⚖ AutoJudge: Predicting Programming Problem Difficulty

Author: Manish Kumar Gupta
Program: B.Tech, IIT Roorkee
Email: manish_kg@bt.iitr.ac.in
GitHub: https://github.com/manishkg27

Problem statement & Objectives

Problem Statement

Competitive programming platforms host thousands of problems, yet difficulty labeling is often **manual, subjective, and inconsistent**. In the competitive programming (CP) ecosystem, problem difficulty is typically assigned by the problem setter. As a result, a problem labeled as Medium by one author may be considered Hard by another.

This subjectivity leads to inconsistency across platforms and makes it difficult for learners to accurately select problems that match their skill level. Moreover, there is a lack of automated systems that can **standardize difficulty classification** based on the inherent complexity reflected in a problem's description.

The goal of this project is to **automatically predict the difficulty of programming problems** using only their **textual descriptions**, including the problem statement and input–output specifications, without relying on solution code or manual annotations.

Objectives

This project aims to:

- Predict **difficulty class** (Easy / Medium / Hard)
- Predict a **numerical difficulty score (1–10)**
- Use **textual analysis and machine learning**
- Provide results through a **web-based interface**

Dataset Description

Dataset Overview

The dataset (**`problems_data.jsonl`**) consists of competitive programming problems.
Although the raw dataset contains additional metadata (such as **sample I/O, URLs, and titles**), only the following fields are used for difficulty prediction and modeling:

Field	Description
description	Problem statement
input_description	Input format & constraints
output_description	Output format
problem_class	Difficulty label (Easy/Medium/Hard)
problem_score	Numerical difficulty score (1-10)

Dataset Challenges and its handling

- **Missing values in difficulty class and score** :

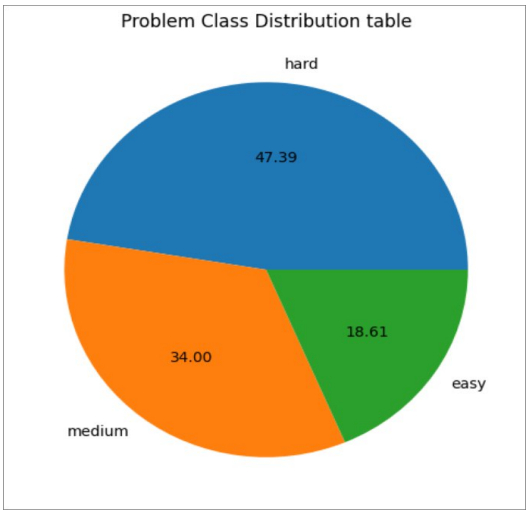
  The dataset contained missing values in both difficulty class and numerical difficulty score. To handle this, a two-step imputation strategy was applied.

  The problem_score column was first converted to numeric form, with invalid values treated as missing. Class-wise average scores for Easy, Medium, and Hard were computed. If the difficulty class was missing but the score was available, the class label was inferred using midpoint thresholds between adjacent class averages. Records missing both class and score were removed.

  For cases where the difficulty score was missing but the class label was present, the score was imputed using the average value of the corresponding class, with a fixed representative value used for the Hard category.

  This approach maintained consistency between categorical and numerical difficulty labels while maximizing usable data.

- **Class imbalance**
  The distribution of difficulty classes was imbalanced, with some classes appearing more frequently than others, which can bias model training.



problem_class	
hard	1941
medium	1405
easy	766

- **Empty textual fields:**
  Empty strings in textual fields were converted to missing values (NaN) to ensure consistent preprocessing.
- **Missing essential text:**
  Rows missing essential textual information, such as the problem description or input specification, were removed to maintain data quality.

Data Preprocessing steps

The raw problem statements contained noise, missing values, and mathematical notation that required careful preprocessing before feature extraction.

Text Normalization

- All text was converted to **lowercase** to ensure consistency.
- **Tokenization** was performed using NLTK.
- **Stopwords and punctuation** were removed to reduce noise.
- **Stemming** was applied using the *Porter Stemmer* to reduce words to their root forms (e.g., *“calculating” → “calcul”*).

Handling Mathematical and LaTeX-Style Text

Competitive programming problems often contain mathematical symbols and LaTeX-style expressions. To preserve meaningful signals:

- Inequality and operator symbols (≤, ≥, ×) were converted to textual equivalents (<=, >=, *).
- **Scientific notations** such as `10^5`, `10^6`, and `10^18` were normalized to unified forms.
- **Modular arithmetic expressions** like `10^9 + 7` were standardized to a special token (`mod_const`).
- **LaTeX** delimiters and escape characters were removed.

