Automatic Programming Problem Difficulty Prediction System(Deployed-app-link)

1. Introduction

Online competitive programming platforms such as Codeforces, CodeChef, and Kattis have thousands of programming problems and usually they classify them into difficulty levels such as Easy, Medium, and Hard. Apart from these classification, some platforms also assign difficulty scores with problems. These difficulty levels and scores play an important role in guiding learners, organizing contests, and maintaining balanced problem sets.

However, difficulty labeling is often performed manually or derived from user feedback, making it subjective, inconsistent, and time-consuming. With the increasing scale of problem repositories, there is a need for automated systems that can estimate problem difficulty reliably using objective criteria.

This project presents an **automatic problem difficulty prediction system** that predicts:

- A difficulty class (Easy / Medium / Hard)
- A numerical difficulty score between 1 and 10

The prediction is based **only on the textual description** of programming problems, without using solution code or user statistics. The system is implemented using classical machine learning techniques and deployed through an interactive web interface.

2. Problem Statement

The objective of this project is to design and implement an intelligent system that can automatically predict the difficulty of programming problems using their textual descriptions.

The system performs two tasks:

1. **Classification Task:**Predicts difficulty class (Easy, Medium, or Hard.)
2. **Regression Task:** Predicts a numerical difficulty score on a scale from 1 (easiest) to 10 (hardest).

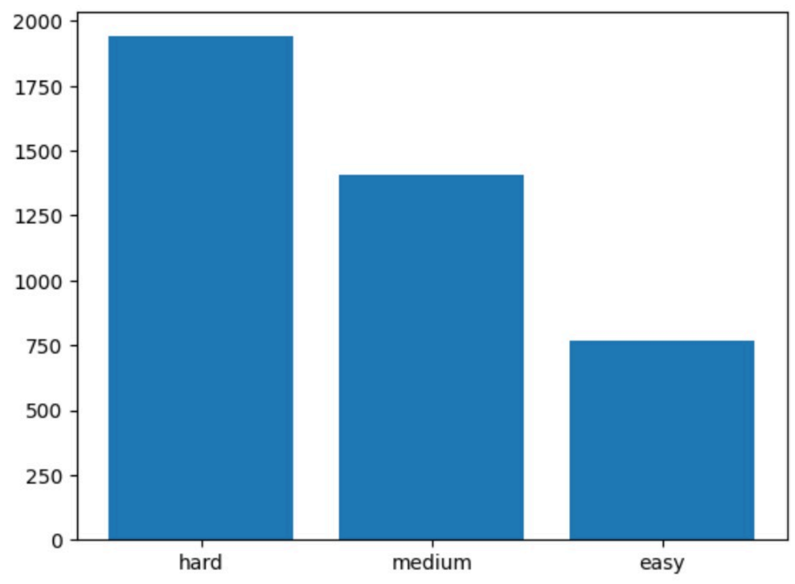
The prediction must rely exclusively on the problem statement text, including the problem description, input format, and output format.

3. Dataset Description

The dataset used in this project is obtained from **TaskComplexityEval-24** repository.(Dataset-link)

- **Format:** JSON Lines (JSONL)
- **Each record contains:**
    - description
    - input_description
    - output_description
    - problem_class (Easy / Medium / Hard)
    - problem_score (numeric)
    - URLs
    - Titles
    - Sample I/O

Irrelevant fields such as **problem URLs, titles, and sample I/O** were removed, as they do not contribute to difficulty estimation.



The dataset exhibits **class imbalance**, with Hard problems being more frequent than Easy and Medium problems.

problem_class	count	mn	mx	mean	std	length_mean
easy	766	1.1	2.8	1.970888	0.433289	1193.221932
hard	1940	5.5	9.7	7.071443	1.049919	1627.293299
medium	1405	2.8	5.5	4.125836	0.774216	1495.689680

4. Data Preprocessing

4.1 Handling Missing Values

The dataset was examined for missing or null values.
No missing values were found in the relevant textual or label fields.

```
description           0
input_description     0
output_description    0
problem_class         0
problem_score         0
dtype: int64
```

4.2 Text Combination

All textual components related to a problem were combined into a single field named full_text.
This combined text includes:

- Problem description
- Input specification
- Output specification

This unified representation simplifies feature extraction and ensures consistent input formatting.

4.3 Duplicate Handling

Duplicate problem statements were identified based on **exact matches of the combined text**.
Since duplicate entries shared identical labels,one of them was removed.

5. Feature Engineering

5.1 TF-IDF Text Features

Term Frequency–Inverse Document Frequency (TF-IDF) was applied to the combined problem text to capture important words and phrases associated with problem complexity.

