Vishakha Sinha 24114104 2nd yr CSE student

# AUTOJUDGE

***PREDICTING PROGRAMMING PROBLEMS DIFFICULTY***

## Problem Statement

Online competitive programming platforms like Codeforces , Codechef , Kattis etc. categorize problems into difficulty levels such as *Easy, Medium,* and *Hard*, often accompanied by a numerical difficulty score. This classification process typically relies on human judgment and user feedback, making it subjective and time-consuming.

The objective of this project is to develop **AutoJudge**, a machine learning–based system that automatically predicts:

- The **difficulty class** of a programming problem (classification)
- The **difficulty score** of a programming problem (regression)

using **only the textual description** of the problem.

## Dataset used

The dataset consists of various programming problems obtained through different platforms where each problem has a particular **difficulty class and difficulty score** respectively . Each data sample in the dataset includes the following fields:

- Problem Title
- Problem description
- Input description
- Output description
- Problem class (Easy / Medium / Hard)
- Problem score (numerical value)
- URL link of the problem

## Dataset Characteristics

The dataset is **text-heavy in nature**, consisting primarily of descriptive fields related to programming problems. No external metadata or handcrafted labels beyond the provided difficulty annotations were used.

The dataset is provided in **JSON Lines (.jsonl) format**, where each line represents a single programming problem encoded as a JSON object. This format enables efficient storage and parsing of large textual datasets and allows each problem instance to be processed independently.

Only **textual information extracted from the JSON file** was utilized for both classification and regression tasks, without incorporating any additional features or external sources.

## Data Preprocessing

Data preprocessing was a crucial step to transform the raw dataset into a suitable format for feature extraction and model training.

The dataset was loaded using the **Pandas** library.  Specifically, the *read_json* function with the *lines=True* parameter was used to correctly parse the JSON Lines file into a **tabular DataFrame** structure. This allowed convenient access to individual fields such as the title, problem description, input description, and output description

Once loaded, the following preprocessing steps were applied:

- **Handling missing values:** Any missing textual fields were handled by replacing them with empty strings to ensure uniform input length and prevent errors during text processing.
- **Text consolidation:** Multiple textual fields (title, description, input description, and output description) were concatenated into a single combined text feature. This ensured that all available contextual information about a problem was utilized by the models.
- **Text normalization:** The combined text was normalized by converting all characters to lowercase, removing newline and tab characters, and reducing multiple consecutive spaces into a single space. These steps helped reduce noise and improve the consistency of the textual data.
- **Final cleaned representation:** The resulting cleaned and consolidated text served as the final input for feature engineering techniques such as TF-IDF vectorization.

These preprocessing steps ensured that the raw JSON-based dataset was transformed into a clean, structured, and machine-learning-ready format suitable for both classification and regression tasks.

## Feature Engineering

Feature engineering helps to transform unstructured textual data into numerical representations that can be effectively processed by machine learning algorithm.

1. **Structural and Length-Based Numerical Features -** These help the model understand how hard the problem "looks".

   - **Text length:** total number of characters in the combined text.
   - **Word count:** total number of words in the problem description.
   - **Average word length:** ratio of text length to word count, used as a proxy for linguistic complexity.

These features help distinguish short, simple problems from longer and more detailed problem descriptions, which often correspond to higher difficulty levels.

Harder problems often tend to be more descriptive and have a higher word count.

2. **Symbol-based Complexity Features** - Programming problems often include mathematical expressions and constraints. To capture this aspect, I kept a count of the mathematical symbols used.

   - **Comparison symbol count:** number of occurrences of symbols such as <, >, and =.
   - **Arithmetic symbol count:** number of occurrences of arithmetic operators such as +, -, *, /, and %.

These features helped to look into the **logical and computational complexity** of problem.

3. **Keyword-based features** - Certain words strongly correlate with **algorithmic complexity**. This feature helps the ML model understand that problems associated with certain keywords tend to be more on the hard side.

