CPT S 515 Homework #3

Lubah Nelson

October 13

No late homework!

0.1 Question 1:

We can mathematically estimate the number of bits leaked based on statistical dependencies between A and B, using mutual information, and to compute the maximum matching in a bipartite graph to quantify the leakage.

Let:

- A: unobservable to the intruder
- B: observable to the intruder

$$\mathcal{A}, \mathcal{B} \subset \{0,1\}^{10}$$

be the sets of possible 10-bit strings for A and B. Given a dataset $C = \{(A^{(n)}, B^{(n)})\}_{n=1}^{N}$, our task is to construct a bipartite graph $G = (V_A, V_B, E)$, where:

- $V_A = \{a_1, \ldots, a_{10}\}$ represents the bits of A,
- $V_B = \{b_1, \ldots, b_{10}\}$ represents the bits of B,
- An edge (a_i, b_j) is added if bit B_j reveals significant information about bit A_i, determined by mutual information.

1. Compute Mutual Information

Mutual information quantifies the dependency between bits A_i and B_j. The mutual information between A_i and B_j is computed as:

$$I(A_i; B_j) = \sum_{a\in\{0,1\}} \sum_{b\in\{0,1\}} P_{A_iB_j}(a,b) \log_2 \left(\frac{P_{A_iB_j}(a,b)}{P_{A_i}(a)P_{B_j}(b)} \right),$$

where the joint probability $P_{A_iB_j}(a,b)$ is estimated from the dataset C as:

$$P_{A_iB_j}(a,b) = \frac{1}{N} \sum_{n=1}^{N} \delta(A_i^{(n)} = a, B_j^{(n)} = b),$$

and δ is the indicator function.

2. Construct Bipartite Graph

For each bit pair (i, j), we add an edge (a_i, b_j) to the bipartite graph G if the mutual information $I(A_i; B_j)$ exceeds a threshold θ, indicating a significant dependency. Thus, the bipartite graph $G = (V_A, V_B, E)$ represents dependencies between bits in A and B.

3. Maximum Matching

A matching is a set of edges such that no two edges share a vertex. The maximum matching in the bipartite graph corresponds to the maximum number of independent dependencies between A and B, where each edge represents a bit of A that can be inferred from B.

4. Algorithm

We compute the maximum matching $|M|$ using a breadth-first search (BFS)-based approach to find augmenting paths in the bipartite graph. The size of this matching $|M|$ gives the number of bits leaked from A to B.

Algorithm Steps

1. **Input Dataset**: Take the dataset $C = \{(A^{(n)}, B^{(n)})\}_{n=1}^{N}$ and initialize the bipartite graph G.
2. **Compute Mutual Information**: For each pair of bits (A_i, B_j), compute $I(A_i; B_j)$. If $I(A_i; B_j) > \theta$, add an edge (a_i, b_j) to G.
3. **Build Bipartite Graph**: Construct the bipartite graph $G = (V_A, V_B, E)$ where edges represent significant dependencies between bits in A and B.
4. **Maximum Matching (BFS)**: Perform a BFS to find augmenting paths and compute the maximum matching $|M|$. This will give the maximum number of independent dependencies between A and B.
5. **Estimate Information Leakage**: The number of bits leaked is given by the size of the maximum matching:
$$\text{Leaked Bits} = |M|.$$

1 Question 2

We will transform the given function into an Integer Linear Programming (ILP) model. By representing the program's logic as linear constraints and variables, we can use an ILP solver to check for the existence of inputs that lead to a negative return value.

- Input Variables: $x_1, x_2, \ldots, x_7$ (integer inputs to the function). - Program Variables: x, y, z (declared and updated within the function). - State Variables: For each line i in the program, v_i denotes the value of variable v after executing line i. This tracks the evolution of variable values throughout the program. - Binary Variables: For each ‘if-then-else‘ statement at line i, introduce a binary variable b_i:

$$b_i = \begin{cases} 1 & \text{if the condition at line } i \text{ is true (then branch),} \\ 0 & \text{if the condition is false (else branch).} \end{cases}$$

Each assignment variable := Exp is represented as a linear equality constraint. For an assignment at line i:

$$\text{variable}_i = \text{Exp}_{i-1}$$

where: - variable_i is the value of the variable after line i. - Exp_{i-1} is a linear expression involving variables from the previous state (after line $i-1$).

