AutoJudge: AI-Powered Programming Problem Difficulty Predictor

Project Report

Author: Ishaan Arora
Open Project By: ACM Club IIT Roorkee
Submission Date: 8 January 2026

---

1. Introduction

1.1 Problem Statement

Competitive programming platforms like Codeforces, LeetCode, and CodeChef host thousands of programming problems with varying difficulty levels. Accurately predicting the difficulty of a new problem is challenging and traditionally relies on manual assessment by problem setters. This project aims to automate difficulty prediction using machine learning techniques.

1.2 Objectives

- Develop a classification model to categorize problems into Easy, Medium, or Hard
- Build a regression model to predict numerical difficulty scores
- Create a hybrid approach combining both models for improved accuracy
- Deploy the solution as a web application for practical use

1.3 Scope

The system analyzes problem text (description, input/output format, constraints) and predicts:

1. Difficulty class (Easy/Medium/Hard)
2. Numerical difficulty score (1.1-9.7 scale)

---

2. Dataset

2.1 Data Source

The dataset consists of 4,112 competitive programming problems collected from various online judges provided in the problem statement. Each problem contains:

- **Title**: Problem name
- **Description**: Full problem statement
- **Input Description**: Input format and constraints
- **Output Description**: Expected output format
- **Problem Class**: Difficulty category (Easy/Medium/Hard)
- **Problem Score**: Numerical difficulty rating (1.1-9.7)

2.2 Data Distribution

Class	Count	Percentage
Easy	680	16.5%
Medium	1,310	31.9%
Hard	2,122	51.6%

2.3 Train-Test Split

- **Training Set**: 3,289 samples (80%)
- **Test Set**: 823 samples (20%)
- **Random State**: 42 (for reproducibility)

---

3. Data Preprocessing

3.1 Text Cleaning

The raw problem text undergoes several preprocessing steps:

```
def preprocess_text(text):
    # Replace LaTeX math expressions
    text = re.sub(r'\$[^$]*\$', ' MATHFORMULA ', text)
    # Replace LaTeX commands
    text = re.sub(r'\\[a-zA-Z]+\{[^}]*\}', ' LATEXCMD ', text)
    # Normalize large numbers
    text = re.sub(r'\b\d{4,}\b', ' LARGENUM ', text)
    text = re.sub(r'\b\d+\b', ' NUM ', text)
    # Remove special characters
    text = re.sub(r'[^\w\s]', ' ', text)
    # Normalize whitespace
    text = re.sub(r'\s+', ' ', text)
    return text.strip().lower()
```

3.2 Text Combination

Problem components are combined into a single text field:

```
full_text = title + description + input_description + output_description
```

---

4. Feature Engineering

4.1 TF-IDF Features

Two TF-IDF vectorizers extract text features:

Word-level TF-IDF (8,000 features)

- N-gram range: (1, 2)
- Stop words: English
- Min document frequency: 3
- Max document frequency: 85%
- Sublinear TF scaling: Enabled

Character-level TF-IDF (3,000 features)

- N-gram range: (3, 4)
- Analyzer: Character
- Min document frequency: 5
- Max document frequency: 90%

4.2 Engineered Features (38 features)

Custom features capture domain-specific patterns:

Length & Structure Features (7)

- Description character length
- Full text character length
- Title character length
- Description word count
- Input description word count
- Sentence count
- Newline count

Mathematical Complexity (5)

- Math operators count (+, -, *, /, ^, =, <, >)
- LaTeX math expressions count
- LaTeX commands count
- "log n" mentions
- Polynomial notation (n^2, n^3)

Constraint Analysis (4)

- Total numbers in text
- Numbers ≥ 1,000
- Numbers ≥ 100,000
- Has number ≥ 1,000,000 (binary)

Algorithm Keywords (8 categories)

- Graph/Tree: graph, tree, node, edge, vertex, dfs, bfs, path

- Dynamic Programming: dp, dynamic, memoization, optimal, subproblem
- Greedy: greedy, minimum, maximum, best, optimal
- Sorting: sort, sorted, order, arrange
- Searching: search, find, binary, locate
- Data Structures: array, list, stack, queue, heap, priority
- String Algorithms: string, substring, pattern, match, palindrome
- Number Theory: modular, modulo, gcd, prime, factor

Complexity Indicators (10)

- Words indicating difficulty: complex, complicated, difficult, advanced, sophisticated
- Words indicating simplicity: simple, basic, straightforward, easy, trivial

Input Format Features (4)

- Variable letter count (n, m, q, k, t, i, j)
- "test case" mentions
- Has "multiple" keyword
- Has "array" or "list" keyword

---

5. Model Architecture

5.1 Classification Model

An ensemble VotingClassifier combines three models:

Random Forest Classifier

- Estimators: 300
- Max depth: 20
- Min samples split: 5
- Class weight: Balanced

Gradient Boosting Classifier

- Estimators: 150
- Learning rate: 0.1
- Max depth: 6
- Subsample: 0.8

Logistic Regression

- C: 1.5
- Class weight: Balanced
- Solver: liblinear

Voting method: Soft voting (probability-based)

5.2 Regression Model

An ensemble VotingRegressor combines three models:

Random Forest Regressor

- Estimators: 300
- Max depth: 25
- Min samples split: 3

Gradient Boosting Regressor

- Estimators: 200
- Learning rate: 0.08
- Max depth: 8
- Subsample: 0.8

Ridge Regression

- Alpha: 0.5

5.3 Hybrid Approach

The hybrid model uses classification to constrain regression predictions:

```
class_boundaries = {
    'easy': (1.1, 2.8),
    'medium': (2.8, 5.5),
    'hard': (5.5, 9.7)
}

# Constrain regression score to class boundaries
if score < min_boundary:
    score = min_boundary
elif score > max_boundary:
    score = max_boundary
```

This ensures consistency between predicted class and score.

