ICSC 2025 Submission

Solutions to the Semi-Finals Round

Dario Loi

2025

Problem A.1: Optimal Cake Production 3
Problem A.2: The Lighthouse Code 5
Problem B.1: Defense Against Model Extraction 6
Problem B.2: Circuit Complexity. 8
Part a) Expressing $F_{2,4}$ 8
Part b) Proof of Complexity Lower Bound 8
Part c) Proof of NAND Complexity Upper Bound 9
Problem C.1: Zipf's Meaning-Frequency Law 11
a) Limitations of Dictionary-Based Studies 11
b) The von Mises-Fisher Distribution and Contextual Diversity 11
c) Rationale for Using the von Mises-Fisher Distribution 12
d) Autoregressive vs. Masked Language Models 12
e) The Proposed Method as a Diagnostic Tool 12
f) Implications of the Law's Breakdown 12
(Bonus) Factors Influencing Polysemy 13
Problem C.2: Self-Improvement Capabilities of LLMs 14
a) Self-Improvement Framework and Key Assumption.... 14
b) The Generation-Verification Gap (GV-Gap) 14
c) Greedy Decoding and its Incompatibility with Self-Improvement 14
d) Scaling of the Relative GV-Gap 15
e) Failure of Self-Improvement on Sudoku Puzzles 15
f) Proposed Task Domain for Self-Improvement 15

Solutions

This document presents the solutions for the semi-finals round of the International Computer Science Competition (ICSC) 2025. Each section corresponds to a problem from the problem sheet. For programming problems, the source code is provided with a description and in-line comments, as per the submission guidelines.

Student: Dario Loi

Problem A.1: Optimal Cake Production

This section provides a formal derivation of the solution for Problem A.1, as well as the implementation of the solution in Python.

The problem requires us to determine the optimal production strategy for a baker who wants to minimize ingredient waste, we have two possible cake recipes to choose from, shown in Table 1.1.

Our program needs to find the amount of cakes to bake under recipe A and Recipe B, such that we minimize waste. We are operating under three *ingredient constraints*:

$$\begin{cases} 100x + 50y \leq F & \text{(Flour constraint)} \\ 50x + 100y \leq S & \text{(Sugar constraint)} \\ 20x + 30y \leq E & \text{(Eggs constraint)} \end{cases} \tag{1.1}$$

We can therefore derive the waste function $W(x, y)$ as follows:

$$W(x, y) = (F - (100x + 50y)) + (S - (50x + 100y)) + (E - (20x + 30y)) \tag{1.2}$$
$$= (F + S + E) - (170x + 180y) \tag{1.3}$$
$$= (F + S + E) - U(x, y) \tag{1.4}$$

Where $U(x, y) = 170x + 180y$ is the *used ingredients function*. It is apparent that the number of available ingredients F, S, E are constants and do not affect the optimization process. Therefore, minimizing $W(x, y)$ is equivalent to maximizing $U(x, y)$, the *usage function* under the constraints in Equation (1.1).

Similarly to the solution to the qualification round problem, we can found the bounds for x and y by calculating the maximum number of cakes that can be baked using only one recipe at a time:

$$x_{\max} = \min\left(\left\lfloor \frac{F}{100} \right\rfloor, \left\lfloor \frac{S}{50} \right\rfloor, \left\lfloor \frac{E}{20} \right\rfloor\right) \qquad y_{\max} = \min\left(\left\lfloor \frac{F}{50} \right\rfloor, \left\lfloor \frac{S}{100} \right\rfloor, \left\lfloor \frac{E}{30} \right\rfloor\right) \tag{1.5}$$

We can improve our bounds by considering the constraints in Equation (1.1), to derive $x_{\max}$ and $y_{\max}$ under the assumption that we bake a fixed number of cakes of the other recipe:

$$x_{\max}(y) = \min\left(\left\lfloor \frac{F - 50y}{100} \right\rfloor, \left\lfloor \frac{S - 100y}{50} \right\rfloor, \left\lfloor \frac{E - 30y}{20} \right\rfloor\right) \tag{1.6}$$

$$y_{\max}(x) = \min\left(\left\lfloor \frac{F - 100x}{50} \right\rfloor, \left\lfloor \frac{S - 50x}{100} \right\rfloor, \left\lfloor \frac{E - 20x}{30} \right\rfloor\right) \tag{1.7}$$

