Auto Judge – Open Project by ACM

~Himesh Kushwaha, 24410011, 2nd Yr Mechanical

1.Introduction

Competitive programming platforms host a large number of algorithmic problems that are typically categorized into difficulty levels such as *Easy*, *Medium*, and *Hard*. In addition to categorical labels, many platforms also assign a numerical difficulty score to each problem.

However, difficulty assignment is usually performed manually or inferred from user submissions and contest statistics. As a result, the process is often subjective and may vary across platforms.

This project, **AutoJudge**, provides an ML baked interface that predicts the difficulty of competitive programming problems using only textual information. The system performs two tasks: (i) classification of problems into *Easy*, *Medium*, or *Hard*, and (ii) regression to estimate a numerical difficulty score. The predictions are based solely on the problem statement, input description, and output description, without relying on solution code or metadata.

2.Dataset Description

The dataset used in this project is provided in **JSON Lines (.jsonl)** format, where each line corresponds to a single programming problem.

Each data sample contains the following fields: problem title, problem description, input description, output description, difficulty class label, and a numerical difficulty score.

The dataset was loaded into the Google Colab environment using the Pandas library and converted into a structured DataFrame for further processing. Preliminary inspection of the dataset confirmed the presence of valid labels for both classification and regression tasks, making it suitable for supervised machine learning.

3. Dataset Inspection and Problem Structure

Before applying any preprocessing or machine learning techniques, an initial inspection of the dataset was performed to understand the structure of individual problem records. A single programming problem was extracted from the JSON Lines file and examined in detail, including its title, difficulty class, numerical difficulty score, and full textual content (problem description, input description, and output description).

Sample input–output examples were also inspected and were observed to appear in varying formats across problems. This exploratory step helped in understanding the length, complexity, and variability of problem statements, and informed the design of subsequent text preprocessing and feature engineering steps.

4.Text Combination and Feature Extraction

After inspecting individual problem records, all textual components of each problem—including the title, problem description, input description, and output description—were combined into a single unified text representation. This consolidated text serves as the primary input for subsequent feature extraction and modelling stages.

Basic quantitative features were then extracted from the combined text, such as total character length, word count, and sentence count. In addition, simple counts related to variable usage and numeric values were computed to reflect the presence of constraints and symbolic expressions commonly found in competitive programming problems. These initial features provide a lightweight yet informative numerical representation of problem complexity.

5.Trigger Word–Based Semantic Features

To capture higher-level semantic complexity present in programming problem statements, trigger word groups were manually defined based on commonly used natural language patterns. These groups represent reasoning concepts such as repetition, ordering, optimization, conditional logic, state maintenance, and quantification. Unlike algorithmic keywords, these terms frequently appear in problem descriptions and provide meaningful cues about problem structure.

For each problem, the frequency of trigger words belonging to each group was counted from the combined textual representation. These trigger-based features were then merged with previously extracted length-based and numeric features to form a unified numerical representation of problem complexity.

6.Dataset-Wide Feature Preparation

After validating handcrafted features on individual problems, the feature extraction process was scaled to the entire dataset. For each problem, all engineered features were computed and combined with the corresponding difficulty class and numerical score, resulting in a structured feature table where each row represents a single problem.

Basic exploratory checks were performed on the generated dataset to verify its size, class distribution, and feature statistics. Following this, the raw dataset was reloaded to construct a clean modeling table consisting of a unified text representation and target labels. This table serves as the primary input for subsequent text-based machine learning models.

7. Text-Based Baseline Models

In the first modeling phase, a text-only baseline was established using TF-IDF vectorization. All problem statements were converted into numerical representations by applying TF-IDF on the combined textual input. The dataset was then split into training and testing sets using a stratified split to preserve the distribution of difficulty classes.

For the classification task, a Logistic Regression model was trained to predict difficulty categories (*Easy*, *Medium*, *Hard*) from TF-IDF features. Model performance was evaluated using accuracy, precision, recall, F1-score, and a confusion matrix.

For the regression task, a Linear Regression model was trained using the same TF-IDF features to predict the numerical difficulty score. Regression performance was evaluated using Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE).

