ACM Open Project Report

AutoJudge: ML-Based Programming Problem Difficulty Predictor

Live Application: https://autojudge-difficulty-predictor.streamlit.app/

GitHub Repository: https://github.com/manasvi-sawaria/AutoJudge

Problem Statement

Predicting the difficulty of competitive programming problems is a challenging task that typically requires domain expertise and manual assessment. This project, **AutoJudge**, aims to automate this process using machine learning techniques.

Given a programming problem description (including title, problem statement, and input/output specifications), the system:

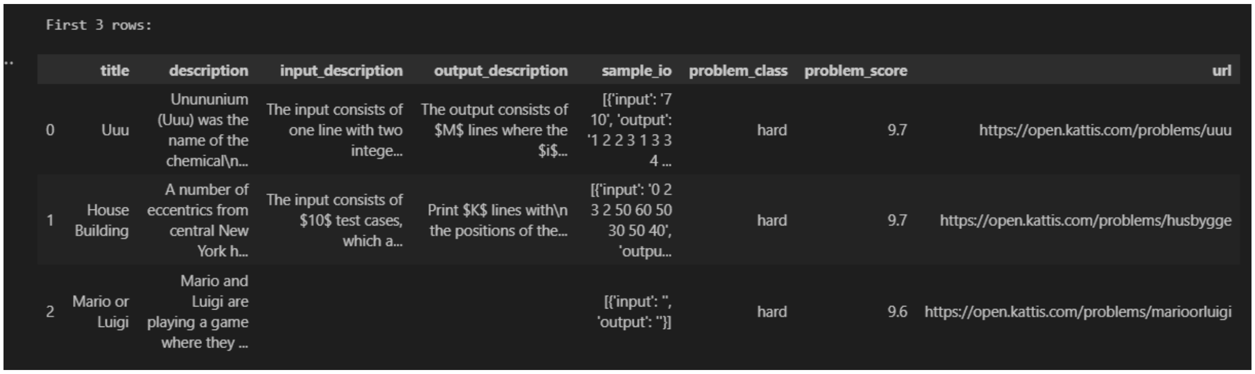
1. **Classifies** the problem into one of three difficulty categories: **Easy**, **Medium**, or **Hard**.
2. **Predicts** a numerical difficulty score on a scale of 1 to 10.

Dataset

The training data consists of **4,112 competitive programming problems** sourced from online judges (primarily Kattis), stored in JSONL format.

Each entry contains the following fields:

- Title
- Description
- Input Description
- Output Description
- Sample I/O
- Problem Class
- Problem Score
- URL



First 3 rows:

	title	description	input_description	output_description	sample_io	problem_class	problem_score	url
0	Uuu	Ununúnium (Uuu) was the name of the chemical\n...	The input consists of one line with two intege...	The output consists of M lines where the i...	[{'input': '7 10', 'output': '1 2 2 3 1 3 3 4 ...	hard	9.7	https://open.kattis.com/problems/uuu
1	House Building	A number of eccentrics from central New York h...	The input consists of 10 test cases, which a...	Print K lines with\n the positions of the...	[{'input': '0 2 3 2 50 60 50 30 50 40', 'outpu...	hard	9.7	https://open.kattis.com/problems/husbygge
2	Mario or Luigi	Mario and Luigi are playing a game where they ...			[{'input': '', 'output': ''}]	hard	9.6	https://open.kattis.com/problems/marioorluigi

Data Preprocessing

All text fields (title, description, input/output specs) were merged into one combined text. The following cleaning steps were applied:

- Convert text to lowercase.
- Remove URLs and HTML tags.
- Normalize whitespace (remove extra spaces).
- Handle missing values by substituting empty strings.

Feature Engineering

TF-IDF Vectorization

Term Frequency-Inverse Document Frequency (TF-IDF) was used to convert text into numerical vectors. The vectorizer was limited to the **top 5,000 terms** to balance feature richness with computational efficiency.

Heuristic Features

Additional hand-crafted features were extracted to capture domain-specific patterns:

- **Length-based:** Text length, word count, sentence count, average word length.
- **Mathematical symbols:** Count of symbols like ∑, ≤, ≥, etc.
- **Constraint patterns:** Detection of patterns like "1 ≤ n ≤ 10^5".
- **Algorithmic keywords:** Binary flags for *graph, dp, recursion, sort, binary search, matrix*.