   Here , I defined a curated list of algorithm-related keywords-

   - **graph, tree, dfs, bfs**
   - **dp , dynamic programming**
   - **binary search, greedy**
   - **recursion, bitmask**

   For each keyword, two types of features were extracted:

   - **Frequency features :** number of times the keyword appears in the text.
   - **Binary indicator features :** whether the keyword appears at least once.

4. **TF-IDF (Primary Feature) –** TF-IDF was used to convert problem statements into numerical vectors by weighing words based on their importance within a problem and their rarity across the dataset.

The primary textual representation was constructed using **TF-IDF** vectorization applied to the combined problem text. The combined text consisted of the problem title, description, input text, and output text.

The TF-IDF vectorizer was configured with the following settings:

- **Unigrams and bigrams (ngram_range = (1, 2)):** This feature was used to capture both individual keywords and meaningful multi-word phrases such as *dynamic programming* and *binary search*.
- **Maximum vocabulary size (max_features = 5000):** This feature helped to keep the balance between capturing sufficient textual detail and avoiding excessive dimensionality.
- **Stop-word removal:** common English stop words(of , for, is, are, the, and) were removed to reduce noise.
- **Minimum document frequency (min_df = 2):** extremely rare terms were filtered using minimum document frequency thresholds. This helped remove noise and improve generalization.

## Further after Feature Engineering – **<u>THE EXPERIMENTAL SETUP</u>** for testing different ML models

1. **Feature Scaling and Combination** - All numerical features were standardized using **z-score normalization** via a *StandardScaler* to ensure that features with different scales contributed equally during model training.

The final feature representation was constructed by **horizontally concatenating**:

- The sparse TF-IDF feature matrix
- The scaled numerical and keyword-based features

This resulted in a unified feature vector that integrates both **semantic** and **structural** information.

2. **Training and Prediction Pipeline** –

The dataset was divided into **training and testing subsets** *(80-20 ratio)* using a stratified split to maintain the distribution of difficulty classes. This ensured that all classes were adequately represented during model evaluation.

During training, the TF-IDF vectorizer and the scaler for numerical features were fitted only on the training data. The test data was transformed using these fitted components without refitting, preventing information leakage and ensuring fair evaluation.

For inference, the same trained TF-IDF vectorizer and scaler were reused to process new problem descriptions provided through the web interface. This ensured that the feature representation during deployment remained consistent with the training process.

## Training Procedure

- Models were trained on training set
- Evaluations were performed on held-out test set

## <u>Models tried in this project</u>

**Classification Models**

The following models were evaluated for predicting difficulty class:

- Logistic Regression
- Random Forest Classifier

**Final Choice:** Logistic Regression

**Reason:** Logistic Regression performed better on high-dimensional sparse TF-IDF features, showed stronger generalization, and was computationally efficient.

**Regression Models**

The following models were evaluated for predicting difficulty score:

- Linear Regression
- Ridge Regression
- Gradient Boosting Regressor

**Final Choice:** Gradient Boosting Regressor

**Reason:** Gradient Boosting effectively modelled non-linear relationships and achieved lower prediction errors compared to linear models.

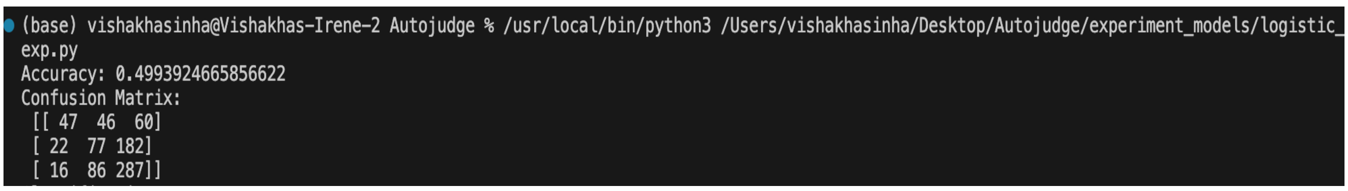
## <u>EVALUATION METRICS</u>

1. **CLASSIFICATION EVALUATION –**

   Metrics used:

- **Accuracy Score**
- **Confusion Matrix**

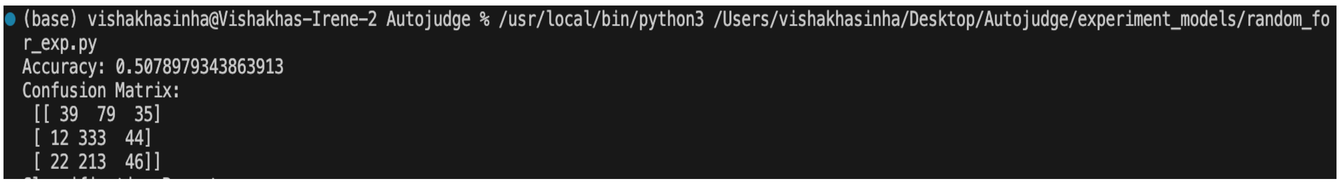
  Logistic Regression **–** ***49.93%*** Accuracy - **FINAL**



```
(base) vishakhasinha@Vishakhas-Irene-2 Autojudge % /usr/local/bin/python3 /Users/vishakhasinha/Desktop/Autojudge/experiment_models/logistic_
exp.py
Accuracy: 0.4993924665856622
Confusion Matrix:
 [[ 47  46  60]
 [ 22  77 182]
 [ 16  86 287]]
```

The confusion matrix follows the order – (easy , medium , hard) – it indicates that the model performs best in identifying Hard problems, while some confusion is observed between Easy and Medium classes.

Random Forest Classifier – ***50.7%*** Accuracy



```
● (base) vishakhasinha@Vishakhas-Irene-2 Autojudge % /usr/local/bin/python3 /Users/vishakhasinha/Desktop/Autojudge/experiment_models/random_fo
  r_exp.py
  Accuracy: 0.5078979343863913
  Confusion Matrix:
   [[ 39  79  35]
   [ 12 333  44]
   [ 22 213  46]]
```

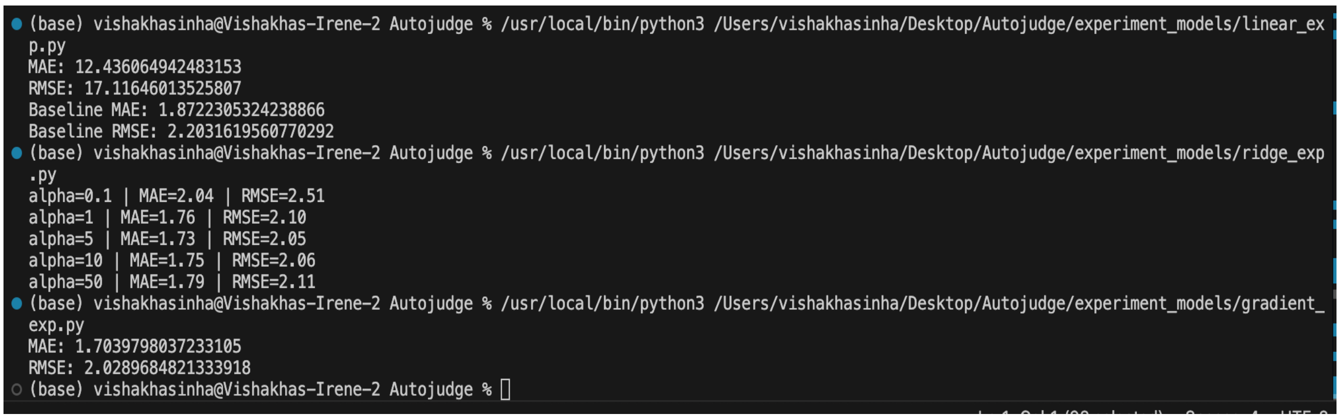
Random Forest Model tended to bias predictions toward the Medium class, resulting in frequent misclassification of both Easy and Hard problems.

