ACM OPEN PROJECT 25-26

Autojudge: Predicting Programming Problem Difficulty

Project Completed By: Sumit Sharma

Branch: Engineering Physics

Enrollment No.: 23123042

1. Introduction

1.1 Motivation

Online competitive programming platforms such as Codeforces, CodeChef, and Kattis host thousands of programming problems and continuously assign difficulty labels such as *Easy*, *Medium*, and *Hard*. In addition, many platforms associate a numerical difficulty rating that reflects how challenging a problem is for the average participant. These labels play a crucial role in guiding learners, organizing contests, and recommending problems.

However, difficulty assignment is largely **subjective**. It is influenced by problem setters, early user submissions, acceptance rates, and post-contest feedback. New problems often lack reliable difficulty estimates until sufficient user interaction occurs. Moreover, two problems with similar textual descriptions may differ significantly in difficulty due to hidden constraints or required optimizations.

The **AutoJudge** project explores whether it is possible to *automatically* predict the difficulty of a programming problem using **only its textual description**, without relying on submission statistics, runtime constraints, or editorial hints. This project frames the task as two complementary machine learning problems:

1. **Difficulty Classification** – Predicting whether a problem is *Easy*, *Medium*, or *Hard*.
2. **Difficulty Regression** – Predicting a continuous numerical difficulty score.

The project demonstrates a complete end-to-end machine learning pipeline, from raw data processing to model deployment via a web interface.

2. Problem Statement and Objectives

2.1 Problem Statement

Given the textual content of a programming problem—including its title, description, input specification, and output specification—build a system that can automatically predict:

- A categorical difficulty label (Easy / Medium / Hard)
- A numerical difficulty score

The system must rely **only on textual information** and must not use user interaction data or external metadata.

2.2 Objectives

The **key objectives** of this project are:

- To investigate how much information about difficulty is encoded in problem text
- To design meaningful text-based and structural features
- To compare classification and regression approaches
- To build an interactive system that demonstrates the final models

3. Dataset Description

3.1 Dataset Format

The dataset is provided in **JSON Lines (.jsonl)** format, where each line corresponds to one programming problem. Each sample contains the following fields:

- title: Short problem title
- description: Main problem statement
- input_description: Description of the input format
- output_description: Description of the output format
- problem_class: Difficulty label (Easy / Medium / Hard)
- problem_score: Numerical difficulty score

Other fields like sample.io and url weren't of much use while designing features and training models.

3.2 Dataset Characteristics

The dataset contains several thousand problems and reflects real-world competitive programming distributions:

- **Class imbalance** is present, with Hard problems appearing more frequently than Easy ones
- Difficulty scores are approximately bounded between **1 and 10**
- Labels are inherently noisy and subjective

No additional labeling or external datasets were used, ensuring fairness and reproducibility.

4. Data Preprocessing

4.1 Text Consolidation

All available textual fields were combined into a single unified text representation. This ensures that the model can leverage every part of the problem description, including formal input/output constraints that often encode difficulty.

4.2 Text Cleaning Strategy

Text preprocessing was carefully designed to avoid destroying technical meaning. The following steps were applied:

- Conversion to lowercase
- Removal of excessive whitespace
- Preservation of mathematical and programming symbols such as +, *, <, >=
- Removal of irrelevant special characters

Aggressive NLP techniques such as stemming and lemmatization were **intentionally avoided**, as they tend to degrade technical vocabulary (e.g., dp, graph, binary search).

