AutoJudge: Predicting Programming Problem Difficulty Using Machine Learning

1. Introduction

In competitive programming platforms and online coding judges, problems are often labeled with difficulty levels such as *Easy*, *Medium*, and *Hard*, and sometimes also assigned a continuous difficulty score. This project, **AutoJudge**, aims to automatically predict the difficulty of programming problems using machine learning techniques. Given a problem statement (title, description, input format, and output format), the system predicts:

1. **Difficulty Class**: Easy / Medium / Hard (classification)
2. **Difficulty Score**: A continuous numerical score (regression)

2. Exploratory Data Analysis (EDA)

Difficulty Class Distribution

The dataset shows a clear class imbalance.Hard problems dominate and Easy problems are the least frequent

This motivated:

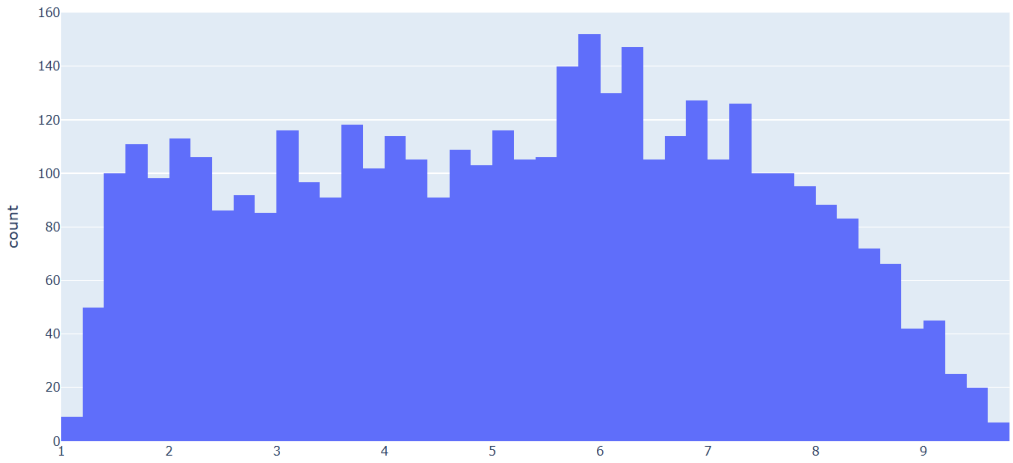
- Use of **Stratified Cross-Validation**
- Optimization of **macro F1-score** instead of accuracy

```
df['problem_class'].value_counts(normalize=True)
```

problem_class	proportion
hard	0.472033
medium	0.341683
easy	0.186284

Difficulty Score Distribution

The difficulty score distribution is approximately uniform with mild concentration in the mid-range.



Mean ≈ 5.1

Standard deviation ≈ 2.1

Range ≈ [1.1, 9.7]

This suggests:

- The regression task contains **high noise**
- Linear models with regularization may generalize better than complex non-linear models

Text Length vs Difficulty

Simple textual statistics such as Character length, Word count show **positive correlation** with difficulty score, but the correlation is moderate (~0.23).

Correlation Matrix

	char_len	word_len	problem_score
char_len	1.000000	0.993434	0.230039
word_len	0.993434	1.000000	0.233438
problem_score	0.230039	0.233438	1.000000

3. Feature Engineering

Feature engineering was a critical component of this project. Features were divided into three categories.

Text Construction

All text fields were merged into a single unified representation:

```
full_text = title + description + input_description + output_description
```

Preprocessing steps:

- Lowercasing
- Whitespace normalization
- Removal of redundant spacing

This unified representation ensures that all relevant information is captured consistently.

Statistical Text Features

The following handcrafted features were extracted:

Character length,
Word count,
Sentence count,
Digit count,
Mathematical symbol count,
Log-transformed lengths,
Constraint operators (`<`, `<=`, `>=`),
Big-O notation hints,
Average word length,
Colon and newline counts

These features capture **structural complexity** of a problem statement.

