Bayesian Mastery Estimation Algorithm for OGE Exam Preparation

July 28, 2025

This document describes a Bayesian algorithm designed to estimate a student's mastery probabilities for skills, topics, and problem number types in the context of preparing for the OGE (Basic State Exam) math exam. The algorithm leverages performance data from a database of math problems, incorporating Bayesian updating, difficulty weighting, time decay, and sequential analysis techniques to provide accurate and dynamic mastery estimates.

1 Overview

The OGE exam consists of 25 problems, each associated with specific topics (e.g., problem 11: correspondence between algebraic formulas and graphs, problem 10: simple probability). Problems are characterized by:

- A list of required skills (e.g., [38, 95, 180, 5, 67]).
- A list of topics (broader categories encompassing skills).
- A problem number type from the set $\{-1, 1, 6, 7, 8, 9, 10,$ $11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25\}$, where -1 represents a unique problem type.

Student performance is recorded in a PostgreSQL `activity` table, capturing attempt details such as correctness, duration, and scores. The algorithm uses this data to model mastery probabilities for skills, topics, and problem types, incorporating sequential analysis to detect performance trends and adjust estimates dynamically.

2 Algorithm Description

2.1 Modeling Skill Mastery

Skills are the foundational units of the OGE syllabus, with each problem requiring a subset of skills. Mastery probabilities are estimated using a Bayesian framework.

2.1.1 Representation

- For each skill i, the mastery probability p_i is modeled as a random variable with a Beta distribution: $p_i \sim \text{Beta}(\alpha_i, \beta_i)$.
  - α_i: Represents "successes" (evidence of mastery).
  - β_i: Represents "failures" (evidence of non-mastery).

Why Beta distribution? The Beta distribution is used in the Bayesian mastery estimation algorithm for the following reasons:

1. Models Probabilities: Naturally represents probabilities (values between 0 and 1), ideal for estimating mastery likelihoods.
2. Conjugate Prior: Simplifies Bayesian updates, as the Beta distribution is a conjugate prior for the Bernoulli/binomial likelihood, making calculations straightforward.
3. Flexibility: Captures a wide range of uncertainty levels (e.g., from uniform to highly confident) via parameters α and β.
4. Interpretable Parameters: α and β directly represent "successes" (evidence of mastery) and "failures" (evidence of non-mastery).
5. Uncertainty Quantification: Provides variance and confidence intervals, enabling robust mastery thresholds (e.g., $P(p_i > 0.7) > 0.95$).
6. Incremental Updates: Supports efficient, real-time updates by incrementing α or β based on new performance data.

2.1.2 Beta distribution basic facts

- Definition: A continuous probability distribution defined on [0, 1], ideal for modeling probabilities or proportions.
- Parameters: α (successes) and β (failures), both positive, shape the distribution.
- Probability Density Function:

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}, \quad 0 \le x \le 1$$

  where $B(\alpha,\beta)$ is the beta function.
- Mean:

$$\mu = \frac{\alpha}{\alpha+\beta}$$

- Variance:

$$\text{Var} = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$$

- Conjugate Prior: Conjugate to the Bernoulli/binomial distribution, simplifying Bayesian updates.
- Shapes: Flexible shapes (e.g., uniform for $\alpha = \beta = 1$, U-shaped, skewed) based on α and β.
- Applications: Used in Bayesian inference, A/B testing, and modeling uncertainty in proportions (e.g., skill mastery).

2.1.3 Initialization

- Each student passes a diagnostic test upon registration. As a result, each skill is classified as weak or non-weak. Beta parameters priors are assigned as follows:

```
def initialize_skill_prior(is_weak: bool) -> tuple[float, float]:
    return (1, 40) if is_weak else (5, 45)
```

- Weak skills: Initial mastery probability ≈ 0.02.
- Non-weak skills: Initial mastery probability $= 0.1$.
- A note: for each skill i there is a pair of (α_i, β_i), for each topic t there is a pair of (α_t, β_t), for each problem type number T there is a pair of (α_T, β_T). These data should be written to PostgreSQL table called student_mastery for each `user_id`. And this data will be constantly updated after each problem attempted.