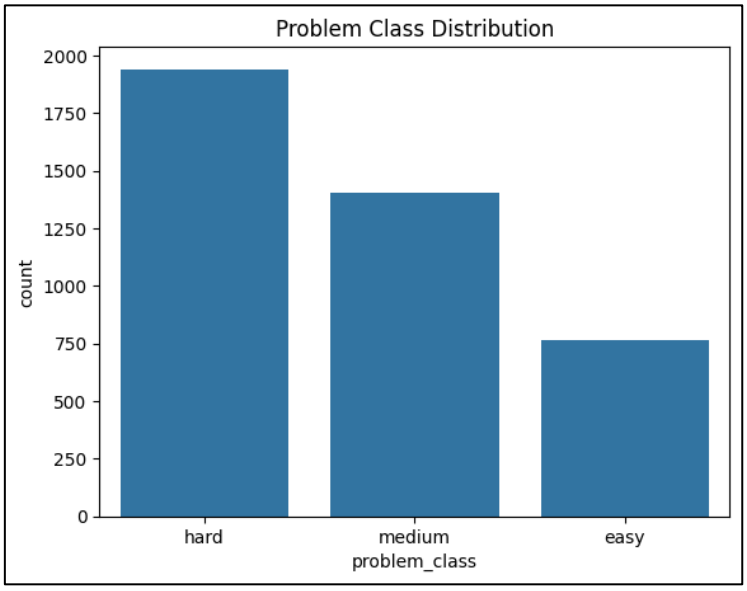
4.3 Handling Missing Values

Any missing textual fields were supposed to be replaced with empty strings to prevent loss of data but there were no missing values.

5. Exploratory Data Analysis (EDA)

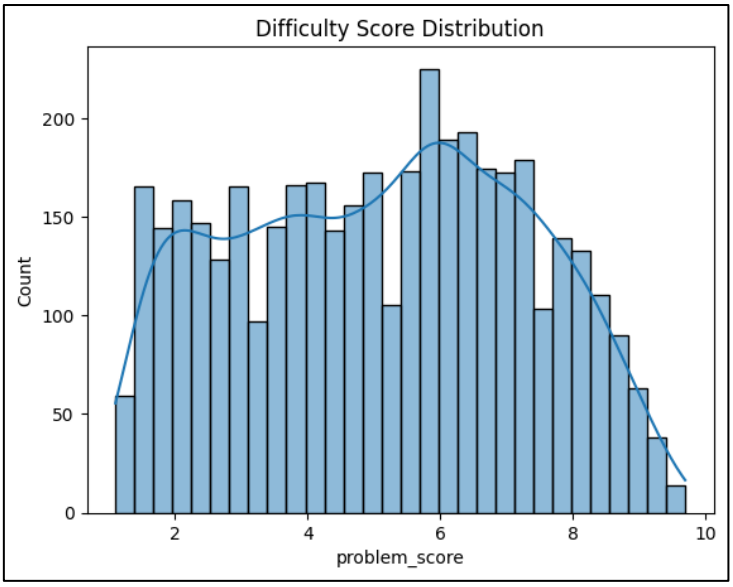
5.1 Difficulty Class Distribution

Exploratory analysis revealed a clear imbalance among difficulty classes. Hard problems dominate the dataset, followed by Medium and Easy. This imbalance reflects realistic competitive programming data and informed later modeling decisions, particularly around evaluation metrics.



5.2 Difficulty Score Distribution

The numerical difficulty scores follow a unimodal distribution bounded roughly between 1 and 10. The distribution is moderately skewed but does not contain extreme outliers. This observation justified modelling difficulty score as a continuous regression target rather than discretizing it further.



6. Feature Engineering

Feature engineering was the most critical component of the project.

6.1 TF-IDF Text Representation

The primary textual representation used **TF-IDF (Term Frequency–Inverse Document Frequency)** with unigrams and bigrams. TF-IDF highlights informative keywords such as graph, dp, recursion, and constraints, while down-weighting common filler words.

The resulting representation is high-dimensional and sparse, capturing semantic information effectively.

6.2 Handcrafted Complexity Features

To complement TF-IDF, a set of handcrafted features was designed to capture structural complexity:

- Total character length and word count
- Frequency of loop constructs (for, while)
- Conditional statements (if)
- Algorithmic indicators (graph, tree, dp, recursion)
- Numeric density and arithmetic operators

These features encode domain knowledge that pure bag-of-words models may miss.

6.3 Feature Scaling

Feature scaling decisions were model-dependent:

- For **classification**, scaling was unnecessary because tree-based models are scale-invariant.
- For **regression**, handcrafted features were standardized using StandardScaler to prevent length-based features from dominating the learning process.

