ACM Open Project

Auto Judge: Predicting Programming Problem Difficulty

Awani Soni 23322008, BS-MS Economics, IIT Roorkee

Introduction

Online competitive programming platforms such as Codeforces, CodeChef, and Kattis host thousands of programming problems. These problems are usually assigned a difficulty level (Easy, Medium, Hard) and a numerical difficulty score, which are determined using human judgment and community feedback.
However, manual difficulty assignment is subjective, time-consuming, and inconsistent across platforms. With the increasing availability of problem data, there is a strong motivation to automate difficulty prediction using machine learning techniques.

This project, **Auto Judge**, aims to automatically predict:
- The **difficulty class** (Easy / Medium / Hard)
- The **difficulty score** (numerical)

using **only the textual description** of programming problems. No metadata, submission statistics, or solution code is used.

Problem Statement

The objective of this project is to design and implement a machine learning system that can:
1. Predict the **difficulty category** of a programming problem (classification task)
2. Predict the **numerical difficulty score** of a problem (regression task)

The predictions are based solely on: Problem Description, Input Description, Output Description.
This makes the task challenging, as difficulty must be inferred from **language complexity, constraints, and algorithmic hints** present in text.

Dataset Description

The dataset provided for this project consists of programming problems with labeled difficulty information. Each data sample contains the following fields:

- `title`
- `description`
- `input_description`
- `output_description`
- `problem_class` (Easy / Medium / Hard)
- `problem_score` (numerical value between 1 and 10)

No manual labeling was required, as the dataset already included difficulty labels and scores.
There were **no missing values** in the dataset.

For modeling, irrelevant fields such as problem URL and sample I/O were removed.

Dataset Preprocessing

- **Text Combination:**
  To represent the complete problem statement, the following fields were concatenated:
  - Problem Description
  - Input Description
  - Output Description

This combined text was used as the main input for feature extraction.

- **Text Cleaning**
  The combined text was cleaned using standard NLP preprocessing techniques:
  - Conversion to lowercase
  - Removal of HTML tags
  - Removal of URLs
  - Normalization of whitespace

These steps helped reduce noise and ensure consistent feature extraction.

Feature Extraction

To capture both semantic and structural characteristics of problems, two types of features were used.

- **TF-IDF Features**
  - TF-IDF vectorization was applied on the cleaned text
  - Unigrams and bigrams were used
  - Maximum features: 30,000
  - Minimum document frequency: 5
  - Sublinear term frequency scaling was applied

TF-IDF captures important words and phrases that indicate problem complexity and algorithmic nature.

- **Handcrafted Numeric Features**
  In addition to text embeddings, several handcrafted features were designed:
  Text Statistics
  - Log-transformed text length
  - Log-transformed count of mathematical symbols

  Constraint-Aware Features
  - Presence of constraints
  - Presence of large input sizes (e.g., 10^5, 10^6)
  - Presence of time limit mentions

  Algorithm Keyword Frequencies
  Keyword counts (log-transformed) for major algorithm categories:
  - Dynamic Programming
  - Graph Algorithms
  - Data Structures
  - Mathematics
  - Geometry
  - Strings

- Greedy Techniques

These numeric features help capture difficulty patterns that pure text embeddings may miss. Finally TF-IDF features and numeric features were concatenated into a single feature matrix.

Models Used

- **Classification Models:**
  The following models were evaluated for predicting difficulty class:
  - Logistic Regression
  - Random Forest Classifier
  - Support Vector Machine (LinearSVC)

After cross-validation, **Logistic Regression** was selected as the final classification model due to its stable performance and lower complexity.

