AutoJudge: Predicting Programming Problem Difficulty Project Report

Online competitive programming platforms such as Codeforces, Kattis, and CodeChef categorize problems into difficulty levels like Easy, Medium, and Hard, and often assign a numerical difficulty score as well. However, these ratings are typically based on human judgment, community feedback, and historical performance data, which can be subjective, slow to update, and inconsistent across platforms. As a result, new problems may lack reliable estimatations for difficulty, and as a result the percieved difficulty compared to the labels can be different.

The objective of this project is to build an intelligent system that can automatically predict both the difficulty class (Easy, Medium, Hard) and a numerical difficulty score using only the textual content of a problem. The model relies on the problem description, input format, and output format, without using any external metadata. Using a labeled dataset of programming problems, the system learns patterns that correlate textual structure and technical cues with problem complexity. Finally, the model is deployed through a simple web interface that allows users to paste a new problem statement and instantly receive a predicted difficulty class and score.

```python
import numpy as np
import pandas as pd
import seaborn as sns
import re

data =
pd.read_json("/kaggle/input/problems-difficulty/problems_data.jsonl",
lines=True)

print(data.shape)
print(data.columns)
```

```
(4112, 8)
Index(['title', 'description', 'input_description',
'output_description',
       'sample_io', 'problem_class', 'problem_score', 'url'],
      dtype='object')
```

Dataset

The dataset used is the same as the one provided with the problem statement, consisting of the problem data along with already assigned category from 'easy', 'medium' or 'hard' and a difficulty score between 1.1 and 0.7

Testing Parameters

Accuracy and **Macro-F1 score** are the two metrics used to evaluate how well the models are doing. Accuracy can oftentimes not be enough as a model highly biased towards one category give high accuracy but that does not make it a good model. Macro F1 on the other hand rewards models when they are correct and treats all the classes equally. For our use case, a combination of the two give an idea of where the model is falling short. If it has high accuracy and low F1 score, it is because the model is highly biased towards a certain category. Confusion matrices are furthur used to check how well the models are performing for different categories.

Feature Engineering

Stage-1

1) All the text input including "title", "description", "input description", "output description" were combined to create a single input. Since models expect a single text input per sample, this gives the models all the context required in a single feature. 2) Text length and word count were created as new features for the combined text feature above with the assumption that harder problems have comparatively more text than the other categories 3) A feature for math symbol count was created as hard problems often include formulas, inequalities, constraints and mathematical expressions.

