AutoJudge

A Machine Learning Approach for Competitive Programming

by: Umar Faruqe
MNC (2nd Year)

Contents

1 Problem Statement

In the domain of competitive programming, assessing the difficulty of a problem is subjective and time-consuming for human setters. Incorrect difficulty labels can lead to unbalanced contests and frustrated participants, or mostly us competitive programmers strive for days till codeforces releases official ratings of the problems, while submission count is a good metric but for OJs which are not popular this metric can give false results. So our project aims to automate this process by developing a machine learning system that predicts:

1. **Difficulty Label (Classification):** Categorizing problems into Easy, Medium, or Hard.
2. **Difficulty Score (Regression):** Predicting a precise numerical score (800–3500 rating scale normalized to 0-100) based on the problem text.

2 Dataset Description

The model was trained on the **TaskComplexity** dataset [**?**], a comprehensive collection of programming problems designed for complexity analysis.

2.1 Data Source

We utilized the `TaskComplexityEval-24` repository hosted on GitHub. The dataset consists of **4,112** unique programming tasks collected via web scraping from various online coding platforms. Each entry in the dataset includes:

- **Problem Description:** The full narrative text of the coding task.
- **Input/Output Specifications:** Detailed constraints and formatting rules.
- **Complexity Label:** Ground truth classification (Easy, Medium, Hard).
- **Complexity Score:** A numerical rating indicating the precise difficulty level (1-10). Although for our project to have a better understanding it was scaled to (1-100)

The data is publicly available at:
`https://github.com/AREEG94FAHAD/TaskComplexityEval-24`

3 Data Preprocessing

Raw text data contains noise that hinders model performance. The following preprocessing pipeline was implemented using Python (NLTK/Scikit-learn):

1. **Text Cleaning:**
    - Lowercasing all text (to treat subarray Subarray as same)
    - Removing stopwords (e.g., "the", "is", "at") which donot contribute anything to determine problem's difficulty
    - Removing special characters and punctuation.

2. **Vectorization (TF-IDF):** We used Term Frequency-Inverse Document Frequency (TF-IDF) to convert text into numerical vectors.
    - **Max Features:** Limited to the top 3,000 most important words to reduce dimensionality.
    - **N-grams:** Unigrams and Bigrams were considered to capture context (e.g., "shortest path", "expected value", "convex hull").

4 Feature Engineering

To capture the complexity of competitive programming problems, we implemented a hybrid feature extraction pipeline combining domain-specific manual features with statistical text analysis.

4.1 Manual Feature Extraction

We extracted **7 numerical features** designed to detect specific complexity indicators in the problem statement.

1. **Hard Topic Count:** We defined a dictionary of advanced algorithmic topics and counted their occurrences in the text. The specific topics tracked are:
    - *convex hull, mobius, segment tree, flow, centroid, geometry, gcd, mex, dynamic programming, modulo, bitwise, graph, expected value, permutations, xor, shortest path, grid, query, range query.*
2. **Math Density:** Calculated as the ratio of mathematical symbols to the total text length. The symbol set includes: $\{\$, \hat{}, \{, \}, \_, n, =, <, >, *\}$.
3. **High Constraints Check:** A binary feature detecting the presence of large constraint numbers using the regex pattern: $10^5|10^9|1000000007|1e9|1e5$.
4. **Statement Length Features:**
    - `text_len`: The raw character count of the problem.
    - `is_short_statement`: Binary flag if length < 300 characters.

4.2 Statistical Feature Engineering

Beyond standard TF-IDF, we developed a probability-based approach to filter noise and amplify signal words.

4.2.1 Class-wise Probability & Noise Removal

We calculated the probability of every word appearing in each difficulty class ($P(w|C_{easy})$, $P(w|C_{medium})$,

- **Noise Detection:** Words were classified as "Noise" if they appeared in more than 25% of problems across **all three** classes ($P > 0.25$).
- **Extending the stopwords:** These statistically noisy words were merged with a manual domain stopword list (e.g., *"input", "output", "print", "test case"*) and standard English stopwords to form the `final_stopwords` list.

4.2.2 Signal Count Features

We identified "Signal Words" that strongly differentiate between difficulty levels.