Key configurations include:

- Use of **unigrams and bigrams**
- Minimum and maximum document frequency thresholds
- Sublinear term frequency scaling

These features help represent linguistic patterns commonly associated with varying difficulty levels.

5.2 Hand-Crafted Numerical Features

In addition to text features, several numeric and heuristic features were extracted:

- **Text length**: Total number of characters in the problem text
- **Mathematical symbol count**: Frequency of symbols such as =, <, >, +, -, *, %
- **Keyword indicators**: Binary features indicating the presence of common algorithmic concepts such as dynamic programming, graphs, binary search, DFS, and BFS

These features capture structural and algorithmic complexity often correlated with harder problems.

5.3 Feature Combination

TF-IDF features and numerical features were combined using a **column-wise transformation pipeline**.
Numeric features were standardized to ensure balanced scaling relative to text features.

6. Classification Model

A **two-stage hierarchical classification approach** was employed to improve discrimination between difficulty levels.

6.1 Two-Stage Hierarchical Classification

Stage 1: Hard vs Not-Hard

- A binary classifier predicts whether a problem is **Hard** or **Not-Hard**
- **Model**: Random Forest Classifier
- Class imbalance handled using weighted classes
- A probability threshold was optimized using cross-validated performance and then fixed for evaluation

This stage focuses on reliably identifying Hard problems, which tend to have stronger distinguishing characteristics.

Stage 2: Easy vs Medium

- Applied only to problems classified as Not-Hard

- **Model**: Linear Support Vector Classifier (LinearSVC)
- Class weights adjusted to reduce bias between Easy and Medium classes

The final difficulty label is obtained by combining predictions from both stages.

6.2 Rationale for Two-Stage Classification and Model Choice

A two-stage classification strategy was chosen to simplify the multi-class difficulty prediction task.
Separating the detection of Hard problems from Easy and Medium problems allows the model to first capture strong signals associated with harder problem descriptions before handling finer distinctions. This hierarchical approach helps isolate Hard problems early and reduces error propagation when distinguishing between Medium and Hard problems.

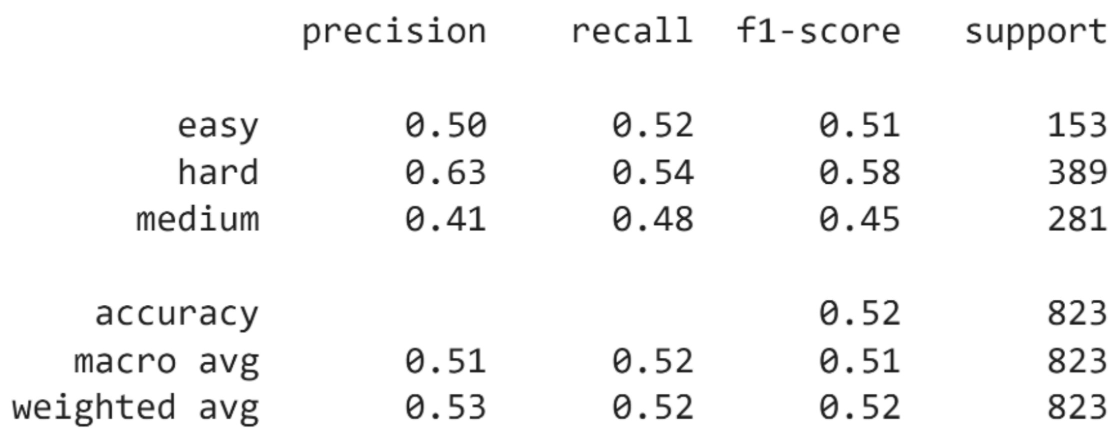
A Random Forest classifier was selected for the Hard vs Not-Hard stage because it is robust to noisy features and can effectively combine textual TF-IDF features with additional numeric indicators.
For the Easy vs Medium stage, a Linear Support Vector Classifier was used due to its strong performance on high-dimensional text features and its ability to separate closely related classes.

7. Classification Results and Evaluation

The classification system was evaluated using an **80–20 train–test split**.