Finally, our solution consists of iterating over all possible values of $x \in \{0, 1, \ldots, x_{\max}\}$ and for each value of x, determining the corresponding $y_{\max}(x)$, and then iterating over all possible values of $y \in \{0, 1, \ldots, y_{\max}(x)\}$, for each of these coordinates, we compute $U(x, y)$ and keep track of the maximum value found. We defer the computation of $W(x, y)$ to the end, once we have found the optimal (x^*, y^*) pair, saving the costs of subtracting from the constant term at each iteration.

The implementation of this solution in Python is provided in Listing 1.1.

Recipe	Flour	Sugar	Eggs
Recipe A	100	50	20
Recipe B	50	100	30

Table 1.1: Ingredient requirements for each cake recipe.

```python
def optimal_cakes(flour: int, sugar: int, eggs: int) -> int:
    """
    Determines the optimal combination of cakes from two recipes that maximizes
    total cakes and minimizes waste.

    Recipe 1: 100 flour, 50 sugar, 20 eggs
    Recipe 2: 50 flour, 100 sugar, 30 eggs

    Args:
        flour: An integer larger than 0 specifying the amount of available flour.
        sugar: An integer larger than 0 specifying the amount of available sugar.
        eggs: An integer larger than 0 specifying the amount of available eggs.

    Returns:
        An integer representing the total waste (sum of leftover ingredients)

    Raises:
        ValueError: If inputs are not positive.
    """
    if not (isinstance(flour, int) and isinstance(sugar, int) and isinstance(eggs, int)):
        raise ValueError("Inputs must be integers.")
    if flour <= 0 or sugar <= 0 or eggs <= 0:
        raise ValueError("Inputs must be positive integers.")

    total_available = flour + sugar + eggs

    # Calculate the bounds for Recipe A cakes
    max_x = min(flour // 100, sugar // 50, eggs // 20)

    best_usage = -1

    for x in range(max_x + 1):

        # Use the x-dependent bounds to calculate the maximum possible Recipe B cakes
        # under the currently considered Recipe A cake amount (x)
        y_by_flour = (flour - 100 * x) // 50
        y_by_sugar = (sugar - 50 * x) // 100
        y_by_eggs = (eggs - 20 * x) // 30

        # Select the minimum among the three constraints (worst case is the feasibility limit)
        ymax = min(y_by_flour, y_by_sugar, y_by_eggs)
        if ymax < 0:
            continue

        # Maximize usage function
        usage = 170 * x + 180 * ymax
        if usage > best_usage:
            best_usage = usage

    final_waste = total_available - best_usage
    return final_waste

```

Listing 1.1: Python implementation of the solution for Problem A.1: Optimal Cake Production

Problem A.2: The Lighthouse Code

The problem asks us to decode a message transmitted by a lighthouse using four different colors of light. We are given a hint that the code is based on ASCII.

Since there are four distinct colors (red, blue, green, yellow), we can hypothesize that each color represents a 2-bit binary number ($2^2 = 4$). A sequence of four colors would then form an 8-bit binary number, which corresponds to a single ASCII character.

Let's denote the 2-bit values for green, red, blue, and yellow as v_g, v_r, v_b, v_y respectively. The known decoded sequences give us a system of equations:

$$\begin{aligned} \text{decode}(\text{●}, \text{●}, \text{●}, \text{●}) &= \text{F} \\ \text{decode}(\text{●}, \text{●}, \text{●}, \text{○}) &= \text{G} \\ \text{decode}(\text{●}, \text{●}, \text{●}, \text{●}) &= \text{H} \end{aligned}$$

We can look up the ASCII values for these characters thanks to the table provided in the problem statement[1] and represent them in binary, as shown in Table 1.2.

Sequence	Character	ASCII (Decimal)	ASCII (8-bit Binary)
(●, ●, ●, ●)	F	70	$01\ 00\ 01\ 10_2$
(●, ●, ●, ○)	G	71	$01\ 00\ 01\ 11_2$
(●, ●, ●, ●)	H	72	$01\ 00\ 10\ 00_2$

Table 1.2: Known signal sequences and their ASCII representations.