---

6. Experimental Results

6.1 Classification Metrics

Metric	Value
Accuracy	**54.80%**
Macro Precision	0.5008
Macro Recall	0.4797
Macro F1-Score	0.4846

Per-Class Performance:

Class	Precision	Recall	F1-Score	Support
Easy	0.4722	0.3750	0.4180	136
Medium	0.4118	0.3206	0.3605	262
Hard	0.6184	0.7435	0.6752	425

6.2 Confusion Matrix

```
                  Predicted
                  Easy    Medium    Hard
Actual Easy        51       42       43
Actual Medium      26       84      152
Actual Hard        31       78      316
```

Analysis:

- The model performs best on Hard problems (74.35% recall)
- Medium problems are most challenging (32.06% recall)
- Easy problems show moderate performance (37.50% recall)

6.3 Regression Metrics

Metric	Raw Regression	Hybrid Model
MAE	1.6547	**1.6536**
RMSE	1.9922	2.0527
R^2 Score	0.1732	0.1222

6.4 Hybrid Constraint Impact

Metric	Value
Adjustments Made	376/823 (45.7%)
Predictions Improved	208 (25.3%)
Predictions Worsened	167 (20.3%)
Net Benefit	+41 predictions

---

7. Web Interface

7.1 Architecture

The application uses a client-server architecture:

- **Backend**: Flask REST API (Python)
- **Frontend**: HTML/CSS/JavaScript (Single Page Application)
- **Communication**: JSON over HTTP

7.2 API Endpoints

POST /predict

- Input: Problem description, input format, output format
- Output: Difficulty class, scores, confidence, explanation, tags

GET /health

- Output: Server status and model loading status

7.3 User Interface Features

1. **Input Form**: Text areas for problem description and formats
2. **Pie Chart**: Visual representation of class probabilities
3. **Score Display**: Raw score, hybrid score, Codeforces rating
4. **Confidence Bars**: Progress bars showing class probabilities
5. **Explanation**: AI-generated analysis of the problem
6. **Tags**: Relevant algorithm categories

7.4 Sample Prediction

Input:

```
 Description: Find the sum of two numbers a and b.
Input: Two integers a and b
Output: Print the sum
```

Output:

- Class: Easy (73% confidence)
- Hybrid Score: 2.40
- Codeforces Rating: 1100
- Tags: implementation, basic logic, beginner friendly

---

8. Conclusions

8.1 Summary

This project successfully developed a hybrid machine learning system for predicting programming problem difficulty. Key achievements:

1. Built a classification model achieving 54.80% accuracy on three-class prediction
2. Developed a regression model with 1.6536 MAE for numerical scoring
3. Implemented a hybrid approach that improves prediction consistency
4. Deployed a functional web application for practical use
5. Developed a feature that would prvide tags to a problem

8.2 Challenges

- **Class Imbalance**: Hard problems dominate the dataset (51.6%)
- **Subjective Difficulty**: Problem difficulty is inherently subjective
- **Medium Class Ambiguity**: Medium problems share characteristics with both Easy and Hard

8.3 Future Work

- Incorporate code solution analysis for better predictions
- Use transformer-based models (BERT, GPT) for text understanding
- Add problem tags as additional features
- Expand dataset with more balanced class distribution
- Implement user feedback loop for continuous improvement

---

9. References

1. Scikit-learn Documentation - https://scikit-learn.org/
2. Flask Documentation - https://flask.palletsprojects.com/
3. TF-IDF Vectorization - Manning et al., "Introduction to Information Retrieval"
4. Ensemble Methods - Breiman, "Random Forests", Machine Learning, 2001
5. Gradient Boosting - Friedman, "Greedy Function Approximation", Annals of Statistics, 2001

---

10. Appendix

A. Project Structure

```
AutoJudge/
├── README.md
├── requirements.txt
├── problems_data.jsonl
├── train_both_models_same_split.py
├── backend_service.py
├── simple_frontend.html
└── models_same_split/
    ├── classification.pkl
    ├── regression.pkl
    ├── class_tfidf_word.pkl
    ├── class_tfidf_char.pkl
    ├── class_scaler.pkl
    ├── reg_tfidf_word.pkl
    ├── reg_tfidf_char.pkl
    └── reg_scaler.pkl
```

B. Dependencies

```
flask>=2.0.0
flask-cors>=3.0.0
pandas>=1.5.0
numpy>=1.21.0
scikit-learn>=1.3.0
scipy>=1.9.0
joblib>=1.2.0
```

C. GitHub Repository

https://github.com/Tech-Ishaan13/Auto-Judge-Model-ACM