Compression Outline

Introduction: Applications, Lossy vs. Lossless, Model

Information Theory Concepts: Entropy, Conditional entropy etc.

Probability Coding:

- Prefix codes and relationship to Entropy
- Huffman codes

Lossy compression: Quantization

Scalar Quantization

Quantize regions of values into a single value

E.g. Drop least significant bit

(For images, can be used to reduce # of bits for a pixel)

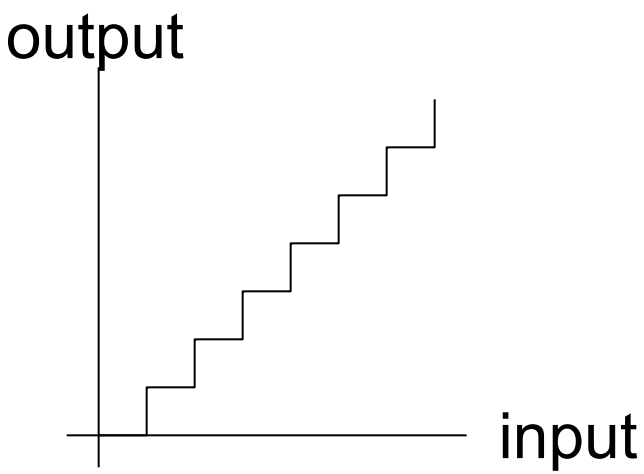
Q: Why is this lossy?