Variables not updated in line i remain unchanged:

$$\forall \text{variables } v \text{ not assigned at line } i, \quad v_i = v_{i-1}$$

For each ‘if-then-else‘ statement: - Condition: A comparison between two linear expressions, e.g., $\text{LHS}_{i-1} > \text{RHS}_{i-1}$. - Binary Variable: $b_i \in \{0, 1\}$ represents the outcome of the condition. - Condition Constraints: - Use the Big-M method to linearize the condition:

$$\text{LHS}_{i-1} - \text{RHS}_{i-1} \geq 1 - M(1 - b_i)$$

- M is a large positive constant. - This ensures that when $b_i = 1$, the condition holds (since $\text{LHS}_{i-1} - \text{RHS}_{i-1} \geq 1$). - When $b_i = 0$, the constraint is relaxed due to $M(1 - b_i)$. - Branch Assignments: - Then Branch ($b_i = 1$):

$$\text{variable}_i = \text{Exp}_{i-1}^{\text{then}} + M(1 - b_i)$$

- Else Branch ($b_i = 0$):

$$\text{variable}_i = \text{Exp}_{i-1}^{\text{else}} + M b_i$$

This setup ensures that the assignments for the inactive branch are relaxed by the Big-M term.Variables not updated in either branch maintain their previous values, adjusted with Big-M to account for branching.

1.1 ILP solution –

1.1.1 Line 1: Assignment

- Code: ‘x = 2*x1 + 3*x2 - 5;‘ - Constraint:

$$x_1 = 2x_1^{\text{in}} + 3x_2^{\text{in}} - 5$$

- x_1^{in} and x_2^{in} are the input variables. - Unchanged Variables:

$$y_1 = y_0, \quad z_1 = z_0, \quad x_{i,1} = x_{i,0} \quad \forall i \neq 1$$

1.1.2 Line 2: Assignment

- Code: ‘y = x + x3;‘ - Constraint:

$$y_2 = x_1 + x_3^{\text{in}}$$

- Unchanged Variables:

$$x_2 = x_1, \quad z_2 = z_1, \quad x_{i,2} = x_{i,1} \quad \forall i \neq 1, 3$$

1.1.3 Line 3: If-Then-Else Statement

- Condition: ‘if (y ¿ 12*x1 - z)‘ - Binary Variable: b_3 - Condition Constraint:

$$y_2 - 12x_1^{\text{in}} + z_2 \geq 1 - M(1 - b_3)$$

- Then Branch Assignments ($b_3 = 1$): - Code: ‘x2 = 3*x7 - 15;‘ - Constraint:

$$x_{2,3} = 3x_7^{\text{in}} - 15 + M(1 - b_3)$$

- Else Branch Assignments ($b_3 = 0$): - Code: ‘x5 = 18*x4 - 6*x7 + 6;‘ - Constraint:

$$x_{5,3} = 18x_4^{\text{in}} - 6x_7^{\text{in}} + 6 + Mb_3$$

- Unchanged Variables: - For variables not assigned in a branch, their values are carried over with Big-M adjustments. - For example, $x_{2,3} = x_{2,2} + Mb_3$ when $b_3 = 0$ (else branch). - Similarly for $x_{5,3}$ when $b_3 = 1$.

1.1.4 Lines 4 to 9

- Assignments and Conditionals: - Repeat the above process for each line, translating assignments into constraints and handling conditionals with binary variables and Big-M constraints. - Variables: - Track the state variables v_i for all program variables at each line.

1.1.5 Line 10: Return Statement

- Code: ‘return x;‘ - Constraint:

$$x_{10} < 0$$

- Ensures that the final value of x is negative.