2.1.4 Update Rule

- For each attempt in the `activity` table:
  - Query the `skills` list for the problem (via `question_id`).
  - If `finished_or_not = True` and `is_correct = True`, compute current mastery probability $p_i = \frac{\alpha_i}{\alpha_i+\beta_i}$, then determine the velocity factor f based on p_i: If $p_i < 0.2$: $f = 2$, Else if $p_i < 0.5$: $f = 0.725$, Else if $p_i < 0.7$: $f = 0.6667$, Else: $f = 0.5$. Then increment α_i by $f \cdot w(d)$ for each skill i in the problem's skill list (credit for success).
  - If `finished_or_not = True` and `is_correct = False`, increment β_i by $w(d)$ for each skill i in the skill list (penalty for failure).
  - If `finished_or_not = False` (problem skipped):
    - If difficulty ≥ 3, increment β_i by $w(d)$ for each skill i (penalty for failure).
    - If difficulty ≤ 2, increment β_i by $0.5 \cdot w(d)$ for each skill i.
- Difficulty Weighting ($w(d)$): Choose between:
  - Linear Scaling: $w(d) = d$. Example: A difficulty-5 problem contributes $5\times$ more than a difficulty-1 problem.
  - Exponential Scaling: $w(d) = 2^{d-1}$. Prioritizes mastery of advanced problem types.

  Goal: Harder problems provide stronger evidence of mastery (correct answers) or gaps (incorrect answers), preventing easy problems from overinflating skill probabilities.
- Sensitivity Tests:
  - After 3 failures on a difficulty-5 problem, no skill's mastery probability should drop by more than 0.3.
  - Solving 5 difficulty-1 problems correctly should increase mastery by less than 0.2.

2.1.5 Mastery Estimate

- The probability of mastering skill i is:

$$p_i = \frac{\alpha_i}{\alpha_i + \beta_i}$$

- To account for uncertainty (e.g., Beta(2,18) and Beta(20,180) both have mean 0.1 but different variances), use a probability threshold:
  - Mastery is achieved if $P(p_i > 0.7) > 0.95$.

2.1.6 Dependency Graph Propagation

- If a student struggles with a skill (e.g., skill 2: "Scientific notation", with dependencies [1, 39]), penalize both the skill and its prerequisites more heavily.
- Implement a validated cognitive model:
  - Use Bayesian Knowledge Tracing (BKT) weights.
  - Precompute prerequisite weights via expert tagging.
  - Example: Penalty = $\frac{1}{1+\text{graph\_distance}}$.

2.1.7 Difficulty-Weighted Bayesian Updates

- Adjust updates based on problem difficulty $d \in \{1, 2, 3, 4, 5\}$:

$$\alpha_i += w(d) \cdot \text{correct}, \quad \beta_i += w(d) \cdot \text{incorrect}$$

- Goal: Ensure harder problems provide stronger evidence, maintaining balanced skill probability estimates.

2.2 Modeling Topic Mastery

Topics are broader categories encompassing multiple skills. Topic mastery is derived from skill mastery, reflecting the hierarchical syllabus structure.

2.2.1 Representation

- For each topic t, the mastery probability p_t is a function of the mastery probabilities of its associated skills.

2.2.2 Skill-to-Topic Mapping

- A predefined mapping links skills to topics (e.g., skills [38, 95] belong to "Linear Equations"). This can be stored in a database table or hardcoded.

2.2.3 Aggregation Method

- Compute p_t as the average mastery probability of the skills S_t associated with topic t:

$$p_t = \frac{1}{|S_t|} \sum_{i \in S_t} p_i = \frac{1}{|S_t|} \sum_{i \in S_t} \frac{\alpha_i}{\alpha_i + \beta_i}$$

- Alternative Options:
  - Minimum: $p_t = \min_{i \in S_t} p_i$ (mastery requires all skills).
  - Product: $p_t = \prod_{i \in S_t} p_i$ (assuming independence and conjunctive mastery).
- The average is recommended for simplicity and robustness, but experimentation can determine the strictness of topic mastery.

