Problem Difficulty Prediction and Scoring System

Ankit Kumar

24117013

B. tech DSAI

IIT ROORKEE

1. Introduction

Competitive programming platforms host thousands of problems of varying difficulty. Manually assigning difficulty levels and numerical scores to new problems is time-consuming and subjective. This project aims to automatically classify programming problems into difficulty categories (Easy / Medium / Hard) and predict a continuous difficulty score using machine learning techniques.

The system takes textual problem descriptions as input, performs extensive preprocessing and feature engineering, and applies both classification and regression models. A lightweight web interface allows users to input new problems and view predicted difficulty levels and scores.

---

2. Problem Statement

The objectives of this project are:

1. To classify programming problems into predefined difficulty classes: Easy, Medium, Hard.
2. To predict a numerical difficulty score that reflects the relative complexity of a problem.
3. To build a complete pipeline consisting of data preprocessing, feature extraction, model training, evaluation, and deployment via a web interface.

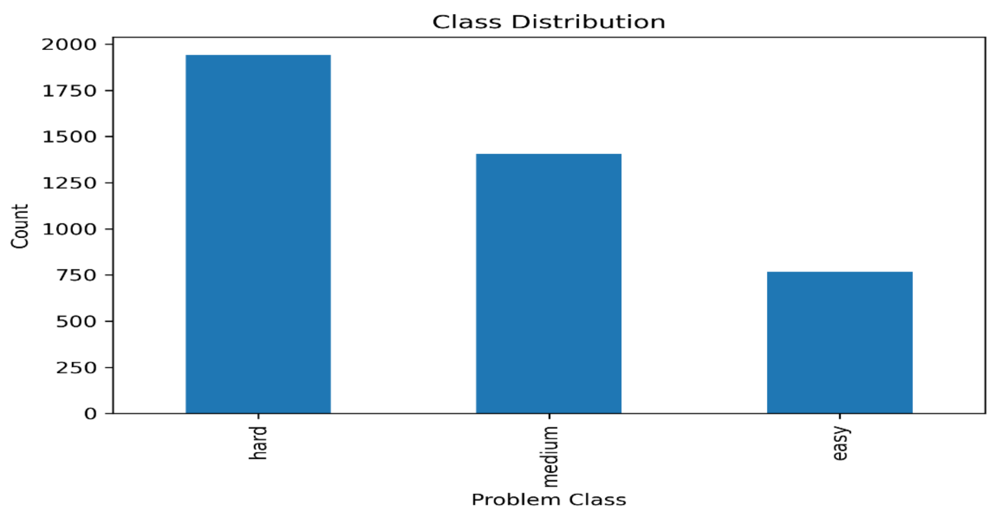
---

3. Dataset Description

I used the dataset given in the problem statement: “problem_data.jsonl”. The dataset consists of programming problems collected in JSON Lines (.jsonl) format. Each record corresponds to one problem and contains the following fields:

Dataset Characteristics

- No explicit NaN values, but empty strings were present in several text fields.
- Significant class imbalance, with Easy problems being overrepresented.



-

4. Data Exploration

Initial exploration revealed:

- No missing values in terms of NaN.
- Several records with empty input/output descriptions.
- Certain placeholder text such as *"There is no input in this problem."*.

Group-wise descriptive statistics were also computed to understand trends across difficulty levels.

5. Data Preprocessing

5.1 Handling Missing and Empty Fields

To ensure completeness of textual information:

- Empty input_description and output_description fields were replaced with the main description.
- Rows containing empty or whitespace-only fields were safely handled using fallback logic.

This ensured that every problem had sufficient textual content for downstream processing.

5.2 Text Cleaning

A custom text cleaning function was applied to normalize mathematical and textual symbols:

- LaTeX operators such as \\le, \\ge, \\times were converted to standard symbols.
- Newlines and excessive whitespace were removed.
- Commas inside numbers (e.g., 1,000) were normalized.

