AutoJudge: Predicting Programming Problem Difficulty

Report (Sohan Awate 23119044)

1.Introduction

Online competitive programming platforms host thousands of problems whose difficulty levels are usually assigned manually. This process is subjective, time-consuming, and may vary across platforms or problem setters. An automated system capable of predicting problem difficulty using textual descriptions can help standardize difficulty labelling and assist problem curators.

This project, **AutoJudge**, aims to predict the difficulty of programming problems using machine learning. The problem is formulated in two ways:

1. **Classification** – predicting the difficulty category (*Easy, Medium, Hard*).
2. **Regression** – predicting a numerical difficulty score in the range 0–10.

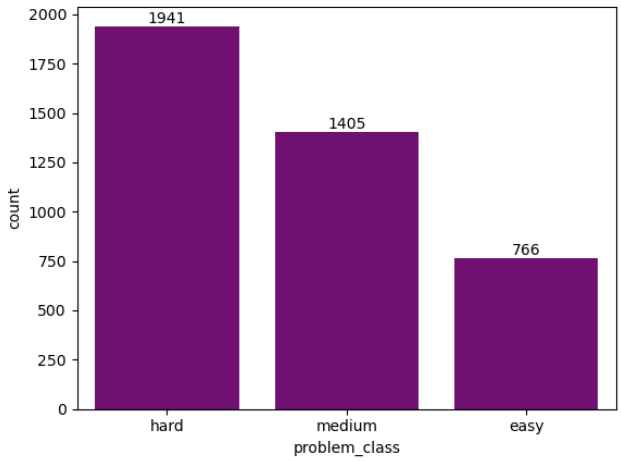
The system relies solely on the textual content of problems and combines natural language processing techniques with feature engineering and ensemble learning. A web-based interface is developed using Streamlit to demonstrate real-time predictions.

2.Exploratory Data Analysis (EDA)

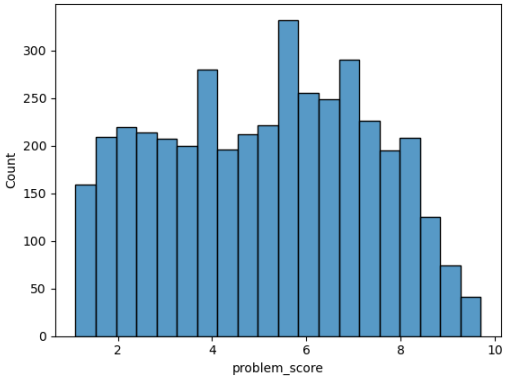
Exploratory Data Analysis was conducted to understand the structure and characteristics of the dataset.

Key analyses included:

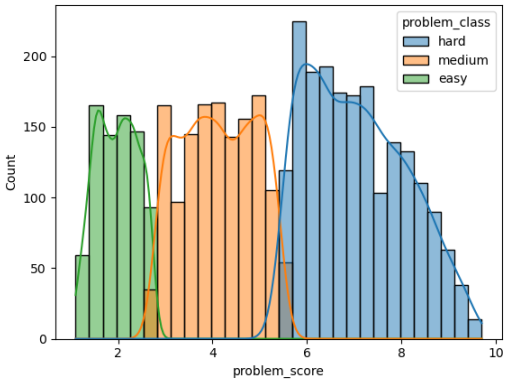
- Frequency of difficulty classes (Easy, Medium, Hard)



- Distribution of difficulty scores



- Overlap between difficulty score ranges across categories



- Length distribution of problem descriptions

The EDA revealed:

- Significant class imbalance, with more problems labeled as *Hard*.
- Overlapping difficulty score ranges between Easy–Medium and Medium–Hard categories.
- Wide variation in text length, indicating varying complexity of problem statements.

These findings justified the use of both classification and regression approaches and highlighted the subjective nature of difficulty labelling instead of just predicting score and hardcoding the classes (like 0-3 easy, 3-6 medium and 6-10 hard).

3. Data Preparation

The dataset contained multiple textual fields such as:

- Title
- Problem description

- Input description
- Output description
- Sample input/output

All text fields were concatenated into a single unified text input to preserve complete contextual information.