The final ILP formulation is –

- Input Variables:

$$x_1^{\text{in}}, x_2^{\text{in}}, x_3^{\text{in}}, x_4^{\text{in}}, x_5^{\text{in}}, x_6^{\text{in}}, x_7^{\text{in}} \in \mathbb{Z}$$

- Program Variables: - For each line i, variables:

$$x_i, y_i, z_i, x_{2,i}, x_{5,i}, \ldots \in \mathbb{Z}$$

- Binary Variables: - For each ‘if-then-else‘ statement at line i:

$$b_i \in \{0, 1\}$$

- Assignment Constraints: - For each assignment at line i:

$$\text{variable}_i = \text{Exp}_{i-1}$$

- Conditional Constraints: - For each condition at line i:

$$\text{LHS}_{i-1} - \text{RHS}_{i-1} \geq 1 - M(1 - b_i)$$

- Branch assignments with Big-M adjustments as detailed earlier. - Variable Update Constraints: - Ensure variables not assigned in a line carry over their previous values, adjusted with Big-M terms based on the branch. - Return Value Constraint:

$$x_{10} < 0$$

- Feasibility Objective:

$$\text{Minimize } 0$$

- Since we are only interested in the existence of a solution, not in optimizing any particular value.

Questions 3

The objective is probelm is to assign a unique, minimal-length binary code C_P for each partition P of a finite set $K = \{1, 2, \ldots, k\}$, such that the code lies in the range $\{1, 2, \ldots, B_k\}$, where B_k is the Bell number, representing the total number of partitions of K. We aim to ensure that this mapping is injective and uses the min bits possible, with $n = \lceil \log_2 B_k \rceil$ bits. First, we can enumerate all partitions of K, we represent each partition $P = \{K_1, K_2, \ldots, K_m\}$, where:

$$K_1 \cup K_2 \cup \cdots \cup K_m = K \quad \text{and} \quad K_i \cap K_j = \emptyset \quad \text{for all} \quad i \neq j.$$

We recursively generate partitions by inserting each element of K into existing blocks K_i or creating new blocks.

To formalize this generation, we use Restricted Growth Functions (RGFs) An RGF is a sequence $(a_1, a_2, \ldots, a_k)$, where each RGF uniquely represents a partition of K, as it encodes how elements are grouped into blocks. The number of possible RGFs is B_k, the Bell number.:

$$a_1 = 1, \quad a_i \leq \max\{a_1, a_2, \ldots, a_{i-1}\} + 1 \quad \text{for} \quad i > 1.$$

Once we generate all RGFs, we sort them lexicographically. Let $\text{RGFs} = \{R_1, R_2, \ldots, R_{B_k}\}$ represent the ordered list of RGFs. For two RGFs $A = (a_1, \ldots, a_k)$ and $B = (b_1, \ldots, b_k)$, we define:

$$A < B \quad \text{if} \quad \exists\, i \text{ such that} \quad a_j = b_j \quad \forall j < i \quad \text{and} \quad a_i < b_i.$$

The lexicographical ordering ensures a consistent and unique ranking for each partition. For each partition P_i, represented by the corresponding RGF R_i, assign the code:

$$C_{P_i} = i \quad \text{for} \quad i = 1, 2, \ldots, B_k.$$

This guarantees that each partition is uniquely assigned a code between 1 and B_k. The minimal number of bits n required to encode the partitions is:

$$n = \lceil \log_2 B_k \rceil.$$

This ensures that the encoding is optimal, as B_k partitions require exactly n bits. The range of C_P lies between 1 and B_k, and we assign the smallest possible binary representation for each partition.

The reason this works can be shown with math! First, the uniqueness of the code mapping stems from the fact that each partition of K is represented by a unique RGF, and we assign a unique integer to each lexicographically ordered RGF. Since RGFs correspond to partitions, this is a 1-to-1 correspondence.

$$\text{Let } C : \text{Partitions} \to \{1, 2, \ldots, B_k\} \quad \text{be defined by} \quad C(P_i) = i.$$

Given that no two partitions share the same RGF, this function is injective.

