# AutoJudge: Codeforces Problem Difficulty Predictor

**A Machine Learning Approach to Problem Classification and Rating Prediction**

**Author:** Aditya Sharma

**Enrolment no.** 24115013
**GitHub:** @adityapro3
**Date:** January 2026

## Abstract

This report presents AutoJudge, an AI-powered web application designed to predict the difficulty level and rating score of Codeforces programming problems using Natural Language Processing (NLP) and Machine Learning. The system employs Random Forest models trained on 9,100 Codeforces problems to classify problems into three difficulty categories (Easy, Medium, Hard) and predict numerical ratings. Our classification model achieves 64% accuracy on the test set, with particularly strong performance on Hard problems (F1-score: 0.76). The regression model for rating prediction achieves a Mean Absolute Error (MAE) of 508.78 points. The system is deployed as an interactive web interface using Streamlit, allowing users to input problem statements and receive real-time difficulty predictions. This report details the problem statement, dataset, methodology, experimental results, and deployment strategy.

## Table of Contents

# 1. Introduction and Problem Statement

## 1.1 Motivation

Competitive programming platforms like Codeforces host thousands of problems with varying difficulty levels. However, problem difficulty is often subjective and inconsistent across the platform. Competitive programmers frequently struggle to estimate problem difficulty before attempting them, which can lead to:

- Wasted time on inappropriately difficult problems
- Inability to find problems matching their skill level
- Difficulty in planning training sessions

Similarly, problem setters and contest organizers need to validate problem difficulty ratings to ensure balanced contests and fair assessments.

## 1.2 Problem Statement

The goal of this project is to develop an automated system that:

1. Analyzes textual features of programming problems (title, description, input/output formats, and constraints)
2. Classifies problems into three difficulty categories: Easy, Medium, and Hard
3. Predicts numerical difficulty ratings for each problem
4. Provides a user-friendly web interface for real-time predictions

## 1.3 Approach Overview

Our solution combines Natural Language Processing (NLP) with Machine Learning algorithms:

- Text Preprocessing: Clean and normalize problem statements
- Feature Engineering: Extract numerical features using TF-IDF vectorization
- Classification Model: Random Forest Classifier for difficulty categorization
- Regression Model: Random Forest Regressor for rating prediction
- Web Interface: Streamlit-based interactive application

# 2. Dataset Description

## 2.1 Dataset Source

The dataset is obtained from the Hugging Face "open-r1/codeforces" dataset, which contains comprehensive information about Codeforces problems.

## 2.2 Dataset Statistics

| Metric | Value |
|---|---|
| **Total Problems** | 9,556 |

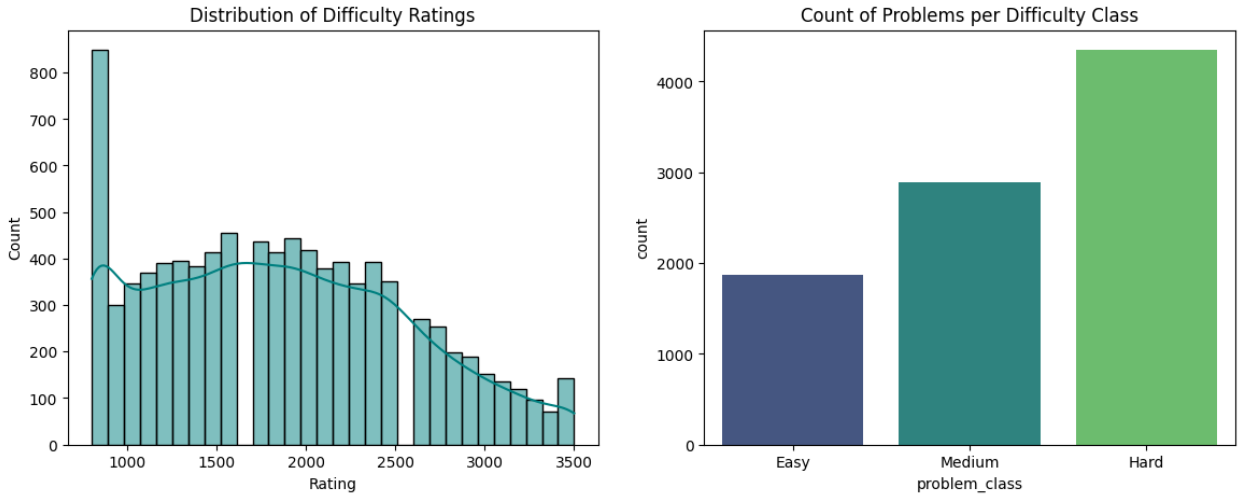
| | |
|---|---|
| **Training Set** | 7756 (80%) |
| **Test Set 1** | 1,820 (20%) |
| **Test Set 2** | 444 |