Indeed, we can see that since F, G, and H are consecutive alphabetically (and thus in ASCII), only the last four bits change in their ASCII encoding, and the same is true for the color sequences. This reinforces our intuition that each color directly maps to a 2-bit segment of an 8-bit ASCII character. Moreover, since the color-ACSII mappings are consistent across the known signals, we can directly obtain the correspondence between colors and their 2-bit values, as shown in Table 1.3.

Color	2-bit Value
Green (●)	01_2
Red (●)	00_2
Blue (●)	10_2
Yellow (○)	11_2

Table 1.3: Deduced binary mapping for each color.

$$\begin{aligned} &s_1 = (\text{●}, \text{●}, \text{●}, \text{●}) && s_2 = (\text{●}, \text{●}, \text{●}, \text{○}) \\ &\text{Binary: } 01\ 00\ 10\ 01 = 01001001_2 && \text{Binary: } 01\ 00\ 00\ 11 = 01000011_2 \\ &\text{Character: } \mathbf{I} && \text{Character: } \mathbf{C} \\ &s_3 = (\text{●}, \text{●}, \text{●}, \text{○}) && s_4 = (\text{●}, \text{●}, \text{●}, \text{○}) \\ &\text{Binary: } 01\ 01\ 00\ 11 = 01010011_2 && \text{Binary: } 01\ 00\ 00\ 11 = 01000011_2 \\ &\text{Character: } \mathbf{S} && \text{Character: } \mathbf{C} \end{aligned} \tag{1.8}$$

The complete decoded message sent by the lighthouse is **ICSC**.

[1] ASCII Table: https://www.asciitable.com/

Problem B.1: Defense Against Model Extraction

This problem asks us to find a strategy that minimizes the total information leaked from a machine learning model while ensuring that a user trust score remains positive. We are faced with a sequence of decisions over T time periods. At each step, we can either add noise to the model's responses, which prevents information leakage at the cost of decreasing user trust, or provide a clean response, which leaks information but increases trust.

This problem can be effectively modeled as finding the shortest path in a decision tree, where the *length* of a path is the total information leaked. Since the optimal future leakage from a given time t and trust score is independent of the path taken to reach that state, we can use dynamic programming with memoization to efficiently compute the minimum leakage.

We define a function `min_leak(t, trust)` that represents the minimum possible leakage from time period t to the end, given a starting `trust` score at time t. Our goal is to compute `min_leak(0, initial_trust)`.

For any state (t, trust), we have two possible choices, we can either add noise or provide a clean response. The outcomes of these choices are as follows:

1. **Add Noise:**
   - New trust: $\text{trust}' = \text{trust} - q_t$. This is only a valid move if $\text{trust}' > 0$.
   - Total future leakage: $0 + \text{min\_leak}(t+1, \text{trust}')$.
2. **Clean Response:**
   - New trust: $\text{trust}'' = \min(\text{trust} \times 2, \text{max\_trust})$.
   - Total future leakage: $q_t + \text{min\_leak}(t+1, \text{trust}'')$.

The value of $\text{min\_leak}(t, \text{trust})$ will be the minimum of the outcomes of these two choices. If a choice is invalid (e.g., adding noise results in non-positive trust), we consider its resulting leakage to be infinite, in order to effectively prune the invalid path. The base case for our recursion is when t reaches the end of the time periods, at which point the future leakage is 0. We use a dictionary to memoize results for each state (t, trust), cutting down on redundant calculations and ensuring an efficient solution. The Python implementation of this dynamic programming approach is provided in Listing 1.2.

```python
def minimize_extraction(query_volumes: list, initial_trust: int, max_trust: int) -> int:
    """
    Determines the minimum information leaked while keeping trust above 0.

    Args:
        query_volumes: A list of integers representing information that would leak at each time period if no defense is applied
        initial_trust: An integer representing the starting user trust score
        max_trust: An integer representing the maximum possible trust score

    Returns:
        An integer representing the minimum information that must be leaked

    Raises:
        ValueError: If inputs are invalid.
    """
    if not all(isinstance(i, int) for i in [initial_trust, max_trust]):
        raise ValueError("initial_trust and max_trust must be integers.")
    if not isinstance(query_volumes, list) or not all(isinstance(q, int) for q in query_volumes):
        raise ValueError("query_volumes must be a list of integers.")

    memo = {}

    def solve(t, current_trust):
        # base case, all queries processed
        if t == len(query_volumes):
            return 0

        # try to hit the memoized result
        if (t, current_trust) in memo:
            return memo[(t, current_trust)]

        # add noise case, only if trust remains positive
        leakage_noise = float('inf')
        trust_after_noise = current_trust - query_volumes[t]
        if trust_after_noise > 0:
            # no leakage, so we just inherit future leakage
            leakage_noise = solve(t + 1, trust_after_noise)

        # clean response case
        trust_after_clean = min(current_trust * 2, max_trust)
        # leakage occurs proportionally to the query volume
        leakage_clean = query_volumes[t] + solve(t + 1, trust_after_clean)

        # pick the best option and memoize it
        result = min(leakage_noise, leakage_clean)
        memo[(t, current_trust)] = result
        return result

    return solve(0, initial_trust)
```