Cleaned versions of the following fields were created:

- clean_input
- clean_description
- clean_output
- sample_io

5.3 Text Consolidation

A unified textual representation named combined_text was constructed using:

Title + Clean Input Description + Clean Problem Description

Other columns were intentionally omitted due to high noise.

---

6. Feature Engineering

6.1 Constraint-Based Feature Extraction

Programming problems often specify constraints such as 10^5 or large integer limits. These were extracted using regex and transformed into a log-scaled feature:

- Maximum numeric constraint detected in the input
- Logarithmic scaling to reduce skewness

This resulted in the feature: log_max_constraint.

6.2 Structural Text Features

A simple but informative feature was added:

- text_len: Length of the combined problem text

6.3 Keyword-Based Difficulty Signals

A manually curated keyword dictionary was used to encode algorithmic hints:

- Easy signals: swap, palindrome, min, max, sort
- Medium signals: DP, BFS, DFS, greedy, binary search
- Hard signals: segment tree, bitmask, flow, FFT

Each keyword was encoded as a binary feature indicating its presence.

6.4 TF-IDF Text Representation

Textual features were extracted using TF-IDF vectorization:

- Maximum features: 2000
- N-grams: unigrams and bigrams
- English stopword removal

This captures semantic and frequency-based importance of words.

6.5 Feature Scaling and Final Feature Matrix

Numeric features (log_max_constraint, text_len) were standardized using StandardScaler.

All feature groups were combined using sparse matrix stacking:

Final Features = TF-IDF + Numeric + Keyword Features

The final matrix was saved as a sparse .npz file for efficient reuse.

---

7. Classification Models and Experiments

7.1 Models Evaluated

Multiple classification models were explored to predict the categorical difficulty level of problems.

Logistic Regression

Logistic Regression with L2 regularization was used as a strong linear baseline. Class imbalance was handled using class_weight="balanced". The multinomial setting enabled direct multi-class optimization.

Random Forest Classifier

Random Forest was chosen due to its robustness with mixed feature types (sparse TF-IDF + numeric + binary keyword features). Hyperparameters such as tree depth, minimum samples per split, and number of estimators were tuned empirically.

Linear SVM (Calibrated)

A Linear Support Vector Machine was trained and calibrated using sigmoid calibration to obtain probabilistic outputs. This allowed its integration into a soft-voting ensemble.

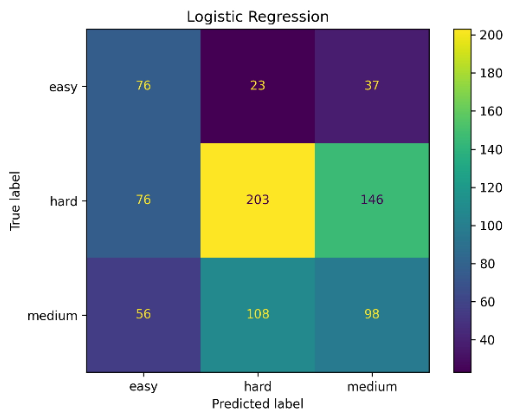
Ensemble (Voting Classifier)

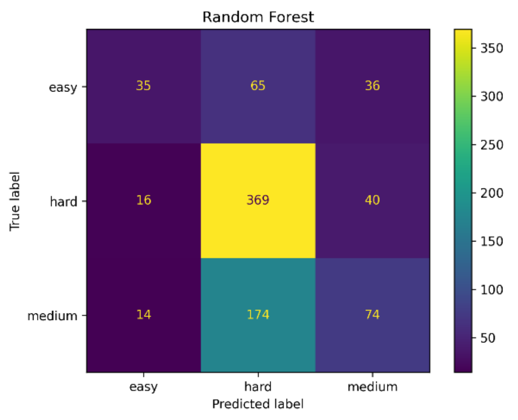
A soft-voting ensemble combining Logistic Regression, Random Forest, and SVM was trained. Higher weight was assigned to Random Forest as it captured non-linear interactions more effectively.

