AutoJudge: Predicting Programming Problem Difficulty Using Machine Learning

---

Introduction

Online competitive programming platforms such as Codeforces, CodeChef, and Kattis host thousands of programming problems categorized by difficulty levels like *Easy*, *Medium*, and *Hard*, along with numerical difficulty scores. These difficulty labels are crucial for learners to select appropriate problems and for platforms to organize contests effectively.

However, difficulty assignment is usually performed manually by problem setters or inferred from user feedback and submission statistics, making the process **subjective, slow, and sometimes inconsistent**.

This project, **AutoJudge**, aims to automate the prediction of programming problem difficulty using **only the textual description of the problem**. The system predicts:

- A **difficulty class** (Easy / Medium / Hard) — classification task
- A **difficulty score** — regression task
  Additionally, a web-based interface is developed to allow users to input a new problem description and instantly obtain predictions.

Problem Statement

The objective of this project is to design and deploy a machine learning system that:

–Predicts the **difficulty class** of a programming problem.
–Predicts a **numerical difficulty score**.
–Uses **only textual information** (no tags, solutions, or user statistics).
–Is deployable via a **simple web interface**.
–Produces interpretable and reproducible results.

The problem is challenging due to:

–Subjective human labeling
–Noisy text descriptions
–Overlapping vocabulary between difficulty classes
–Class imbalance (Medium and Hard problems dominate)

3. Dataset Description

The dataset consists of programming problems stored in **JSONL format**, where each entry includes:

–`title`
–`description`
–`input_description`
–`output_description`
–`problem_class` (Easy / Medium / Hard)
–`problem_score` (numerical difficulty)

Dataset Characteristics:

–Total samples: ~4100 problems
–Labels are **human-assigned**
–Difficulty scores are **continuous but noisy**

. Data Preprocessing

Data preprocessing plays a crucial role in handling noisy and inconsistent text.

Text Cleaning

All text fields were concatenated into a single input
Lowercasing applied
Mathematical symbols and operators normalized
Excess whitespace removed

Missing Value Handling

–Missing text fields replaced with empty strings
–Problems with insufficient content removed

Noise Removal

Extremely short or malformed problem descriptions filtered
Outlier samples with inconsistent labels removed
This reduced the dataset from **4112 → ~3948 problems**
Noise handling was **centralized in the feature engineering pipeline** to maintain consistency between training and inference.

Feature Engineering

Feature engineering is the most critical component of the AutoJudge system, as the entire prediction pipeline relies solely on **textual information** from problem statements. Unlike approaches that use metadata (tags, submissions, constraints tables), this project intentionally restricts itself to raw text, making robust feature design essential.

To capture both **semantic meaning** and **structural complexity**, a **hybrid feature representation** combining sparse text features and handcrafted numeric signals was developed.

Word-Level TF-IDF

Word-level TF-IDF vectors were used to capture the **semantic content** of problem statements. Common algorithmic terms such as *graph*, *dp*, *recursion*, *matrix*, and *greedy* often correlate strongly with problem difficulty.

Character-Level TF-IDF

Character-level TF-IDF was added to complement word-level features.

Rationale:

- Programming problems often contain:
  –Mathematical symbols
  –Pseudocode-like syntax
  –Formatting patterns

- Character n-grams capture:
  –Operator density
  –Numeric formatting
  –Structural complexity not visible at word level

Handcrafted Numeric Features

To move beyond bag-of-words representations, a set of **handcrafted linguistic, cognitive, and structural features** was designed. These features aim to approximate how humans perceive problem difficulty.