Listing 1.2: Python implementation of the solution for Problem B.1: Defense Against Model Extraction

Problem B.2: Circuit Complexity

This problem explores the complexity of the "at least k" Boolean function, denoted $F_{k,n}$, focusing on the case where $k = 2$. We are asked to express the function for specific parameters, prove a lower bound on its complexity in terms of the number of gates, and then establish an upper bound for its recursive construction using only NAND gates.

Part a) Expressing $F_{2,4}$

The function $F_{2,4}(x_1, x_2, x_3, x_4)$ outputs 1 if and only if at least two of its four inputs are 1. This is equivalent to stating that at least one pair of inputs (x_i, x_j) with $i \neq j$ are both 1. We can express this as a disjunction over all possible pairs of inputs. For four inputs, there are $\binom{4}{2} = 6$ such pairs. The Boolean formula in Disjunctive Normal Form (DNF) is therefore:

$$F_{2,4} = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4) \vee (x_3 \wedge x_4) \tag{1.9}$$

This formula uses 6 AND ($\wedge$) operators and 5 OR ($\vee$) operators, for a total of 11 gates.

Part b) Proof of Complexity Lower Bound

We will prove by induction that any Boolean formula computing $F_{2,n}$ has a complexity of at least $N(2, n) \geq 2n - 3$ for $n > 1$. The complexity is the number of operators in the formula.

Base Cases:

- For $n = 2$, the function is $F_{2,2}(x_1, x_2) = x_1 \wedge x_2$. This requires exactly 1 gate. The formula gives $2(2) - 3 = 1$. The base case holds.
- For $n = 3$, the function is $F_{2,3} = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. An equivalent, more optimal formula is $(x_1 \wedge x_2) \vee (x_3 \wedge (x_1 \vee x_2))$, which requires 3 gates. The bound is $2(3) - 3 = 3$. The base case holds.

Inductive Hypothesis: Assume that for any integer m such that $1 < m < n$, any formula computing $F_{2,m}$ has a complexity of at least $N(2, m) \geq 2m - 3$.

Inductive Step: Let ϕ be an optimal formula for $F_{2,n}$. We aim to show that $N(2, n) \geq 2n - 3$. We can relate $F_{2,n}$ to $F_{2,n-1}$ by considering the role of one variable, say x_n. The condition "at least two of $\{x_1, \ldots, x_n\}$ are 1" can be split into two cases:

1. $x_n = 0$: The condition becomes "at least two of $\{x_1, \ldots, x_{n-1}\}$ are 1". This is precisely the function $F_{2,n-1}$.
2. $x_n = 1$: The condition becomes "at least one of $\{x_1, \ldots, x_{n-1}\}$ is 1". This is the OR function on $n - 1$ variables, which we denote OR_{n-1}.

This gives the recursive identity: $F_{2,n} = (\neg x_n \wedge F_{2,n-1}) \vee (x_n \wedge \text{OR}_{n-1})$.

Now, consider an optimal formula for $F_{2,n}$. If we set the input x_n to 0, the formula must simplify to a formula that computes $F_{2,n-1}$. The simplification can, at most, remove the gates to which x_n was a direct input and propagate the constant value. At least one gate must be affected if the function depends on x_n. Thus, $N(2, n) > N(2, n - 1)$.