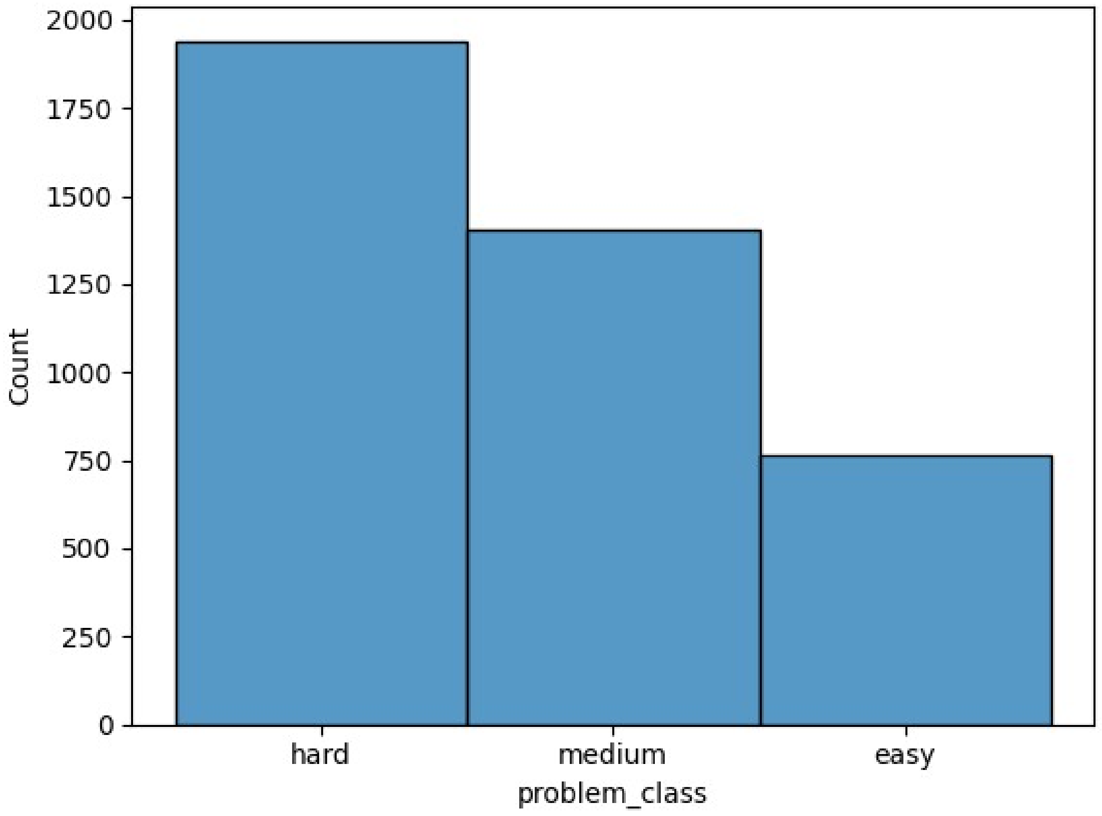
To evaluate the impact of feature engineering, we trained models using only basic features such as combined text, text length, and mathematical symbol count. While Random Forest achieved moderate performance (Macro F1 ≈ 0.40), linear models such as Logistic Regression and SVM exhibited poor class balance. This indicates that surface-level features alone are insufficient for difficulty classification. Incorporating domain-specific features such as algorithmic keywords, constraint indicators, and complexity metrics significantly improved class separation, particularly for Hard problems.

4) A count keywords feature which counts how many times advanced algorithm keywords appear in each problem. More frequency indicate medium or tough problems.

```
data["combined_text"] = (
    data["title"].fillna("") + " " +
    data["description"].fillna("") + " " +
    data["input_description"].fillna("") + " " +
    data["output_description"].fillna("") + " "

    )
data["text_length"] = data["combined_text"].apply(len)
data["word_count"] = data["combined_text"].apply(lambda x:
len(x.split()))

def count_math_symbols(text):
    return len(re.findall(r"[+\-*/%=<>]", text))

data["math_symbol_count"] =
data["combined_text"].apply(count_math_symbols)
```