6.4 Dimensionality Reduction (Regression Only)

TF-IDF features are extremely sparse. For regression, **Truncated SVD (Latent Semantic Analysis)** was applied to reduce dimensionality and capture latent semantic structure. SVD significantly improved linear regression models and slightly improved Gradient Boosting performance.

SVD was deliberately **not applied** to classification to preserve interpretability and because classification models already handled sparse features well.

7. Modeling and Experiments

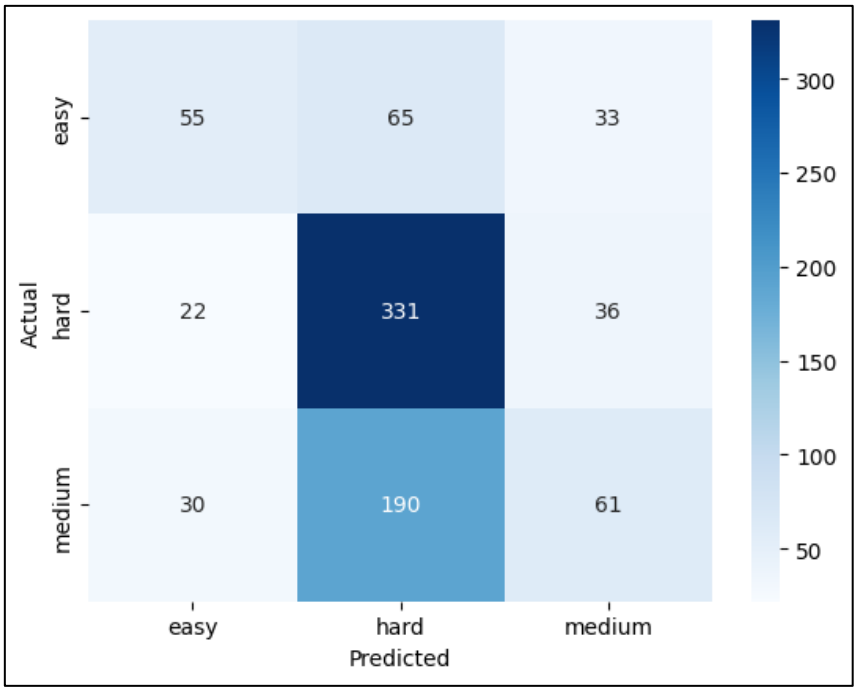
7.1 Difficulty Classification

Multiple models were evaluated, including Logistic Regression, Linear SVM, and Random Forest. After experimentation, **Random Forest Classifier** was selected due to its robustness to noisy labels and non-linear feature interactions.

Final classification performance:

- Accuracy ≈ **0.54**

Most errors occurred between Medium and Hard classes, which is expected given the subjective nature of difficulty.



7.2 Difficulty Score Regression

Regression experiments included Linear Regression, Ridge Regression, Random Forest Regressor, and Gradient Boosting Regressor.

Key findings:

- Plain Linear Regression performed very poorly due to sparsity and non-linearity
- Regularized linear models improved dramatically
- Tree-based models performed best overall

After extensive tuning and experimentation, **Gradient Boosting Regressor with SVD-enhanced features** was selected as the final model.

Final regression performance:

- MAE ≈ **1.68**
- RMSE ≈ **2.02**

This represents strong performance given the inherent noise in difficulty labels.

Model	MAE	RMSE
Linear Regression	1.698	2.064
Ridge Regression	1.716	2.308
Random Forest	1.727	2.045
Gradient Boosting	1.684	2.019

8. Web Interface

A Streamlit-based web application was developed to demonstrate the system.

