

AutoJudge: Predicting Programming Problems Difficulty

Submitted by: Shivam Agarwal (Enrollment Number: 24115134)

Abstract

Online competitive programming platforms host a large number of programming problems that are categorized into difficulty levels such as Easy, Medium, and Hard. Assigning these difficulty levels manually is subjective, time-consuming, and inconsistent across platforms. To address this issue, this project proposes **AutoJudge**, a machine learning-based system that automatically predicts the difficulty of programming problems using textual descriptions.

The system utilizes natural language processing techniques and feature engineering to extract meaningful representations from problem statements. A **Logistic Regression model** is employed for difficulty classification, while a **Random Forest Regressor** predicts a numerical difficulty score. The models are evaluated using accuracy, confusion matrix, MAE, and RMSE. Additionally, a web-based user interface is developed using **Streamlit** to provide real-time predictions. Experimental results demonstrate that the proposed system achieves reasonable performance for an academic-level dataset and provides a complete end-to-end solution for difficulty prediction.

1. Introduction

1.1 Problem statement

Programming platforms such as Codeforces, CodeChef, and LeetCode categorize problems into difficulty levels to guide learners and participants. However, assigning difficulty levels is typically done manually by problem setters or platform administrators. This process is subjective and may vary depending on the experience and perspective of the evaluator. Moreover, as the number of problems increases, manual classification becomes inefficient and error-prone.

The goal of this project is to build an automated system that can predict the difficulty of programming problems using machine learning techniques based solely on the textual information of the problem statement.

1.2 Objectives

The main objectives of this project are:

- To predict the difficulty class (Easy, Medium, Hard) of a programming problem.
- To predict a numerical difficulty score using regression.
- To apply text preprocessing and feature engineering techniques.
- To evaluate classification and regression models using appropriate metrics.
- To deploy the trained models through an interactive web interface.

2. Dataset description

2.1 Dataset Source

The dataset used in this project was provided by the college. The data is stored in JSONL format, where each line represents one programming problem.

2.2 Dataset Structure

Each problem contains the following fields:

title: Title of the problem

description: Full problem Description

input_description: Input format

output_description: Output format

sample_io: Sample input and output

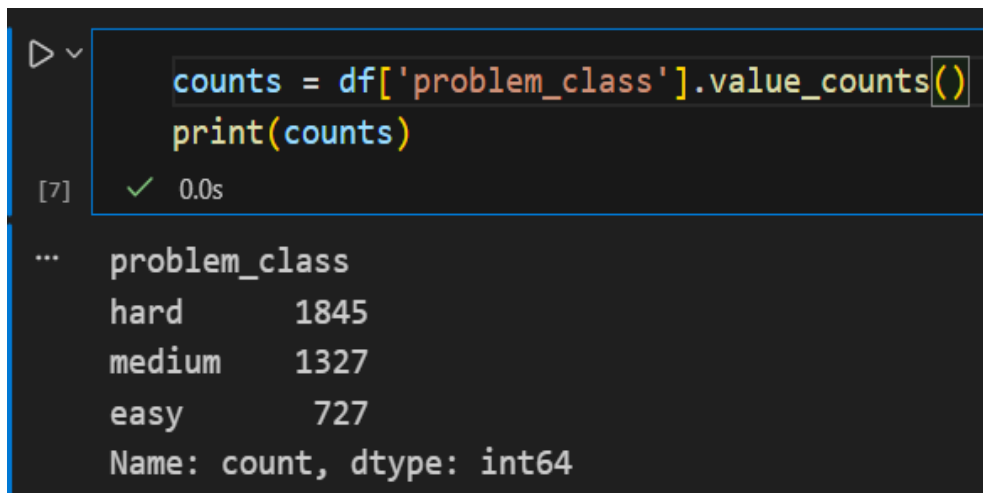
problem_class: Difficulty class(Easy/Medium/Hard)

problem_score: Numerical Difficulty Score(ranging from 1 to 10)

url: Problem source link

2.3 Dataset distribution

The dataset exhibits **class imbalance** as the count of easy problems is significantly lower than the count of hard/medium problems. This imbalance affects classification performance and is discussed later in the results section.



```
counts = df['problem_class'].value_counts()
print(counts)
```

[7] ✓ 0.0s

```
... problem_class
hard      1845
medium    1327
easy       727
Name: count, dtype: int64
```

3. Data Preprocessing

3.1 Text Cleaning

The textual fields were cleaned using the following steps:

- Conversion to lowercase
- Removal of newline and tab characters
- Removal of special characters and punctuation
- Normalization of whitespace

This ensured consistent and noise-free textual data.