*"On comparing both confusion matrices it was observed that **Logistic Regression** demonstrated more balanced behaviour across classes and greater stability when applied to high-dimensional TF-IDF features. Therefore it was chosen as **Final Classification Model**"*

2. **REGRESSION EVALUATION –**

   Metrics Used :

   - **Mean Absolute Error (MAE)**
   - **Root Mean Squared Error (RMSE)**



```
● (base) vishakhasinha@Vishakhas-Irene-2 Autojudge % /usr/local/bin/python3 /Users/vishakhasinha/Desktop/Autojudge/experiment_models/linear_ex
  p.py
  MAE: 12.436064942483153
  RMSE: 17.11646013525807
  Baseline MAE: 1.8722305324238866
  Baseline RMSE: 2.2031619560770292
● (base) vishakhasinha@Vishakhas-Irene-2 Autojudge % /usr/local/bin/python3 /Users/vishakhasinha/Desktop/Autojudge/experiment_models/ridge_exp
  .py
  alpha=0.1 | MAE=2.04 | RMSE=2.51
  alpha=1 | MAE=1.76 | RMSE=2.10
  alpha=5 | MAE=1.73 | RMSE=2.05
  alpha=10 | MAE=1.75 | RMSE=2.06
  alpha=50 | MAE=1.79 | RMSE=2.11
● (base) vishakhasinha@Vishakhas-Irene-2 Autojudge % /usr/local/bin/python3 /Users/vishakhasinha/Desktop/Autojudge/experiment_models/gradient_
  exp.py
  MAE: 1.7039798037233105
  RMSE: 2.0289684821333918
○ (base) vishakhasinha@Vishakhas-Irene-2 Autojudge % ▯
```

To summarize above data –

| **Model** | **MAE** | **RMSE** |
|---|---|---|
| **Linear Regression** | 12.44 | 17.12 |
| **Ridge Regression (α = 5)** | 1.73 | 2.05 |
| **Gradient Boosting Regressor** | 1.70 | 2.02 |

**Model Performance Evaluation -**

- **Linear Regression** showed high error values (MAE ≈ 12.44, RMSE ≈ 17.12), indicating that linear models were insufficient to capture the complexity of the data.
- **Ridge Regression** significantly improved performance (MAE ≈ 1.73–2.04, RMSE ≈ 2.05–2.51), demonstrating the effectiveness of regularization in improving generalization.
- **Gradient Boosting Regressor** achieved the lowest error values (MAE ≈ 1.70, RMSE ≈ 2.03), making it the most accurate and reliable model.

## WEB INTERFACE

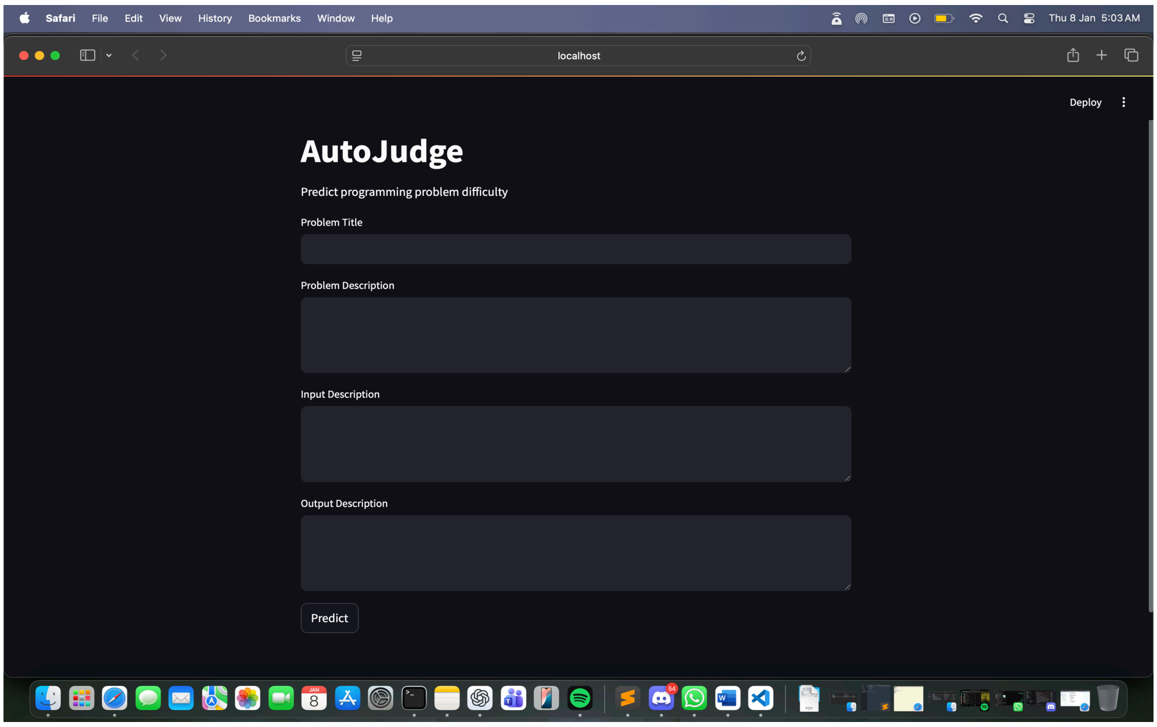
A simple web UI interface was developed using ***Streamlit*** to allow interactive predictions.