Evaluation Metrics:

- Accuracy
- Confusion Matrix
- Precision
- Recall
- F1-score

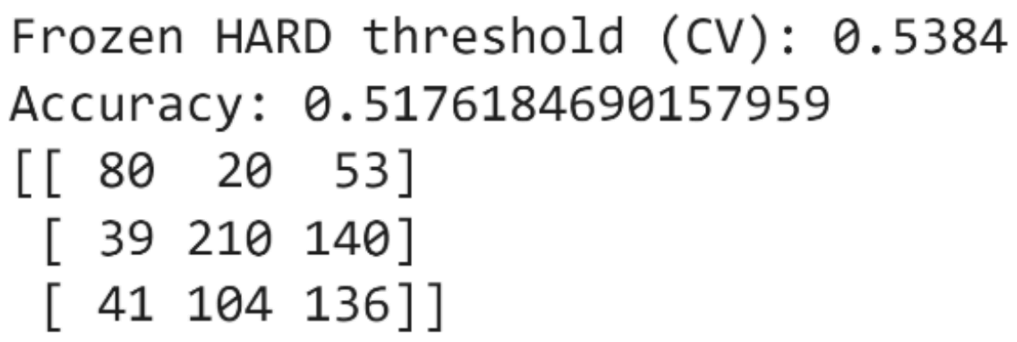


```
              precision    recall  f1-score   support

        easy       0.50      0.52      0.51       153
        hard       0.63      0.54      0.58       389
      medium       0.41      0.48      0.45       281

    accuracy                           0.52       823
   macro avg       0.51      0.52      0.51       823
weighted avg       0.53      0.52      0.52       823
```

Confusion Matrix Order: Easy, Hard, Medium



```
Frozen HARD threshold (CV): 0.5384
Accuracy: 0.5176184690157959
[[ 80  20  53]
 [ 39 210 140]
 [ 41 104 136]]
```

The observed performance shows an **overall accuracy of approximately 51.8%**, reflecting the inherent difficulty and overlap between problem difficulty classes.

8. Regression Model for Difficulty Score

A single **Ridge Regression** model is trained using all training data to predict the numerical difficulty score of a problem.
After prediction, the score is adjusted based on the **predicted difficulty class** (Easy, Medium, or Hard).This helps keep the predicted score consistent with the predicted class.

8.1 Regression Strategy

Instead of training separate regression models for each difficulty level, one common regression model is used.
The difficulty class predicted by the classification stage is then used only to **limit**

the score range during post-processing.
This approach keeps the system simple and makes better use of the available data.

8.2 Model Characteristics

Ridge Regression is used because it works well with **high-dimensional TF-IDF features**.
The same feature extraction process used for classification is also used for regression to ensure consistency.
Sparse feature vectors are converted to dense format where required by the regression model.

8.3 Why Ridge Regression Was Used

Ridge Regression was chosen because:

- It handles large numbers of text features effectively
- It reduces overfitting using regularization
- It provides stable and reliable predictions

8.4 Score Constraints

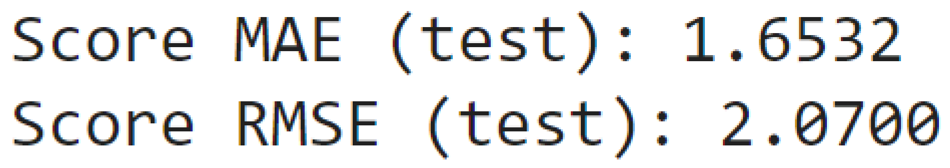
To ensure that the predicted score matches the predicted difficulty class, the output scores are limited to fixed ranges:

- **Easy:** up to 2.8
- **Medium:** between 2.9 and 5.5
- **Hard:** between 5.6 and 10.0

This prevents incorrect cases, such as assigning a very high score to an Easy problem.

8.5 Evaluation Metrics:

- Mean Absolute Error (MAE)
- Root Mean Squared Error (RMSE)


```
Score MAE (test): 1.6532
Score RMSE (test): 2.0700
```

9. Experimental Setup

- Train-test split: 80% training, 20% testing
- Stratified sampling used for classification
- Fixed random seed for reproducibility

Libraries Used:

- Python
- pandas
- NumPy
- scikit-learn
- Streamlit

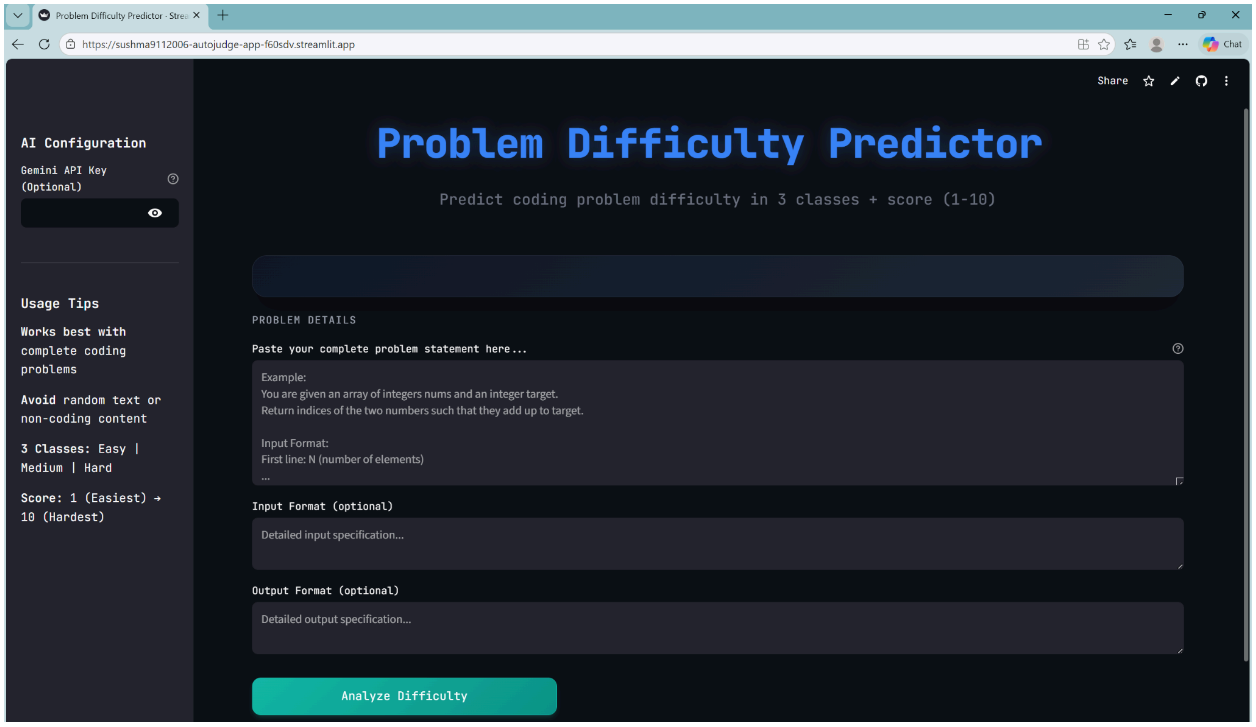
All reported results correspond directly to outputs produced by the implemented system.

10. Web Interface

A web-based application was developed using **Streamlit** to allow interactive use of the trained models.

Features:

- Text input for problem description
- Optional fields for input and output formats
- Real-time prediction of:
  - Difficulty class
  - Difficulty score (1–10)


Problem Difficulty Predictor · Strea

https://sushma9112006-autojudge-app-f60sdv.streamlit.app

Share

AI Configuration

Gemini API Key (Optional)

Usage Tips

Works best with complete coding problems

Avoid random text or non-coding content

3 Classes: Easy | Medium | Hard

Score: 1 (Easiest) → 10 (Hardest)

Problem Difficulty Predictor

Predict coding problem difficulty in 3 classes + score (1-10)

PROBLEM DETAILS

Paste your complete problem statement here...

Example:
You are given an array of integers nums and an integer target.
Return indices of the two numbers such that they add up to target.

Input Format:
First line: N (number of elements)
...

Input Format (optional)

Detailed input specification...

Output Format (optional)

Detailed output specification...

Analyze Difficulty

11. Optional AI-Based Validation Guardrail

As an additional robustness measure, the system includes an **optional AI-based input validation guardrail** at the web interface level.

This component:

- Checks whether user input resembles a valid programming problem
- Is used only for **high-confidence invalid input detection**
- Does **not influence** difficulty classification or score prediction
- Allows users to proceed even if the input is flagged as invalid, ensuring that no valid problem statement is rejected.
- For AI validation,

  1)Download required packages

  2) Enter the API key in the sidebar

This component does not influence any predictions made by the machine learning models.

12.Demo-link : Autojudge-demo.mp4

13. Conclusion

This project demonstrates that it is feasible to estimate programming problem difficulty using **only textual descriptions** and **classical machine learning techniques**.

Key contributions include:

- A hybrid feature representation combining TF-IDF and structural features
- A two-stage hierarchical classification strategy
- An interactive web interface for real-time difficulty prediction

Although the system achieves moderate accuracy, it establishes a strong foundation for automated difficulty estimation.

Explicit constraint extraction was not included, as constraints are often inconsistently specified, and incorrect or ambiguous numeric values can introduce noise rather than improve model performance. Instead, relevant information is partially captured through textual and structural features.