2.2.4 Update

- Recalculate p_t whenever skill mastery probabilities are updated.

2.3 Modeling Problem Number Type Mastery

Problem number types (e.g., 11 for algebraic formulas and graphs) are tied to specific topics and skills but are modeled independently to assess type-specific proficiency.

2.3.1 Representation

- For each problem number type $T \in \{-1, 1, 6, \ldots, 25\}$, model the probability of correctly solving a problem of that type, p_T, as Beta(α_T, β_T).

2.3.2 Initialization

- Use a less harsh prior (e.g., Beta$(1, 25)$) to avoid overly pessimistic initial estimates.
- For problem types with higher average difficulty, initialize with a lower prior (e.g., Beta$(1, 40)$ for difficulty-4 and difficulty-5 types).

2.3.3 Update Rule

- For each attempt where `problem_number_type = T` and difficulty $d \in \{1, 2, 3, 4, 5\}$:
  - If `finished_or_not = True` and `is_correct = True`, compute current mastery probability $p_i = \frac{\alpha_T}{\alpha_T + \beta_T}$, then determine the velocity factor f based on p_T: If $p_T < 0.2$: $f = 2$, Else if $p_T < 0.5$: $f = 0.725$, Else if $p_T < 0.7$: $f = 0.6667$, Else: $f = 0.5$. Then increment α_T by $f \cdot w(d)$ for each problem type T.
  - If `finished_or_not = True` and `is_correct = False`, increment β_T by $w(d)$.
  - If `finished_or_not = False`, treat as incorrect with a smaller penalty, e.g., increment β_T by $0.5 \cdot w(d)$.
  - If `finished_or_not = False` (problem skipped):
    - If difficulty ≥ 3, increment β_T by $w(d)$ for each problem type T (penalty for failure).
    - If difficulty ≤ 2, increment β_T by $0.5 \cdot w(d)$ for each problem type T.
- Difficulty Weighting $(w(d))$ should be one of:
  - Linear Scaling: $w(d) = d$.
  - Exponential Scaling: $w(d) = 2^{d-1}$.
- Rationale: Directly reflects the student's performance on each problem type, critical for OGE exam preparation where each number has a fixed structure.

2.3.4 Mastery Estimate

- The probability of mastering problem type T is:

$$p_T = \frac{\alpha_T}{\alpha_T + \beta_T}$$

2.4 Change Detection with CUSUM

2.4.1 Description

- Use the Cumulative Sum (CUSUM) method to detect significant shifts in performance, such as when a student masters a skill or experiences a sudden drop in proficiency.

2.4.2 Implementation

- Track a cumulative sum for each skill i or problem type T:

$$S_n = \max(0, S_{n-1} + (x_n - k))$$

  where:

  - $x_n = 1$ if the answer is correct, 0 if incorrect,
  - k is a reference value (e.g., 0.5, representing expected performance),
  - $S_0 = 0$.

- Set upper and lower thresholds (e.g., $h = 3$):
  - If $S_n > h$, increase confidence in mastery (e.g., boost α_i or α_T by an additional amount, like 0.5).
  - If S_n drops below a negative threshold (e.g., $-h$), flag a potential decline (e.g., increase β_i or β_T).
- Apply to skills and problem types directly; for topics, aggregate CUSUM signals from related skills.

2.4.3 Benefit

- Quickly identifies turning points in student performance, allowing the model to adjust mastery estimates proactively.

2.5 Sequential Probability Ratio Test (SPRT)

2.5.1 Description

- Implement SPRT to formally test whether a student has mastered a skill, topic, or problem type, using accumulating evidence to make decisions efficiently. This determines the final mastery levels displayed in progress bars.

2.5.2 Implementation

- Incorporate a time decay factor to prioritize recent performance, reflecting the student's current mastery level. Older attempts are less relevant.
- For skill i (or problem type T):
  - Query `answer_time_start` for the most recent problem involving skill i (or type T), denoted t_{attempt}.
  - Calculate the current timestamp, t_{current}.
  - Compute the time decay function:

$$w(t) = \exp(-\lambda \cdot (t_{\text{current}} - t_{\text{attempt}}))$$

    where λ is a decay rate (e.g., 0.01 per day, tunable based on data).

