

## CMSC 39600: Online Algorithms

Lecture 21

Course Instructor: Adam Kalai

Date: November 15, 2004

### Prediction, compression and investing

Today, we're going to relate the problems of probabilistic prediction, data compression, and investing. First, we will discuss how a probability distribution that assigns probability  $p(x)$  to  $x$  can be thought of as a compression algorithm that encodes  $x$  using  $-\log_2 p(x)$  bits (give or take).

Without knowing your backgrounds, I thought I'd give a brief introduction to coding. Some of you may be familiar with this stuff already.

## 1 Compression, entropy and probability

We would like to encode an element  $x$  in some countable set  $X$ , using a  $\{0, 1\}$  string. A code  $c : X \rightarrow \{0, 1\}^*$  is a function from  $x \in X$  to its encoding  $c(x) \in \{0, 1\}^*$ . One property that we would like to have of a code is that  $c$  is one-to-one, i.e., we can uniquely decode  $x$  from its encoding  $c(x)$ . A second property (which subsumes this) is that no encoding  $c(x)$  should be a prefix of another  $c(x')$  for  $x \neq x'$ . This is important so that, as we read the encoding, we know when to stop (we are not assuming any special STOP symbol).<sup>1</sup> From now on, we will assume that all of our codes are *prefix codes*, i.e. they have this property.

Finally, the last property that we would like of a code is a type of efficiency. Namely, we can never predict the next bit of an encoding with certainty. Let us illustrate with some examples. Suppose  $X = \{A, B, C\}$ . An example of an efficient prefix code is  $c(A) = 0$ ,  $c(B) = 10$ ,  $c(C) = 11$ . A non-prefix example would be  $c(A) = 0$ ,  $c(B) = 1$ ,  $c(C) = 00$ . An inefficient prefix example would be  $c(A) = 0$ ,  $c(B) = 11111$  and  $c(C) = 1110001110101$ . From now on we only consider efficient prefix codes. This means that we can represent the code as a binary tree, where each internal node has exactly two children. Each edge corresponds to either a 0 or 1, and each leaf corresponds to an encoding  $c(x)$  for some  $x \in X$ .

Now, for each (efficient prefix) code  $c$ , we define a probability distribution  $p_c$  over  $X$  by randomly choosing a  $\{0, 1\}$  string until we get an encoding, equivalently randomly descending the tree until we reach a leaf. Thus the probability  $p_c(x) = 2^{-|c(x)|}$ , where  $|c(x)|$  is the length of the encoding of  $x$ . (Rearranging, we get that  $|c(x)| = -\log_2 p_c(x)$ .) Now, for any probability distribution  $p$  over  $X$ , the quantity in concern is the *average length of the encoding*, which is  $E_{x \leftarrow p}[|c(x)|]$ . For example, suppose the distribution was  $p(A) = 1/2$ ,  $p(B) = 1/4$  and  $p(C) = 1/4$  and the code was  $c(A) = 0$ ,  $c(B) = 10$ , and  $c(C) = 11$ . Then the average length of the encoding would be  $(1/2)1 + (1/4)2 + (1/4)2 = 1.5$  bits. This is optimal. Also notice that with this encoding,  $p_c = p$ .

**Theorem 1** *The average length of an encoding of  $x \in X$  chosen according to distribution  $p$  is at least*

$$E_{x \leftarrow p}[|c(x)|] \geq E_{x \leftarrow p}[-\log_2 p(x)].$$

---

<sup>1</sup>This is also nice, in case we want to encode a sequence  $x_1, x_2, \dots, x_T \in X$  using  $c(x_1)c(x_2) \dots c(x_T)$ . Such prefix codes are like languages where you don't need spaces between words.

**Proof.** Notice that  $|c(x)| = -\log_2 p_c(x)$ . Hence,

$$E_{x \leftarrow p}[|c(x)|] = E_{x \leftarrow p}[-\log_2 p_c(x)].$$

Using Jensen's inequality (which states that  $E[f(x)] \geq f(E[x])$  for concave functions  $f$  like  $-\log$ ) we have

$$E[-\log_2 \frac{p_c(x)}{p(x)}] \geq -\log_2 E[\frac{p_c(x)}{p(x)}] = -\log_2 1 = 0.$$

We have used the fact that  $E[p_c(x)/p(x)] = \sum_{x \in X} p(x)(p_c(x)/p(x)) = \sum p_c(x) = 1$ . Using  $\log(x/y) = \log(x) - \log(y)$  and the above two displayed equations gives the theorem.  $\square$  Notice

that in our previous example, we achieved  $p = p_c$  and so we had equality in the above theorem. The quantity,  $E_{x \leftarrow p}[-\log p(x)]$ , is called the *entropy* of the distribution  $p$  and gives a lower bound on how efficiently we can encode random  $x \in X$  drawn according to  $p$ .

## 2 Huffman codes

Next, we want to see how close one can get to  $E_{x \leftarrow p}[-\log p(x)]$ , the *entropy* of a distribution  $p$ . From now on, all  $\log$ 's will be base 2. Actually, we can get very close (within 1 bit!).

The Huffman code is the optimal code (minimum average encoding length) for a given distribution, and can be constructed in time polynomial in  $|X|$ . The code is best described by the binary tree that represents the code, as described before. See [http://theory.csail.mit.edu/classes/6.897/spring03/scribe\\_notes/L13/lecture13.pdf](http://theory.csail.mit.edu/classes/6.897/spring03/scribe_notes/L13/lecture13.pdf) for a more complete presentation of Huffman codes. But the idea is to build a bunch of trees (a forest) and slowly reduce the number of trees to 1. We start with each  $x \in X$  with non-zero probability in its own tree, which is a trivial tree with a single leaf. We define the probability of a tree to be the sum of the probabilities of all  $x$  in the tree. The algorithm is iterative: each time create merge the two trees of minimal weight into a single tree with a new root (arbitrarily labeling one child 0 and the other 1). This repeats until there is a single node.

**Theorem 2** *For any probability distribution  $p$ , the expected length of an encoding for the Huffman code is optimal. Moreover,*

$$E[|c(x)|] \leq E[-\log p(x)] + 1.$$

**Proof.** We only prove the second part. Suppose that  $p$  was a *power of 2 distribution*, meaning that for every  $x$  with nonzero probability,  $p(x) = 2^{-k_x}$  for some integer  $k_x$ , i.e.,  $-\log p(x) \in \mathbb{Z}$ . Then one can imagine reverse-engineering  $p$  and construct a code  $c$  such that  $p_c = p$ , as we saw in our example.

However, in general  $p$  may not be such a nice distribution. For a distribution  $p$ , define  $p'(x) = 2^{-\lceil \log p(x) \rceil}$ . Hence  $p'(x) \geq p(x)/2$ . Unfortunately,  $p'$  is not quite a distribution because  $\sum p'(x)$  may be less than 1. So let's add some new element  $\nu$  to  $X$  and give it probability  $p'(\nu) = 1 - \sum p'(x)$ .

It is not too difficult to see that we can now reverse-engineer  $p'$  to construct a code  $c$  such that  $p_c = p'$ . (I won't prove this, but the basic idea is that if one has a bunch of powers of two that sum to one, then there must be a way of partitioning them into two groups that each sum to  $1/2$ . Then one can put each half in a different part of the binary tree.)

Finally, we have that

$$E[|c(x)|] = E[-\log p_c(x)] = E[-\log p'(x)] \leq E[-\log p(x)/2] = E[-\log p(x)] - 1$$

Since Huffman codes are optimal, which we did not show, they achieve this bound. □

### 3 Dynamic Huffman codes