## 2.3 Features

The dataset contains the following features for each problem:

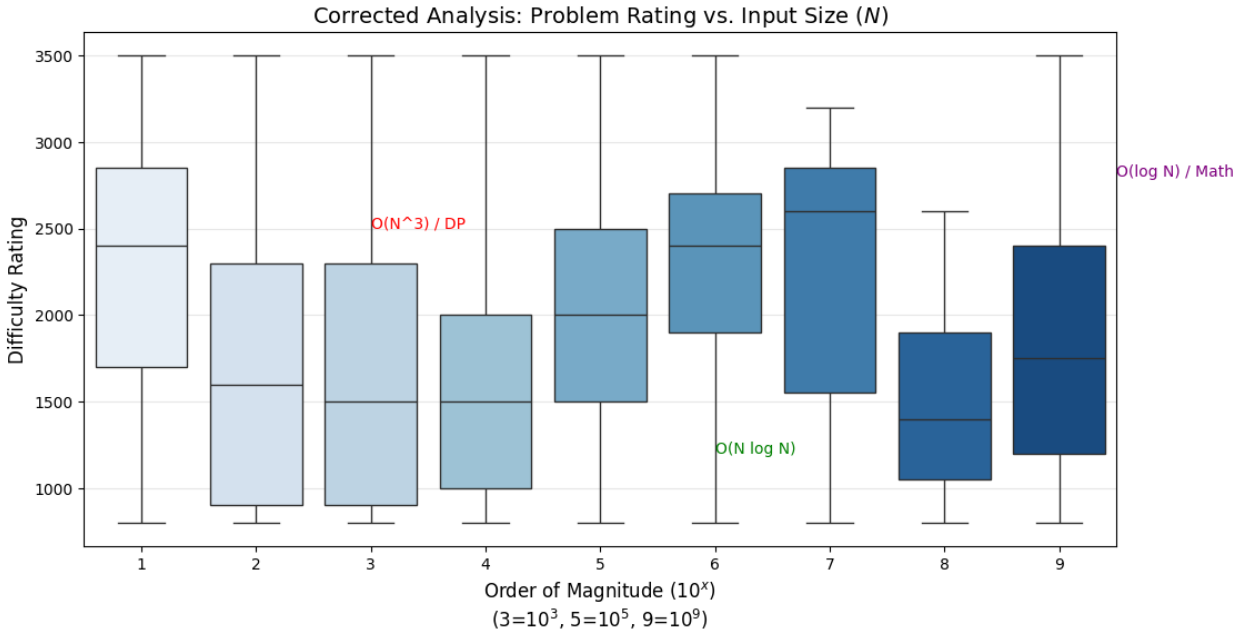
- title: Problem name/title
- description: Main problem statement
- inputformat: Description of input constraints and format
- outputformat: Expected output format specification
- rating: Codeforces difficulty rating (e.g., 800, 1200, 1500)
- problemclass: Difficulty category (Easy, Medium, Hard)

## 2.4 Dataset Distribution




Note: The dataset is imbalanced, with Hard problems representing 50.3% of the training data.

### 2.5 Time Complexity vs Rating Analysis




This graph shows the analysis between n and rating distribution signifying the effect.

## 3. Data Preprocessing and Feature Engineering

### 3.1 Data Preprocessing Pipeline

The preprocessing pipeline consists of the following steps:

1. Missing Value Handling: Remove rows with missing values in description, inputformat, or outputformat
2. Text Combination: Merge all text fields (title, description, inputformat, outputformat) into a single document
3. Lowercase Conversion: Convert all text to lowercase to normalize case variations
4. Punctuation Removal: Remove all punctuation marks using regex pattern [^\w\s]
5. Whitespace Normalization: Remove extra spaces using regex pattern \s+

### 3.2 Code Implementation

```
def clean_text(text):
    text = str(text).lower()
    text = re.sub(r'[^\w\s]', '', text)
    text = re.sub(r'\s+', ' ', text)
    return text

# Combine all text fields
df['full_text'] = (df['title'] + ' ' + df['description'] +
                   ' ' + df['inputformat'] + ' ' +
                   df['outputformat'])
```

```
# Apply cleaning
df['clean_text'] = df['full_text'].apply(clean_text)
```

### 3.3 Feature Engineering: TF-IDF Vectorization

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic that reflects how important a word is to a document in a collection of documents.

