AutoJudge: Automatic Programming Problem Difficulty Prediction

1. Introduction

Online competitive programming platforms classify problems into difficulty levels such as *Easy*, *Medium*, and *Hard*, and often assign a numerical difficulty score. These labels are usually determined manually using expert judgment and user feedback, which is subjective and time-consuming.

This project, **AutoJudge**, aims to automate this process using Natural Language Processing (NLP) and Machine Learning. Given only the **textual description** of a programming problem, the system predicts:

1. **Problem Class** – Easy / Medium / Hard (classification)
2. **Problem Score** – a continuous numerical difficulty value (regression)

The project also includes a simple **web interface** where users can paste a new problem statement and obtain predictions in real time.

---

2. Dataset Description

The dataset used in this project was **provided as part of the problem statement itself**. No external data sources were collected or labeled by the authors.

Each data sample contains the following fields:

- `title`
- `description`
- `input_description`
- `output_description`
- `problem_class` (Easy / Medium / Hard)
- `problem_score` (numerical difficulty score)

All labels (`problem_class` and `problem_score`) were already included in the dataset, as explicitly specified in the official project problem statement provided by the club. The task was strictly limited to **using the provided dataset only**, without any manual relabeling or data augmentation.

For modeling purposes, the textual fields (`description`, `input_description`, and `output_description`) were concatenated into a single field called **combined_text**, following the official project instructions.

---

3. Data Preprocessing

The following preprocessing steps were applied:

1. Lowercasing all text
2. Removal of English stop words
3. Tokenization handled internally by TF-IDF
4. Numerical constraints and symbols retained as they convey problem complexity

No data leakage was introduced; all transformations were fitted only on training data.

---

4. Feature Engineering

4.1 TF-IDF Features

A **TF-IDF Vectorizer** was used to capture semantic and syntactic patterns from problem descriptions.

- Max features: 12,000
- N-grams: (1, 2)
- Stop words: English

4.2 Handcrafted Complexity Features

To complement TF-IDF, domain-specific features were engineered:

- Text length
- Number of mathematical symbols
- Frequency of algorithmic keywords (DP, graph, BFS, DFS, etc.)
- Average word length
- Sentence count
- Presence of explicit algorithm hints
- Maximum numerical constraint
- Control-flow keyword density (`if`, `for`, `while`)

These features capture **structural and logical complexity** beyond raw text semantics.

4.3 Feature Scaling and Selection

- Min-Max scaling was applied to handcrafted features (chi-square safe)
- TF-IDF and handcrafted features were concatenated
- **Chi-square feature selection** reduced features to the top 5000

Final feature shape: **(4112, 5000)**

---

5. Classification Model

5.1 Motivation for Hierarchical Classification

A flat 3-class classifier struggled with the *Easy* class due to overlap with Medium problems. To address this, a **two-stage hierarchical approach** was adopted.

5.2 Stage 1: Easy vs Non-Easy

- Model: Linear SVM (LinearSVC)
- Class weights: Easy boosted (2.5×)
- Objective: Maximize recall for Easy problems

Stage-1 Accuracy: 80.80%

5.3 Stage 2: Medium vs Hard

- Model: Linear SVM (LinearSVC)
- Balanced class weights
- Trained only on non-easy samples

5.4 Final Prediction Logic

1. Predict Easy vs Non-Easy
2. If Non-Easy → predict Medium or Hard

---

6. Classification Results

Overall Accuracy

Final Hierarchical Accuracy: 79.95%

Classification Report

- Easy: Precision 0.67, Recall 0.41
- Medium: Precision 0.79, Recall 0.88
- Hard: Precision 0.84, Recall 0.89

The hierarchical approach significantly improved robustness compared to a flat classifier, especially for Medium and Hard classes.

---

7. Regression Model (Difficulty Score Prediction)

7.1 Motivation

Difficulty scores vary significantly across classes. A single regressor underperformed due to heterogeneity. Hence, **class-conditional regression** was adopted.

7.2 Model

- Separate **Linear SVR** models for Easy, Medium, and Hard
- Shared feature space from classification

7.3 Evaluation Metrics

- **MAE:** 0.795
- **RMSE:** 1.017
- **R^2 Score:** 0.787

This demonstrates strong predictive performance without overfitting or data leakage.

---

8. Web Interface

A lightweight web interface was developed using **Streamlit** and deployed publicly for live demonstration and verification.

- **Live Application URL:** https://autojudge-hxoigre646axmf2kpmdtbi.streamlit.app/

User Input Format

The interface strictly follows the project specification:

- **Problem Description** – Main problem statement
- **Input Description** – Description of input format and constraints
- **Output Description** – Description of expected output

The user fills all three text boxes and clicks **Predict**.

Backend Processing

1. All three inputs are concatenated internally into a single text string
2. TF-IDF and handcrafted features are extracted
3. Hierarchical SVM predicts the difficulty class
4. The corresponding class-conditional SVR predicts the difficulty score

Output Displayed

- Predicted difficulty class (Easy / Medium / Hard)

- Predicted numerical difficulty score

This deployed interface loads the same serialized models (`.pkl` files) submitted in the GitHub repository, ensuring full consistency between reported results, live predictions, and the submitted code.

---

9. Conclusion

This project demonstrates that programming problem difficulty can be effectively predicted using only textual descriptions. Key contributions include:

- Hybrid TF-IDF + handcrafted feature engineering
- Cost-sensitive hierarchical classification
- Class-conditional regression for robust score prediction

Future improvements may include transformer-based embeddings and cross-platform generalization.

---

10. Repository Structure

The submitted GitHub repository contains **all necessary executable code and trained models** required to verify the project results.

Current repository contents: - `app.py` – Streamlit web application (loads trained models and performs inference) - `tfidf.pkl` – Trained TF-IDF vectorizer - `extra_feature_scaler.pkl` – MinMax scaler for handcrafted features - `selector.pkl` – Chi-square feature selector (top 5000 features) - `stage1_easy_vs_noneasy.pkl` – Stage-1 hierarchical classifier (Easy vs Non-Easy) - `stage2_medium_vs_hard.pkl` – Stage-2 classifier (Medium vs Hard) - `svr_models.pkl` – Class-conditional regression models - `requirements.txt` – Python dependencies - `README.md` – Project overview and usage instructions - `AutoJudge_Training_Reference.ipynb` – Reference Colab notebook

Note on Training Code

The primary training experiments for this project were conducted offline, and the final trained models are stored as serialized `.pkl` files in the repository. These trained artifacts are the ones used by the web application and reported in this document.

For additional clarity and transparency, a **reference Colab notebook (**`AutoJudge_Training_Reference.ipynb`**)** has been included in the repository. This notebook reproduces the same training pipeline and methodology and is provided **only for reference and reproducibility**. The reported results in this document correspond to the trained models and not to any subsequent retraining.

This setup ensures: - Consistency between reported results and deployed models - Reproducibility for reviewers - Faster verification without requiring model retraining