

# AutoJudge: Predicting Programming Problem Difficulty

Machine Learning – based system

Eshika Suresh Katekhaye

24114036

CSE (2<sup>nd</sup> yr)

IIT Roorkee

## 1. Introduction

Online programming platforms such as competitive coding websites and learning portals categorize problems into difficulty levels like *Easy*, *Medium*, and *Hard*. These labels help learners choose appropriate problems and assist platforms in organizing content. However, assigning difficulty levels is often subjective and requires human judgment.

This project presents **AutoJudge**, a machine learning - based system that automatically predicts the difficulty of programming problems using their textual descriptions alone. The system predicts both a **numerical difficulty score** and a **categorical difficulty label**, enabling a consistent and interpretable representation of problem difficulty.

The primary objective of this project is to explore whether textual features extracted from programming problem statements can be used to accurately model problem difficulty, and to design a system that produces reliable and consistent predictions.

## 2. Problem Statement

Given the textual description of a programming problem, the task is to:

1. Predict a **numerical difficulty score** representing the complexity of the problem.
2. Assign a **difficulty class** (*Easy* / *Medium* / *Hard*) based on the predicted score.

The challenge lies in modeling difficulty, which is inherently continuous, while also producing meaningful categorical labels without contradictions between the score and class.

## 3. Dataset Description

The dataset used in this project is provided in JSON format (problems\_data.json) and contains programming problems collected from online judges.

Each problem includes the following fields:

- Problem title
- Problem description
- Input description
- Output description
- Difficulty class (Easy / Medium / Hard)
- Numerical difficulty score

The dataset consists of **4112 programming problems**. No manual relabeling was performed, and the dataset was used as provided.

## 4. Data Preprocessing

All relevant textual fields - including the title, problem description, input description, and output description - are combined into a single text representation for each problem.

The preprocessing steps include:

- Converting text to lowercase
- Removing extra whitespace
- Combining all text fields to capture full context

This unified text representation ensures that the model has access to all available problem information during training.

## 5. Feature Engineering

Three categories of features are extracted from the text:

### 5.1 TF-IDF Features

Term Frequency - Inverse Document Frequency (TF-IDF) is used to capture important words and phrases in problem descriptions. Both unigrams and bigrams are included to represent algorithmic terms and patterns.

Mathematically, TF-IDF is defined as:

$$TF-IDF(t, d) = TF(t, d) \times \log_{\sqrt{2}}(N / DF(t))$$

where:

- $t$  is a term
- $d$  is a document
- $N$  is the total number of documents
- $DF(t)$  is the document frequency of term  $t$

## 5.2 Structural Text Features

To capture structural complexity, the following numerical features are extracted:

- Number of characters
- Number of words
- Count of mathematical symbols

These features provide signals about problem length and formulation complexity.

## 5.3 Keyword-Based Features

Binary and count-based features are added for common algorithmic keywords such as:

- *dynamic programming*
- *graph*
- *tree*
- *BFS / DFS*
- *greedy*

These keywords often correlate with higher problem difficulty.

All extracted features are concatenated into a single feature vector.

## 6. System Architecture

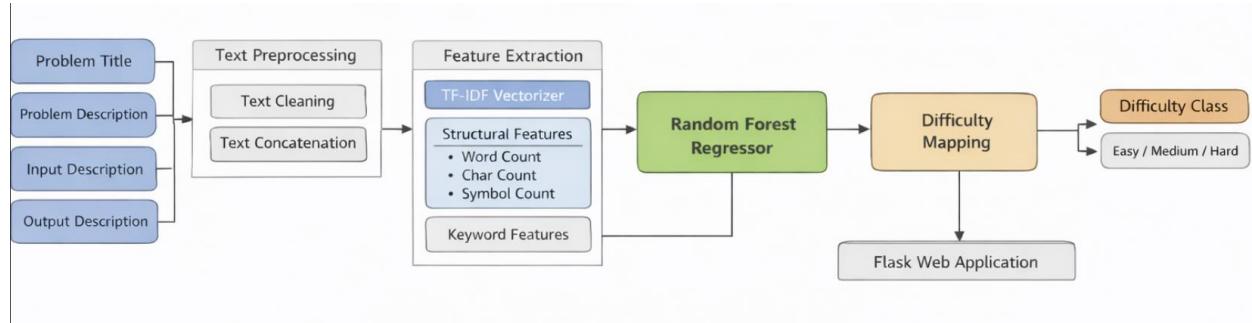


Figure 1: Architectural overview of the AutoJudge system

Figure 1 illustrates the overall architecture of the AutoJudge system. The pipeline represents the complete flow from raw problem statements to the final difficulty prediction.