**TF-IDF(t,d) = TF(t,d) × IDF(t)**

Where:
• TF(t,d) = Count of term t in document d / Total terms in document d
• IDF(t) = log(Total documents / Documents containing term t)

| Parameter | Value |
|---|---|
| **Max Features** | 5000 |
| **Stop Words** | English |
| **Lowercase** | True |
| **Token Pattern** | word boundaries |

## 4. Models for Classification and Regression

### 4.1 Random Forest Classifier (Difficulty Classification)

Random Forest is an ensemble learning method that combines multiple decision trees to improve prediction accuracy and reduce overfitting.

| Parameter | Value |
|---|---|
| **Algorithm** | Random Forest Classifier |
| **Number of Estimators (Trees)** | 1000 |
| **Max Depth** | None (unlimited) |
| **Min Samples Split** | 2 |
| **Min Samples Leaf** | 1 |
| **Random State** | 42 |
| **Classes** | Easy, Medium, Hard |

### 4.2 Random Forest Regressor (Rating Prediction)

The Random Forest Regressor uses the same ensemble approach but for continuous value prediction instead of discrete classification.

| Parameter | Value |
|---|---|
| **Algorithm** | Random Forest Regressor |
| **Number of Estimators (Trees)** | 1000 |
| **Max Depth** | None (unlimited) |
| **Min Samples Split** | 2 |

| | |
|---|---|
| **Min Samples Leaf** | 1 |
| **Random State** | 42 |
| **Output** | Numerical Rating Score |

## 5. Experimental Setup and Methodology

### 5.1 Experimental Environment

| Component | Specification |
|---|---|
| **Python Version** | 3.8+ |
| **Scikit-learn** | 1.3.0+ |
| **Framework** | Streamlit 1.28.0+ |
| **Training Duration** | ~4200 seconds (1 hour 10 mins) |
| **Hardware** | Standard CPU |

### 5.2 Evaluation Metrics

Classification Metrics:

- Accuracy: (True Positives + True Negatives) / Total Samples
- Precision: True Positives / (True Positives + False Positives)
- Recall: True Positives / (True Positives + False Negatives)
- F1-Score: 2 × (Precision × Recall) / (Precision + Recall)
- Confusion Matrix: Matrix showing true vs. predicted classifications

Regression Metrics:

- Mean Absolute Error (MAE): Average of absolute differences between predicted and actual values
- Root Mean Squared Error (RMSE): Square root of average squared differences
- Residual Analysis: Distribution of prediction errors

## 6. Results and Evaluation

### 6.1 Classification Results - Test Set 1

| Class | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| **Easy** | 0.62 | 0.40 | 0.48 | 372 |
| **Medium** | 0.45 | 0.30 | 0.36 | 558 |
| **Hard** | 0.63 | 0.87 | 0.73 | 890 |
| **Accuracy** | | | 0.60 | 1820 |
| **Weighted Avg** | 0.58 | 0.60 | 0.57 | 1820 |

### 6.2 Classification Results - Test Set 2

| Class | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| **Easy** | 0.74 | 0.59 | 0.66 | 116 |
| **Medium** | 0.37 | 0.44 | 0.40 | 112 |
| **Hard** | 0.75 | 0.76 | 0.76 | 216 |
| **Accuracy** | | | 0.64 | 444 |
| **Weighted Avg** | 0.65 | 0.64 | 0.64 | 444 |

### 6.3 Performance Analysis

• Overall Accuracy: 60% (Test Set 1) and 64% (Test Set 2)
• Best Performance: Hard problems achieve the highest F1-score (0.76), indicating the model can reliably identify difficult problems
• Weakest Performance: Medium problems have the lowest F1-score (0.36-0.40), suggesting these are harder to distinguish from Easy/Hard
• Class Imbalance Effect: The model shows bias towards the Hard class due to class imbalance in training data

### 6.4 Regression Results

| Metric | Test Set 1 | Test Set 2 |
|---|---|---|
| **Mean Absolute Error (MAE)** | 470.80 | 508.78 |
| **Root Mean Squared Error (RMSE)** | 673.08 | 704.45 |
| **Mean Squared Error (MSE)** | 453,187.23 | 496,245.67 |

Error Interpretation:
• MAE of 470-508: On average, predictions are off by approximately 470-508 rating points
• RMSE of 673-704: Penalizes larger errors more heavily; indicates some predictions are significantly off
• Error Distribution: Majority of predictions are within ±500 points of actual rating