To achieve the bound of $2n-3$, we argue more formally that at least two gates must be added to a circuit for $F_{2,n-1}$ to create one for $F_{2,n}$. Any formula for $F_{2,n}$ must incorporate the logic for when $x_n = 1$. This new logic, which is active when $x_n = 1$, corresponds to OR_{n-1}. This requires connecting x_n with the other variables, costing at least one gate (e.g., to compute $x_n \wedge \text{OR}_{n-1}$). The result of this new term must then be combined with the original logic for $F_{2,n-1}$, which requires at least one more gate (e.g., an OR gate to combine $F_{2,n-1}$ with the new term). Therefore, any valid construction must increase the gate count by at least two.

$$N(2,n) \geq N(2,n-1) + 2 \tag{1.10}$$

Applying our inductive hypothesis, $N(2,n-1) \geq 2(n-1) - 3 = 2n - 5$. Substituting this into our inequality:

$$N(2,n) \geq (2n-5) + 2 = 2n - 3 \tag{1.11}$$

This completes the inductive step. Thus, the complexity of any Boolean formula computing $F_{2,n}$ is at least $2n-3$.

Part c) Proof of NAND Complexity Upper Bound

We want to prove that for all $n \geq 3$, the minimum number of NAND gates required to compute $F_{2,n}$, denoted $N_{\text{NAND}}(2,n)$, satisfies the inequality $N_{\text{NAND}}(2,n) \leq N_{\text{NAND}}(2,n-1) + n + 6$.

We will provide a constructive proof. We can define the function $F_{2,n}$ recursively based on the number of inputs. Let's define two functions to be computed at each stage:

- $S_{1,n}(x_1, \ldots, x_n)$: The function is 1 if *at least one* input is 1 (this is OR_n).
- $S_{2,n}(x_1, \ldots, x_n)$: The function is 1 if *at least two* inputs are 1 (this is $F_{2,n}$).

Let $c_1 = S_{1,n-1}$ and $c_2 = S_{2,n-1}$ be the outputs from a circuit for the first $n-1$ inputs. We can compute the outputs for n inputs, $y_1 = S_{1,n}$ and $y_2 = S_{2,n}$, using the new input x_n as follows:

$$y_1 = c_1 \vee x_n \tag{1.12}$$
$$y_2 = c_2 \vee (c_1 \wedge x_n) \tag{1.13}$$

The second equation holds because for $S_{2,n}$ to be true, we need either at least two of the first $n-1$ inputs to be 1 (making $c_2 = 1$), or we need exactly one of the first $n-1$ inputs to be 1 (making $c_1 = 1, c_2 = 0$) and the new input x_n to be 1.

We can now design a recursive module that takes c_1, c_2, x_n as inputs and produces y_1, y_2 as outputs using only 2-input NAND gates. The cost of computing $F_{2,n}$, which is $S_{2,n}$, will be the cost of computing $F_{2,n-1}$ plus the number of gates in this module.

NAND Implementation of the Module: We need to implement $y_1 = c_1 \vee x_n$ and $y_2 = c_2 \vee (c_1 \wedge x_n)$.

1. **Compute y_1:** The expression $A \vee B$ can be implemented as $\text{NAND}(\text{NAND}(A,A), \text{NAND}(B,B))$, which requires 3 NAND gates. So, computing y_1 requires 3 gates.
2. **Compute y_2:**
   - First, compute the term $T = c_1 \wedge x_n$. This can be implemented as $\text{NAND}(\text{NAND}(c_1, x_n), \text{NAND}(c_1, x_n))$, which requires 2 NAND gates.
   - Next, compute $y_2 = c_2 \vee T$. As before, this OR operation requires 3 NAND gates.

The total number of gates to compute y_2, given c_1, c_2, x_n, is $2 + 3 = 5$ gates.