8.1 User Interaction

Users can paste:

- Problem description
- Input description
- Output description

Upon clicking *Predict*, the system displays:

- Predicted difficulty class
- Predicted difficulty score
- Visual indicators such as progress bars and color-coded labels

8.2 Inference Consistency

The web interface strictly mirrors the training pipelines:

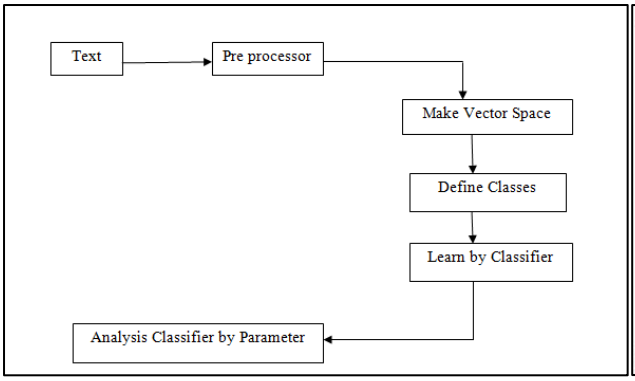
- Classification uses sparse TF-IDF + handcrafted features
- Regression uses TF-IDF → SVD → scaled handcrafted features

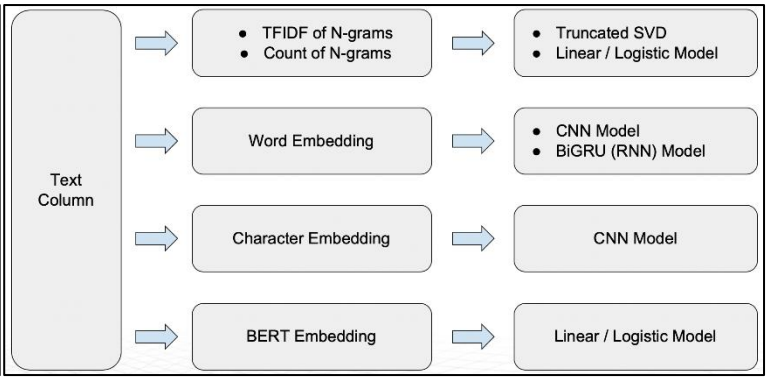
This guarantees consistency between training and deployment.

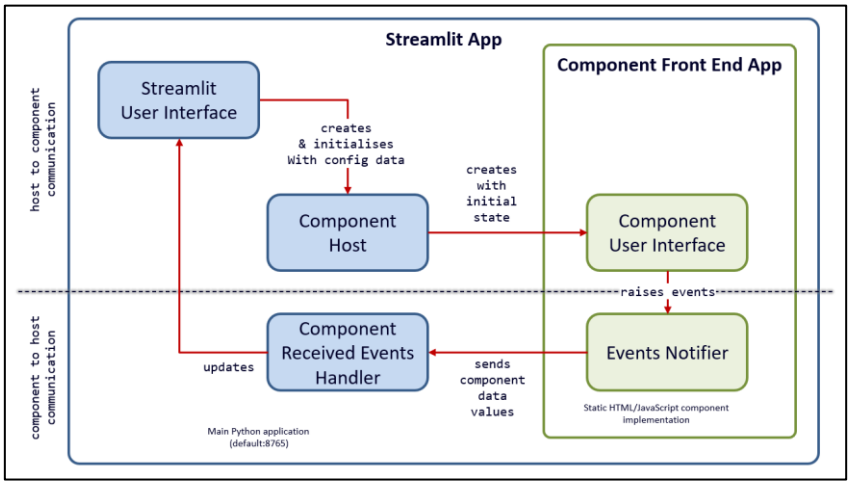
9. System Architecture

The overall system follows a modular pipeline:

1. User input
2. Text preprocessing
3. Feature extraction
4. Classification prediction
5. Regression prediction
6. Visualization of results







10. Limitations

- Difficulty labels are subjective and noisy
- Hidden constraints and optimizations are not captured by text
- Performance is bounded by dataset quality

11. Future Work

- Ordinal classification methods
- Model explainability (e.g., SHAP)
- Integration with contest metadata (if allowed)
- Cloud deployment

12. Conclusion

This project demonstrates that a significant amount of information about programming problem difficulty is encoded in textual descriptions. While classification provides coarse guidance, regression offers a more informative and reliable estimate of difficulty. AutoJudge highlights the importance of careful feature engineering, realistic evaluation, and disciplined experimentation in applied machine learning.