- **Hard Signals:** Words where $P(w|Hard) > 2 \times P(w|Medium)$.
- **Medium Signals:** Words where $P(w|Medium) > 2 \times P(w|Hard)$.
- **Impact of this feature:** While weighting the `medium_signal_count` by a factor of 5 may initially appear heuristic, it proved critical for addressing the class overlap between *Medium* and *Hard* problems. Since text-based features alone often struggle to distinguish these adjacent categories, the model initially misclassified approximately 300 Medium problems as Hard. This weighted feature significantly improved separability, reducing false positives in the Hard class to roughly 220–230 samples. Although some overlap persists due to the subjective nature of difficulty, this reduction represents a measurable improvement in classification precision.

We selected the top 30 words from each list and created two new features:

- `high_difficulty_signal_count`: Count of hard signal words in the text.
- `medium_signal_count`: Count of medium signal words (Weighted $\times 5$ to increase importance).

4.3 Final Feature Matrix

The final feature set was constructed by stacking:

1. **TF-IDF Vectors:** Top 3,000 features (unigrams and bigrams) using the custom stopword list.
2. **Manual Features:** The 7 extracted metrics: [`hard_topic_count`, `math_density`, `text_len`, `high_difficulty_signal_count`, `is_short_statement`, `has_high_constraints`, `medium_signal_count`].

This resulted in a final sparse matrix X_f of shape $(N, 3007)$.

4.4 Feature Stacking

For the regression model, we used 'SelectKBest' to reduce the TF-IDF vectors to the top 800 features and stacked them with the 7 manual features, resulting in an **807-dimensional input vector**.

5 Models and Experimental Setup

5.1 Classification Experiments

We explored two distinct strategies for hyperparameter optimization to improve the Random Forest Classifier.

5.1.1 Strategy 1: Automated Grid Search

We initially performed a `GridSearchCV` with Stratified K-Fold validation to automatically select the best hyperparameters.

- **Result:** The best automated model achieved an accuracy of **0.50**.
- **Limitation:** The model exhibited severe bias towards the majority class ("Hard"), yielding a very poor recall of 0.15 for the "Easy" class.

5.1.2 Strategy 2: Manual Tuning (Selected)

We shifted to a manual tuning approach, explicitly setting `n_estimators=500` and using `class_weight='balanced_subsample'` to penalize misclassification of minority classes.

- **Result:** This approach improved accuracy to **0.53**.
- **Key Improvement:** The recall for "Easy" problems improved significantly from 0.15 to **0.35**, indicating the model was no longer just guessing the most frequent class.

Metric	Grid Search (Baseline)	Manual Tuning (Final)
Accuracy	0.50	**0.53**
Easy Class Recall	0.15	**0.35**
Hard Class Precision	0.53	**0.58**

Table 1: Performance Comparison: Automated Grid Search vs. Manual Tuning

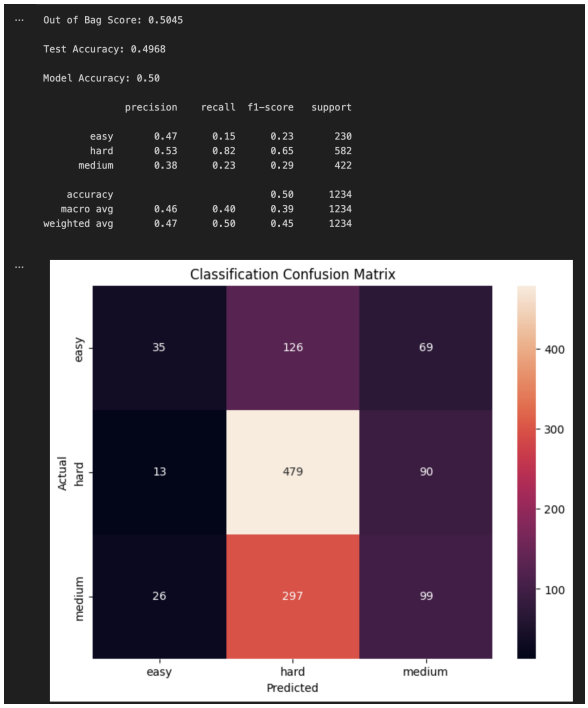


Figure 1: Grid Search Baseline (Acc: 0.50)