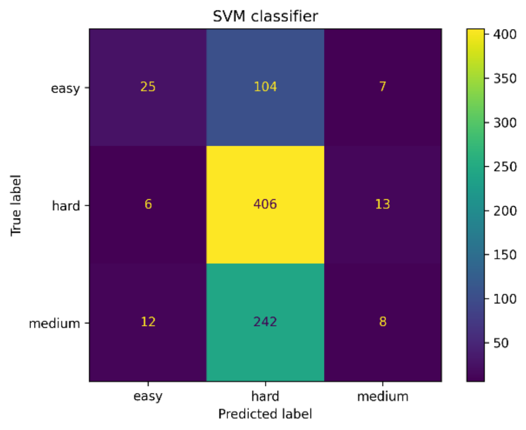
7.2 Evaluation Metrics

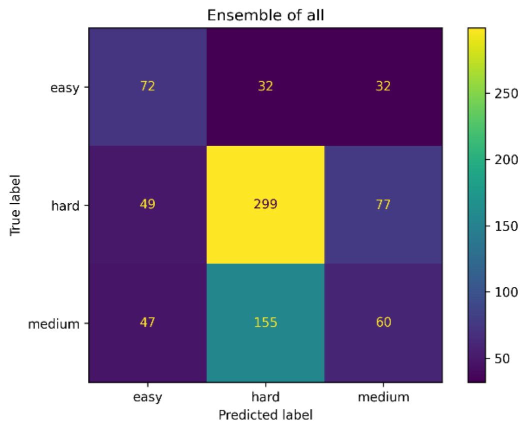
The models were evaluated on a held-out test set using:

- Accuracy : 45.81%, 58.08%, 53.34% and 52.37%
- Confusion Matrix : Plots are added below
- Precision, Recall, and F1-score (per class)









7.3 Final Classification Model

Based on empirical performance and stability, Random Forest Classifier was selected as the final model. It achieved an accuracy of approximately 58.08%, with balanced performance across difficulty classes.

The trained model was serialized for deployment using joblib.

---

8. Regression Models and Experiments

8.1 Objective

The regression task aimed to predict a continuous difficulty score corresponding to each problem. This complements the categorical difficulty prediction and provides finer-grained insights.

8.2 Baseline and Ensemble Regression Models

Baseline Models

- Linear Regression: Tested but discarded due to poor fit and negative R^2 score.
- Random Forest Regressor: Captured non-linear relationships effectively.
- Gradient Boosting Regressor: Improved generalization using shrinkage and subsampling.

These models were combined using a Voting Regressor, with higher weight given to Random Forest. Still, the r2 score was quite low.

---

9. Final Regression Model with Meta-Features

9.1 Motivation

Difficulty score is strongly correlated with difficulty class. To exploit this, classifier probability outputs were used as meta-features for the regression model.

9.2 Meta-Feature Construction

- Class probability vectors were generated using cross-validation on training data to avoid leakage.
- Probabilities for the test set were obtained from the fully trained classifier.
- These probabilities were concatenated with the original feature set.

9.3 Regression Ensemble

A second-stage ensemble was trained using:

- XGBoost Regressor: Captures complex non-linear interactions.
- Ridge Regression: Handles high-dimensional sparse features effectively.

Predictions from both models were averaged using a Voting Regressor.

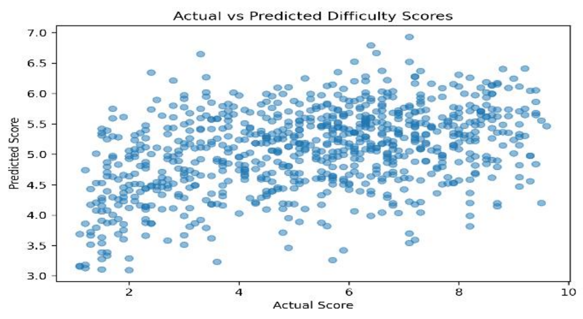
This approach significantly improved MAE and RMSE compared to baseline regressors and a dummy mean predictor.