Many-to-one mapping

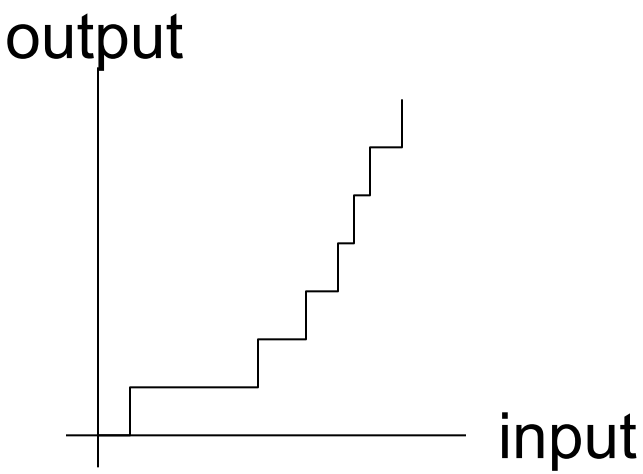
Two types

- Uniform: Mapping is linear
- Non-uniform: Mapping is non-linear

Scalar Quantization



uniform



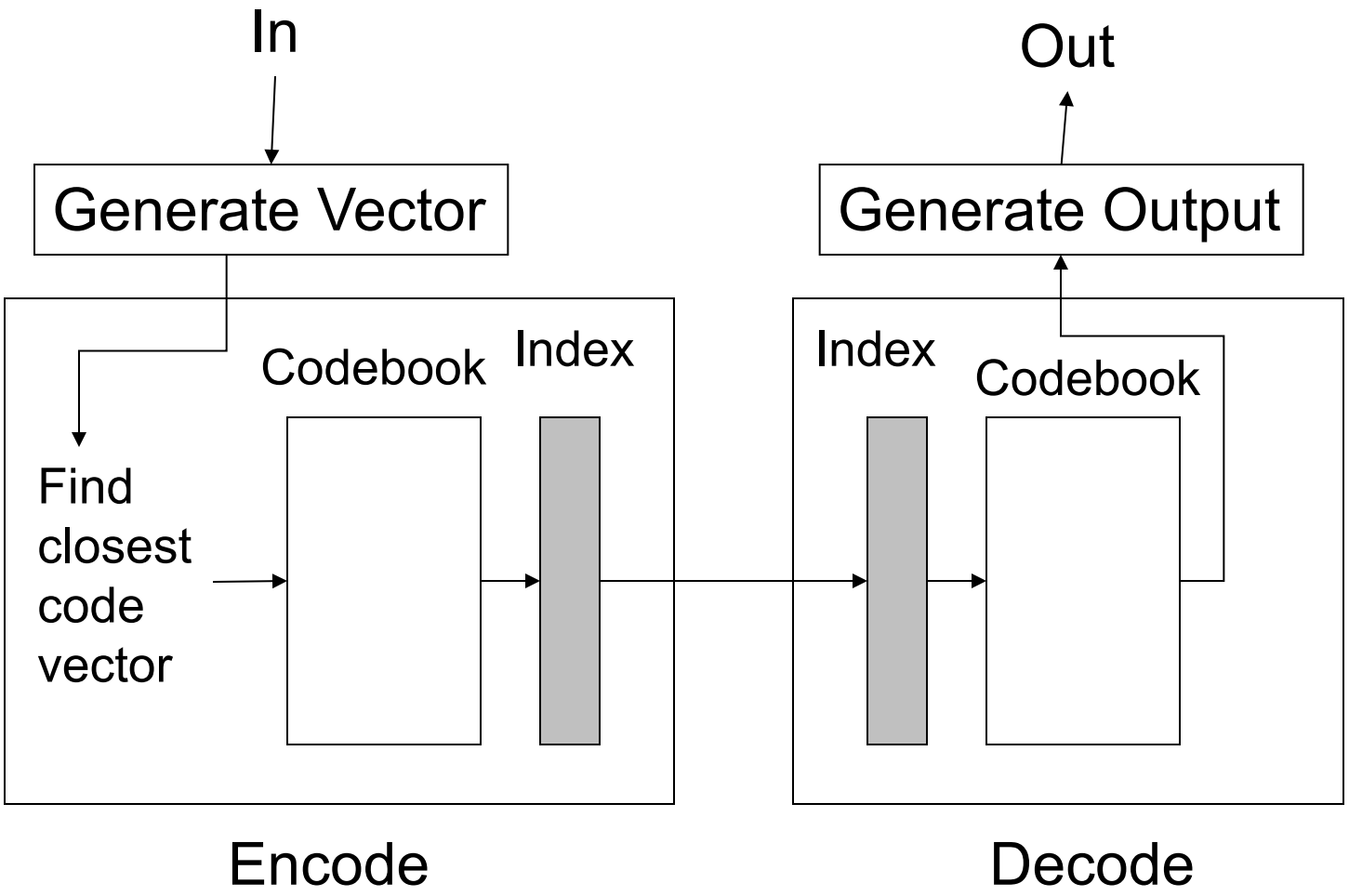
non uniform

Q: Why use non-uniform?
Error metric might be non-uniform.
E.g. Human eye sensitivity to specific color regions

Can formalize the mapping problem as an optimization problem

Vector Quantization

Mapping a multi-dimensional space into a smaller set of messages



Vector Quantization

Examples of what are used as vectors:

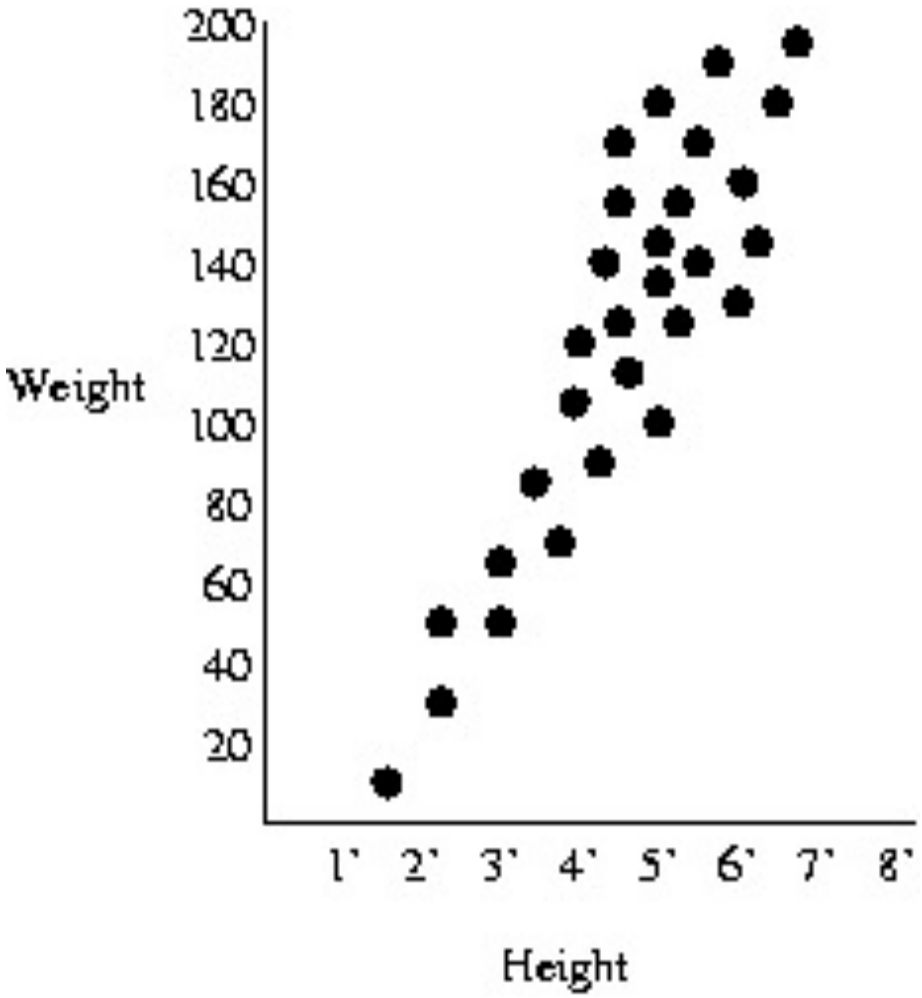
- Color (Red, Green, Blue)
  - Can be used, for example to reduce 24bits/pixel to 8bits/pixel
- K consecutive samples in audio
- Block of K pixels in an image
- Weights of a neural network layer