This text-only setup serves as a baseline against which more advanced feature-enhanced models can be compared

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_class_test, y_pred_class)
```

```
array([[ 33,  74,  46],
       [ 17, 292,  80],
       [ 18, 192,  71]])
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
import numpy as np

mae = mean_absolute_error(y_score_test, y_pred_score)
rmse = np.sqrt(mean_squared_error(y_score_test, y_pred_score))

mae, rmse
```

```
(2.412295453633488, np.float64(3.0462931275714182))
```

```
from sklearn.metrics import classification_report

print(classification_report(y_class_test, y_pred_class))
```

```
              precision    recall  f1-score   support

        easy       0.49      0.22      0.30       153
        hard       0.52      0.75      0.62       389
      medium       0.36      0.25      0.30       281

    accuracy                           0.48       823
   macro avg       0.46      0.41      0.40       823
weighted avg       0.46      0.48      0.45       823
```

```
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_class_test, y_pred_class)
accuracy
```

```
0.48116646415552855
```

8.Enhanced Classification Using Combined Features

8.1 logistic Regression with Combined Features

To improve upon the text-only baseline, handcrafted textual features capturing structural and semantic characteristics were incorporated into the model. These features included text length statistics, numeric and variable counts, and trigger word frequencies reflecting logical patterns in problem statements.

The handcrafted features were standardized and concatenated with TF-IDF vectors to form a combined feature representation. A Logistic Regression classifier was then trained using this enriched feature space. Model performance was evaluated using the same classification metrics as the baseline experiment.

Comparing the results of the baseline and enhanced models highlights the benefit of incorporating handcrafted features alongside TF-IDF representations, demonstrating improved modeling of problem complexity beyond raw text alone.

```
from sklearn.metrics import accuracy_score

accuracy_combined = accuracy_score(y_class_test, y_pred_class_combined)
accuracy_combined
```

```
0.5103280680437424
```

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_class_test, y_pred_class_combined)
```

```
array([[ 50,  55,  48],
       [ 22, 287,  80],
       [ 22, 176,  83]])
```

```
from sklearn.metrics import classification_report

print(classification_report(y_class_test, y_pred_class_combined))
```

```
              precision    recall  f1-score   support

        easy       0.53      0.33      0.40       153
        hard       0.55      0.74      0.63       389
      medium       0.39      0.30      0.34       281

    accuracy                           0.51       823
   macro avg       0.49      0.45      0.46       823
weighted avg       0.50      0.51      0.49       823
```

8.1 Random Forest Classification

To explore non-linear decision boundaries, a **Random Forest** classifier was trained on the same combined feature set. The performance of both models was evaluated using accuracy, precision, recall, F1-score, and confusion matrices. This comparison provides insight into the effectiveness of linear versus ensemble-based approaches for difficulty classification.

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_class_test, y_pred_rf)
```

```
array([[ 46,  78,  29],
       [ 10, 338,  41],
       [ 21, 212,  48]])
```

```
from sklearn.metrics import accuracy_score

rf_accuracy = accuracy_score(y_class_test, y_pred_rf)
rf_accuracy
```

```
0.5249088699878494
```

```
from sklearn.metrics import classification_report

print(classification_report(y_class_test, y_pred_rf))
```

```
              precision    recall  f1-score   support

        easy       0.60      0.30      0.40       153
        hard       0.54      0.87      0.66       389
      medium       0.41      0.17      0.24       281

    accuracy                           0.52       823
   macro avg       0.51      0.45      0.44       823
weighted avg       0.50      0.52      0.47       823
```

8.3 Gradient Boosting–Based Classification (XGBoost)

To further evaluate the effectiveness of ensemble learning, an XGBoost classifier was trained using the combined TF-IDF and handcrafted feature representation. XGBoost employs gradient boosting, allowing it to model complex non-linear interactions between features more effectively than linear models.

Class labels were encoded numerically to satisfy model requirements, and the classifier was trained using the same train–test split as previous experiments. Model performance was evaluated using accuracy, precision, recall, F1-score, and confusion matrices. The results provide a comparative perspective on boosting-based approaches relative to linear and bagging-based classifiers.