Keyword-Based Features

Domain-specific keywords were manually curated, covering:

Algorithmic paradigms (DP, Greedy, Graph),
Data structures (Segment Tree, Fenwick Tree, Heap),
Mathematical indicators (Modulo, Probability),
Brute-force hints

For each keyword group:

- Binary presence feature

- Count-based feature

These features inject **expert knowledge** into the model.

TF-IDF Features

TF-IDF (Term Frequency–Inverse Document Frequency) was used to capture semantic information from text.

Key configuration:

- Unigrams + bigrams
- Minimum document frequency threshold
- Maximum feature limit to control dimensionality

TF-IDF transforms text into a high-dimensional sparse vector where:

- Common words are down-weighted
- Rare but informative words are emphasized

This proved especially effective for both classification and regression.

4. Regression Modeling (Difficulty Score)

Model Selection Rationale

Several models were evaluated:

- XGBoost Regressor
- Residual learning pipelines
- Ridge Regression

Due to:

- High label noise
- Near-linear relationships in TF-IDF space
- Better generalization

Ridge Regression with TF-IDF + structured features consistently outperformed tree-based models.

Final Regression Model

- Model: Ridge Regression

- Features:
    - TF-IDF
    - Statistical features
    - Keyword features
- Hyperparameters tuned using **Optuna**
- Metric optimized: **RMSE**

Key observations:

- Regularization reduces overfitting
- Linear model handles sparse, high-dimensional data efficiently

```
Best RMSE: 2.0547407176407386
Best Params: {'alpha': 9942.524780082125, 'fit_intercept': False}
```

5. Classification Modeling (Difficulty Class)

Metric Choice

Due to class imbalance, **macro F1-score** was chosen instead of accuracy.

Logistic Regression with TF-IDF

Logistic Regression was chosen because:

- Extremely effective for sparse text data
- Fast and stable
- Interpretable coefficients

Pipeline:

- TF-IDF + structured features
- StandardScaler (sparse-safe)
- Logistic Regression

Hyperparameters (C, class weighting) were tuned using Optuna.

```
Best macro-F1: 0.4548674007084618
Best params: {'C': 0.00010021197815889274, 'class_weight': 'balanced'}
```