Preprocessing steps included:

- Lowercasing text
- Removing special characters and excessive whitespace
- Minimal cleaning to preserve algorithmic and mathematical terms

Heavy linguistic preprocessing such as stopwords removal or lemmatization was avoided to prevent loss of semantic information important for technical texts.

4. Feature Engineering

Feature engineering played a crucial role in improving model performance beyond baseline NLP methods.

4.1 Text-Based Features

- TF-IDF vectorization was applied to the processed text.
- Vocabulary size was limited to the top 10,000 features to balance expressiveness and computational efficiency.

4.2 Engineered Numerical Features

Additional handcrafted features were extracted, including:

- Word count and character count
- Count of numeric tokens
- Maximum numeric value present
- Count of mathematical symbols
- Count of brackets and operators

4.3 Keyword-Based Features

Binary indicators were created for the presence of common algorithmic concepts such as:

- Dynamic Programming
- Graphs
- Greedy algorithms
- Trees, DFS, BFS
- Sorting and searching
- Bit manipulation

These features capture domain-specific cues often correlated with problem difficulty.

5. Baseline Models

As a baseline, models were trained using only TF-IDF features without extensive feature engineering.

5.1 Classification Baseline

The following models were evaluated:

- Logistic Regression
- Decision Tree
- Random Forest
- XGBoost
- K-Nearest Neighbors

Metrics used:

- Accuracy
- Macro F1-score

Baseline performance showed moderate accuracy (~50%), highlighting the difficulty of the task when relying solely on text features.

5.2 Regression Baseline

Regression models included:

- Linear Regression
- Ridge Regression
- Random Forest Regressor

- XGBoost Regressor

Metrics used:

- R^2 Score
- RMSE

Baseline regression models achieved low R^2 scores (0.141), motivating further feature engineering and model tuning.

6. Final Models

To improve performance, engineered features were combined with TF-IDF representations and advanced modelling strategies were applied.

6.1 Ensemble Learning

Classification Ensemble

A **soft voting classifier** was built using:

- Logistic Regression
- Random Forest
- XGBoost

Soft voting was chosen to leverage probabilistic outputs from individual models. Also we gave weight of 1.5 to logistic regression and random forest.

Regression Ensemble

A **voting regressor** was built using:

- Ridge Regression
- Random Forest Regressor
- XGBoost Regressor

This ensemble improved stability compared to individual regressors.

6.2 Hyperparameter Tuning with Optuna

Optuna was used to tune hyperparameters for key models:

- Random Forest
- XGBoost
- Logistic Regression

Optimization objectives included:

- Macro F1-score for classification
- R^2 score for regression

The tuning process helped:

- Improve generalization
- Reduce overfitting
- Balance model complexity

6.3 Inference-Time Calibration

Regression models can produce unbounded outputs. To ensure valid predictions:

- Regression outputs were clipped to the range 0–10 during inference.
- This calibration step ensures interpretability without affecting training.

7. Results

Classification Results

- Accuracy: ~51.15%
- Macro F1-score: ~0.467

Performance reflects the subjective and overlapping nature of difficulty labels.

Regression Results

- R^2 Score: ~0.199
- RMSE: ~1.97

Although the R^2 score is modest, it is reasonable given:

- Subjective target labels
- Text-only input
- High variability in problem difficulty

8. Future Improvements

Several enhancements can be explored:

- Use transformer-based embeddings (BERT, RoBERTa)
- Incorporate user interaction data or submission statistics
- Use difficulty calibration using human feedback
- Expand keyword taxonomy for algorithmic concepts

9. Conclusion

This project demonstrates an end-to-end machine learning pipeline for predicting programming problem difficulty using textual data. By combining NLP techniques, feature engineering, ensemble learning, and hyperparameter optimization, the system achieves reasonable performance on a challenging and subjective task.

The inclusion of a web-based Streamlit interface makes the solution practical and user-friendly. Overall, AutoJudge showcases how data-driven methods can assist in automating complex decision-making processes in educational and competitive programming platforms.