9.4 Evaluation Metrics

Regression performance was measured using:

- Mean Absolute Error (MAE): 1.6614
- Root Mean Squared Error (RMSE): 2.0193

- R^2 score: 0.1505



---

10. Web Interface and Deployment

10.1 Overview

A lightweight web application was developed using Streamlit to demonstrate the practical usability of the trained models. The interface allows users to input a new programming problem and obtain:

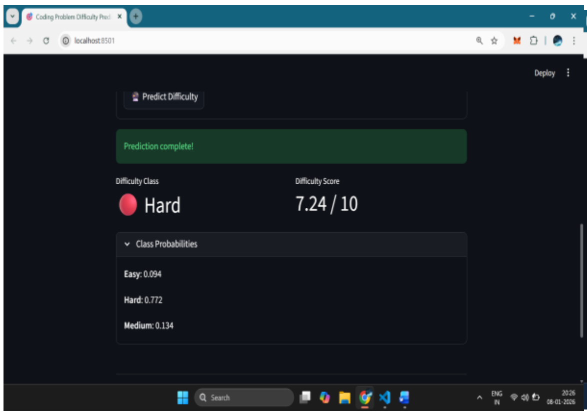
- Predicted difficulty class (Easy / Medium / Hard)
- Predicted numerical difficulty score (regression output)
- Class probability distribution for transparency

The web application strictly reuses the same preprocessing, feature extraction, and model artifacts as the training pipeline, ensuring consistency between offline evaluation and deployment.

10.2 User Interface

The interface provides:

- Input fields for title, description, input, and output formats
- Color-coded difficulty class display
- Numerical difficulty score (scaled to 10)
- Expandable section showing class probabilities



Prediction complete!

Difficulty Class
Hard

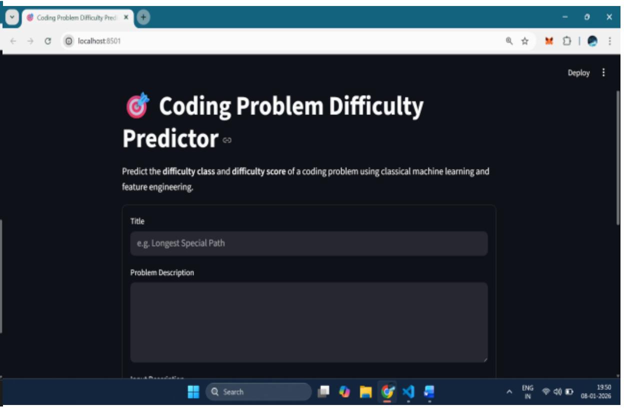
Difficulty Score
7.24 / 10

Class Probabilities

Easy: 0.094

Hard: 0.772

Medium: 0.134



Coding Problem Difficulty Predictor

Predict the **difficulty class** and **difficulty score** of a coding problem using classical machine learning and feature engineering.

Title
e.g. Longest Special Path

Problem Description

---

11. Final Results Summary

Task	Model	Key Metrics
Classification	Random Forest	Accuracy ≈ 0.58
Regression	XGBoost + Ridge Ensemble	Lower MAE & RMSE than baseline

---

12. Conclusion

This project demonstrates an end-to-end machine learning system for automatic difficulty assessment of programming problems. By combining textual analysis, handcrafted features, ensemble learning, and meta-feature stacking, the system achieves robust and interpretable performance.

Future work may include:

- Robust handling of noisy textual data
- Meaningful extraction of both semantic and structural features
- Compatibility with multiple machine learning models
- Incorporating code-based features
- Using transformer-based language models
- Further calibration of difficulty scores across platforms