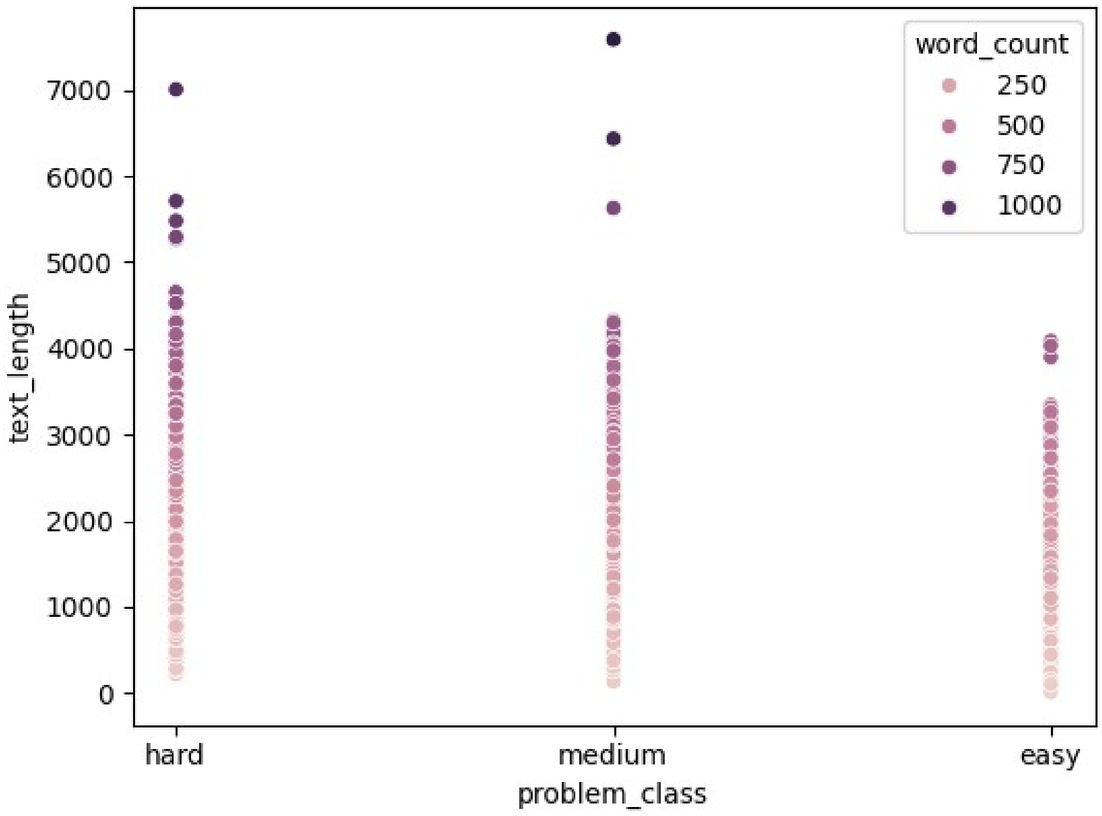
```
sns.histplot(data['problem_class'])
```

```
<Axes: xlabel='problem_class', ylabel='Count'>
```



```
sns.scatterplot(x="problem_class", y="text_length", hue =
"word_count", data= data)
```

```
<Axes: xlabel='problem_class', ylabel='text_length'>
```



As can be seen, most problems in the dataset lie in hard category, and there are very less samples for easy category. Also, features such as text_length and word_count do not add much to the categorisation. At best, they might help distinguish easy from medium+hard.

Models(for classification)

Random Forest, SVM, Logistic Regression

Initial models showed strong bias toward the 'Medium' class, with most 'Hard' problems being misclassified. After introducing class-specific features such as advanced algorithm keyword counts, numeric density, and complexity indicators, the Random Forest classifier showed a significant improvement in detecting 'Hard' problems, achieving high recall for the Hard class and more meaningful difficulty separation.

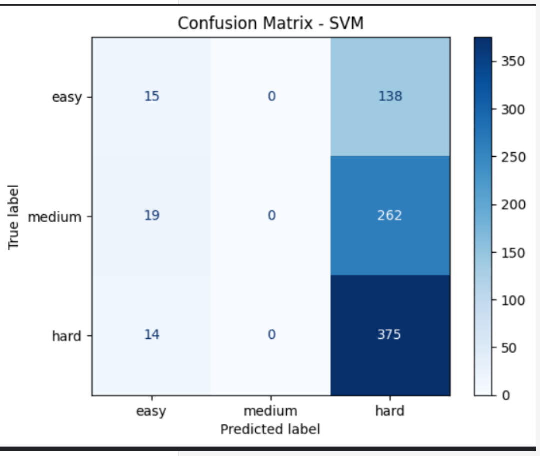
Although the Linear SVM achieved very high recall for the 'Hard' class, the confusion matrix revealed severe class bias, with the majority of Easy and Medium problems being misclassified as Hard. This resulted in poor Macro F1-score and huge bias towards Hard category. In contrast, the Random Forest classifier maintained strong Hard detection while preserving meaningful separation across all three difficulty levels. Therefore, Random Forest was selected as the final classification model.

