Project Report: AutoJudge

Predicting Programming Problem Difficulty using Machine Learning

Author: Gaurav Arora (3rd year BS-MS Economics Student)

1. Introduction

Competitive programming platforms like Codeforces and LeetCode host thousands of algorithmic problems. Currently, assigning difficulty levels (e.g., Easy, Medium, Hard) and numerical scores to these problems is a manual, subjective process reliant on community feedback and moderator judgment. This leads to inconsistencies and delays in categorizing new content.

AutoJudge is an automated machine learning system designed to solve this by predicting the difficulty of a problem solely based on its **textual description**. By analyzing the problem statement, constraints, and input/output formats, the system removes human bias and provides instant difficulty estimation.

2. Problem Statement

The objective of this project is to build a dual-prediction system that operates without access to metadata (like user submission rates or tags).

The system performs two core tasks:

1. **Classification:** Categorizing problems into **Easy, Medium,** or **Hard**.
2. **Regression:** Predicting a precise **Numerical Difficulty Score** (normalized to a 1–10 scale).

This is a challenging Natural Language Processing (NLP) task because "difficulty" is often hidden in subtle clues—such as mathematical constraints ($N \le 10^9$) or specific keywords implying complex algorithms (e.g., "shortest path", "minimize cost").

3. Dataset & Preprocessing

The model was trained on a dataset of competitive programming problems collected from various online judges.

3.1 Data Fields

- **Input Features:** Problem Title, Description, Input Format, Output Format.
- **Target Variables:**
  - problem_class: Categorical labels (Easy, Medium, Hard).
  - problem_score: Continuous values (scaled to 1–10).

3.2 Expert Preprocessing Pipeline

Standard text cleaning is insufficient for technical domains. We implemented a custom **"Expert Cleaning"** pipeline:

- **Constraint Normalization:** Mathematical constraints are strong indicators of difficulty. Terms like 10^9 or 10^{18} were replaced with a special token `heavy_constraint` to signal high computational complexity to the model.
- **Time Limit Detection:** explicit mentions of time limits (e.g., "2.0 seconds") were captured as `time_limit` tokens.
- **Symbol Preservation:** Unlike standard cleaning which removes punctuation, we preserved mathematical symbols (`%`, `<`, `>`, `^`) as they often indicate specific algorithmic requirements (e.g., modulo arithmetic).
- **Lemmatization:** Words were reduced to their root forms using NLTK to reduce dimensionality.

4. Methodology

We avoided "black box" Deep Learning models in favor of an interpretable, high-performance Classical Machine Learning approach.

4.1 Feature Engineering

The system utilizes a hybrid feature set:

1. **TF-IDF Vectors (N-grams):** Captures the semantic context of the problem statement (Unigrams and Bigrams).
2. **Handcrafted Cognitive Features:** We engineered specific features to mimic how a human programmer judges difficulty:
   - **Topic Counters:** Frequency of keywords related to *Dynamic Programming*, *Graph Theory*, *Number Theory*, *Geometry*, and *Brute Force*.
   - **Structural Features:** Length of the description and density of mathematical symbols.
   - **Rarity Score:** A metric representing the "information density" or vocabulary complexity of the text.

4.2 Model Architecture

- Classifier (Voting Ensemble):

A Soft-Voting Ensemble combining Logistic Regression and Random Forest.
   - *Logistic Regression* handles the high-dimensional sparse data from TF-IDF.
   - *Random Forest* captures non-linear relationships in the handcrafted features.
   - **Result:** robust classification that outperforms individual models.

- Regressor (Gradient Boosting):

A Gradient Boosting Regressor (GBR) was selected to minimize the Mean Absolute Error (MAE) for the precise score prediction, effectively handling the ordinal nature of difficulty scores.

5. Web Interface

To demonstrate the practical utility of AutoJudge, a web-based user interface was developed using **Streamlit**.

Workflow:

1. **Input:** The user pastes the problem description, input format, and output format into the provided text areas.
2. **Processing:** The app applies the same Expert Cleaning and feature extraction logic used during training.
3. **Output:**

○ **Difficulty Class:** A class (Easy/Medium/Hard).
○ **Difficulty Score:** A precise numerical value (e.g., "7.4 / 10").
○ **Visual Indicator:** A progress bar visualizing the complexity level.

6. Results & Evaluation

The model was evaluated on a held-out test set (20% of data).

- **Classification Accuracy:** The Ensemble model successfully distinguishes between 'Easy' and 'Hard' problems with high reliability. Confusion primarily occurs between adjacent classes (e.g., Easy vs. Medium), which is expected due to the subjective nature of the ground truth.
- **Regression Performance:** The Gradient Boosting model achieved a low Mean Absolute Error (MAE), indicating that the predicted scores closely track the actual difficulty ratings.

7. Conclusion

AutoJudge demonstrates that programming problem difficulty can be effectively predicted using text analysis and domain-specific feature engineering. By explicitly modeling "hard" signals like constraints and algorithmic keywords, the system achieves results comparable to human categorization without the need for complex neural networks.

Future Scope:

- Integration with Transformer models (BERT/CodeBERT) for deeper semantic understanding.
- Expansion to include code snippet analysis for problems that provide starter code.