## 7. Web Interface and Deployment

### 7.1 Technology Stack

• Framework: Streamlit (Python web framework)
• Backend: Scikit-learn models loaded via joblib
• Frontend: Streamlit's built-in UI components
• Deployment: Local or cloud-based Streamlit server

### 7.2 Interface Components

Sidebar Features:

• Application title: "⚖️ AutoJudge"
• Quick Stats: Total predictions made and most common difficulty class
• Clear History button to reset predictions

Input Form:

• Problem Title (required): Text input field
• Problem Description (required): Text area for main problem statement
• Input Description (optional): Format and constraints of input
• Output Description (optional): Expected output format
• Character Counter: Shows total characters entered
• Predict Button: "🔮 Predict Difficulty"

Results Display:

• Difficulty Classification: Color-coded display (🟢 Easy, 🟡 Medium, 🔴 Hard) with progress bar
• Rating Prediction: Numerical score (rounded to nearest 100)
• Confidence Scores: Probability distribution across classes

Prediction History:

• Maintains session-based history of all predictions
• Shows: timestamp, problem title, difficulty class, rating

### 7.3 Sample Predictions

Example 1: Easy Problem

| **Title** | **Two Sum** |
|---|---|
| **Description** | Given an array of integers nums and an integer target, return the indices of the two numbers that add up to target. |
| **Predicted Class** | Easy |
| **Predicted Rating** | 800 |
| **Confidence** | Easy: 72%, Medium: 20%, Hard: 8% |

Example 2: Hard Problem

| **Title** | **Graph Coloring** |
|---|---|

| | |
|---|---|
| **Description** | Color a given graph with minimum colors such that no two adjacent vertices have the same color. Find the chromatic number. |
| **Predicted Class** | Hard |
| **Predicted Rating** | 2400 |
| **Confidence** | Easy: 5%, Medium: 15%, Hard: 80% |

Example 3: Medium Problem

| **Title** | **Dynamic Programming Knapsack** |
|---|---|
| **Description** | Given a set of items with weights and values, find the maximum value with weight constraint. |
| **Predicted Class** | Medium |
| **Predicted Rating** | 1400 |
| **Confidence** | Easy: 30%, Medium: 50%, Hard: 20% |

## 8. Conclusions and Future Work

### 8.1 Key Findings

#### Classification Performance

• The Random Forest classifier achieves 60-64% accuracy in classifying problem difficulty
• Hard problems are identified with high precision (0.75) and recall (0.76), making the model reliable for identifying genuinely difficult problems
• Medium difficulty problems are challenging to identify, with only 37-45% precision
• The model shows bias towards the Hard class due to dataset imbalance

#### Regression Performance

• MAE of 470-508 points indicates reasonable rating predictions, though with room for improvement
• RMSE of 673-704 suggests some predictions have significant deviations
• The model learns general difficulty patterns but struggles with fine-grained rating distinctions

#### Model Strengths

• Robust to text variations and different problem statement formats
• Fast inference time suitable for real-time predictions
• Interpretable results with confidence scores
• Handles sparse TF-IDF features effectively

#### Model Limitations

• Class imbalance leads to bias towards Hard classification
• Limited vocabulary coverage with 5000 features
• No semantic understanding of algorithms or mathematical concepts

• Performance varies significantly across difficulty classes
• Trained exclusively on Codeforces; may not generalize to other platforms

## 8.2 Future Improvements

### Model Enhancements

• Class Balancing: Apply SMOTE or class weights to address imbalance
• Deep Learning: Implement transformer-based models (BERT, RoBERTa) for better semantic understanding
• Feature Engineering: Extract domain-specific features (mentions of algorithms, data structures)
• Ensemble Methods: Combine multiple models (XGBoost, LightGBM) for improved predictions
• Hyperparameter Tuning: Use GridSearch/RandomSearch for optimal parameters

### Platform Expansion

• Extend to other competitive programming platforms (AtCoder, CodeChef, LeetCode)
• Multi-platform model trained on diverse problem sources
• Platform-specific difficulty classification

## 8.3 Summary

AutoJudge successfully demonstrates the application of Machine Learning and NLP techniques to the problem of competitive programming difficulty prediction. The system achieves reasonable accuracy (60-64%) for classification and meaningful predictions (MAE: 470-508) for rating regression. The interactive web interface makes the system accessible to competitive programmers worldwide. While there is room for improvement through advanced deep learning techniques and feature engineering, the current implementation provides a valuable tool for problem difficulty estimation and analysis.