The process begins with combining multiple textual components of a programming problem, including the title, description, input, and output sections. The combined text is then preprocessed and transformed into feature representations using TF-IDF, structural text features, and keyword-based features.

These features are passed to a Random Forest Regressor, which predicts a continuous difficulty score. The predicted score is subsequently mapped to a difficulty class (Easy, Medium, or Hard) using fixed thresholds. This design ensures consistency between the predicted score and the difficulty label.

The trained model and preprocessing components are reused by the web application to generate real-time predictions for user-provided problem statements.

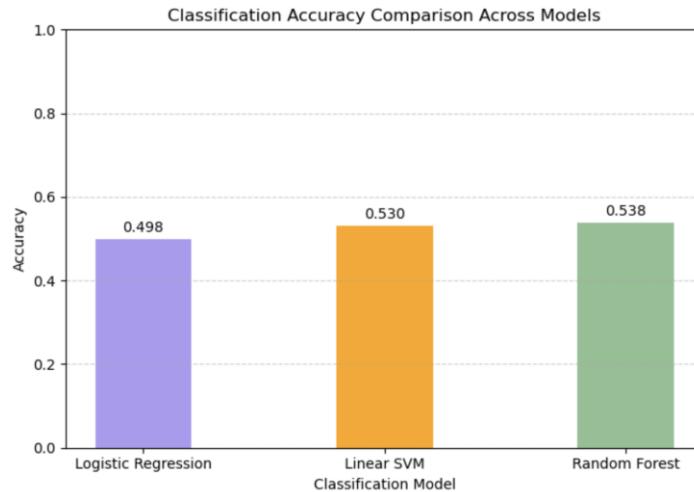
## 7. Models Used

### 7.1 Baseline Classification Models

As a baseline experiment, direct multiclass classification was performed using:

- Logistic Regression
- Linear Support Vector Machine (SVM)
- Random Forest Classifier

These models attempt to directly predict difficulty classes without considering the ordinal nature of difficulty.



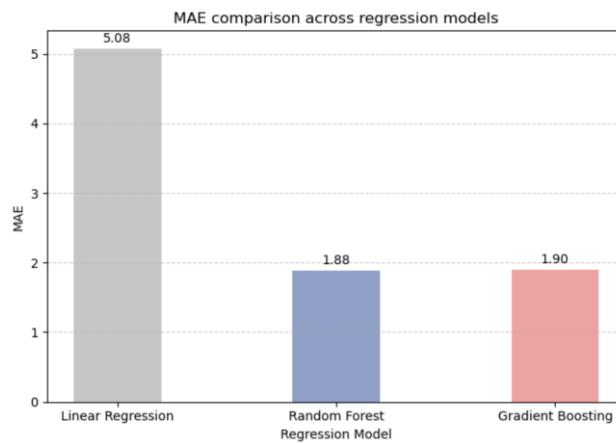
*Figure 2: Classification Accuracy Comparison*

## 7.2 Regression Models for Difficulty Score

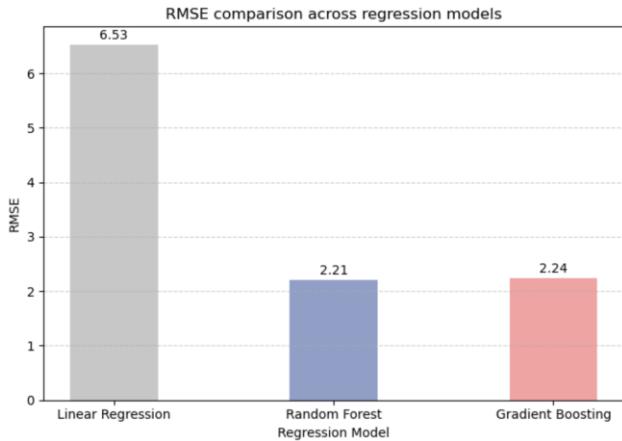
To model difficulty as a continuous variable, the following regression models were evaluated:

- Linear Regression
- Random Forest Regressor
- Gradient Boosting Regressor

Performance was measured using MAE and RMSE.