3.2 Handling missing values

Although `df.isnull()` reported zero missing values, several columns contained **empty strings**, which are not detected as null values by default. These empty strings were explicitly handled during preprocessing.

- Empty strings were temporarily replaced with NaN.
- The total number of rows containing missing values was very less in comparison to the dataset containing more than 4000 problems, hence those rows were dropped.

```
df.isnull().sum()
[3]  ✓ 0.0s
... title 0
description 81
input_description 120
output_description 131
sample_io 0
problem_class 0
problem_score 0
url 0
dtype: int64
```

3.3 Text Consolidation

Multiple text columns (title, description, input_description, output_description) were combined into a single column named `full_text`. This ensured that all relevant information about a problem was captured in one unified text field.

4. Feature Engineering

Feature engineering played a crucial role in improving model performance.

4.1 TF-IDF Features

The primary text representation was created using **TF-IDF (Term Frequency–Inverse Document Frequency)**.

Configuration:

- Maximum features: 8000
- N-gram range: (1, 2)
- Stop words: English
- Minimum document frequency: 3

TF-IDF captures the importance of words while reducing the influence of commonly occurring terms.

4.2 Hand Crafted Numerical Features

In addition to TF-IDF, several numerical features were manually engineered:

- `text_length`: Length of the combined text.
- `num_math_symbols`: Count of mathematical symbols(+, -, *, /, <, >, =).
- `kw_dp`: Count of keyword “dp”.
- `kw_graph`: Count of keyword “graph”.
- `kw_tree`: Count of keyword “tree”.
- `kw_greedy`: Count of keyword “greedy”.
- `kw_math`: Count of keyword “math”.
- `kw_string`: Count of keyword “string”.

These features help capture problem complexity beyond raw text.

4.3 Final Feature Matrix

Sparse TF-IDF features were horizontally stacked with numerical features using `scipy.sparse.hstack` to form the final input matrix used for both classification and regression models.

5. Models Used

5.1 Classification Model

Model: Logistic Regression

Reason for selection:

- Simplicity and interpretability
- Stable performance on sparse text features
- Better performance than Linear SVM on this dataset

The classification model predicts difficulty classes: Easy, Medium, and Hard.

5.2 Regression Model

Model: Random Forest Regressor

Reason for selection:

- Handles non-linear relationships
- Robust to noise and feature interactions
- Performs well with mixed feature types

The regression model predicts a continuous difficulty score.

6. Experimental Setup

- Train-test split: 80% training, 20% testing
- Libraries used: scikit-learn, pandas, numpy, scipy
- Evaluation metrics:
 - Classification: Accuracy, Confusion Matrix
 - Regression: MAE, RMSE

7. Results and Evaluation

7.1 Classification Results

The classification accuracy and the confusion matrix were:

```
Accuracy: 0.47435897435897434
[[ 71  40  34]
 [ 47 191 131]
 [ 40 118 108]]
```

The classifier achieves an accuracy of approximately **47%**, as shown by the confusion matrix. The model predicts the Medium class with relatively higher confidence, while misclassifications are observed between Easy–Medium and Medium–Hard classes. Although the dataset is skewed toward the Hard category, the prediction bias is primarily due to semantic overlap in problem descriptions and the limitations of text-only features.

7.2 Regression Results

The MAE and RMSE were:

```
... MAE: 1.7266542536970804
    RMSE: 2.047922227585502
```

The regression model demonstrates stable performance and is effective for estimating relative difficulty levels.