  - Query α and β for the skill or type, then multiply by $w(t)$ to obtain the most relevant Beta distribution parameters for display.
- Define hypotheses:

- H_0: Student has not mastered the skill ($p_i < 0.7$).
- H_1: Student has mastered the skill ($p_i \geq 0.7$).

- For the most recent attempt, compute the likelihood ratio:

$$\Lambda_n = \prod_{j=1}^{n} \frac{P(x_j|H_1)}{P(x_j|H_0)}$$

where x_j is the outcome (correct/incorrect).

- Use the current Beta distribution parameters to estimate $P(x_j|H_1)$ and $P(x_j|H_0)$.

- Set decision thresholds A (e.g., 0.05) and B (e.g., 20):
  - If $\Lambda_n \leq A$, accept H_0 (not mastered).
  - If $\Lambda_n \geq B$, accept H_1 (mastered).
  - Otherwise, continue collecting data.
- Apply to skills and problem types; for topics, test mastery based on aggregated skill probabilities.
- Result: A conclusion for each skill, topic, and problem type (mastered or not) for display in progress bars.

2.5.3 Benefit

- Provides a rigorous, evidence-based method to confirm mastery, reducing uncertainty in estimates.

2.6 Incorporating Additional Data (Optional Enhancements)

2.6.1 Duration (`duration_answer`)

- Longer times may indicate struggle, even if correct. Weight the α_i or α_T increment (e.g., +0.5 instead of +1) if `duration_answer` exceeds a threshold (e.g., median time for that problem).
- Threshold Definition: If insufficient data exists to calculate the median time for a problem, use a default threshold (e.g., 300 seconds) or estimate based on similar problems (e.g., same difficulty or problem type).

2.6.2 Scores (`scores_fipi`)

- For problem types 20–25 with scores of 0, 1, or 2:
  - Score = 0: Treat as incorrect ($\beta + 1$).
  - Score = 1: Treat as partial mastery ($\alpha + 0.5$).
  - Score = 2: Treat as full mastery ($\alpha + 1$).

3 Implementation Notes

- Database: Assumes a PostgreSQL `activity` table with fields for `attempt_id`, `user_id`, `question_id`, `answer_time_start`, `finished_or_not`, `is_correct`, `duration_answer`, `scores_fipi`, `skills`, `topics`, and `problem_number_type`. Note also that we should store pairs (alpha, beta) for each skill, topic, problem number type in the table `student_mastery`.

- Storage: Store α, β, and timestamps in a `student_mastery` table for efficient updates and retrieval.
- Tuning: Adjust parameters like λ, $w(d)$, and thresholds based on empirical data to meet sensitivity requirements.
- Scalability: Use caching and periodic updates to handle time decay efficiently, avoiding recalculation of $w(t)$ for every display.

4 Storage and Retrieval of Mastery Weights

We use a single `student_mastery` table with columns:

- `user_id` (BIGINT): Identifies the user.
- `entity_type` (VARCHAR): Distinguishes between 'skill' and 'problem_type'.
- `entity_id` (INTEGER): Unique identifier for the skill or problem type.
- `alpha` (FLOAT): Alpha parameter of the Beta distribution.
- `beta` (FLOAT): Beta parameter of the Beta distribution.
- Primary Key: Composite key `(user_id, entity_type, entity_id)` to ensure uniqueness.

Note: Topics: Since p_t is computed as the average of p_i for associated skills, we don't store (α_t, β_t) in the table. Instead, we calculate it dynamically when needed. Computing p_t on the fly avoids redundant storage and keeps the table lean, though it requires an efficient `get_skills_for_topic` implementation. IMPORTANT: We need to write a function `get_skills_for_topic(` that returns a list of skill numbers that correspond to the given topic.

