AutoJudge: Automated Programming Problem Difficulty Prediction

PROJECT REPORT

AUTHOR: **CHANDRAKANT** (ENROLLMENT NO: **23112027**)

Abstract

Competitive programming platforms host a vast array of problems, but often lack a unified or objective standard for difficulty classification. This project, **AutoJudge**, presents a Machine Learning-based system designed to automatically predict the difficulty level (Easy, Medium, Hard) and a numerical complexity score for programming problems based on their textual descriptions. Leveraging Natural Language Processing (NLP) techniques and Random Forest models on a merged dataset from LeetCode and Kattis, the system achieves an accuracy of **61.88%** in classification and a Mean Absolute Error (MAE) of **0.94** in regression. A user-friendly web interface powered by Streamlit allows real-time analysis, bridging the gap between problem curation and automated difficulty assessment.

1. Introduction

1.1 Problem Statement

In the domain of competitive programming and computer science education, assessing the difficulty of a problem is crucial for creating balanced contests, guiding student learning learning paths, and recommending appropriate challenges. Currently, difficulty tags are often assigned manually by problem setters or community voting, leading to inconsistency, subjectivity, and bias across different platforms. There is a need for an automated system that can objectively evaluate problem difficulty based on the inherent properties of the problem statement itself.

1.2 Motivation

The subjective nature of "difficulty" means what is "Medium" on one platform might be "Hard" on another. An automated tool can standardize this metric. Furthermore, for educators and platform maintainers, automating this process saves significant time and ensures immediate classification for new problems.

1.3 Objectives

The primary objectives of the AutoJudge project are:

1. To construct a diverse and comprehensive dataset of programming problems from multiple sources.
2. To develop a robust preprocessing pipeline to handle unstructured text data including mathematical notation.
3. To implement and evaluate machine learning models for both multi-class classification (Difficulty Level) and regression (Complexity Score).
4. To deploy the best-performing model via an interactive web interface for real-time predictions.

2. Dataset Used

To ensure robust model training, a heterogeneous dataset was constructed by merging data from three prominent competitive programming platforms. This diversity prevents the model from overfitting to the specific style of a single platform.

2.1 Data Sources

- **LeetCode**: Known for interview-style questions. Data includes 'Easy', 'Medium', and 'Hard' labels.
- **Kattis**: Hosts problems from various contests with difficulty scores ranging from ~1.0 to 9.0+.
- **Dataset given in problem statement**

2.2 Data Statistics

The final merged dataset contains approximately **10,937** samples.

- Labels: Mapped to a unified scheme:
  - **Easy**: Introductory problems, LeetCode Easy, Kattis < 3.0.
  - **Medium**: Interview level, LeetCode Medium, Kattis 3.0-6.0.
  - **Hard**: Competition level, LeetCode Hard, Kattis > 6.0.
- **Scores**: A continuous scale (1-3) was synthesized where Easy ≈ 1, Medium ≈ 2, Hard ≈ 3 to support fine-grained regression analysis.

3. Methodology

3.1 Data Preprocessing

Raw text data from problem descriptions often contains HTML tags, LaTeX math formulas, and irrelevant noise. The **src/preprocessing.py** module handles this:

1. **HTML Removal**: Regular expressions strip HTML tags (<p>, <code>, etc.) to isolate plain text.
2. **Normalization**: Text is converted to lowercase to ensure consistency.
3. **Symbol Preservation**: Unlike standard NLP pipelines, we preserve the $ symbol, which is commonly used to delimit mathematical expressions in LaTeX. This is a critical domain-specific decision.
4. **Tokenization** & Stopword Removal: We use NLTK to tokenize text and remove common English stopwords (e.g., "the", "is") to focus on content-bearing words

3.2 Feature Engineering

We extract both statistical and semantic features:

1. **TF-IDF Vectors**: Term Frequency-Inverse Document Frequency (TF-IDF) is used to convert the cleaned text into numerical vectors (max features: 1000). This captures the importance of specific terms (e.g., "tree", "array", "dynamic") relative to the difficulty.
2. Explicit Features:
   - **word_count**: The total number of words in the description. Longer problems often correlate with higher complexity.
   - **math_symbol_count**: The count of $ symbols. This serves as a proxy for the mathematical density of the problem, which is a strong indicator of difficulty (e.g., number theory or combinatorics problems).
   - **text_len**: Total character length.