*Figure 3: MAE Comparison Across Regression Models*



*Figure 4: RMSE Comparison Across Regression Models*

The **Random Forest Regressor** achieved the lowest error values and was selected as the final regression model.

## 8. Difficulty Classification Strategy

Two strategies were explored for assigning difficulty classes:

1. **Direct Classification**

Difficulty class is predicted directly using a classifier.

2. **Regression-Based Classification**

A numerical difficulty score is predicted first, and the difficulty class is derived using fixed thresholds.

The fixed thresholds used are:

- Easy:  $\text{score} \leq 3.0$
- Medium:  $3.0 < \text{score} \leq 5.5$
- Hard:  $\text{score} > 5.5$

Although quantile-based thresholds achieved higher accuracy, they depend on dataset-specific statistics and are not suitable for deployment. Fixed thresholds provide consistent and interpretable predictions for unseen problems.

## 9. Experimental Results

Direct classification achieved slightly higher accuracy compared to regression-based threshold classification. However, direct classification occasionally produced contradictory outputs where the predicted class did not align with the predicted difficulty score.

To avoid such inconsistencies, the final system uses **regression as the backbone**, deriving difficulty classes from predicted scores using fixed thresholds.

## 10. Web Interface

A web application was developed using **Flask** to demonstrate the system.

The interface allows users to:

- Enter a problem title and description
- Receive a predicted difficulty score
- View the derived difficulty class
- Understand the prediction process through an interactive “How does this work?” section

The screenshot shows the AutoJudge web application interface. At the top, it says "AutoJudge" and "Predicting programming problem difficulty using textual analysis".

**Problem Title:**

**Problem Description:** sum only contains numbers 1, 2 and 3. Still, that isn't enough for Xenia. She is only beginning to count, so she can calculate a sum only if the summands follow in non-decreasing order. For example, she can't calculate sum 1+3+2+1 but she can calculate sums 1+1+2 and 3+3.

You've got the sum that was written on the board. Rearrange the summands and print the sum in such a way that Xenia can calculate the sum.

**Input Description:** The first line contains a non-empty string s — the sum Xenia needs to count. String s contains no spaces. It only contains digits and characters "+". Besides, string s is a correct sum of numbers 1, 2 and 3. String s is at most 100 characters long.

**Output Description:** Print the new sum that Xenia can count

**Predict Difficulty:**

**Predicted Difficulty Class:** Medium  
**Predicted Difficulty Score:** 4.92

[How does this work?](#)

Figure 5: Screenshots of web UI and sample prediction

## 11. Conclusion

This project demonstrates that programming problem difficulty can be effectively predicted using textual features alone. Modeling difficulty as a continuous variable through regression provides a more natural representation of problem complexity.

While direct classification achieves higher accuracy, a regression-based approach ensures consistency, interpretability, and better alignment with real-world deployment scenarios. The final AutoJudge system successfully balances performance and design reliability.

## 12. Future Work

Potential future improvements include:

- Incorporating code solution features
- Exploring transformer-based language models
- Learning adaptive but deployment-safe thresholding strategies
- Expanding the dataset with problems from multiple platforms

## References

1. Salton, G., and Buckley, C., *Term-weighting approaches in automatic text retrieval*, Information Processing & Management, 1988.
2. Friedman, J. H., *Greedy function approximation: A gradient boosting machine*, Annals of Statistics, 2001.
3. Pedregosa, F. et al., *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research, 2011.
4. Joachims, T., *Text categorization with Support Vector Machines*, European Conference on Machine Learning, 1998.

## Demo video link

[https://drive.google.com/file/d/1QcawqqzZ8T2SFvwJh3ZwtLzLwaTMplqJ/view?usp=share\\_link](https://drive.google.com/file/d/1QcawqqzZ8T2SFvwJh3ZwtLzLwaTMplqJ/view?usp=share_link)