AutoJudge: Predicting Programming Problem Difficulty

1. Introduction and Problem Statement

Online competitive programming platforms such as Codeforces, CodeChef, and Kattis host a large number of programming problems. These problems are usually categorized into difficulty levels such as *Easy*, *Medium*, and *Hard*, and are often assigned a numerical difficulty score. Traditionally, this classification is done manually based on expert judgment and user feedback, which can be subjective, time consuming, and inconsistent.

The objective of this project is to design an automated system that can predict the difficulty of programming problems using only their textual descriptions. The problem is formulated as two machine learning tasks:

1. **Classification Task** – Predicting the difficulty class (Easy / Medium / Hard)
2. **Regression Task** – Predicting a numerical difficulty score

The system relies solely on natural language information such as the problem statement, input description, and output description, without using any user statistics or solution data.

2. Dataset Description

The dataset used in this project is a provided collection of programming problems annotated with difficulty labels and scores. Each problem is stored as a single JSON object in a JSONL (JSON Lines) format.

Each data sample contains the following fields:

- title
- description
- input_description
- output_description
- problem_class (Easy / Medium / Hard)
- problem_score (numerical value)

The dataset consists of **4112 samples** in total and is stored locally in the file: data/problems.jsonl

Each line in the file represents one programming problem, making it suitable for line by line parsing and preprocessing.

3. Data Preprocessing

Since the dataset consists of raw textual data, preprocessing is a crucial step to improve model performance. The following preprocessing steps were applied:

1. Conversion of all text to lowercase
2. Removal of special characters and punctuation
3. Tokenization using SpaCy
4. Stopword removal using the NLTK stopwords list
5. Lemmatization to reduce words to their base forms

All text fields (title, description, input_description, output_description) were merged into a single field called full_text. This combined representation ensures that the model has access to all relevant information about a problem.

The cleaned text was stored in a new column called clean_text, which was later used for feature extraction.

4. Feature Engineering

In addition to standard text vectorization, several handcrafted features were extracted to capture structural and semantic properties of programming problems.

4.1 TF-IDF Features

The cleaned text was converted into numerical vectors using Term Frequency Inverse Document Frequency (TF-IDF). This helps capture the importance of words across the dataset.

4.2 Additional Features

The following additional features were included:

- **Text Length**: Number of words in the combined problem text
- **Mathematical Symbol Count**: Frequency of symbols such as +, -, *, /, <, >
- **Keyword Frequency**: Count of important keywords such as *graph*, *dp*, *recursion*, *binary*, *search*, and *matrix*

These features were inspired by the observation that harder problems often contain longer descriptions, mathematical expressions, and advanced algorithmic terminology.

5. Models Used

The project uses classical machine learning models rather than deep learning approaches to keep the system simple, interpretable, and computationally efficient.

5.1 Classification Models

The following models were evaluated for difficulty classification:

- Logistic Regression
- Support Vector Machine (SVM)
- Random Forest Classifier

5.2 Regression Models

For predicting the numerical difficulty score, the following models were used:

- Linear Regression
- Random Forest Regressor
- Gradient Boosting Regressor

Each model was trained and evaluated using the same feature set to ensure fair comparison.

6. Experimental Setup

The dataset was split into training and testing sets using a standard train test split. All experiments were conducted on a local machine using the following libraries:

- Pandas and NumPy for data handling
- Scikit-learn for machine learning models and evaluation
- NLTK and SpaCy for natural language processing
- Streamlit for the web interface

The evaluation metrics used were:

- **Accuracy** and **Confusion Matrix** for classification
- **Mean Absolute Error (MAE)** and **Root Mean Squared Error (RMSE)** for regression

7. Results and Evaluation

7.1 Classification Results

Model	Accuracy
Logistic Regression	0.53
Support Vector Machine	0.50
Random Forest Classifier	**0.56**

The Random Forest Classifier achieved the highest accuracy of **56.38%** and was selected as the final classification model.

Confusion Matrix (Random Forest):

[[50 66 20]

[24 372 29]

[17 203 42]]

7.2 Regression Results

Model	RMSE	MAE
Linear Regression	2.45	1.98
Random Forest Regressor	**2.03**	**1.69**
Gradient Boosting Regressor	2.05	1.72

The Random Forest Regressor produced the lowest error values and was selected as the final regression model.