How do we decide on a codebook

- Typically done with **clustering**

VQ most effective when the variables along the dimensions of the space are correlated

Vector Quantization: Example



Observations:

1. Highly correlated:
   Concentration of representative points

2. Higher density is more common regions.

15-750: Graduate Algorithms

Review of modules:

- Hashing

Hashing

Setting:

A large set of (possible) values: called universe U

Interested in only a subset of this: S

Let |S| = N (typically N << |U|)

Roughly, hashing is a way to map elements of U onto smaller number of values such that with high probability there are not too many collisions among elements of S.

- We will assume a family of hash functions H.
- When it is time to hash S, we choose a random function $h \in H$

Hashing: Desired properties

Let [M] = {0, 1, ..., M-1}

We design a hash function h: U -> [M]

1. Small probability of distinct keys colliding:
    1. If $x \neq y \in S$, P[h(x) = h(y)] is "small"
2. Small range, i.e., small M so that the hash table is small
3. Small number of bits to store h
4. h is easy to compute

Ideal Hash Function

Perfectly random hash function:

For each $x \in S$, $h(x)$ =a uniformly random location in [M]

Properties:

- Low collision probability: $P[h(x) = h(y)] = 1/M$ for any $x \neq y$
- Even conditioned on hashed values for any other subset A of S, for any element $x \in S$, $h(x)$ is still uniformly random over [M]

Universal Hash functions

Captures the basic property of non-collision.

Due to Carter and Wegman (1979)

Definition: A family H of hash functions mapping U to [M] is universal if for any $x \neq y \in U$,

$$P[h(x) = h(y)] \leq 1/M$$

Note: Must hold for every pair of distinct x and $y \in U$.

Universal Hash functions

A simple construction of universal hashing:

Assume |U| = 2^u and |M| = 2^m

Let A be a m x u matrix with random binary entries.

For any x ∈U, view it as a u-bit binary vector, and define

$$h(x) := Ax$$

where the arithmetic is modulo 2.

Theorem. The family of hash functions defined above is universal.

Addressing collisions in hash table

One of the main applications of hash functions is in hash tables (for dictionary data structures)

Handling collisions:

Closed addressing

Each location maintains some other data structure

One approach: "**separate chaining**"

Each location in the table stores a **linked list** with all the elements mapped to that location.

Look up time = length of the linked list

To understand lookup time, we need to study the number of many collisions.

Addressing collisions in hash table

Using universal hashing:

$M >= N^2$

P[there exists a collision] = ½

⇒ Can easily find a **collision free hash table**!

⇒ Constant lookup time for all elements! (worst-case guarantee)

Can we do better? O(N)? (while providing worst-case guarantee?)

Perfect hashing

Handling collisions via "**two-level hashing**"

- First level hash table has size O(N)
- Each location in the hash table performs a collision-free hashing

Let C(i) = number of elements mapped to location i in the first level table

For the second level table, use C(i)^2 as the table size at location i. (We know that for this size, we can find a collision-free hash function)

Collision-free and O(N) table space!

k-wise independent hash functions

In addition to universality, certain independence properties of hash functions are useful in analysis of algorithms

Definition. A family H of hash functions mapping U to [M] is called k-wise-independent if for any k distinct keys

$x_1, x_2, \ldots x_k$ and any k values $\alpha_1, \alpha_2, \ldots, \alpha_k$

we have

$$P(h(x_1) = \alpha_1 \cap h(x_2) = \alpha_2 \cap \ldots \cap h(x_k) = \alpha_k) \leq \frac{1}{m^k}$$

Case for k=2 is called "pairwise independent.

