# AUTOJUDGE
# Programming Problem Difficulty Prediction

**Author :** Varun Bhimani

**Enrolment No :** 24114022

This report presents the design, implementation, and evaluation of AutoJudge, an end-to-end machine learning system for automatically predicting the difficulty of programming problems using only their textual descriptions. The entire model was divided into 6 different phases.

## 1. Problem Statement

Difficulty estimation of programming problems is a crucial task for competitive programming platforms and online judges. Traditionally, problem difficulty is assigned manually by problem setters or inferred over time using user submission statistics such as acceptance rate and average solving time. This approach is subjective, time-consuming, and unreliable for newly added problems.
The objective of this project is to automate difficulty estimation by analyzing the textual content of programming problems. AutoJudge aims to predict both a categorical difficulty class (Easy, Medium, or Hard) and a numerical difficulty score, without relying on any user interaction data or platform-specific metadata.

## 2. Dataset Used

The dataset used for training and evaluating the AutoJudge system consists of a collection of competitive programming problems stored in **JSON Lines (.jsonl) format**. In this format, each line represents a single programming problem encoded as a JSON object, enabling efficient storage, streaming, and parsing of large textual datasets.
The dataset was chosen to reflect the diversity and complexity typically observed on online coding platforms. It contains problems ranging from simple introductory tasks to advanced algorithmic challenges, making it suitable for both difficulty classification and numerical difficulty regression.

### 2.1 Dataset Fields

Each problem entry includes the following attributes:

- **description**: The full problem statement

- **input_description**: Description of input format and constraints
- **output_description**: Description of expected output format
- **problem_class**: Difficulty label (Easy / Medium / Hard)
- **problem_score**: Numerical difficulty score

The dataset covers a wide range of problem complexities, from basic arithmetic problems to advanced algorithmic challenges involving dynamic programming, graphs, and data structures.

# 3. Data Preprocessing

Raw textual data often contains noise and inconsistencies that can negatively affect model performance. Therefore, careful preprocessing was performed before feature extraction.

## 3.1 Text Field Combination

The problem description, input description, and output description were concatenated into a single unified text input. This ensures that the model has access to complete contextual information about each problem.

## 3.2 Text Cleaning

Text cleaning involved converting all text to lowercase, removing HTML tags, and normalizing whitespace. Mathematical symbols and operators were intentionally preserved, as they often indicate algorithmic or mathematical complexity.

## 3.3 Handling Missing Values

Missing textual fields were replaced with empty strings to avoid data loss. Rows with missing difficulty labels or scores were excluded from training.

# 4. Feature Engineering & Feature Extraction

Feature engineering plays a critical role in enabling machine learning models to capture problem difficulty from text. A hybrid feature representation was adopted.

## 4.1 TF-IDF Features

Term Frequency–Inverse Document Frequency (TF-IDF) was used to convert text into numerical vectors. Unigrams and bigrams were included to capture both individual terms and important phrases such as 'dynamic programming'.

## 4.2 Handcrafted Numerical Features

In addition to TF-IDF, several handcrafted features were extracted:

• Total text length
• Count of mathematical symbols
• Frequency of algorithmic keywords such as graph, dp, recursion, bfs, and dfs
These features were scaled using standard normalization and combined with TF-IDF vectors to form a single sparse feature matrix.

# 5. Models Used

The AutoJudge system is designed to predict programming problem difficulty at two levels: a categorical difficulty class and a continuous numerical difficulty score. To address these two complementary tasks, two different machine learning models were employed—each selected based on the nature of the task, data representation, and performance considerations.