```
from sklearn.metrics import classification_report

print(classification_report(y_class_test, y_pred_xgb))
```

```
              precision    recall  f1-score   support

        easy       0.48      0.34      0.40       153
        hard       0.55      0.71      0.62       389
      medium       0.36      0.27      0.31       281

    accuracy                           0.49       823
   macro avg       0.46      0.44      0.44       823
weighted avg       0.47      0.49      0.47       823
```

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_class_test, y_pred_xgb)
```

```
array([[ 52,  52,  49],
       [ 25, 276,  88],
       [ 32, 173,  76]])
```

```
from sklearn.metrics import accuracy_score

xgb_accuracy = accuracy_score(y_class_test, y_pred_xgb)
xgb_accuracy
```

```
0.4908869987849332
```

9.Threshold Selection for Difficulty Classification

Initial Quantile-Based Thresholding (Exploratory Step)

During the early stages of the project, a quantile-based approach was explored to convert the continuous difficulty score predicted by the regression model into discrete difficulty classes.

Specifically, the 33rd and 66th percentiles of the predicted difficulty scores were used as provisional thresholds to divide problems into *Easy*, *Medium*, and *Hard* categories.

This approach was chosen as an **unsupervised and data-driven baseline**, requiring no prior assumptions about score ranges.

Refinement

After the initial quantile-based thresholding step, fixed score boundaries were refined to better align with the labeled difficulty classes. This was done by examining how numerical difficulty scores were distributed across *Easy*, *Medium*, and *Hard* problems and evaluating candidate thresholds against ground-truth labels.

Multiple threshold pairs were tested by mapping predicted regression scores to difficulty classes and measuring classification accuracy. The thresholds **2.7** and **5.4** provided the best alignment with the dataset's difficulty labels and achieved the highest accuracy among the tested configurations.

10. Web Interface and Sample Predictions

10.1 Web Interface Description

To make the AutoJudge system accessible and easy to use, a lightweight web interface was developed using **Streamlit**.

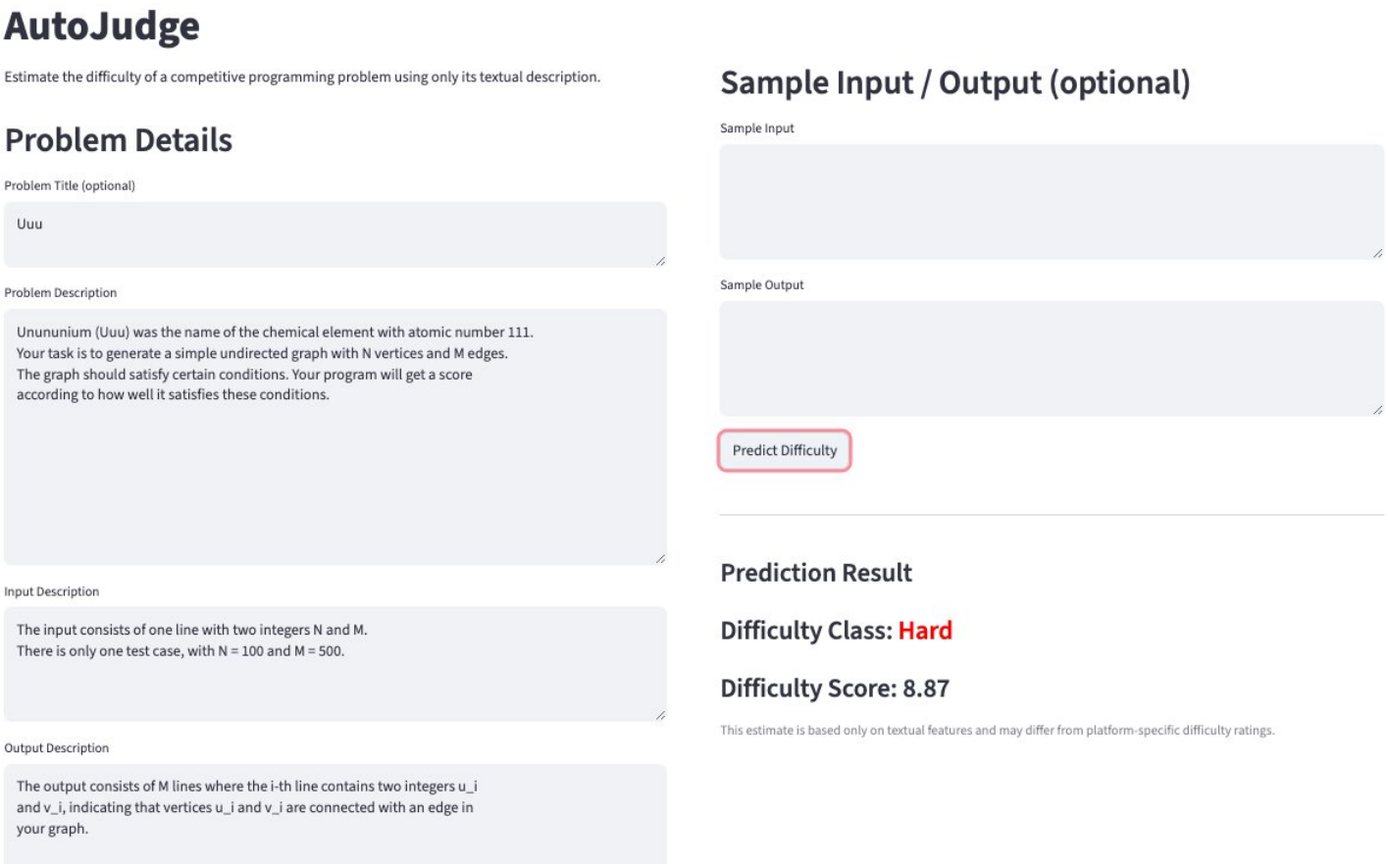
The web application provides multiple input fields where users can enter different components of a programming problem. These include the problem title (optional), problem description, input description, and output description. Additionally, optional fields for sample input and sample output are provided to enrich the textual context when available.