Class Imbalance Handling

SMOTE (Synthetic Minority Over-sampling Technique) was applied to the training data to generate synthetic samples for minority classes, addressing the imbalanced distribution of Easy, Medium, and Hard problems.

Experimental Setup

Data was partitioned into:

- **Training:** 70%
- **Validation:** 15%
- **Test:** 15%

Models Used:

- **Classification:** Random Forest
- **Regression:** XGBoost Regressor

Results and Evaluation

Classification Results

Random Forest Classifier was selected as the classification model.

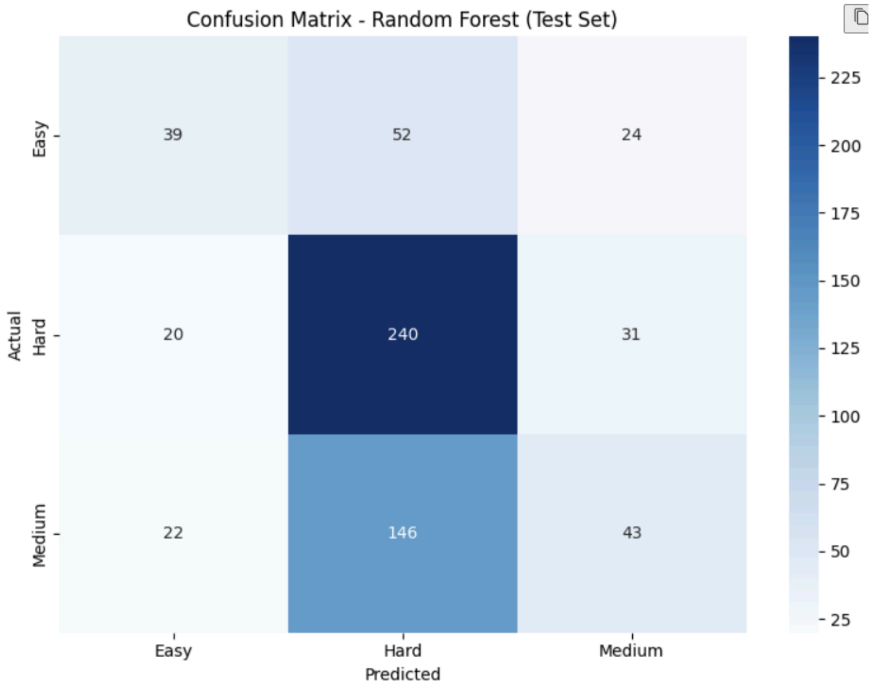
- **Overall Accuracy:** 52.19%
- **F1-Score (Macro):** 0.4860

Class-wise Accuracy:

- **Easy:** 33.91%
- **Medium:** 20.38%
- **Hard:** 82.47%

Confusion Matrix:

	Predicted Easy	Predicted Hard	Predicted Medium
Actual Easy	39	52	24
Actual Hard	20	240	31
Actual Medium	22	146	43