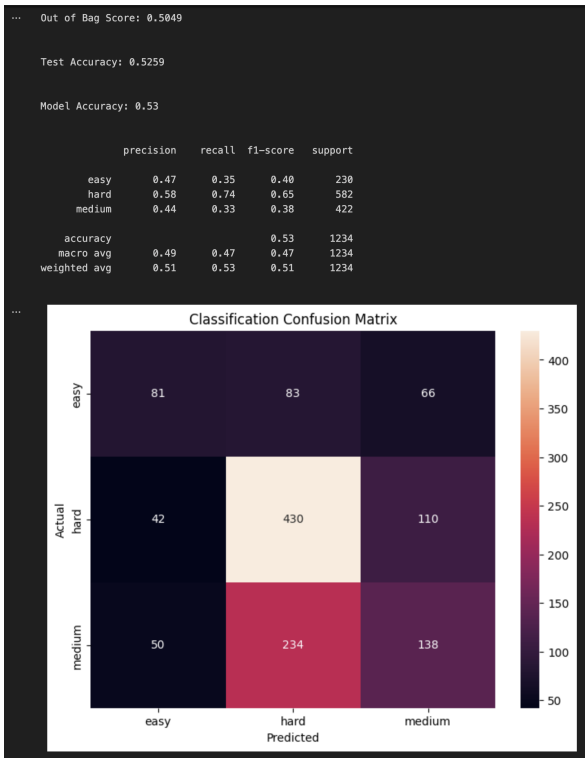


Figure 2: Manual Tuning Final (Acc: 0.53)

Figure 3: Confusion Matrix Comparison. Note the improvement in the 'Easy' class prediction in the Final Model (Right).

5.2 Feature Importance Analysis

To interpret the model's decision-making process, we extracted the feature importance scores from the trained Random Forest model.

The results, shown in Table 2, validate our feature engineering strategy. The top 5 most influential features were all **manually engineered domain features**, significantly outperforming raw TF-IDF keywords.

Feature Name	Importance Score
`text_len`	0.0442
`high_difficulty_signal_count`	0.0220
`math_density`	0.0215
`hard_topic_count`	0.0090
`medium_signal_count`	0.0075
`"100"` (TF-IDF Token)	0.0071
`"minimum"` (TF-IDF Token)	0.0069
`"characters"` (TF-IDF Token)	0.0064
`"different"` (TF-IDF Token)	0.0050
`"separated"` (TF-IDF Token)	0.0049

Table 2: Top 10 Most Important Features used by the Model

Key Observation: The fact that `text_len` (0.0442) and `math_density` (0.0215) are

among the top predictors confirms that problem difficulty is strongly correlated with the length of the problem statement and the density of mathematical notation, more so than any single specific keyword.

5.3 Regression Model

The goal of the regression task was to predict the exact difficulty score (normalized 0–100). We iterated through three distinct approaches to minimize the Root Mean Squared Error (RMSE).

5.3.1 Strategy 1: Random Forest + Grid Search (Baseline)

Our initial approach used a pipeline combining **Truncated SVD** (for dimensionality reduction) and a **Random Forest Regressor**.

- **Method:** We used `GridSearchCV` to optimize ‘$\mathrm{n}_e stimators$‘$(200, 500)$$and$‘$max_d epth$‘.

- **Outcome:** The model performed poorly with high error variance. SVD resulted in information loss, and the Random Forest struggled to capture the subtle linear relationships in the text data. This model was discarded.

5.3.2 Strategy 2: Standard LightGBM

We switched to **LightGBM**, a gradient boosting framework, due to its ability to handle sparse data without SVD.

- **Setup:** We used ‘SelectKBest‘ ($k = 1000$) to select the top textual features and applied a **Log-Transformation** (‘np.log1p‘) to the target variable to handle skewness.

- **Outcome:** Performance improved over the baseline, but the model still treated the problem purely as a ”bag of words,” ignoring structural complexity.

5.3.3 Strategy 3: LightGBM + Optuna + Feature Stacking (Final)

Our final model incorporated domain knowledge and automated hyperparameter tuning.

1. **Feature Stacking:** Instead of using only text, we explicitly stacked the top **800 TF-IDF features** with our **7 manual domain features** (e.g., `math_density`, `hard_topic_count`), creating a rich hybrid input vector.

2. **Optuna Optimization:** We used the Optuna framework to perform 30 trials of Bayesian optimization, tuning parameters such as ‘learning rate‘, ‘num leaves‘, and ‘lambda l1‘ regularization.