Final Classification Results

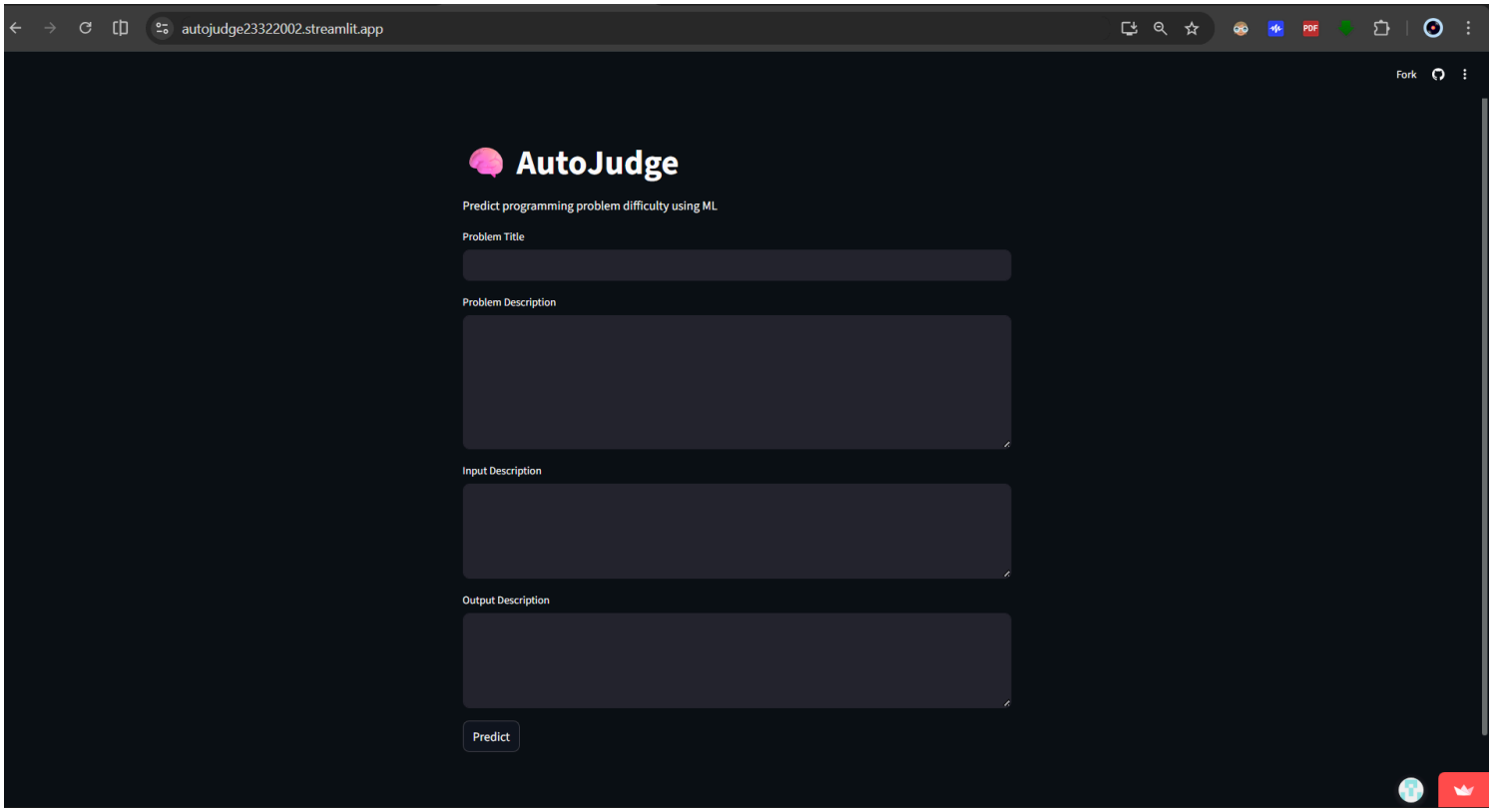
- Macro F1-score improved significantly after adding TF-IDF