3.3 Models Used

We experimented with several algorithms, selecting **Random Forest** for its robustness and ability to handle high-dimensional text data.

- **Classification**: RandomForestClassifier (n_estimators=100) predicts the class {Easy, Medium, Hard}.
- **Regression**: RandomForestRegressor (n_estimators=100) predicts the continuous complexity score.

4. Experimental Setup

- **Training Split**: The dataset was split into 80% training and 20% testing sets using train_test_split with a fixed random seed (42) for reproducibility.
- **Pipeline**: A Scikit-Learn Pipeline was constructed to chain the ColumnTransformer (which applies TF-IDF to text and Scaling to numeric features) with the Random Forest estimator.
- **Environment**: Python 3.8+, Scikit-Learn, Pandas, NLTK.

5. Results and Evaluation

The models were evaluated on the held-out test set (20% of data).

5.1 Evaluation Metrics

Metric	Value
Accuracy (Classification)	61.88%
MAE (Regression)	0.9409
RMSE (Regression)	1.3933

Table 1: Performance metrics for the Random Forest models.

5.2 Confusion Matrix

The confusion matrix below illustrates the performance across different classes.

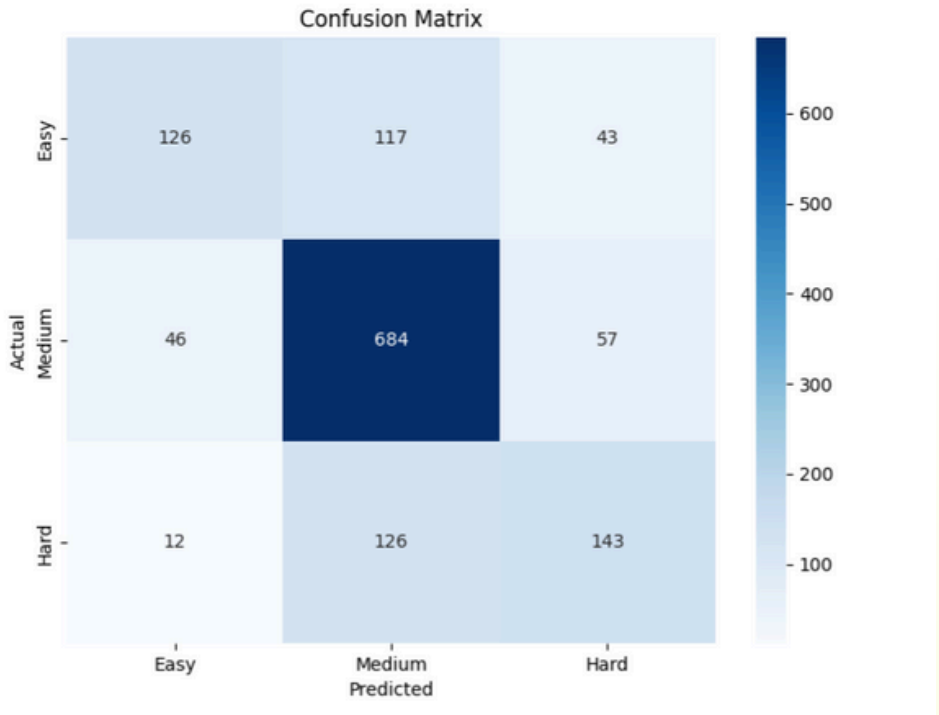

Figure 1: Confusion Matrix for Difficulty Classification. The diagonal elements represent correct predictions.

The model shows reasonable performance but, as expected, has some overlap between adjacent classes (e.g., Easy vs. Medium), reflecting the subjective nature of the ground truth labels.

6. Web Interface

To make the model accessible, a web application was developed using **Streamlit**.

6.1 Design

The interface features a modern, dark-themed design for visual appeal.

- **Input Form**: Users can input the Problem Title, Description, Input Format, and Output Format.
- **Real-time** Analysis: Upon clicking "Analyze", the system runs the entire preprocessing and inference pipeline in the background.

6.2 Sample Prediction

Below is a screenshot of the interface predicting a "Medium" problem.