Preservation of Competitive Programming Semantics

- Common competitive programming variables (`n`, `m`, `k`, `q`) were **explicitly preserved**.
- Important numeric and constraint-related tokens were retained to maintain problem complexity information.

These preprocessing steps ensured that the cleaned text retained essential algorithmic and constraint-related information while remaining suitable for statistical feature extraction methods such as TF-IDF.

Feature Engineering Techniques

Text-Based Features (TF-IDF)

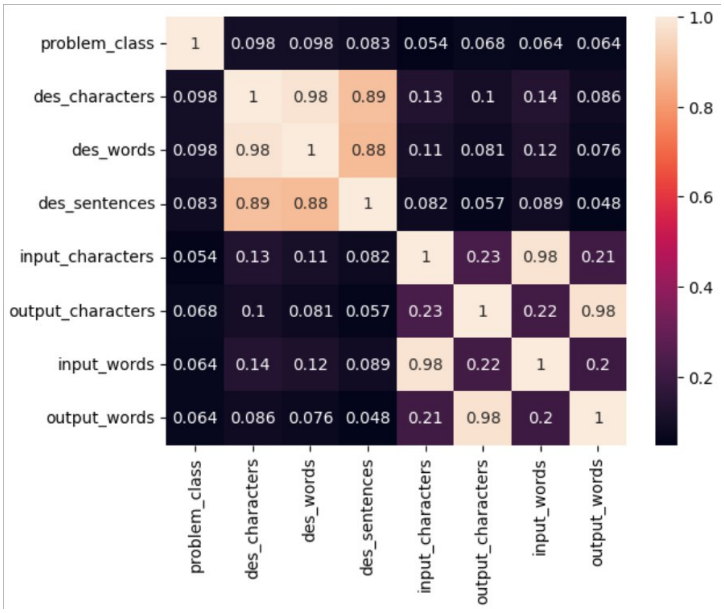
- The description, input, and output texts were combined and vectorized using TF-IDF (max_features = 3000) to capture informative terms without using deep learning models.

Length-Based Features:

To capture the structural complexity of programming problems, length-based features were extracted from different components of the problem text. These include:

- **Word count of the problem description**
- **Word count of the input description**
- **Word count of the output description**

Correlation analysis (as shown in the heatmap) indicates that **word-count–based features exhibit a stronger relationship with problem difficulty** compared to character or sentence counts. In particular, des_words, input_words, and output_words show meaningful associations with the difficulty class, suggesting that problems with longer and more detailed descriptions tend to be more complex.



Based on this observation, word-count features were selected as the primary length-based inputs for model training, as they provide a concise and effective representation of textual complexity.

Domain-Specific Features

In addition to generic text representations, **domain-specific features** were manually engineered to incorporate competitive programming (CP) knowledge that is often strongly correlated with problem difficulty.

Algorithm and Problem Pattern Keywords

A curated list of keywords related to common CP algorithms and problem-solving patterns was used, including:

- **Dynamic Programming:** `dp`, `dynamic programming`
- **Graph Algorithms:** `dfs`, `bfs`, `shortest path`
- **Searching and Optimization:** `binary search`, `greedy`
- **Problem Patterns:** `sliding window`, `two pointers`, `bitmask`, `backtracking`

These keywords capture the underlying algorithmic techniques implied by the problem statement.

Data Structure Indicators

Keywords representing commonly used data structures were extracted, such as:

- Arrays and strings
- Trees and graphs
- Segment trees and Fenwick trees
- Stacks, queues, and hash tables

The presence and frequency of these terms provide insights into the expected solution structure and complexity.

Keyword Frequency Encoding

For each algorithm and data structure keyword, the **frequency of occurrence** in the problem text was computed. These frequency-based features help identify the algorithmic depth and difficulty embedded in the statement.

Time and Memory Constraints

Binary features were used to detect mentions of:

- Time limits (seconds, milliseconds)
- Memory limits (MB, megabytes)

These features help identify problems with strict performance requirements.

Feature Scaling and Integration

All domain-specific features were normalized using **MinMax scaling** and concatenated with:

- TF-IDF text features
- Length-based features (word counts)

This ensured that domain features contributed meaningfully without dominating the overall feature space.

Model Design & Experimental Setup

This project addresses difficulty prediction as both a multi-class classification and a regression problem. Multiple classical and ensemble machine learning models were evaluated to ensure robust performance comparison.

