop"* jeśli są wykorzystywane.

Kodowanie arytmetyczne

Ciąg znaków z $\mathcal{A}$ $\quad\rightarrow\quad$ liczba $x \in [0,1]$.

PRZYKŁAD: $p(A) = 0.6$, $p(B) = 0.3$, $p(C) = 0.1$. $H(\mathcal{A}) = 1.295$.
Kodujemy napis ABAACB, kod Huffmana: $h(A) = 0$, $h(B) = 10$, $h(C) = 11$.
Stąd $\mathbb{E}(h) = 1.4$, $h(ABAACB) = 010001110$.



Odkodowanie: $x = 0.42318 =_2 0.0110110001$

$0 < 0.42318 < 0.6$	litera A, $x = x/0.6$
$0.6 < 0.7053 < 0.9$	druga litera B, $x = (x - 0.6)/0.3$
$0 < 0.351 < 0.6$	kolejne A, $x = x/0.6$
$0 < 0.585 < 0.6$	następne A, $x = x/0.6$
$0.9 < 0.975 < 1.0$	litera C, $x = (x - 0.9)/0.1$
$0.6 < 0.75 < 0.9$	litera B

Algorytm kodowania arytmetycznego

Dane: $\mathcal{A} = \{a_1, a_2, ..., a_n\}$, $p_i = p(A = a_i)$, $D[j] = p_1 + p_2 + \cdots + p_j$

KODOWANIE:

```
a=0.0;  b=1.0;
c = getchar();
while ( !EOF(input) ){
   i = index[c];
   r = b-a;
   b = a + r*D[i];
   a = a + r*D[i-1];
   c = getchar();
}
x = (b-a)/2;
```

ROZKODOWANIE:

```
do {
   i = 0;
   while ( D[i]<x ) i++;
   output ( symbol[i] );
   r = D[i] - D[i-1];
   x = (x - D[i-1])/r;
} while ( symbol[i] != EOF );
```

Jedno miejsce dziesiętne to średnio 3.322 bita. Dla dokładnych pięciu miejsc dzisiętnych potrzeba więc ok. 16–17 bitów, tu jednak wystarczy mniej. Wynika to z efektywniejszej w tym wypadku arytmetyki binarnej.