## 5.1 Classification Model: Linear Support Vector Machine (Linear SVM)

The difficulty classification task involves assigning each programming problem to one of three discrete categories: **Easy**, **Medium** or **Hard**. This is formulated as a multi-class text classification problem, where the input features are high-dimensional and sparse due to TF-IDF vectorization.
To solve this task, a **Linear Support Vector Machine (Linear SVM)** was chosen as the classification model.
Linear SVMs are widely regarded as one of the most effective models for text classification tasks. The reasons for selecting Linear SVM in this project include:

- Suitability for High-Dimensional Sparse Data
- Robust Decision Boundaries
- Resistance to Overfitting
- Empirical Effectiveness in NLP Tasks

The Linear SVM was trained with:

- A linear kernel
- Class-weight balancing to mitigate class imbalance
- A regularization parameter tuned to allow separation between Easy, Medium, and Hard problems without collapsing predictions into a single class

The class-weighted loss function ensures that minority classes, such as Hard problems, contribute proportionally to the optimization objective.

## 5.2 Regression Model: Gradient Boosting Regressor

While difficulty classification provides a coarse-grained categorization, it does not capture subtle differences between problems within the same category. To address this limitation, a regression model was used to predict a **continuous numerical difficulty score**.
For this task, a **Gradient Boosting Regressor** was selected.
Gradient Boosting is an ensemble learning technique that builds a strong predictive model by combining multiple weak learners, typically decision trees. The reasons for selecting Gradient Boosting Regressor include:

- Ability to Model Non-Linear Relationships
- Strong Predictive Performance
- Robustness to Feature Heterogeneity
- Controlled Bias–Variance Tradeoff

The Gradient Boosting Regressor was configured with:

- A moderate number of boosting stages
- A small learning rate to ensure gradual improvement
- Limited tree depth to avoid overfitting

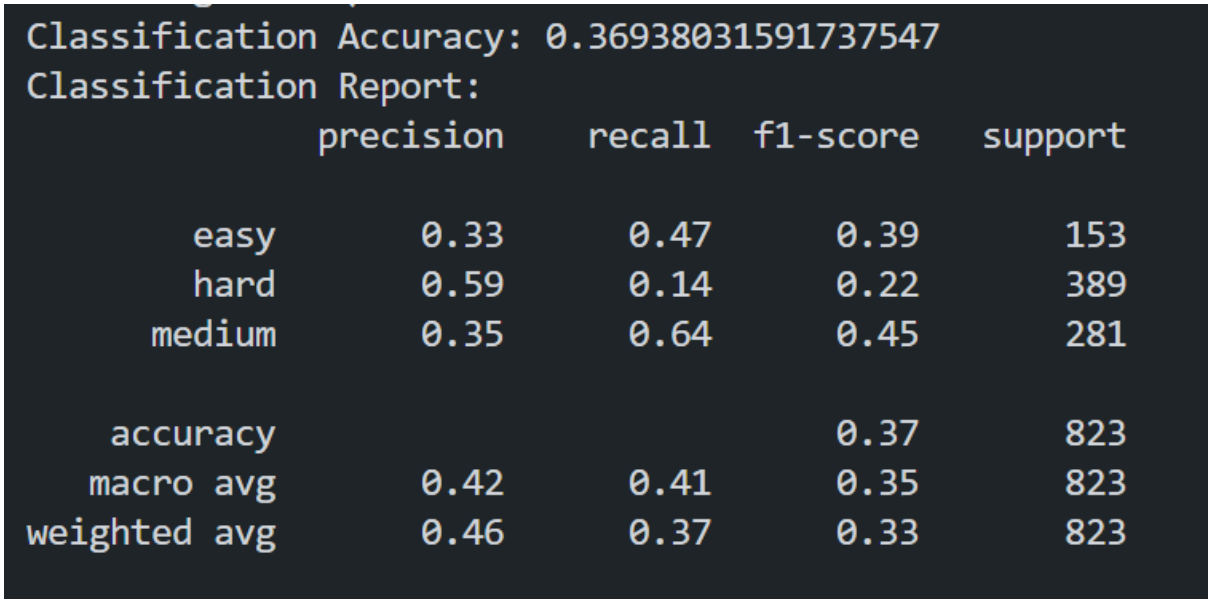
These choices ensure stable learning and smooth prediction behavior across the difficulty score range.

# 6. Experimental Setup & Evaluation

The dataset was split into training and testing sets using an 80–20 stratified split to preserve class distribution. The same split was used for both classification and regression tasks.