Classification Models

The classification task aims to predict the difficulty category of a problem as **Easy, Medium, or Hard**. The following models were trained and evaluated:

- **Naive Bayes**
- **Logistic Regression**
- **Support Vector Machines (SVM)**
  - Linear kernel
  - RBF kernel
- **Random Forest**
- **Extra Trees**
- **Gradient Boosting**
- **XGBoost**
- **Voting Classifier** (ensemble of Random Forest and Extra Trees)

Regression Models

The regression task predicts a continuous **difficulty score (1–10)**. The following models were used:

- **Linear Regression**
- **Ridge Regression**
- **Bayesian Ridge Regression**
- **Support Vector Regression (SVR)**
- **Random Forest Regressor (tuned)**
- **Extra Trees Regressor (tuned)**
- **Gradient Boosting Regressor (tuned)**
- **XGBoost Regressor**

Hyperparameter tuning was applied to ensemble models to improve generalization performance.

Train–Test Setup

- The dataset was split using a **stratified train–test split**
- **Test size:** 20%
- The same split was used across all models to ensure **fair comparison**
- Models were evaluated strictly on **unseen test data**

Results & Evaluation

Classification Results:

Models were evaluated using accuracy and confusion matrices on the held-out test set.

Accuracy Comparison

Model	Accuracy (%)
Multinomial Naive Bayes	49.04
Bagging classifier	49.80
SVM (Linear)	50.44
Random Forest	**53.5**
Extra Trees	51.21
Gradient Boosting	48.78
XGBoost	50.83
Voting Classifier (RF + ETC)	**52.11**

The Voting Classifier and tree-based ensemble models consistently outperformed linear and probabilistic models due to their ability to capture non-linear relationships between textual, structural, and domain-specific features.

Confusion Matrix (Best Model)

The **best-performing classifier** (Random Forest) was further analyzed using a confusion matrix.

```
For  rfc
Accuracy -  0.5351213282247765
confusion matrix -
 [[ 70  59  17]
 [ 31 310  30]
 [ 49 178  39]]
```

Observations:

- The model performs best on the Medium class, correctly classifying a large majority of Medium-level problems.
- Easy and Hard problems show higher confusion with the Medium class.
- Misclassifications are primarily between adjacent difficulty levels, indicating that the model captures relative difficulty ordering.

Voting Classifier(**RF + ETC**)

```
For Voting Classifier (RF + ETC): Accuracy - 0.5211
confusion matrix -
 [[ 41  82  23]
 [ 13 331  27]
 [ 22 208  36]]
```

Observations:

- The Voting Classifier also performs strongly on the Medium class.
- Slightly lower performance on Easy and Hard categories compared to Random Forest.
- Ensemble averaging improves stability but does not outperform the tuned Random Forest.

Regression Results

Model performance was evaluated using Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R^2 score on the unseen test set.

Regression Performance Comparison

Model	MAE	RMSE	R^2
AdaBoost	1.741	2.063	0.084
Ridge Regression	1.630	1.946	0.184
Bayesian Ridge	1.634	1.948	0.182
SVR	1.626	1.956	0.176
Random Forest (Tuned)	1.638	1.965	0.169
Extra Trees (Tuned)	**1.619**	**1.939**	**0.190**
Gradient Boosting (Tuned)	1.662	1.993	0.145
XGBoost	1.706	2.076	0.071

Key Observations

- **Extra Trees Regressor (tuned)** achieved the lowest MAE and RMSE, making it the best-performing regression model.
- Gradient Boosting and Random Forest also performed competitively.
- Linear models showed higher error, indicating the non-linear nature of difficulty prediction.

Feature Importance Analysis

To improve interpretability, **feature importance** was extracted from the trained **Random Forest classifier**.

Interpretation:

- Length-based features dominate
- Constraint & variable usage matters
- Confirms interpretability

Web Interface

To demonstrate the practical applicability of the proposed system, a web-based interface was developed that allows users to input problem statements and receive instant difficulty predictions.

Architecture

- **Backend:** Flask
- **Frontend:** HTML, CSS, JavaScript
- **Model Deployment:**
    - All trained models and preprocessing components are serialized using **Joblib** and loaded during inference.

This architecture enables lightweight, fast, and fully local execution without requiring cloud deployment.