Once the user enters the problem details, they can click the **"Predict Difficulty"** button. Internally, all provided text fields are safely cleaned and concatenated into a single text input. This combined text is then transformed using a pre-trained **TF-IDF vectorizer**, and the resulting feature vector is passed to a trained **linear regression model**.

The regression model predicts a numerical difficulty score. This score is then mapped to a difficulty class using predefined thresholds:

The predicted difficulty class is displayed using color-coded text (green for Easy, orange for Medium, and red for Hard), along with the predicted numerical difficulty score. A short disclaimer is also shown to clarify that predictions are based solely on textual features and may differ from platform-specific ratings.

10.2 Sample Predictions



11.Conclusion

Classification Models

Model	Features Used	Accuracy	Key Observation
Logistic Regression	TF-IDF	~0.48	Baseline model, Hard class detected well
Logistic Regression	TF-IDF + Handcrafted	~0.51	Small improvement, limited by linearity
Random Forest	TF-IDF + Handcrafted	**~0.52**	Best classification accuracy, Medium still weak
XGBoost	TF-IDF + Handcrafted	~0.49	No improvement due to label ambiguity

Regression Models

Model	Features Used	MAE	RMSE	Observation
Linear Regression	TF-IDF	~2.41	~3.05	Stable predictions, good relative difficulty estimation

Regression-to-Classification Experiments

Method	Threshold Strategy	Train Accuracy	Test Accuracy	Conclusion
Regression → Class	Quantile-based	–	~0.43	Misaligned with true labels
Regression → Class	Supervised thresholds	~0.99	~0.43	Severe overfitting, not generalizable

In this project, we investigated the feasibility of predicting competitive programming problem difficulty using only textual information. Multiple machine learning approaches were explored, including linear models, ensemble methods, and gradient-boosted models, combined with TF-IDF representations and handcrafted structural features.

Experimental results showed that **difficulty classification into discrete categories (Easy/Medium/Hard) consistently plateaued around 50–52% accuracy**, regardless of

model complexity. This indicates an inherent ambiguity in difficulty labels when inferred solely from problem statements, especially for the Medium class.

In contrast, **difficulty score prediction via regression demonstrated more stable and meaningful performance**, with relatively low prediction error and better preservation of relative difficulty ordering. Based on this empirical evidence, the final system prioritizes **regression-based difficulty estimation**, with class labels treated as secondary, approximate interpretations rather than strict predictions.

Overall, the findings highlight that competitive programming difficulty is better modeled as a **continuous quantity rather than a rigid categorical label** when only textual data is available.