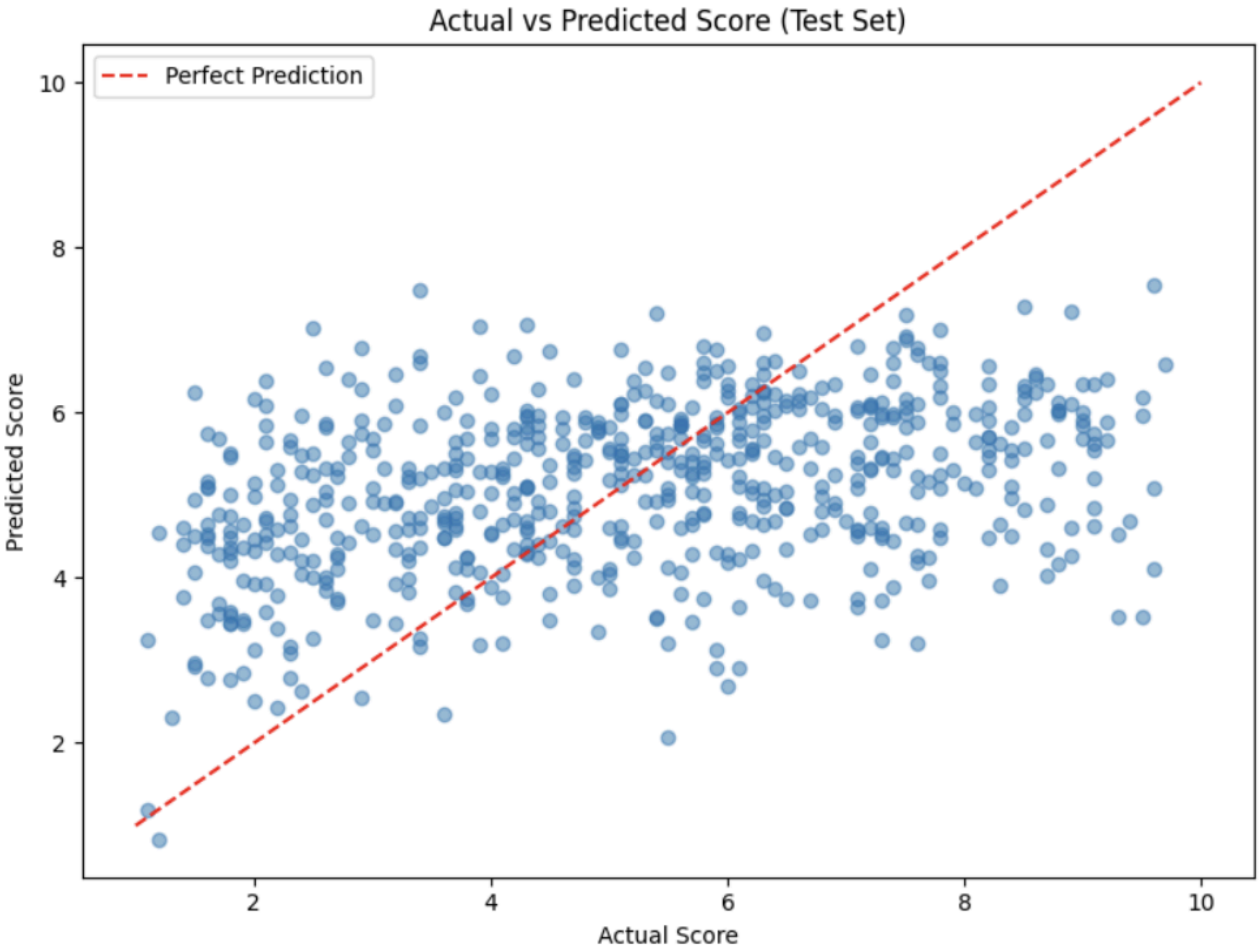


The model predicts **Hard** problems most reliably (82.47%), while Medium problems are often misclassified as Hard, indicating overlap in intermediate difficulty levels.

Regression Results

XGBoost Regressor was selected as the regression model.