The total number of gates for the entire module (to compute both y_1 and y_2) is $3+5 = 8$ gates, as there are no shared gates in this straightforward implementation. This construction gives the recurrence relation:

$$N_{\text{NAND}}(2, n) \leq N_{\text{NAND}}(2, n-1) + 8 \tag{1.14}$$

We need to prove that $N_{\text{NAND}}(2, n) \leq N_{\text{NAND}}(2, n-1) + n + 6$. Our construction shows that the cost increases by a constant 8 gates at each step. We must check if this constant is bounded by $n + 6$:

$$8 \leq n + 6 \tag{1.15}$$

This inequality simplifies to $2 \leq n$. The problem statement specifies that the proof must hold for all $n \geq 3$. Since our condition $n \geq 2$ is met, our construction is valid and proves the given inequality.

□.

Problem C.1: Zipf's Meaning-Frequency Law

This section provides answers to the questions regarding the scientific article "A New Formulation of Zipf's Meaning-Frequency Law through Contextual Diversity" by Nagata and Tanaka-Ishii (2025).

a) Limitations of Dictionary-Based Studies

Traditional studies of Zipf's Meaning-Frequency Law rely on dictionaries to count the number of meanings for a word. The authors identify several key limitations with this approach:

- **Inconsistency and Ambiguity:** The number of meanings recorded for a word can vary significantly from one dictionary to another, making research findings dependent on the choice of lexicon.
- **Corpus Mismatch:** Not all meanings listed in a dictionary will appear in a given corpus. Conversely, a corpus may contain neologisms or domain-specific usages not found in a standard dictionary.
- **Limited Vocabulary:** To avoid ambiguity, previous studies often had to exclude function words, high-frequency words, and inflected forms, as these are poorly handled by sense inventories like WordNet. This means the law has only been verified for a limited subset of any language's vocabulary.
- **Difficulty of Scaling:** Manually annotating words with their senses in a corpus is a labor-intensive and difficult task, which makes it hard to analyze the law on the large-scale datasets required for statistical robustness.

b) The von Mises-Fisher Distribution and Contextual Diversity

The authors propose a new method to quantify the number of meanings by measuring the diversity of contexts in which a word appears. They model this using the von Mises-Fisher (vMF) distribution, which is a probability distribution for directional data (i.e., points on the surface of a hypersphere).

The vMF distribution is defined as $f(x;\mu,\kappa) \propto \exp\left(\kappa\mu^T x\right)$, where x is a contextualized word vector (normalized to be a unit vector), μ is the mean direction of all vectors for that word type, and κ is the *concentration parameter*.

- A high value of κ indicates that all contextual vectors for a word are tightly clustered around the mean direction μ. This implies the word is used in very similar contexts and thus has low meaning diversity.
- A low value of κ indicates that the vectors are widely dispersed, implying the word is used in many different contexts and has high meaning diversity.

The authors define *contextual diversity* v as the reciprocal of the concentration parameter:

$$v = 1/\kappa \tag{1.16}$$

This value, v, serves as their proxy for the number of meanings, m, allowing them to reformulate Zipf's law without a dictionary.

c) Rationale for Using the von Mises-Fisher Distribution

The paper's core assumption is that the variation in a word's meaning corresponds to the variation in the directions of its contextualized word vectors. The von Mises-Fisher distribution is the canonical and most natural probability distribution for modeling directional data on a hypersphere. Using a formal probability distribution provides a more principled and robust method for quantifying this variation compared to simpler ad-hoc measures.

While a measure like average pairwise cosine similarity could also capture the notion of vector spread, the vMF distribution provides a more complete model of the data. It estimates a single, interpretable parameter (κ) that represents the concentration around a central tendency (μ). This is conceptually cleaner and statistically more powerful than simply averaging similarity scores, which ignores the overall geometric structure of the vector distribution.

d) Autoregressive vs. Masked Language Models

The authors find a significant difference between Masked Language Models (MLMs) like BERT and autoregressive models like GPT-2 in their ability to capture the meaning-frequency law.

- **MLMs (BERT):** Even the moderately-sized `bert-base` model is able to observe the law, showing a clear positive correlation between frequency and contextual diversity.
- **Autoregressive Models (GPT-2):** These models struggle significantly. The `gpt2-medium` model shows no correlation at all. Only the much larger `gpt2-xl` model, which is approximately 14 times larger than `bert-base`, begins to show an observable, albeit weak, meaning-frequency relationship.