3. **Outcome:** This model achieved the lowest RMSE and the highest R^2 score, proving that combining domain features with gradient boosting yields the best results.

5.4 Regression Results

Table 3 summarizes the improvement across experiments. The integration of Optuna and feature stacking reduced the error significantly.

Model	RMSE	MAE	R^2 Score
Random Forest (Grid Search)	20.05	16.99	0.1310
Standard LightGBM	20.15	16.55	0.1227
Optuna LightGBM (Final)	**19.66**	**16.34**	**0.1641**

Table 3: Comparison of Regression Model Performance

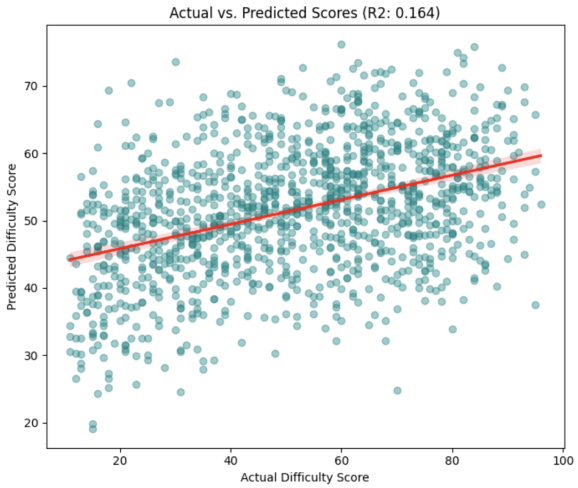


Figure 6: Actual vs Predicted Scores. The linear trend indicates correlation.

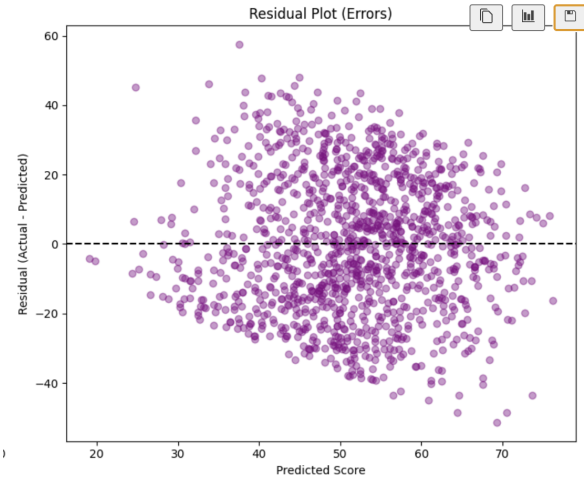


Figure 7: Residual Plot. Errors are centered around zero, indicating low bias.

Figure 8: Performance Visualization of the Final Regression Model

6 Web Interface

A web application was developed using **Flask** to serve the models.

6.1 Implementation

- **Backend:** Python (Flask) handles the API requests, loads the '.pkl' models, and processes input text.
- **Frontend:** HTML5 provides a clean interface for users to paste problem details.

6.2 Sample Prediction Web Interface

To make this tool accessible, we built a user-friendly web interface where users can simply paste a problem statement to get an instant difficulty rating.