7.3 Discussion

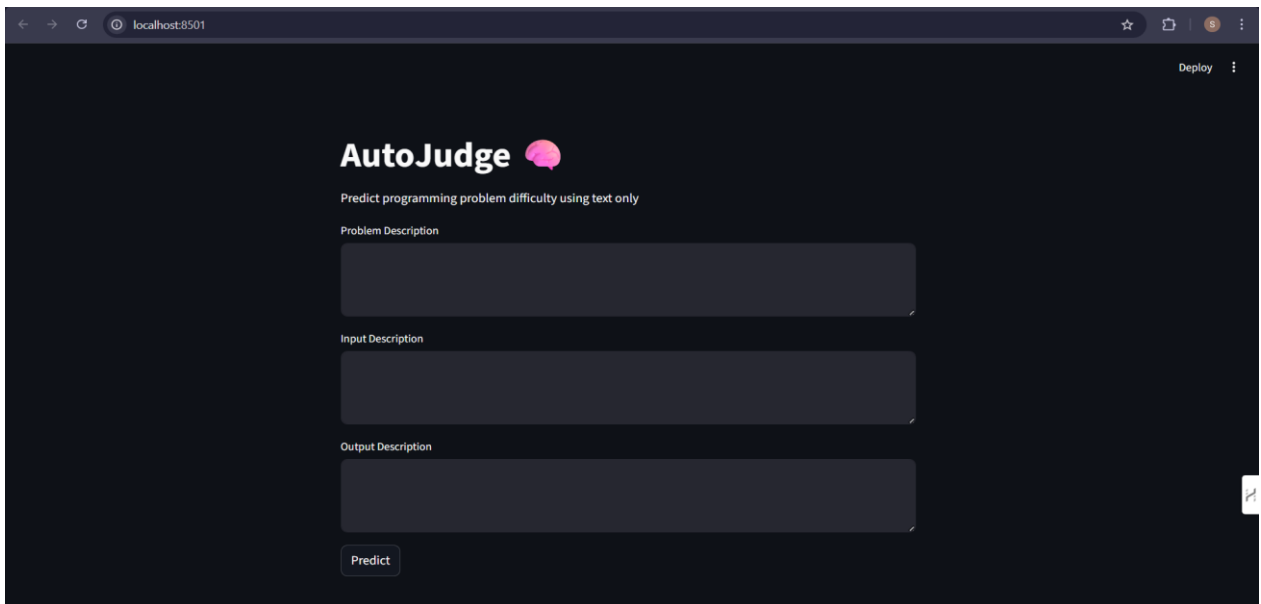
Predicting problem difficulty using text alone is inherently challenging and subjective. Despite these limitations, the models achieved reasonable performance. The regression model performed better than classification due to continuous output representation.

8. Web Interface

8.1 Streamlit Application

A user-friendly web interface was developed using **Streamlit**. The interface allows users to:

- Enter problem descriptions
- Submit inputs for prediction
- View predicted difficulty class and score



The screenshot shows a web browser window with the address bar displaying 'localhost:8501'. The web application, titled 'AutoJudge' with a brain icon, has the subtitle 'Predict programming problem difficulty using text only'. It features three text input fields labeled 'Problem Description', 'Input Description', and 'Output Description'. A 'Predict' button is positioned below the 'Output Description' field. In the top right corner, there is a 'Deploy' button and a menu icon. The interface is dark-themed.

8.2 Sample Predictions

Sample predictions demonstrate correct handling of different problem types and reasonable output consistency

Predict programming problem difficulty using text only

Deploy

Problem Description

eccentrics use Manhattan distance to measure this, and is defined as $\sum |x_i - A_i| + |y_i - y_c|$ over all other people's squares (x_i, y_i) . The eccentrics now want your help in placing their houses optimally, so that the sum of the happiness they experience is as high as possible. Can you help them?

Input Description

The input consists of 10 test cases, which are described below.

Output Description

Print 10 lines with the positions of the houses. Each line should contain two numbers: first the row for the house (between 1 and 5), then the column (between 1 and 5). Two houses may not be placed at the same position.

Predict

Predicted Difficulty Class: hard

Predicted Difficulty Score: 7.25

9. Conclusion

This project successfully implemented an end-to-end machine learning system for predicting programming problem difficulty. It integrates data preprocessing, feature engineering, classification, regression, and web deployment into a single pipeline.