Constructions: 2-wise independent

Construction 1 (variant of random matrix multiplication):

Let A be a m x u matrix with uniformly random binary entries.

Let b be a m-bit vector with uniformly random binary entries.

$$h(x) := Ax + b$$

where the arithmetic is modulo 2.

Constructions: 2-wise independent

Construction 3 (Using finite fields)

Consider $GF(2^u)$

Pick two random numbers $a, b \in GF(2^u)$. For any $x \in U$, define

$h(x) := ax + b$

2-wise independent.

Constructions: k-wise independent

Construction 4 (k-wise independence using finite fields):

Consider GF(2^u).

Pick k random numbers $a_0, a_1, \ldots, a_{k-1} \in GF(2^u)$

$h(x) = a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}$

where the calculations are done over the field GF(2u).

Application: Cuckoo hashing

Another open addressing hashing method.

Invented by Pagh and Rodler (2004).

Take two tables T1 and T2, both of size M = O(N).

Take two hash functions h1, h2: U -> [M] from hash family H.

Let H be fully-random

Cuckoo hashing

Inserting an element x:

1. If either T[h1(x)] or T[h2(x)] is empty, put the element x in that location.
2. If not bump out the element (say y) in either of these locations and put x in.
3. When an element gets bumped out, place it in the other possible location. If that is empty then done. If not, bump the element in that location and place y there.
4. If any element relocated more than once then rehash everything.

Query/delete:

An element x will be either in T[h1(x)] or T[h2(x)].

O(1) operations

Cuckoo hashing

Theorem. The expected time to perform an insert operation is O(1) if M >= 4N.

Proof sketch used “cuckoo graph”:

- M vertices corresponding to hashtable locations
- Edges correspond to the items to be inserted.
    - For all x in S, e_x=(h1(x),h2(x)) will be in the edge set

Bloom filter

Representing a dictionary with far fewer bits when only need membership query.

Possible if we:

Allow to make mistakes on membership queries

No deletions

Data structure: “Bloom filter” [Bloom 1970]

- Only false positives; no false negatives
  - may report that a key is present when it is not

Bloom filter

Space efficient data structure for *approximate* membership queries.

- Keep an array T of M bits
  - initially all entries are zero.
- k hash functions: h_1, h_2, .., h_k: U -> [M]

Adding a key:

- To add a key $x \in S \subseteq U$, set bits $T[h_1(x)]$, $T[h_2(x)]$, ..., $T[h_k(x)]$ to 1

Bloom filter

Membership query:

- For a query for key $x \in U$: check if all the entries $T[h_i(x)]$ are set to 1
- If so, answer Yes else answer No.

If an item x is present, then corresponding bits will be set.

Other elements could have set the same bits.

Let ϵ denote the prob. of false positives.

$$M \geq 1.44 N \log(1/\epsilon)$$

$1.44 \log(1/\epsilon)$ bits per element

Load balancing

N balls and N bins. Randomly put balls into bins

Question of interest: Understanding the number of balls in the **maximally loaded** bin.

Theorem: The max-loaded bin has $O(\frac{\log N}{\log \log N})$ balls with probability at least 1 – 1/N.

Theorem. With high probability the max load is $\Omega\left(\frac{\log n}{\log \log n}\right)$

Uniformly randomly placing balls into bins does not balance the load after all!

Load balancing: power-of-2-choice

When a ball comes in, pick two bins and place the ball in the bin with smaller number of balls.

Turns out with just **checking two bins** maximum number of balls drops to **O(log log N)**!

=> called “power-of-2-choices”

Intuition:

Even though max loaded bins has $O(\frac{\log N}{\log \log N})$ balls, most bins have far fewer balls.

Load balancing: power-of-d-choice

When a ball comes in, **pick d bins** and place the ball in the bin with smallest number of balls.

Theorem:

For any d>=2 the d-choice process gives a maximum load of

$$\frac{\log \log N}{\log d} \pm O(1)$$

with probability at least 1 – O(1/N)

Observations:

Just looking at two bins gives huge improvement.

Diminishing returns for looking at more than 2 bins.

Concentration Bounds

Central question:

What is the probability that a R.V. deviates much from its expected value

- Typically want to say a R.V. stays “close to” it expectation “most of the time”

Useful in analysis of randomized algorithms

Markov's Inequality

The most basic concentration bound.