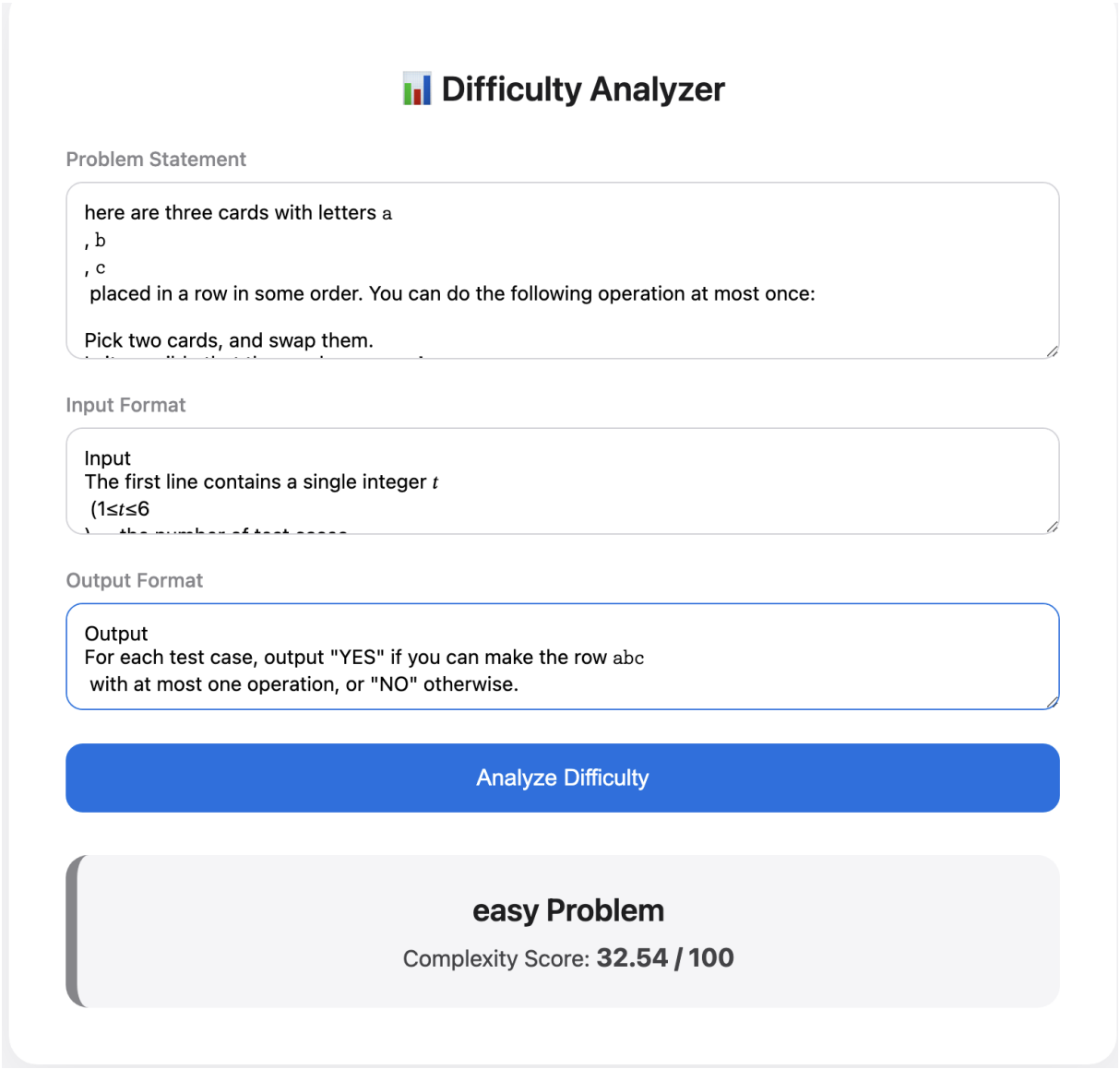


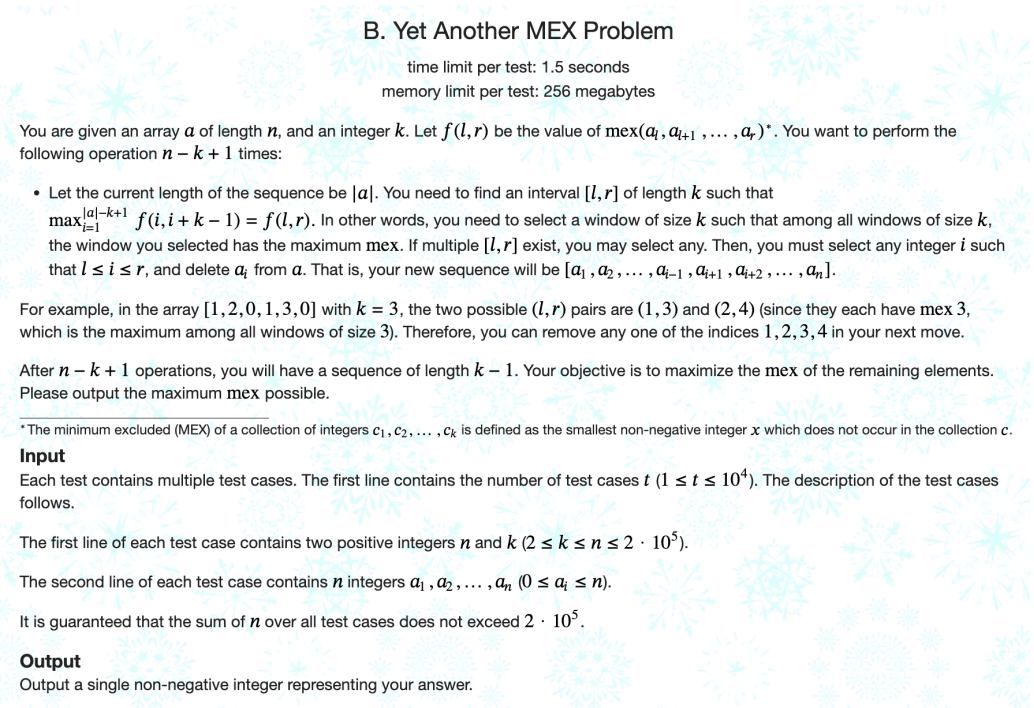
Figure 9: The Web Interface in action: Predicting a 'Easy' difficulty problem'