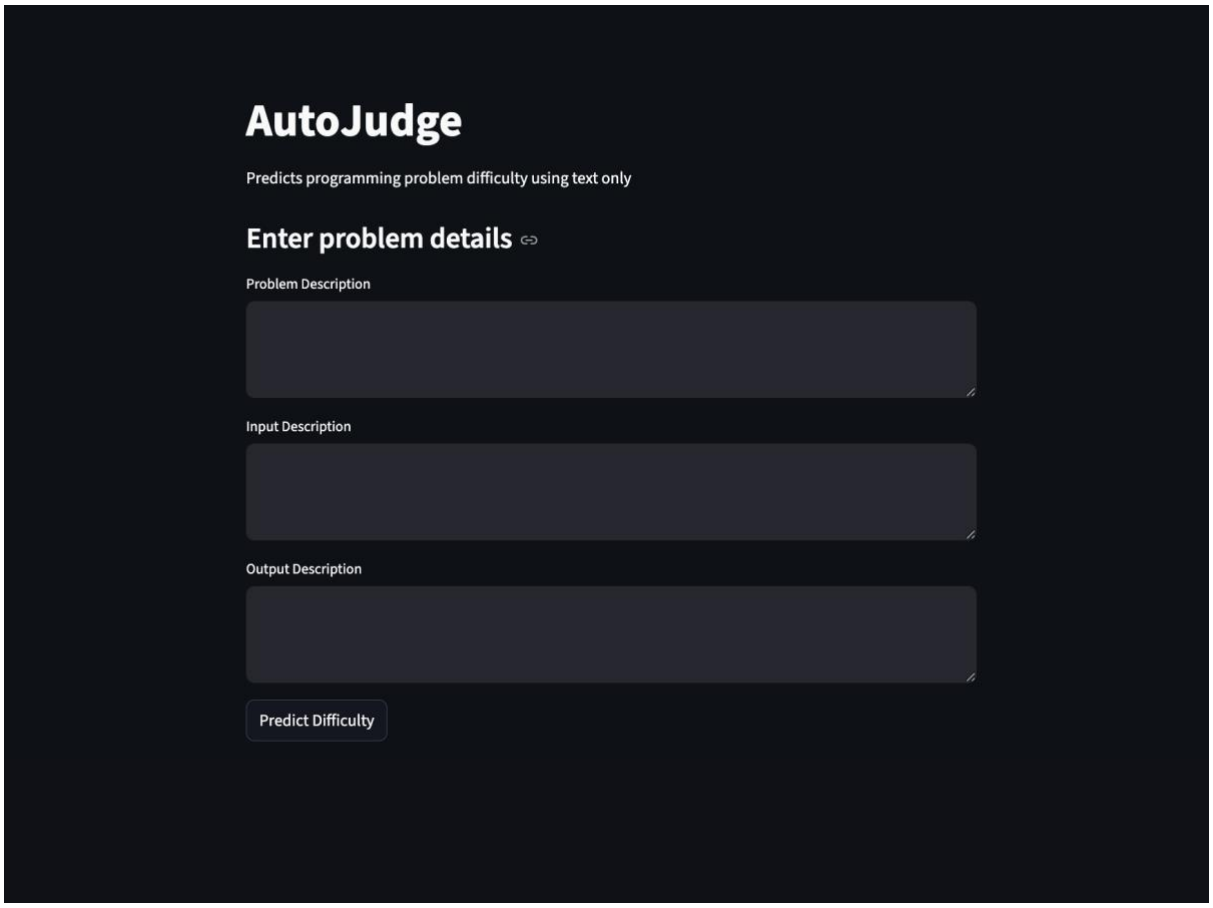
8. Web Interface Description

A simple web interface was developed using Streamlit to allow users to interact with the system. The interface provides text boxes where users can paste a programming problem description.

After clicking the **Predict** button, the application displays:

- The predicted difficulty class
- The predicted difficulty score

The interface runs locally and does not require authentication or a database.



9. Conclusion

This project demonstrates that programming problem difficulty can be reasonably predicted using only textual descriptions and classical machine learning techniques. By combining TF-IDF features with handcrafted features and ensemble models such as Random Forests, the system achieves meaningful classification and regression performance.

Although the accuracy can be further improved using larger datasets or deep learning models, the current system provides a simple, interpretable, and efficient baseline solution. Future work may include using transformer-based language models, incorporating code samples, or expanding the dataset.

Gorla Thannuj

23116036