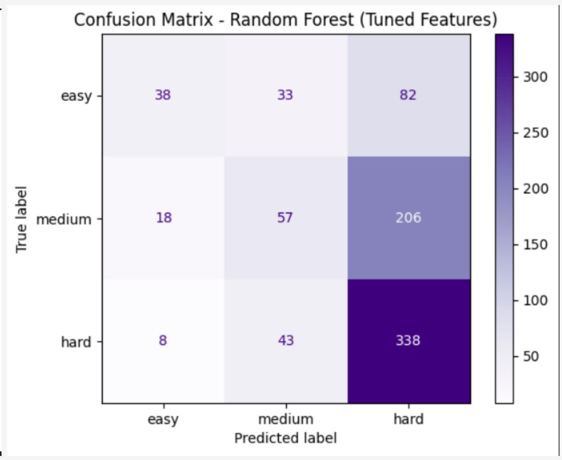
Models(for regression)

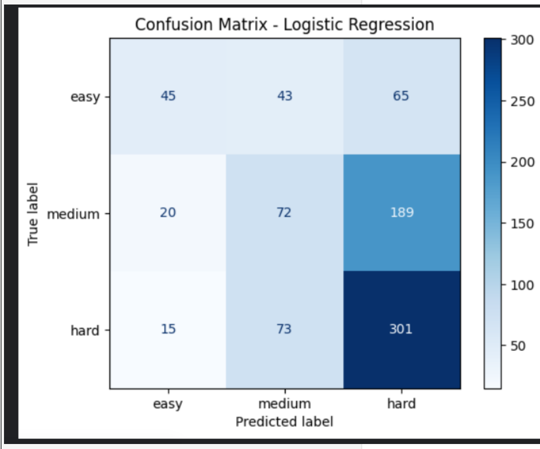
Ridge, Random Forest, and XGBoost

Although all three regression models (Ridge, Random Forest, and XGBoost) showed similar performance, their results clustering around an R2 of roughly 0.11–0.15 suggested that the main limitation came from the dataset rather than the models themselves. The textual features were able to capture broad difficulty patterns, but they did not contain enough detailed information to support highly precise score predictions. Random Forest performed best as an out-of-the-box model, successfully learning non-linear relationships between constraints, algorithmic keywords, and numeric patterns.

However, after hyperparameter tuning, Random Forest Regression achieved the highest overall accuracy and was better able to extract weak but distributed signals from the high-dimensional feature space. As a result, Random Forest Regression was chosen as the final regression model, offering the best balance between performance and optimization potential, while also acknowledging that any further improvements are largely limited by the quality and size of the dataset.







Stage - 2

1) Max constraint extraction In coding problems, the input size strictly dictates the required time complexity of the solution.If it is less than 20, The solution can be Exponential. This implies Recursion or Bitmask DP (usually Hard).If less than 100, the solution can be Floyd-Warshall or Matrix Multiplication (usually Medium).If less than 10000,the solutionimplies Sorting, Greedy, or Segment Trees (Could be Easy or Medium).If less than 10^18, the solution implies Math or Number Theory (usually Medium/Hard).

2) TF-IDF We used TF-IDF instead of a simple word-count approach so that important technical terms would carry more weight than common filler words. Words that appear frequently in a single problem but rarely across the dataset—such as algorithm names or mathematical terms are emphasized, while generic words like “input” or “program” are down-weighted. We also used N-grams (up to three words) so the model could recognize meaningful phrases like “binary search” or “lowest common ancestor,” rather than treating each word in isolation. This allowed the model to focus on complete algorithmic concepts that are more closely tied to problem difficulty.

3) Domain specific keywords

We introduced keyword-based features to inject basic domain knowledge into the model. By explicitly counting groups of words associated with advanced algorithms, simple operations, and optimization tasks, we helped the model identify difficulty signals more directly. Terms related to graphs, dynamic programming, or data structures tend to indicate harder problems, while words like “print” or “sum” are more common in easy tasks. This guided the model toward important technical cues without relying entirely on statistical learning from raw text.

Web UI Interface

The web UI is built with **Streamlit** for simplicity and speed.

1. **Input Fields:**
   - **Problem Description:** Paste the main body of the problem.
   - **Input Description:** Paste the constraints (e.g., "The first line contains an integer T...").
   - **Output Description:** Paste the required output format.
2. **Predict Button:** Click to run the model.
3. **Results:**
   - **Difficulty Class:** Displays whether the problem is Easy, Medium, or Hard.
   - **Difficulty Score:** Displays the estimated numerical rating (e.g., 4.5).

The predictions were done on some samples from the dataset only and can be seen in the demo video.