7 Conclusion and Future Work

7.1 Limitations The Buzzword Trap

While the project successfully demonstrates that machine learning can approximate difficulty using text, our error analysis highlights a critical limitation: the disconnect between **Textual Complexity** and **Algorithmic Logic**.

A relatable example of this limitation can be seen in the recent *Hello 2026* contest on Codeforces (Problem B). As shown below, the problem statement appears mathematically dense, filled with complex terms like "ranges," "MEX," and "intervals."



B. Yet Another MEX Problem

time limit per test: 1.5 seconds
memory limit per test: 256 megabytes

You are given an array a of length n, and an integer k. Let $f(l,r)$ be the value of $\text{mex}(a_l, a_{l+1}, \ldots, a_r)^*$. You want to perform the following operation $n-k+1$ times:

- Let the current length of the sequence be $|a|$. You need to find an interval $[l,r]$ of length k such that $\max_{i=1}^{|a|-k+1} f(i, i+k-1) = f(l,r)$. In other words, you need to select a window of size k such that among all windows of size k, the window you selected has the maximum mex. If multiple $[l,r]$ exist, you may select any. Then, you must select any integer i such that $l \le i \le r$, and delete a_i from a. That is, your new sequence will be $[a_1, a_2, \ldots, a_{i-1}, a_{i+1}, a_{i+2}, \ldots, a_n]$.

For example, in the array $[1,2,0,1,3,0]$ with $k=3$, the two possible (l,r) pairs are $(1,3)$ and $(2,4)$ (since they each have $\text{mex}\ 3$, which is the maximum among all windows of size 3). Therefore, you can remove any one of the indices $1,2,3,4$ in your next move.

After $n-k+1$ operations, you will have a sequence of length $k-1$. Your objective is to maximize the mex of the remaining elements. Please output the maximum mex possible.

*The minimum excluded (MEX) of a collection of integers $c_1, c_2, \ldots, c_k$ is defined as the smallest non-negative integer x which does not occur in the collection c.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \le t \le 10^4$). The description of the test cases follows.

The first line of each test case contains two positive integers n and k ($2 \le k \le n \le 2 \cdot 10^5$).

The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ ($0 \le a_i \le n$).

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

Output a single non-negative integer representing your answer.

Figure 10: Case Study: Problem B from Hello 2026 (Yet Another Mex Problem)

When we feed this problem into our model, it detects these "scary" keywords and predicts a high difficulty score. However, any competitive programmer solving this would realize the logic is actually quite simple—often reducing to a basic observation like $\min(mex, k-1)$.