4.1 SQL Code to Create Table

```sql
CREATE TABLE IF NOT EXISTS student_mastery (
    user_id BIGINT,
    entity_type VARCHAR(20),
    entity_id INTEGER,
    alpha FLOAT,
    beta FLOAT,
    PRIMARY KEY (user_id, entity_type, entity_id)
);
```

4.2 Examples of How to Retrieve Weights from student_mastery Table

```python
import psycopg2

def get_alpha_beta(conn, user_id, entity_type, entity_id):
    """Generic function to retrieve alpha and beta for a given entity."""
    with conn.cursor() as cur:
        cur.execute("""
            SELECT alpha, beta FROM student_mastery
            WHERE user_id = %s AND entity_type = %s AND entity_id = %s
        """, (user_id, entity_type, entity_id))
        result = cur.fetchone()
        return result if result else (None, None)

def get_skill_alpha_beta(conn, user_id, skill_id):
```

```
    """Retrieve (alpha_i, beta_i) for a specific skill."""
    return get_alpha_beta(conn, user_id, 'skill', skill_id)

def get_problem_type_alpha_beta(conn, user_id, problem_type):
    """Retrieve (alpha_T, beta_T) for a specific problem type."""
    return get_alpha_beta(conn, user_id, 'problem_type', problem_type)

def get_topic_mastery(conn, user_id, topic_id, get_skills_for_topic):
    """Compute p_t for a topic as the average p_i of associated skills."""
    skill_ids = get_skills_for_topic(topic_id)  # Assume this function returns skill IDs
    p_i_list = []
    for skill_id in skill_ids:
        alpha, beta = get_skill_alpha_beta(conn, user_id, skill_id)
        if alpha is not None and beta is not None:
            p_i = alpha / (alpha + beta)  # Mastery probability for the skill
            p_i_list.append(p_i)
    return sum(p_i_list) / len(p_i_list) if p_i_list else None

# Example usage:
# conn = psycopg2.connect(...)  # Your connection setup
# alpha_i, beta_i = get_skill_alpha_beta(conn, 1, 38)
# alpha_T, beta_T = get_problem_type_alpha_beta(conn, 1, 11)
# p_t = get_topic_mastery(conn, 1, 2, lambda tid: [38, 39])  # Example skill mapping
```

4.3 Examples of How to Set and Update Weights from student_mastery Table

```
import psycopg2

def set_alpha_beta(conn, user_id, entity_type, entity_id, alpha, beta):
    """Write or update alpha and beta for a given entity."""
    with conn.cursor() as cur:
        cur.execute("""
            INSERT INTO student_mastery (user_id, entity_type, entity_id, alpha, beta)
            VALUES (%s, %s, %s, %s, %s)
            ON CONFLICT (user_id, entity_type, entity_id)
            DO UPDATE SET alpha = EXCLUDED.alpha, beta = EXCLUDED.beta
        """, (user_id, entity_type, entity_id, alpha, beta))
    conn.commit()

def set_skill_alpha_beta(conn, user_id, skill_id, alpha, beta):
    """Write (alpha_i, beta_i) for a specific skill."""
    set_alpha_beta(conn, user_id, 'skill', skill_id, alpha, beta)

def set_problem_type_alpha_beta(conn, user_id, problem_type, alpha, beta):
    """Write (alpha_T, beta_T) for a specific problem type."""
    set_alpha_beta(conn, user_id, 'problem_type', problem_type, alpha, beta)

# Example usage:
# conn = psycopg2.connect(...)  # Your connection setup
# set_skill_alpha_beta(conn, 1, 38, 5.0, 45.0)  # Set skill 38 for user 1
# set_problem_type_alpha_beta(conn, 1, 11, 1.0, 25.0)  # Set problem type 11 for user 1
```

Note: The `INSERT ... ON CONFLICT` syntax (upsert) ensures that if the record exists, it updates the values; otherwise, it inserts a new record.

5 Conclusion

This algorithm provides a robust, probabilistic framework for tracking student progress in OGE exam preparation, balancing accuracy, responsiveness, and computational efficiency.