- **MAE:** 1.68
- **RMSE:** 2.05
- **R²:** 0.12



Technology Stack & Implementation

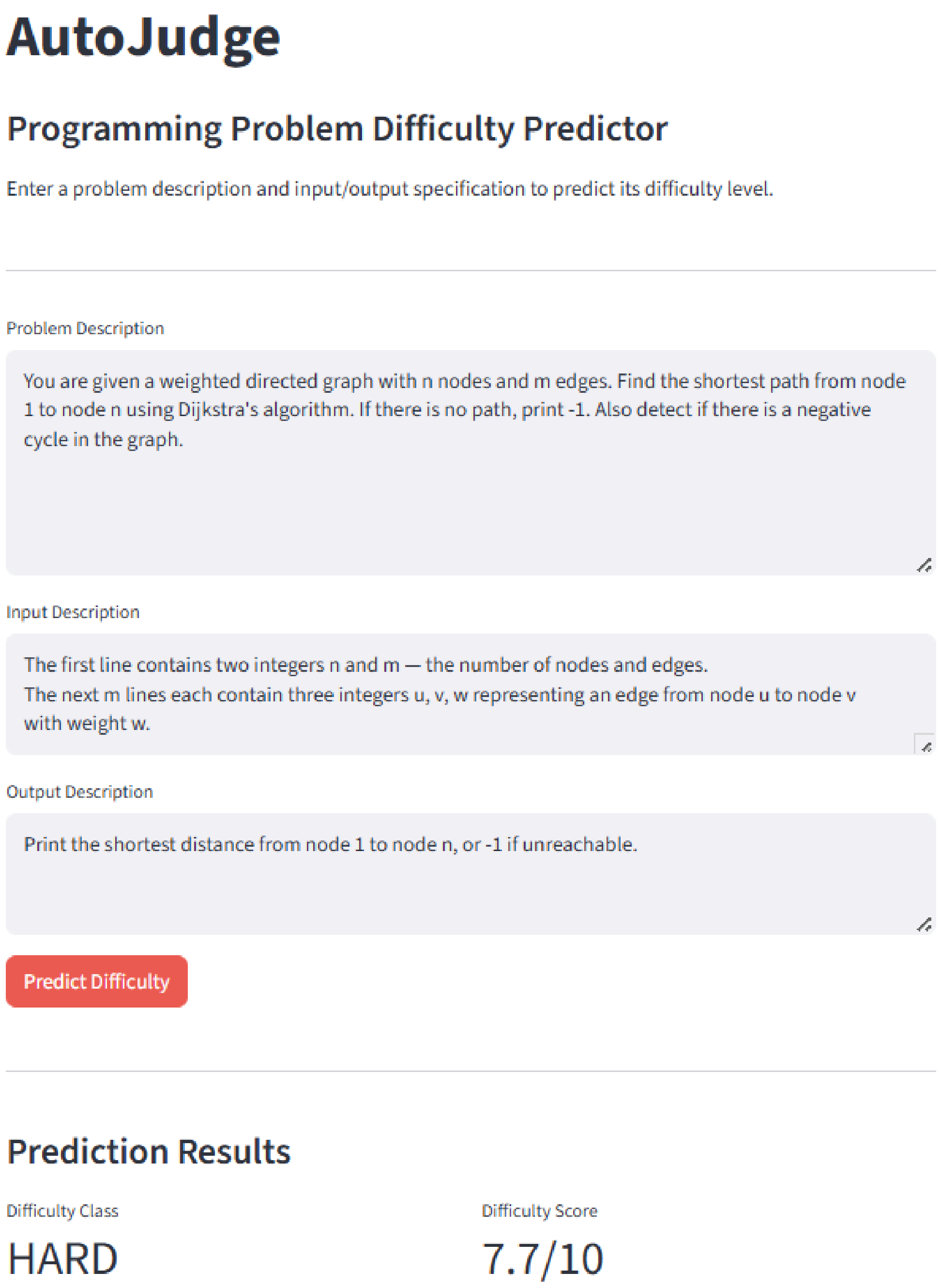
Technology Stack

The web application was built using **Streamlit**, a Python framework that handles both frontend and backend. Other libraries used include:

- Scikit-learn
- XGBoost
- joblib
- NumPy
- SciPy

How It Works

1. The user enters problem description, input format, and output format.
2. Text is cleaned and transformed using the saved TF-IDF vectorizer.
3. Heuristic features are extracted.
4. Models predict the class and score.
5. Results are displayed on the screen.



AutoJudge

Programming Problem Difficulty Predictor

Enter a problem description and input/output specification to predict its difficulty level.

Problem Description

You are given a weighted directed graph with n nodes and m edges. Find the shortest path from node 1 to node n using Dijkstra's algorithm. If there is no path, print -1. Also detect if there is a negative cycle in the graph.

Input Description

The first line contains two integers n and m — the number of nodes and edges.
The next m lines each contain three integers u, v, w representing an edge from node u to node v with weight w.

Output Description

Print the shortest distance from node 1 to node n, or -1 if unreachable.

Prediction Results

Difficulty Class

HARD

Difficulty Score

7.7/10

Conclusion

This project demonstrates the application of machine learning and NLP techniques for programming problem difficulty prediction. The **Random Forest classifier** achieved **52.19% accuracy** for classification, while the **XGBoost regressor** achieved an **MAE of 1.68** for score prediction.

The main challenge is the inherently subjective nature of difficulty labels, what is "medium" for one person may be "hard" for another. The deployed web application allows users to get real-time difficulty estimates for any problem description. Future work could incorporate actual code solutions or submission statistics to better estimate difficulty.

Submitted by:
Manasvi Sawaria
24112072
Chemical Engineering
IIT Roorkee (2nd Year)
Email: manasvi_s@ch.iitr.ac.in
Phone: 8757473685