Let X be a **non-negative** R.V. with mean μ then

$$P(X \geq \alpha) \leq \frac{\mu}{\alpha}$$

Uses expectation only

Chebyshev's Inequality

More powerful than Markov's

Let X be a R.V. with mean μ and variance σ^2

$$P(|X - \mu| \geq \epsilon) \leq \frac{\sigma^2}{\epsilon^2}$$

Stronger since it uses variance information
Smaller the variance more concentrated the R.V. around mean

Chernoff Bound

For any R.V. X, for any t>0

$$P(X \geq a) \leq \frac{E[e^{tX}]}{e^{ta}}$$

$$\Rightarrow P(X \geq a) \leq \min_{t>0} \frac{E[e^{tX}]}{e^{ta}}$$

There are many different variants of Chernoff bounds applied to various different distributions

Hoeffding Bound

Hoeffding bound:

Let Xi's be independent R.V.s taking values in [0,1].

Let X = X1 + X2 + … + Xn

Let $\mu = E[X]$

$$P(X > \mu + \lambda) \leq e^{\frac{-\lambda^2}{2\mu + \lambda}}$$

$$P(X < \mu - \lambda) \leq e^{-\frac{\lambda^2}{3\mu}}$$

Exponential decay!

Much much stronger Markov and Chebyshev.

Chernoff Bounds for Binomial

Binomial = sum of Bernoulli (i.e. Binary valued) R.V.s

Let $X = \sum_{i=1}^{n} x_i$

Where X_i's = Bernoulli (p) and independent.

$\mu = E[X] = np$

Then for all $\delta > 0$

$$P(X - np \geq \delta) \leq e^{-\frac{2\delta^2}{n}}$$

$$P(X - np \leq -\delta) \leq e^{-\frac{2\delta^2}{n}}$$

Data streaming model

- Different computational model: elements going past in a "stream"
- Limited storage space: Insufficient to store all the elements

- Functions of interest:
  - Sum of all elements seen (easy)
  - Max of the elements seen (easy)
  - Median (tricky to do with small space)
  - **Heavy-hitters, i.e., element(s) that have appeared most often)**
  - Number of distinct elements seen

Sampling vs. Hashing

Sampling is a natural option (since it helps reduce the amount of data)

But can lead to incorrect answers if not done correctly.

Example from [1]:

Suppose we want to figure out

#“uniques” = elements that occur exactly once.

Consider this sampling approach:

- Sample 10% of the stream by picking each element with probability 0.1.
- Count uniques and scale up the answer by 10

1. “Mining of Massive Datasets” book from Stanford: http://infolab.stanford.edu/~ullman/mmds/book.pdf

Streams as vectors

A useful abstraction: Viewing streams as vectors (in high dimensional space)

Stream at time t as a vector $x^t \in Z^{|U|}$

$$x^t = (x^t_1, x^t_2, \ldots, x^t_{|U|})$$

Element i = #times i^{th} element of U has been seen until time t

Makes it easy to formulate some of the data stream problems:

- Heavy hitters = estimate "large" entries in the vector x
- Total number of elements seen = Sum of the elements of x (easy one)
- #distinct elements = #non-zero entries in x

Heavy hitters

Many ways to formalize the heavy hitters problem.

ε-heavy-hitters: Indices i such that $x_i > \varepsilon \| x \|_1$

Considered a simpler problem.

Count-Query:

At any time t, given an index i, output the value of x^t_i with an error of at most $\varepsilon\|x^t\|_1$. i.e., output an estimate

$$y_i \in x_i \pm \varepsilon \| x \|_1$$

Hashing-based solution: Count-Min Sketch

By Cormode and Muthukrishnan.

Step 1:

Let h: U -> [M] be a hash function

Let a A [1...M] be an array of non-negative integers

When an add (or del) arrives for item i, perform corresponding increment (or decrement) on A[h(i)]

Showed the expected error is small, assuming universal family of hash functions

Hashing-based solution: Count-Min Sketch

Step 2:

Amplify the probability that we are close to the expectation: Independent repetitions!

ℓ hash functions: $h_1, h_2, .., h_\ell : U \rightarrow [M]$

ℓ arrays $A_1, \ldots A_\ell$

(one for each hash function)

- Same approach as before applied independently on each of the ℓ arrays using the associated hash function
- And take min over ℓ values to get the estimate