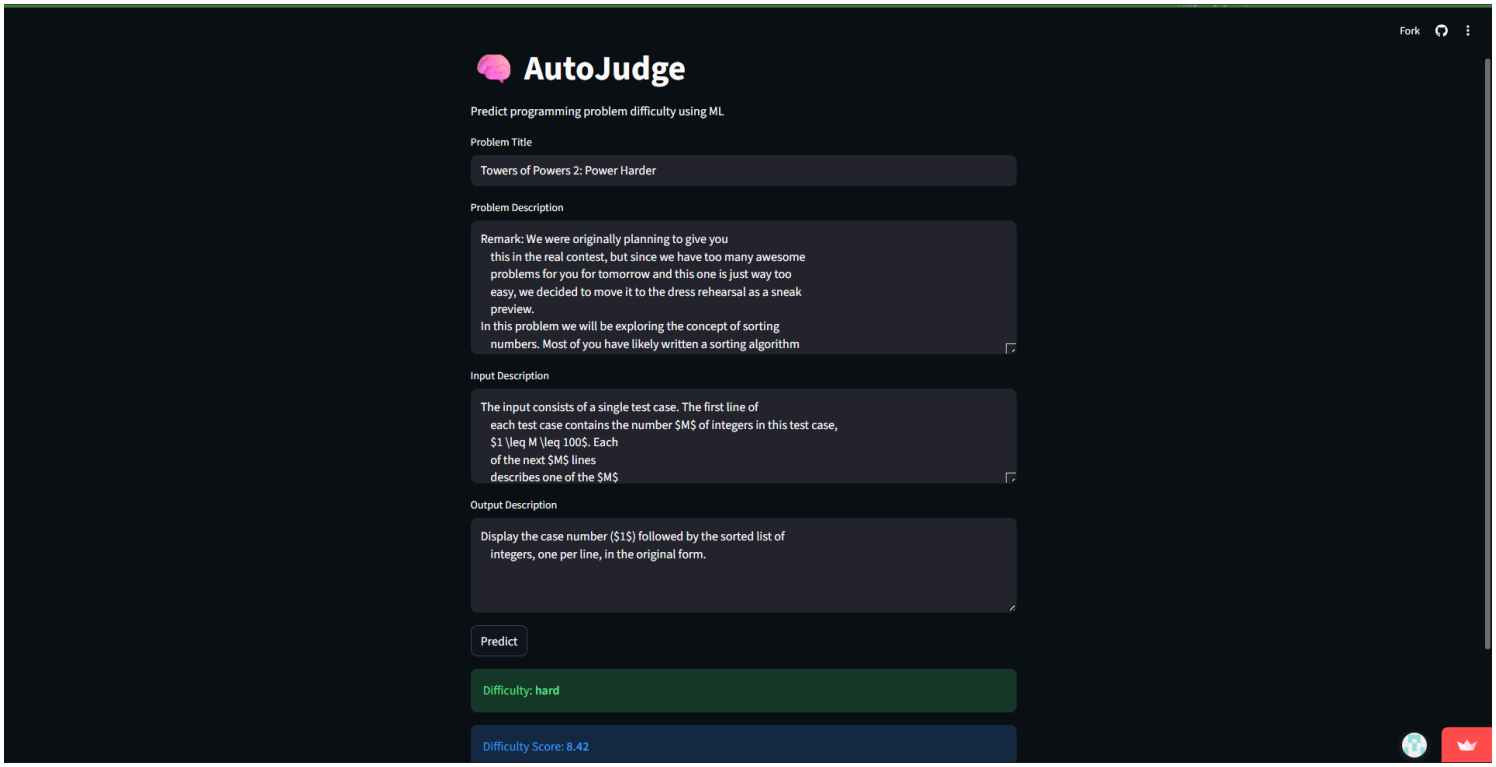
```
              precision    recall  f1-score   support

        easy       0.52      0.37      0.44       153
        hard       0.56      0.64      0.60       389
      medium       0.37      0.35      0.36       281

    accuracy                           0.49       823
   macro avg       0.48      0.46      0.46       823
weighted avg       0.49      0.49      0.49       823

[[ 57  38  58]
 [ 27 250 112]
 [ 25 157  99]]
```

6. Streamlit Application

The deployed application allows users to:

- Input a programming problem
- Instantly receive:
    - Difficulty class
    - Difficulty score


autojudge23322002.streamlit.app
Fork
AutoJudge
Predict programming problem difficulty using ML
Problem Title
Problem Description
Input Description
Output Description
Predict


Fork
AutoJudge
Predict programming problem difficulty using ML
Problem Title
Towers of Powers 2: Power Harder
Problem Description
Remark: We were originally planning to give you this in the real contest, but since we have too many awesome problems for you for tomorrow and this one is just way too easy, we decided to move it to the dress rehearsal as a sneak preview.
In this problem we will be exploring the concept of sorting numbers. Most of you have likely written a sorting algorithm
Input Description
The input consists of a single test case. The first line of each test case contains the number M of integers in this test case, $1 \leq M \leq 100$. Each of the next M lines describes one of the M
Output Description
Display the case number (1) followed by the sorted list of integers, one per line, in the original form.
Predict
Difficulty: hard
Difficulty Score: 8.42

7. Future Improvements

- Add transformer embeddings (BERT) for semantic richness
- Include solution code or constraints as features
- Calibrate class probabilities
- Explainability via SHAP or coefficient analysis