The interface allows users to:

- Enter problem title
- Enter problem description
- Enter input and output descriptions

It then displays the predicted difficulty class and score

Web Interface :-

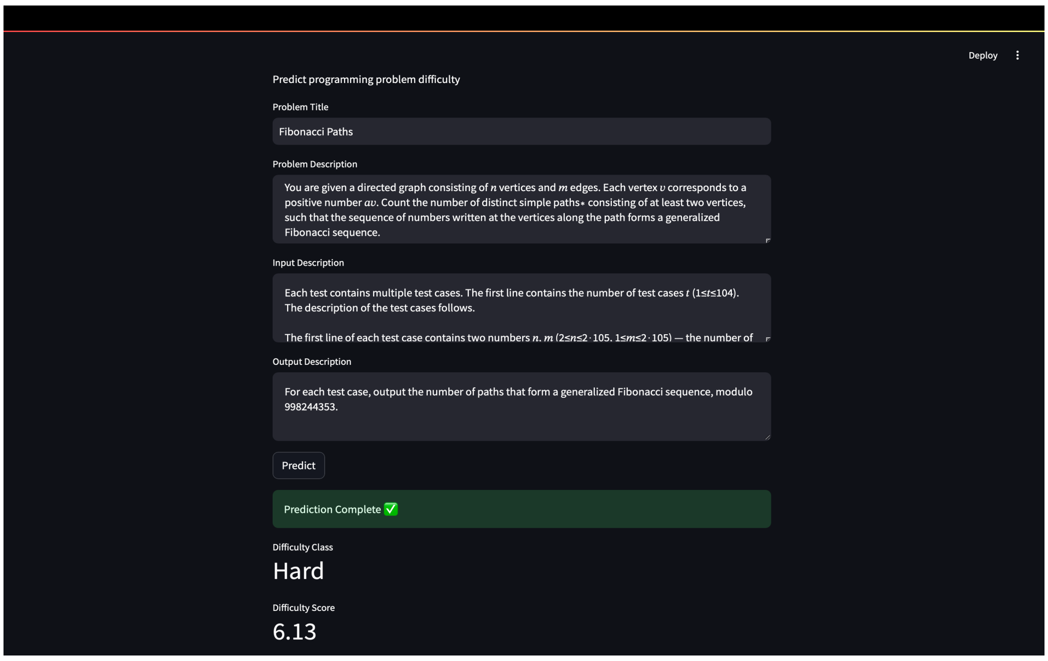


Taking following sample problems –

1. https://codeforces.com/problemset/problem/2176/D

2. https://codeforces.com/problemset/problem/2154/E

Predicted results were respectively obtained as follows -



Predict programming problem difficulty

Problem Title

Fibonacci Paths

Problem Description

You are given a directed graph consisting of $n$ vertices and $m$ edges. Each vertex $v$ corresponds to a positive number $av$. Count the number of distinct simple paths∗ consisting of at least two vertices, such that the sequence of numbers written at the vertices along the path forms a generalized Fibonacci sequence.