- **Regression Models:**
  The following models were tested for predicting difficulty score:
  - Linear Regression
  - Gradient Boosting Regressor

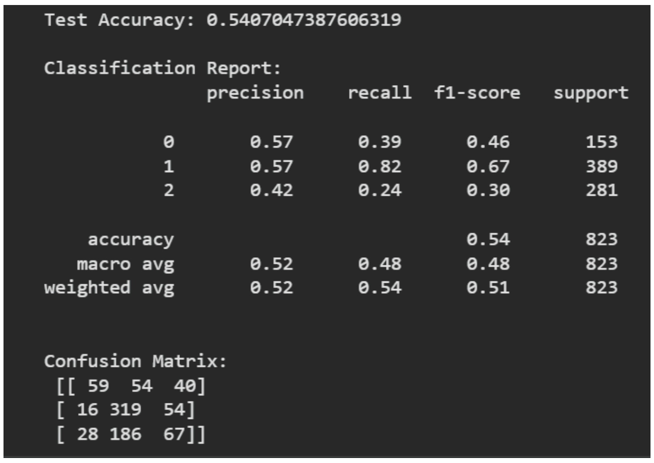
Gradient Boosting Regressor was selected as the final regression model due to its lower RMSE and MAE.
Deep learning models were intentionally not used, as per project guidelines.

Experimental Setup

- Train-test split: 80% training, 20% testing
- Stratified sampling used for classification to maintain class balance
- Cross-validation:
  - 5-fold Stratified CV for classification
  - 5-fold CV for regression
- Hyperparameter Tuning
  - GridSearchCV for Logistic Regression
  - RandomizedSearchCV for Gradient Boosting

Evaluation Results

- **Classification Results:**
  Easy:0, Medium:2, Hard:1



```
Test Accuracy: 0.5407047387606319

Classification Report:
              precision    recall  f1-score   support

           0       0.57      0.39      0.46       153
           1       0.57      0.82      0.67       389
           2       0.42      0.24      0.30       281

    accuracy                           0.54       823
   macro avg       0.52      0.48      0.48       823
weighted avg       0.52      0.54      0.51       823


Confusion Matrix:
 [[ 59  54  40]
 [ 16 319  54]
 [ 28 186  67]]
```

   - Test accuracy: ~54%

A confusion matrix was used to visualize misclassifications across Easy, Medium, and Hard classes, highlighting overlap especially for Medium problems.

Observations:
   - Hard problems were predicted most reliably
   - Medium problems were hardest to classify due to overlap with Easy and Hard
   - Easy problems were sometimes confused with Medium

A confusion matrix was used to analyze misclassifications.

- **Regression Results:**
   - Test RMSE: ~2.01
   - Test MAE: ~1.68

Since Gradient Boosting averages multiple weak learners, predicted scores are slightly smoothed. Exact score matching is not expected in NLP-based regression tasks.