The authors conclude that autoregressive models require far more parameters to observe the law. They attribute this to the architectural difference: MLMs are bidirectional and use both the preceding and following context to generate a word's vector. In contrast, autoregressive models are unidirectional and can only use the preceding context. This bidirectional context gives MLMs a distinct advantage in distinguishing between nuances of meaning.

e) The Proposed Method as a Diagnostic Tool

The authors propose that their formulation of the meaning-frequency law can be used as a *diagnostic tool* or a "sanity check" for the lexical abilities of language models. If a model fails to show the expected positive correlation between word frequency and contextual diversity on a given dataset, it may indicate that the model is not capturing the semantic properties of that data effectively.

This claim is supported by their finding (Table 3) that there is a strong, statistically significant correlation between a model's performance on the meaning-frequency law (measured by the slope coefficient α) and its accuracy on a downstream task like Multi-Genre Natural Language Inference (MNLI). A model that better adheres to this fundamental linguistic law also appears to have a better general understanding of language.

f) Implications of the Law's Breakdown

The observation that the meaning-frequency law breaks down in certain scenarios provides insight into the limitations of current language models.

- **Small Models:** For smaller models, the law fails because they lack the capacity (i.e., enough parameters) to learn the meanings of all words in their training data. They tend to prioritize learning representations for high-frequency words to minimize overall loss, while failing to capture the diverse meanings of less frequent words.

- **Out-of-Domain Data:** When a model like BERT, trained on contemporary text, is applied to historical or learner corpora, the law becomes weaker. The model struggles to recognize the meanings of infrequent words in these unfamiliar contexts, assigning them vectors with artificially high diversity. This highlights that a model's lexical competence is highly dependent on its training data and does not always generalize well to new domains.

In both cases, the breakdown suggests that the model's ability to form nuanced, context-aware semantic representations is compromised, either by a lack of capacity or a lack of relevant training experience.

(Bonus) Factors Influencing Polysemy

While the article focuses on the correlation between frequency and meaning, it does not detail the causal factors. Based on principles of historical linguistics and language evolution, we can identify several factors.

Factors leading to more meanings (higher polysemy):

- **Linguistic Economy:** It is more efficient for speakers to extend the meaning of a common, existing word to a new concept (e.g., the verb "to run" being applied to engines, noses, and campaigns) than to coin a new word. This is a manifestation of Zipf's *Principle of Least Effort*.
- **Semantic Change Over Time:** The most frequent words are often the oldest in a language. Over centuries of use, they have had more opportunities to undergo semantic shifts, broadening, narrowing, or developing metaphorical meanings.
- **Abstraction:** Frequent words often represent fundamental or abstract concepts (e.g., "to be", "to have", "thing"), which can be applied in an extremely wide range of contexts, each subtly different.

Factors leading to fewer meanings (lower polysemy):

- **Domain Specificity:** Words created for a specific technical, scientific, or professional field (e.g., "hemoglobin", "stochastic", "ionosphere") are designed to have precise, unambiguous meanings to avoid confusion.
- **Recency:** Neologisms, or newly coined words, have not been in the language long enough to acquire new senses.
- **Low Frequency:** Infrequent words are used in a limited set of contexts, providing fewer opportunities for their meanings to be metaphorically or functionally extended.

Problem C.2: Self-Improvement Capabilities of LLMs

This section provides answers to the questions regarding the scientific article "Mind the Gap: Examining the Self-Improvement Capabilities of Large Language Models" by Song et al. (2025).

a) Self-Improvement Framework and Key Assumption

In the authors' framework, self-improvement is a process where a Large Language Model (LLM) enhances its own capabilities without external data or human feedback. The process consists of three main steps:

1. **Generation:** The model produces multiple, diverse responses to a given prompt.
2. **Verification:** The same model is then used to evaluate and score its own generated responses.
3. **Update:** The model is fine-tuned on a filtered or re-weighted dataset of its own generations, prioritizing the high-scoring responses.

