PROJECT REPORT – AutoJudge

Name – Parshv Meshiya.

Enrollment No. – 24114059

Branch – CSE.

Year – 2nd

(1) Problem Statement :

Competitive programming platforms such as Codeforces, CodeChef, and AtCoder categorize problems into difficulty levels like Easy, Medium, and Hard. This categorization is usually done manually by problem setters or platform admins, making it subjective, time-taking, and inconsistent across platforms.

The objective of this project is to **automatically predict the difficulty of a programming problem** using its **textual problem statement**. The system predicts:

- A **difficulty class** (Easy / Medium / Hard)
- A **numerical difficulty score** ranging from 0 to 10

This project aims to assist problem setters, learners, and educational platforms by providing an automatic, consistent, and scalable difficulty estimation system using machine learning techniques.

(2) Dataset:

The dataset used in this project consists of programming problem statements collected from competitive programming sources(As provided in project instructions pdf.). Each problem is associated with:

- The full **problem statement text**
- A **difficulty class label** (Easy / Medium / Hard)
- A **numerical difficulty score** (0–10)

The dataset is stored in CSV format and is used for both classification and regression tasks.

Dataset location : data/problems.csv

3. Data Preprocessing:

The raw dataset consists of CP problem statements with multiple text fields such as title, description, input format, output format, and sample input–output pairs.

Several preprocessing steps were applied to prepare the data for feature extraction and modeling:

1. **Handling missing values:**
   Missing textual fields were replaced with empty strings to avoid errors during vectorization.
2. **Text consolidation:**
   All relevant textual fields were concatenated into a single {full_text} column to provide a single representation of each problem.
3. **Sample input–output processing:**
   Sample input–output pairs were converted into text format and added to the problem description.
4. **Text normalization:**
   Basic normalization such as whitespace handling and lowercasing was applied implicitly through the TF-IDF vectorizer.

The preprocessed dataset was then saved and used across all further stages of feature extraction and model training.

Full_text after preprocessing :

```
Juu Unununium (Uuu) was the name of the chemical
    element with atom number 111, until it changed to
    Röntgenium (Rg) in 2004. These heavy elements are very
    unstable and have only been synthesized in a few
    laboratories.
You have just been hired by one of these labs to optimize
    the algorithms used in simulations. For example, when
    simulating complicated chemical reactions, it is important to
    keep track of how many particles there are, and this is done by
    counting connected components in a graph.
Currently, the lab has some Python code (see attachments)
    that takes an undirected graph and outputs the number of
    connected components. As you can see, this code is based on
    everyone's favourite data structure union-find1.
After looking at the code for a while, you notice that it
    actually has a bug in it! The code still gives correct answers,
    but the bug could cause it to run inefficiently. Your task is
    to construct a graph with a given number of vertices and edges
    where the code runs very slowly. We will count how many times
    the third line (the one inside the while loop) is visited, and
    your program will get a score according to this number.
 The input consists of one line with two integers
    $N$ and $M$, the number of vertices and edges
    your graph should have. Apart from the sample, there will be
    only one test case, with $N =
    100$ and $M =
    500$. The output consists of $M$ lines where the $i$:th contains two integers
    $u_ i$ and $v_ i$ ($1 \leq u_ i, v_ i \leq N$). This
    indicates that the vertices $u_
    i$ and $v_ i$ are
    connected with an edge in your graph.
```

(4)Feature Engineering

Feature engineering is a critical component of this project, as the difficulty of a programming problem depends not only on its length but also on the presence of algorithmic concepts, constraints, and input structure.

To capture these aspects, both **textual features** and **hand-crafted structural features** were extracted from the problem statements.

1. **Textual Features:** Textual information was represented using **Term Frequency–Inverse Document Frequency (TF-IDF)** vectorization.
    - Vocabulary size: 5000
    - N-grams: unigrams and bigrams

TF-IDF helps capture the importance of words such as *dynamic programming*, *graph*, *tree*, etc., which are strong indicators of problem difficulty.

2. Hand-Crafted Features: In addition to TF-IDF, several manually designed features were extracted to capture structural and algorithmic complexity. These include:

- Text length

- Word count
- Mathematical symbol count
- Algorithmic keyword indicators (DP, BFS, DFS, greedy, segment tree)
- Constraint indicators (e.g., 10^5, time limit)
- Input structure indicators (matrix, graph, multiple test cases)

A total of **27 hand-crafted features** were used.

The final feature vector consists of **5027 dimensions** obtained by concatenating TF-IDF and manual features.

```
print(X.shape)
print(rf.n_features_in_)

✓ 0.0s
(4112, 5027)
5027
```

(5) Models Used :Two separate machine learning models were trained for the two prediction tasks.

1. Difficulty Classification: Difficulty classification was formulated as a **multi-class classification problem** with three classes: Easy, Medium, and Hard.

- **Model:** Logistic Regression
- **Input:** 5027-dimensional feature vector
- **Output:** Difficulty class

Logistic Regression was chosen due to its simplicity, interpretability, and effectiveness on high-dimensional sparse text features.

5.2 Difficulty Score Regression

Difficulty score prediction was treated as a **regression problem**.

- **Model:** Random Forest Regressor

- **Input:** Same feature vector
- **Output:** Continuous difficulty score (0–10)

Random Forest was selected because it can model non-linear relationships between features and difficulty score.