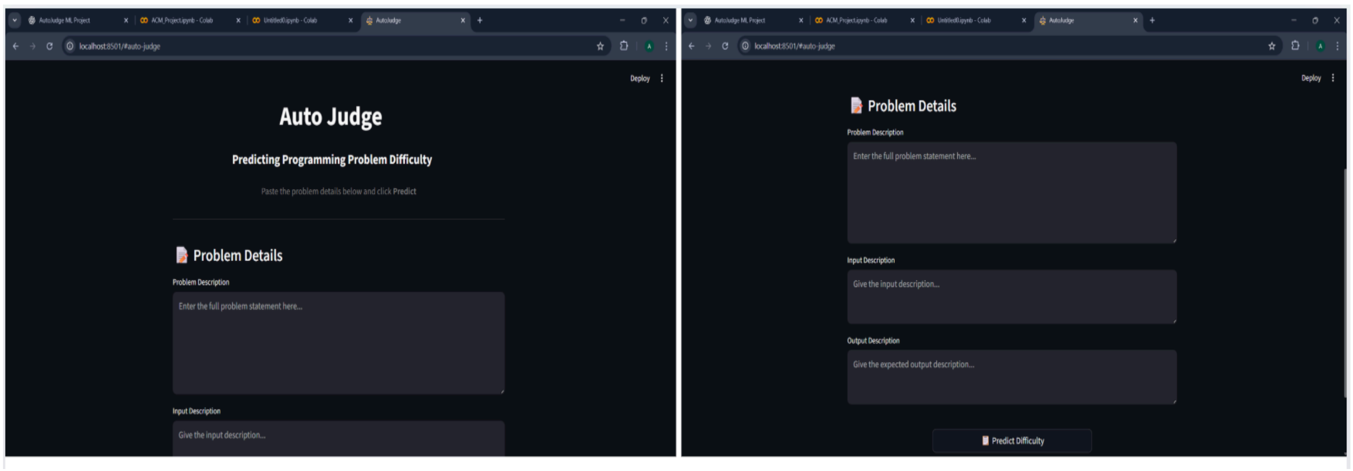
Web Interface

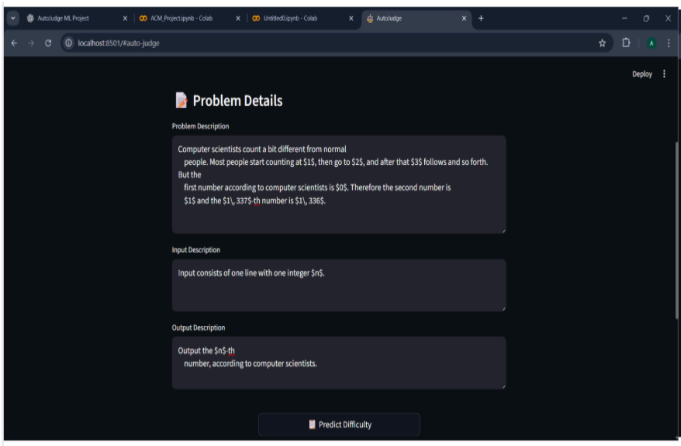
A **Streamlit-based web application** was developed to demonstrate the system.

Workflow

1. User pastes:
   - Problem description
   - Input description
   - Output description
2. User clicks **Predict**
3. The app:
   - Preprocesses the text
   - Extracts TF-IDF and numeric features
   - Applies trained classification and regression models
4. Displays:
   - Predicted difficulty class
   - Predicted numerical difficulty score

The application runs **locally**, and no deployment or database is required. Screenshots of the interface and sample predictions are included in the repository.


localhost:8501/#auto-judge
Deploy
Auto Judge
Predicting Programming Problem Difficulty
Paste the problem details below and click Predict
Problem Details
Problem Description
Enter the full problem statement here...
Input Description
Give the input description...
Output Description
Give the expected output description...
Predict Difficulty



Problem Details

Problem Description

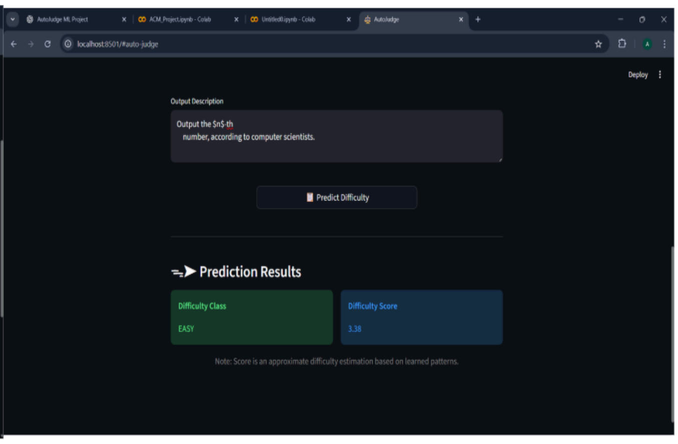
```
Computer scientists count a bit different from normal
   people. Most people start counting at $1$, then go to $2$, and after that $3$ follows and so forth.
But the
   first number according to computer scientists is $0$. Therefore the second number is
   $1$ and the $1\, 337$-th number is $1\, 336$.
```

Input Description

```
Input consists of one line with one integer $n$.
```

Output Description

```
Output the $n$-th
   number, according to computer scientists.
```



Output Description

```
Output the $n$-th
   number, according to computer scientists.
```

Prediction Results

Difficulty Class

EASY

Difficulty Score

3.38

Note: Score is an approximate difficulty estimation based on learned patterns.

Conclusion

This project demonstrates that **programming problem difficulty can be reasonably predicted using only textual information**. Despite moderate accuracy, the model captures meaningful patterns related to algorithmic complexity and constraints.

Key takeaways:

- NLP-based approaches can assist in automating difficulty labeling.
- Handcrafted features significantly improve prediction quality.
- Medium-level problems remain challenging due to overlapping characteristics.

Future work may include:

- Using transformer-based embeddings.
- Incorporating problem constraints more explicitly.
- Platform-specific model tuning.