## 6.1 Evaluation Metrics

Classification performance was evaluated using accuracy and confusion matrix analysis. Regression performance was evaluated using Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). The results for the same are given in below pics :

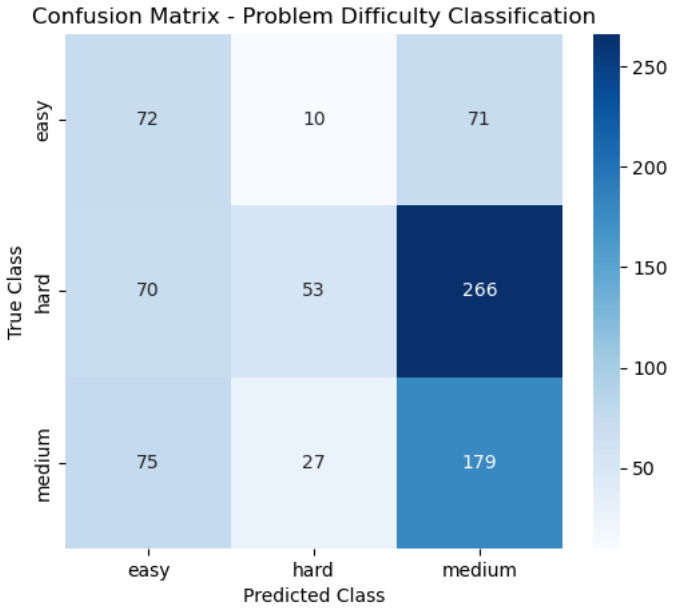


Classification Accuracy: 0.36938031591737547

Classification Report:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| easy | 0.33 | 0.47 | 0.39 | 153 |
| hard | 0.59 | 0.14 | 0.22 | 389 |
| medium | 0.35 | 0.64 | 0.45 | 281 |
| accuracy | | | 0.37 | 823 |
| macro avg | 0.42 | 0.41 | 0.35 | 823 |
| weighted avg | 0.46 | 0.37 | 0.33 | 823 |

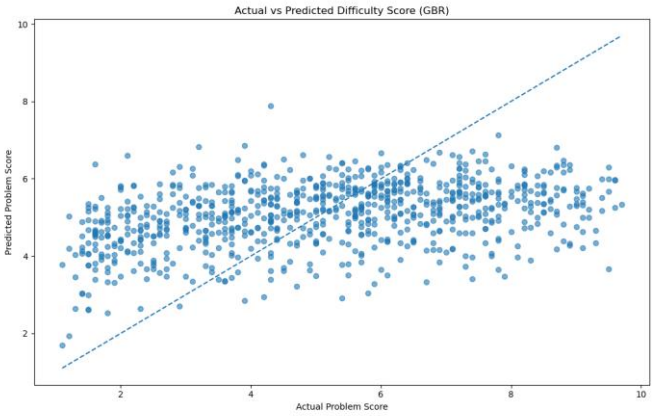
**Classification Results**




**Confusion Matrix**

```
Gradient Boosting Regressor Performance:
MAE : 1.707170923708655
RMSE: 2.0437567372196273
Class distribution: [ 613 1552 1124]
```