User Interface Overview

The interface provides three input fields:

- **Problem Description**
- **Input Description**
- **Output Description**

Two actions are available:

- **Predict Complexity** – triggers model inference
- **Clear All Fields** – resets all inputs

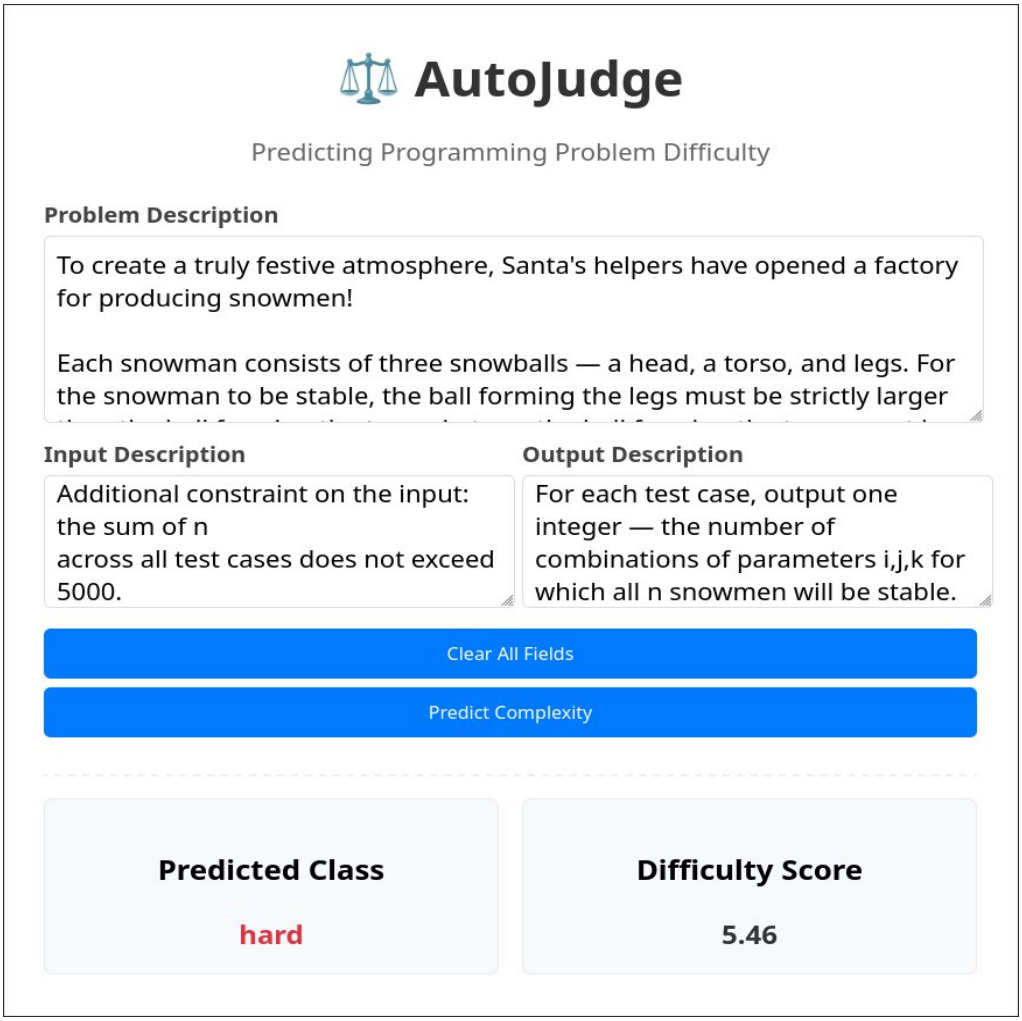
The output section displays:

- **Predicted Difficulty Class** (Easy / Medium / Hard)
- **Numerical Difficulty Score** (1–10)

Workflow

1. The user pastes the problem statement, input format, and output format into the respective fields.
2. The backend performs text preprocessing, including cleaning, tokenization, and feature extraction.
3. The processed input is vectorized using the saved TF-IDF and scaling models.
4. The classification and regression models predict the difficulty class and score.
5. Results are displayed instantly on the web interface.

Sample Predictions



Conclusion

This project presents an end-to-end machine learning system for automatically predicting the difficulty of competitive programming problems using only their textual descriptions. By combining classical machine learning models with carefully designed feature engineering, the system successfully predicts both a categorical difficulty label (Easy, Medium, Hard) and a numerical difficulty score.

The results demonstrate that structural properties of problem statements—such as text length, constraint expressions, and algorithm-specific terminology—carry strong signals about problem complexity. Length-based features and domain-specific indicators (e.g., constraints like 10^5, modulo arithmetic, and algorithmic keywords) were found to significantly improve performance when combined with TF-IDF representations.

Despite not using deep learning models, the proposed approach achieves reliable performance, strong interpretability, and low computational cost. The consistency between offline evaluation metrics and real-time predictions in the web interface further validates the robustness and practical usability of the system. Overall, this work shows that classical machine learning, when paired with domain knowledge, can effectively solve real-world NLP problems in competitive programming.