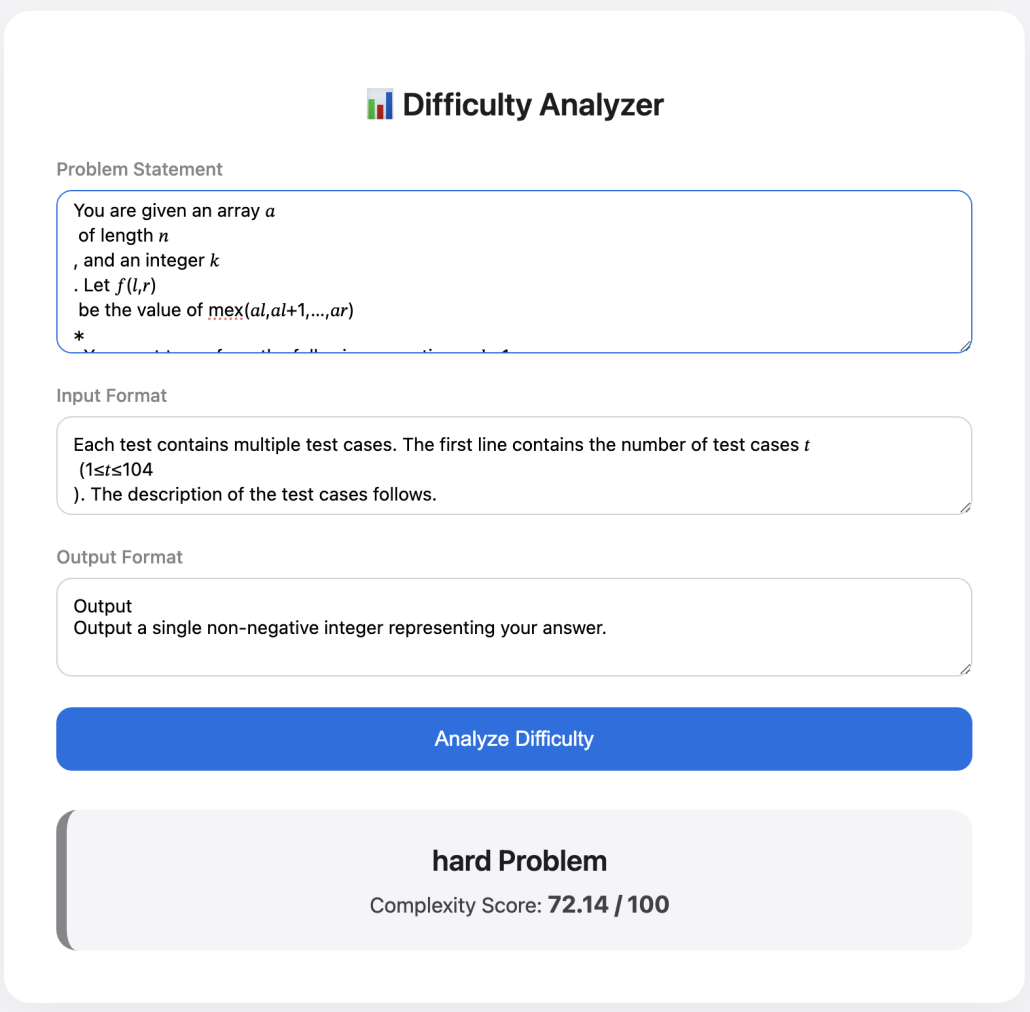


Figure 11: Model Prediction: Overestimating difficulty due to "complex" keywords.

This confirms that our model for some problems acts as a "Buzzword Detector." It struggles to differentiate between problems that are genuinely hard and problems that just *sound* hard. In extreme cases where the solution is trivial (e.g., `cout << 1;`) but the story is long, the model will fail. However, on average, for standard problems where length correlates with effort, it performs reliably.

7.2 Conclusion

The Random Forest classifier achieved a usable baseline accuracy of **53%**, significantly better than random guessing. The regression results ($R^2 \approx 0.16$) further prove that while text is a useful signal, it is not the *whole* story. Difficulty is subjective and often lies in the "trick" of the logic, not the words used to describe it. But this if often helpful when these extreme cases are not present, whilst this extreme cases are present at a good amount. Our Model still is able to correctly tag in many problems which is very helpful to the people who are starting competitive programming or problem setters.

7.3 Possible Improvements

To bridge this gap between perceived and actual difficulty, future iterations could implement:

1. **Submission Statistics Integration:** Incorporating metadata such as the "Accepted Submission Count" or "Accuracy Rate" would be the single strongest predictor of difficulty, serving as a proxy for the community's actual struggle.

2. **Solution Code Analysis:** Instead of analyzing only the problem statement, a dual-input model that also processes the *Editorial* or *Sample Solution* code could detect when a "scary" looking problem actually has a 3-line solution.

3. **Deep Learning (Transformers):** Replacing TF-IDF with Contextual Embeddings (e.g., BERT) could help the model understand the *semantic* context of constraints rather than just counting keyword frequencies.