5.2.1 Text Length and Scale Features

Feature Category	Description
Text Length	Log-scaled word and character counts
Keyword Counts	Presence of algorithmic keywords (dp, graph, recursion, etc.)
Algorithmic Intent	Detection of algorithm families
Lexical Entropy	Measures vocabulary complexity
Cognitive Load	Combination of entropy and length
Constraint Pressure	Numeric bounds and constraints
Grammar Complexity	Structural language cues
Failure Awareness	Presence of warnings and edge-case hints
Numeric Density	Ratio of numbers to words

Symbol Density	**Mathematical operator usage**
Structural Density	**Control-flow language indicators**

This hybrid approach was chosen because:

- TF-IDF captures **what the problem is about**
- Numeric features capture **how difficult it feels**
- Combining both mirrors human reasoning

By integrating semantic, structural, and cognitive signals, the model gains a richer understanding of problem complexity than text-only baselines

Classification Models

Predicting the difficulty of programming problems using only textual descriptions is inherently challenging due to the **subjective and noisy nature of human-assigned labels** and the **overlapping linguistic patterns** across difficulty levels. A standard multi-class classification approach treats *Easy*, *Medium*, and *Hard* as independent categories, ignoring their natural ordering. To better model this structure, we adopt an **ordinal classification approach**, recognizing that difficulty follows an inherent progression (*Easy* < *Medium* < *Hard*). Instead of directly predicting one of three classes, the problem is decomposed into two binary decisions: whether a problem is *Easy* and whether it is *Hard*, with remaining cases classified as *Medium*. This design better reflects human judgment, improves robustness under class imbalance, and allows the model to focus on learning relative difficulty boundaries rather than absolute labels.

Models tested(balanced accuracy) -

- Logistic Regression(0.496)
- Linear SVM(0.490)
- RBF SVM(0.487)
- Random Forest Classifier(0.444)

  Best model was selected from these to be implemented in Ordinal classification

Ordinal Classification

Instead of treating difficulty as a flat 3-class problem, an **ordinal approach** was used:

–One classifier predicts *Easy vs Not Easy*
–Another predicts *Hard vs Not Hard*
–Remaining cases classified as *Medium*
This reflects the natural ordering:

- `Easy < Medium < Hard`

Regression Models

The following regression models were evaluated:

Linear Regression
ElasticNet
Huber Regressor
Random Forest Regressor
Gradient Boosting Regressor
The best model was selected using **Mean Absolute Error (MAE)**.

Experimental Setup

–Train-test split: 80% / 20%
–Random seed fixed for reproducibility
–All preprocessing and feature extraction performed **before training**
–Same preprocessing reused during inference
–Models compared using appropriate metrics:

- Balanced Accuracy for classification
- MAE for regression

Results & Evaluation

Classification Results (Best Model)

Ordinal Classification Report:

	precision	recall	f1-score	support
Easy	0.45	0.42	0.44	121
Hard	0.58	0.79	0.67	388
Medium	0.42	0.23	0.30	281
accuracy			0.53	790
macro avg	0.48	0.48	0.47	790
weighted avg	0.51	0.53	0.50	790

Regression Results (Best Model)

LinearRegression | MAE = 2.9457

ElasticNet | MAE = 1.7544

HuberRegressor | MAE = 2.0269

RandomForestRegressor | MAE = 1.6937

GradientBoostingRegressor | MAE = 1.6736

GradientBoostingRegressor turned out to be the best

➡️ GradientBoostingRegressor

MAE = 1.6736

RMSE = 1.9792

R^2 = 0.1237

Web Interface

A lightweight **Streamlit-based web interface** was developed.

Features:

Input fields for:
–Problem description

–Input description
–Output description

Predict button
Displays:
– Difficulty class
–Difficulty score

Deployment

Models and preprocessing artifacts serialized using `joblib`
Deployed via **Streamlit Community Cloud**
Public URL provided for live demo
GitHub repository hosts code and models

Conclusion

This project demonstrates that programming problem difficulty can be reasonably predicted using **text-only machine learning approaches**. While absolute accuracy is limited by subjective labeling, the system effectively distinguishes between difficulty levels, particularly identifying *Hard* problems.

Key Contributions:

- Ordinal classification approach
- Rich handcrafted feature engineering
- Noise-aware preprocessing
- Deployment-ready ML system
- Real-time web interface