(6) Experimental Setup

The dataset was divided into training and testing sets using an **80–20 split**. Both classification and regression models were trained using the same feature representation. Experiments were conducted on a local machine using Python and scikit-learn. Hyperparameters for the Random Forest model were selected empirically.

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y_encoded,
    test_size=0.2,
    random_state=59,
    stratify=y_encoded
)
```

✓ 0.0s

(7) Results and Evaluation

1. **Classification Results:** The performance of the difficulty classification model was evaluated using **accuracy**. A **confusion matrix** was also generated to analyze class-wise performance and misclassification patterns.

```
acc = accuracy_score(y_test, y_pred)
print("Accuracy:", acc)
print(classification_report(y_test, y_pred))
```

✓ 0.0s

```
Accuracy: 0.503037667071689
              precision    recall  f1-score   support

        easy       0.52      0.35      0.42       153
        hard       0.56      0.71      0.62       389
      medium       0.38      0.30      0.33       281

    accuracy                           0.50       823
   macro avg       0.48      0.45      0.46       823
weighted avg       0.49      0.50      0.49       823
```

```
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)
```

✓ 0.0s

```
Confusion Matrix:
 [[ 54  51  48]
 [ 22 277  90]
 [ 27 171  83]]
```

---

2. Regression Results: The regression model was evaluated using:

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

These metrics quantify the average deviation between predicted and actual difficulty scores.

```
mae = mean_absolute_error(y_test, y_pred)
print("MAE:", mae)
```

✓ 0.0s

```
MAE: 1.7073219014998684
```

```
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("RMSE:", rmse)
```

✓ 0.0s

```
RMSE: 2.052906135185684
```

(8) Web Interface

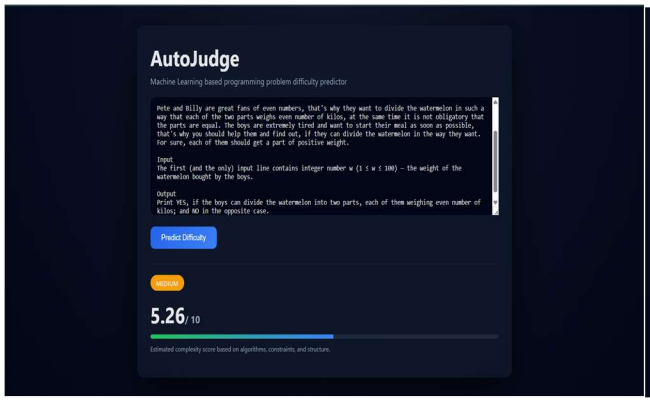
A web interface was developed to demonstrate real-time predictions.

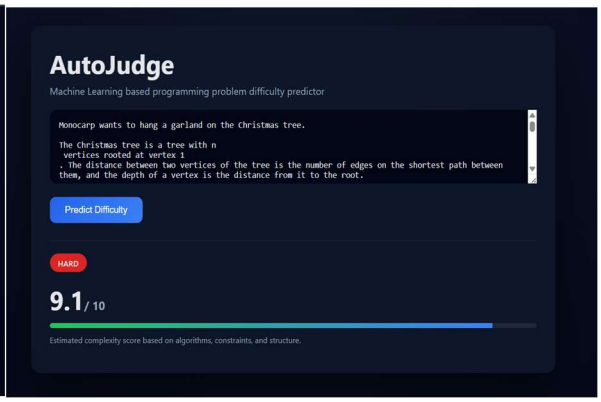
Backend:

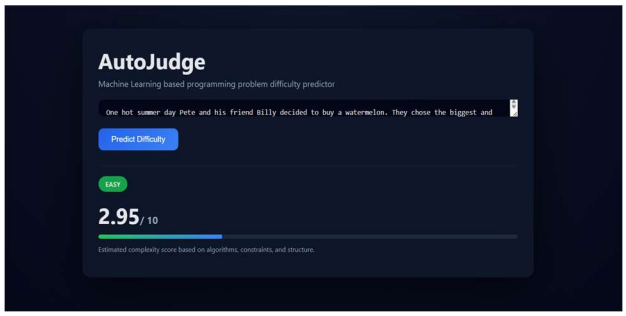
- Implemented using FastAPI
- Performs feature extraction and model inference

Frontend:

- Built using React
- Allows users to paste problem statements
- Displays predicted difficulty class and difficulty score







(9)Conclusion

In this project, we developed **AutoJudge**, a machine learning–based system for predicting the difficulty of programming problems using their textual descriptions. The system addresses both **classification** (easy, medium, hard) and **regression** (numerical difficulty score) tasks by leveraging TF-IDF–based text representations and supervised learning models.

Experimental results show that the proposed approach can capture meaningful patterns from problem statements and generalize reasonably well to unseen data. During evaluation, we identified and mitigated biases caused by superficial features such as text length, highlighting the importance of careful feature engineering in text-based machine learning.

The trained models were successfully deployed using a **FastAPI backend** and a **React frontend**, enabling real-time difficulty prediction through a user-friendly interface. Overall, this work demonstrates the practicality of automated difficulty assessment for programming problems and provides a foundation for future improvements using more advanced language models and richer datasets.