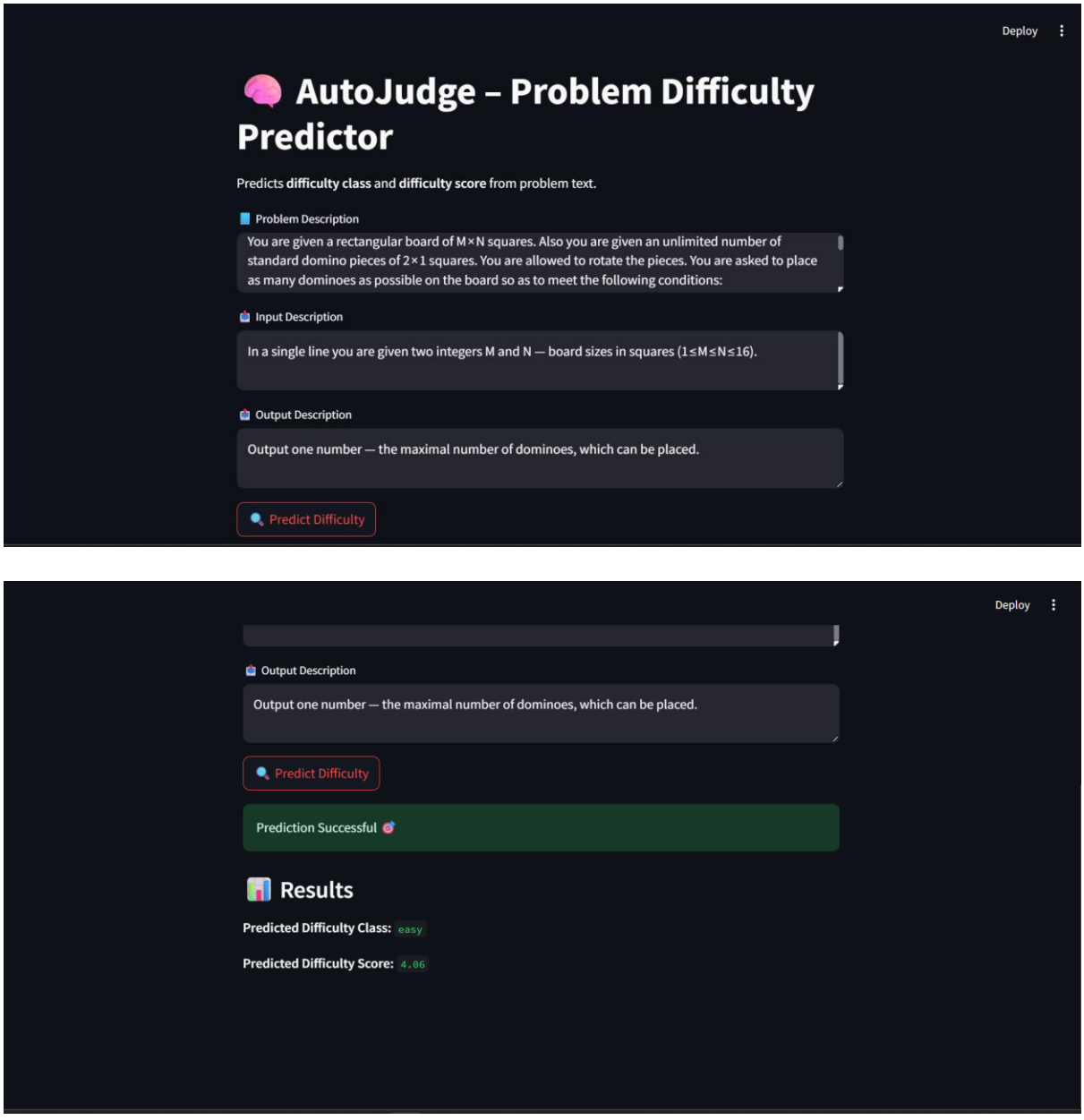
**Regression Results**




**Actual vs Predicted Problem Score**

# 7. Web Interface

A web-based interface was developed using Streamlit to demonstrate the functionality of AutoJudge. The interface allows users to input problem descriptions and receive real-time predictions.
The UI consists of text boxes for problem description, input description, and output description. Upon submission, the input is processed using the same preprocessing and feature extraction pipeline used during training, ensuring consistency.




**Web UI**

# 8. Conclusion

This project demonstrates the effectiveness of combining NLP techniques with classical machine learning models to automate difficulty estimation for programming problems. AutoJudge provides a scalable and interpretable solution that can be extended using transformer-based language models or deployed for real-world use on online coding platforms.

**Note :** There is a small catch that the model sometimes misinterprets problems from their correct category. This can be justified by the imbalance in the problem categories in the dataset. With improved dataset, efficiency of the model can also be improved.