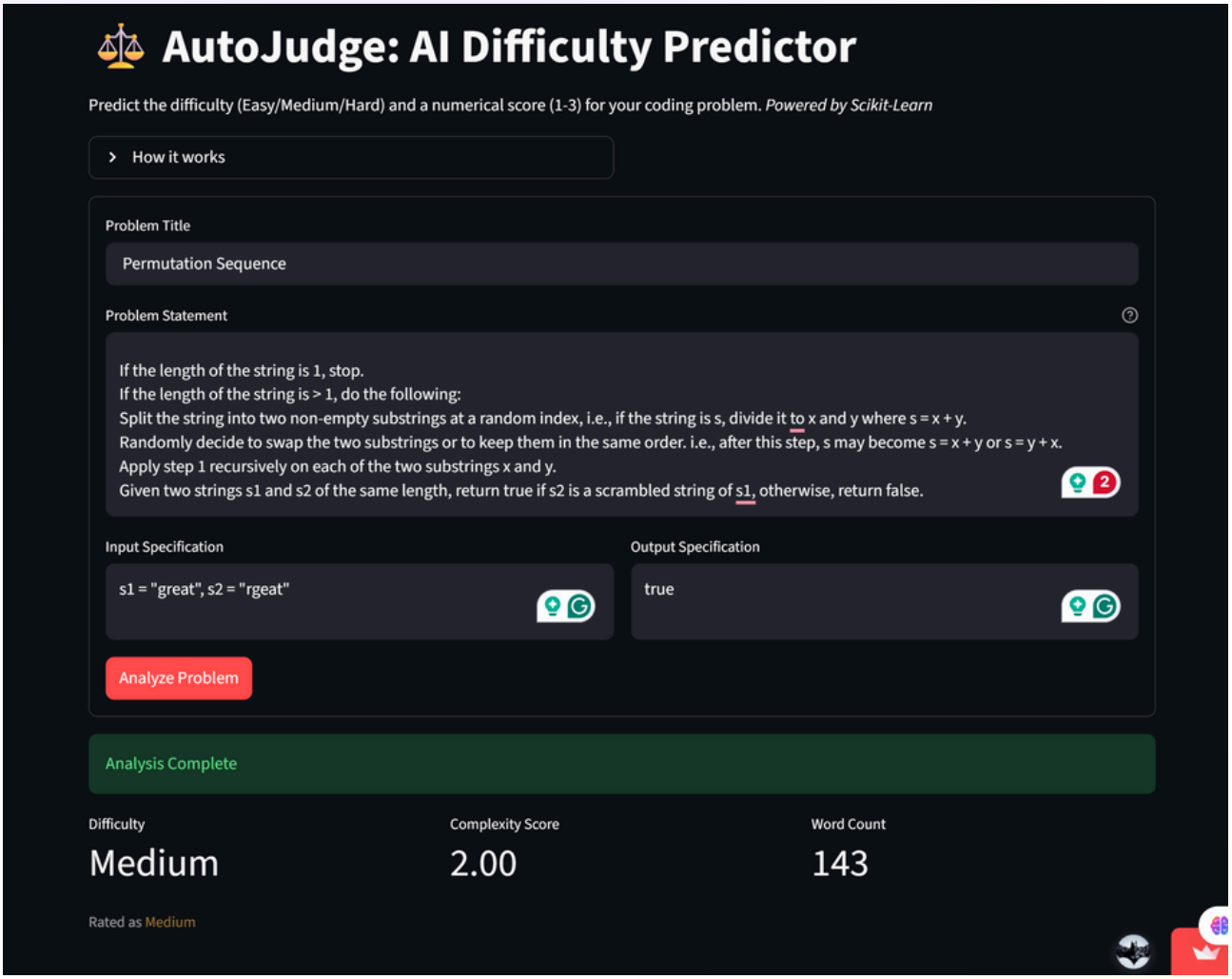

Figure 2: AutoJudge Web Interface showing a prediction result.

In this example, the user entered a graph problem ("Permutation Sequence"). The system correctly identified the keywords and mathematical context, predicting **Medium** with a high complexity score of **2.0.**

7. Conclusion

AutoJudge successfully demonstrates the viability of using NLP and Machine Learning to automate the classification of programming problem difficulty. By merging datasets and engineering domain-specific features like math symbol counts, we achieved a classification accuracy of ~62% and a low regression error. While differentiation between adjacent difficulty levels remains a challenge due to subjectivity, the tool provides a consistent and immediate baseline for problem setters and educators.

Future Work

- Deep Learning: Implementing Transformer-based models (BERT, CodeBERT) to capture deeper semantic meaning.
- Code Analysis: Incorporating actual solution code (if available) as an additional feature for difficulty estimation.
- User Feedback Loop: Allowing users to correct predictions to valid/retrain the model over time.