Since there are B_k partitions of K, the minimal number of bits needed to encode them is $n = \lceil \log_2 B_k \rceil$, as we need to represent B_k distinct numbers. The encoding scheme uses exactly n bits, ensuring minimal bit usage. We have:

$$2^{n-1} < B_k \leq 2^n,$$

which guarantees that the number of bits is minimal.

Algorithm for Generating and Ranking RGFs

Algorithm 1 Generate and Encode RGF from Partition

```
 1: Input: Partition $P = \{K_1, K_2, \ldots, K_m\}$ of set $K = \{1, 2, \ldots, k\}$
 2: Output: RGF $(a_1, a_2, \ldots, a_k)$ and its code $C_P$
 3: Initialize an empty dictionary BlockIndex
 4: Set NextIndex = 1
 5: for each element $i = 1, 2, \ldots, k$ do
 6:     Find the block $B \in P$ that contains $i$
 7:     if $B \notin$ BlockIndex then
 8:         Assign BlockIndex[$B$] = NextIndex
 9:         Increment NextIndex
10:     end if
11:     Set $a_i$ = BlockIndex[$B$]
12: end for
13: Compute lexicographical rank $C_P$ of $(a_1, a_2, \ldots, a_k)$
14: return $(a_1, a_2, \ldots, a_k)$ and $C_P$
```

Algorithm 2 Generate Partition from RGF

```
1: Input: RGF $(a_1, a_2, \ldots, a_k)$
2: Output: Partition $P = \{K_1, K_2, \ldots, K_m\}$
3: Initialize empty sets $K_1, K_2, \ldots, K_m$
4: for each element $i = 1, 2, \ldots, k$ do
5:     Place element $i$ into block $K_{a_i}$
6: end for
7: return $P = \{K_1, K_2, \ldots, K_m\}$
```

Inverse Algorithm: Generating Partition from RGF

By generating all partitions of K using Restricted Growth Functions, sorting them lexicographically, and assigning unique codes C_P to each partition, we achieve a bijective mapping from partitions to integer codes. The use of $n = \lceil \log_2 B_k \rceil$ bits ensures that the encoding is minimal and efficient, satisfying the requirements of injectivity and optimal bit usage.

Question 4

We want to determine the number of Boolean variables required to uniquely represent all nodes and edges in a directed graph G with $n = 2048$ nodes using a Boolean formula. To ensure that we can represent all $n = 2048$ nodes, we require that the number of distinct combinations 2^k is at least as large as the number of nodes. This gives us the inequality:

$$2^k \geq n.$$

$$k \geq \log_2 n.$$

Substituting $n = 2048$, we calculate:

$$k \geq \log_2 2048.$$

Since $2048 = 2^{11}$, it follows that:

$$\log_2 2048 = 11.$$

Thus, $k = 11$ Boolean variables are required to uniquely identify each node. In summary, every node in the graph can be represented using exactly 11 Boolean variables.

To represent a directed edge (u, v) in the graph, where u is the source node and v is the destination node, we need to specify both u and v. Since each node requires 11 Boolean variables, representing an edge from node u to node v requires:

$$11\,(\text{for node } u) + 11\,(\text{for node } v) = 22\,\text{Boolean variables.}$$

Even though multiple edges exist in the graph, the total number of unique Boolean variables used in the Boolean formula remains 22, as the same set of variables is reused for different edges.

Question 5

Each bit has two possible values (0 or 1), and k bits can represent 2^k distinct values. Therefore, the number of bits k must satisfy:

$$2^k \geq 40.$$

Taking the logarithm base 2 on both sides:

$$k \geq \log_2 40.$$

$$\log_2 40 \approx 5.32.$$

Since k must be an integer, we round up:

$$k = \lceil 5.32 \rceil = 6.$$