The key assumption that underpins this entire framework is that *verification is easier than generation*. The authors hypothesize that a model can be a better-than-random judge of its own outputs, even if those outputs are not perfect. This allows the model to identify its more correct or higher-quality responses, creating a useful signal to guide its own learning and subsequent improvement.

b) The Generation-Verification Gap (GV-Gap)

The Generation-Verification Gap (GV-Gap) is the central metric proposed by the authors to measure a model's potential for self-improvement. It is formally defined as the difference between the performance of a re-weighted distribution of the model's outputs and the performance of its original, unfiltered outputs.

$$\text{gap}(f, g) := J(f[w(\hat{u}_g)]) - J(f) \tag{1.17}$$

where f is the generator model, g is the verifier, $J(f)$ is the expected utility (performance) of the original model, and $J(f[w(\hat{u}_g)])$ is the expected utility after re-weighting the outputs based on the verifier's scores. When the generator and verifier are the same model, this is the *self-improvement GV-Gap*.

The GV-Gap is a better metric than simply measuring performance differences after a model update because it isolates the *potential* for improvement that exists within the model's own generations, before the update step. Post-update performance can be influenced by confounding factors, such as the model learning to mimic the format of the desired answers rather than improving its underlying reasoning. The GV-Gap, by contrast, provides a direct, quantitative measure of the quality of the verification signal itself.

c) Greedy Decoding and its Incompatibility with Self-Improvement

Greedy decoding is a deterministic text generation method where the model, at each step, selects the single most probable next token. This results in the generation of only one, single output for any given prompt.

Self-improvement, as defined in the paper, is impossible with greedy decoding because it critically relies on the model's ability to produce a *diverse set of candidate responses*. The self-improvement

process requires the model to generate multiple different outputs, explore various reasoning paths, and then use its verification capability to identify the best one. Since greedy decoding produces only a single, fixed output, there is no variability and no set of options to verify or select from. This lack of diversity means there is no "improvable generation," and thus no opportunity for the verification and update steps to take place.

d) Scaling of the Relative GV-Gap

The authors observe that the *relative* GV-Gap scales monotonically (and often linearly on a log-scale) with the amount of pre-training FLOPs (Floating Point Operations, a proxy for model size and training compute) for certain verification methods, but not for others.

This phenomenon is most clearly observed with stable and robust verification methods, such as Chain-of-Thought Scoring (CoT-S). The reason for this is that these methods provide a reliable and consistent signal of correctness that improves as the model's own capabilities grow. Larger models are better at both generating correct answers and, crucially, at verifying them. This creates a positive feedback loop where the quality of the verification signal (and thus the relative GV-Gap) scales with the model's size.

In contrast, less stable methods like Multiple Choice (MC) verification can be unreliable, especially for smaller models. They may produce noisy or even misleading signals, which breaks the consistent scaling trend. The monotonic scaling is therefore an indicator of a high-quality, reliable verification mechanism.

e) Failure of Self-Improvement on Sudoku Puzzles

Despite the fact that solving a Sudoku puzzle is computationally hard (NP-hard for general cases) while verifying a solution is easy (in P), most models fail to self-improve on this task. Only the very largest models tested show a significant positive GV-Gap.

The authors hypothesize that this is because, while verification is *theoretically* simple, it still requires a certain level of abstract reasoning and planning capabilities. A model needs to be able to understand and apply the rules of Sudoku systematically to verify a solution. Most models, especially smaller ones, have likely not been exposed to enough Sudoku-like reasoning tasks during pre-training to develop these specific capabilities. Therefore, they lack the requisite skills to perform even the "easy" task of verification effectively, causing the self-improvement process to fail. This suggests that a model's inherent, pre-trained capabilities are a crucial prerequisite for self-improvement on any given task.

f) Proposed Task Domain for Self-Improvement

A task domain where one would expect self-improvement to be highly effective is **code generation with unit tests**.

This domain perfectly aligns with the core assumption that verification is easier than generation.

- **Generation is Hard:** Writing functionally correct code that solves a complex problem is a difficult task. There are many possible implementations, and a model may produce code that has subtle bugs or logic errors.
- **Verification is Easy and Unambiguous:** A set of pre-defined unit tests can serve as an objective and automated verifier. The model does not need to "reason" about whether the code is correct; it simply needs to execute the code against the tests. The verification signal is a clear, binary pass/fail.

In this scenario, a model could generate numerous different code solutions for a given problem. It could then execute each solution against the provided unit tests. The solutions that pass all the tests would be identified as correct. The model could then be fine-tuned on this verified, high-quality code, leading to a significant improvement in its coding abilities.