Input Description

Each test contains multiple test cases. The first line contains the number of test cases $t$ ($1 \le t \le 104$). The description of the test cases follows.

The first line of each test case contains two numbers $n, m$ ($2 \le n \le 2 \cdot 105$, $1 \le m \le 2 \cdot 105$) — the number of

Output Description

For each test case, output the number of paths that form a generalized Fibonacci sequence, modulo 998244353.

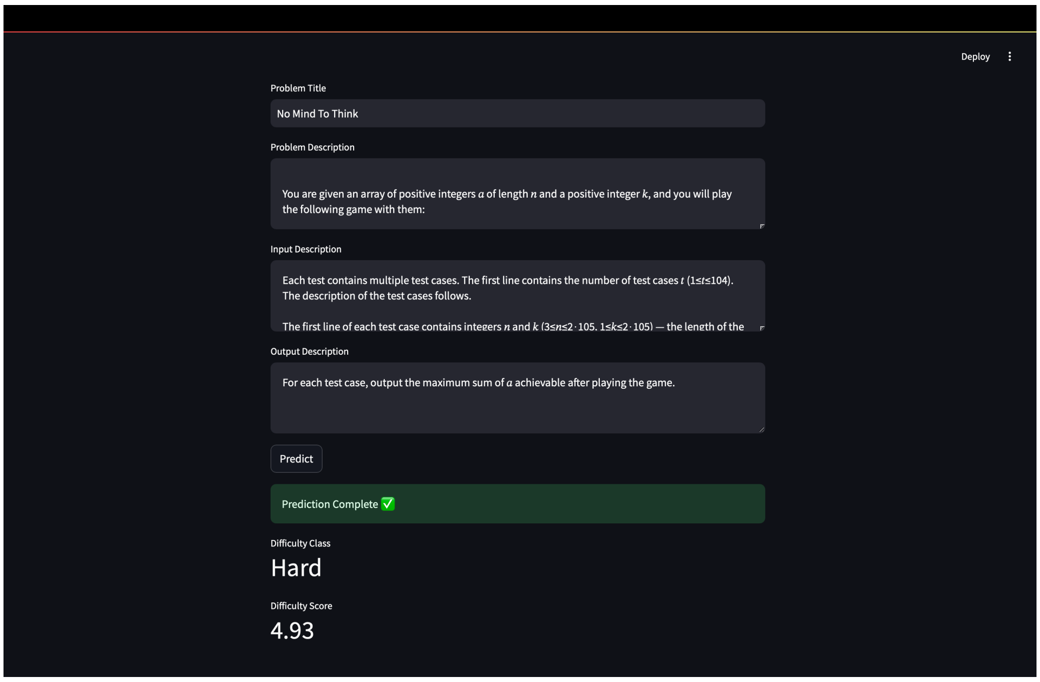
Prediction Complete ✅

Difficulty Class

Hard

Difficulty Score

6.13



Problem Title

No Mind To Think

Problem Description

You are given an array of positive integers $a$ of length $n$ and a positive integer $k$, and you will play the following game with them:

Input Description

Each test contains multiple test cases. The first line contains the number of test cases $t$ ($1 \le t \le 104$). The description of the test cases follows.

The first line of each test case contains integers $n$ and $k$ ($3 \le n \le 2 \cdot 105$, $1 \le k \le 2 \cdot 105$) — the length of the

Output Description

For each test case, output the maximum sum of $a$ achievable after playing the game.

Prediction Complete ✅

Difficulty Class

Hard

Difficulty Score

4.93

## <u>CONCLUSION –</u>

In this project, I developed **AutoJudge**, a system that predicts the difficulty of programming problems using only their textual descriptions. By applying text-based feature engineering and machine learning models, the system was able to achieve reasonable performance for both difficulty classification and score prediction.

However, using only text has its limitations, as it may not fully capture the actual difficulty of implementing a solution. In the future, adding more information such as code examples or solution details could improve the predictions.