AutoJudge: Predicting Programming Problem Difficulty

By: Vaibhavi Shinde (23119042)

1. Problem Statement

Given the **textual description of a competitive programming problem**, including:

- Problem statement
- Input format
- Output format

I aim to:

1. **Classify** the problem as `Easy`, `Medium`, or `Hard`.
2. **Predict** a numeric difficulty score between **0 and 10**.

2. Dataset Description

I used a dataset stored in **`problems_data_acm.jsonl`**, where each line represents one problem.

Each problem contains:

- `description`
- `input_description`
- `output_description`
- `problem_score` (numeric difficulty label)

I converted the numeric score into difficulty classes using:

- `Score ≤ 5.0` → Easy
- `5.0 < Score ≤ 8.5` → Medium
- `Score > 8.5` → Hard

3. Data Preprocessing

3.1 Text Combination

I combined all textual fields into a single column:

```
combined_text = description + input_description +
output_description
```

This ensured that the model sees the **complete problem context**.

3.2 Train-Test Split

I split the data into:

- 80% training
- 20% testing

I used **stratified splitting** to preserve the class distribution.

4. Feature Engineering

4.1 Text Features (TF-IDF + SVD)

I used:

- **TF-IDF Vectorization** with unigrams and bigrams
- **Truncated SVD** to reduce dimensionality

This allowed me to capture semantic patterns while keeping the model efficient.

4.2 Meta Features from Problem Statements

I designed a large set of meta-features that mimic how humans judge problem difficulty.

(a) Text Statistics

- Text length (log-scaled)
- Number of lines
- Token count
- Average word length
- Type-token ratio

(b) Constraint & Numeric Signals

- Number of numeric values
- Maximum numeric constraint (log-scaled)

- Estimated maximum `n`
- Number of constraints mentioned
- Presence of Big-O notation

(c) Structural Features

- Number of examples
- Sample input/output pairs
- Code-like lines
- Use of directive words like *find*, *compute*, *print*

(d) Algorithmic Keywords

I searched for keywords related to:

- Graph algorithms
- Dynamic programming
- Advanced data structures
- Math, strings, geometry, greedy techniques

I also created **grouped keyword counts** such as:

- `graph_kw_count`
- `dp_kw_count`
- `advanced_ds_kw_count`

These features significantly improved model performance and interpretability.

5. Feature Cleaning & Scaling

After feature extraction:

- I removed near-constant features using **VarianceThreshold**.
- I standardized features using **StandardScaler**.

This step was essential for SVM and regression stability.

6. Model Design

6.1 Classification Model (Easy / Medium / Hard)

I used **Linear Support Vector Classifier (LinearSVC)** with:

- Balanced class weights
- Regularization (`C = 1.0`)
- High iteration limit

This model was chosen for:

- High-dimensional text data
- Good performance with sparse features
- Strong generalization

6.2 Regression Model (Difficulty Score 0–10)

For regression, I used a **Stacking Ensemble**, consisting of:

- **Ridge Regression**
- **Histogram Gradient Boosting Regressor**

These base models were combined using **RidgeCV** as the final estimator.

This design helped capture both:

- Linear trends
- Non-linear relationships

7. Model Training

Both models were trained on the same feature space:

- Text embeddings
- Meta-features

All preprocessing objects (text pipeline, scaler, variance filter) were saved together using **joblib**, ensuring reproducibility.

8. Evaluation Results

8.1 Classification Performance

- **Training Accuracy:** 64.24%
- **Test Accuracy:** 57.23%

```
Training Accuracy  : 0.6424
Test Accuracy      : 0.5723

Confusion Matrix (Test Set):
[[231 142  17]
 [119 237  36]
 [ 15  23   3]]
```

```
Classification Report:
              precision    recall  f1-score   support

        Easy       0.63      0.59      0.61       390
        Hard       0.05      0.07      0.06        41
      Medium       0.59      0.60      0.60       392

    accuracy                           0.57       823
   macro avg       0.43      0.42      0.42       823
weighted avg       0.58      0.57      0.58       823
```

Key Observations:

- Easy and Medium classes are predicted reasonably well.
- Hard problems are often misclassified due to low sample count.
- Macro F1 score is low because the Hard class is under-represented.

8.2 Regression Performance

```
--------------------------------------------
Metric                Train           Test
--------------------------------------------
MAE                   1.5160          1.6500
RMSE                  1.8435          1.9903
R²                    0.2788          0.1839
--------------------------------------------
```

Interpretation:

- The model predicts difficulty within ~1.6 points on average.
- Moderate R^2 is expected due to subjectivity in difficulty labels.

9. My Alternative Approach: SBERT + TF-IDF Experiments

During the development of this project, I also experimented with an alternative modeling approach using **SBERT (Sentence-BERT) embeddings** in combination with TF-IDF features. My initial hypothesis was that SBERT, being a deep semantic model, would better understand the meaning of problem statements and therefore improve difficulty prediction.

9.1 Why I Tried SBERT

I chose SBERT because:

- It produces **dense semantic embeddings** that capture sentence-level meaning.
- It is widely used in modern NLP tasks such as semantic similarity and clustering.
- Competitive programming problems often have long descriptions where semantic understanding seems important.

In this alternative pipeline:

- I generated SBERT embeddings for the `combined_text`.
- I concatenated these embeddings with TF-IDF features and meta-features.
- I trained the same classification models for a fair comparison.

9.2 Observed Results

Contrary to my expectations, the model performance **decreased**:

- The **classification accuracy dropped below 50%**.
- The confusion between Easy and Medium increased.
- The Hard class performance remained poor.

After this, I removed SBERT embeddings and trained the model using **only TF-IDF + meta-features**, which resulted in a **clear improvement**, with test accuracy rising to **~57%**.

9.3 Why SBERT Performed Worse in This Project

After detailed analysis, I identified several important reasons for this behavior:

(a) CP Difficulty Depends on Rare Technical Keywords

Competitive programming difficulty is often determined by **specific algorithmic keywords**, such as:

- `segment tree`
- `bitmask`
- `flow`
- `digit dp`
- `LCA`
- `persistent`

TF-IDF is particularly strong at capturing such **rare but highly informative words**, because:

- Rare terms receive **higher TF-IDF weights**.
- These words directly correlate with problem difficulty.

SBERT, on the other hand:

- Compresses text into dense vectors.
- Tends to **smooth out rare technical tokens**.
- Focuses more on general semantic similarity than algorithm-specific signals.

As a result, crucial difficulty indicators were weakened.

example:

- Two problems may have very similar descriptions but vastly different difficulty levels.
- The presence of one keyword like `DP with bitmask` can drastically change difficulty, even if the rest of the text is similar.

TF-IDF preserves these sharp distinctions, while SBERT tends to blur them.

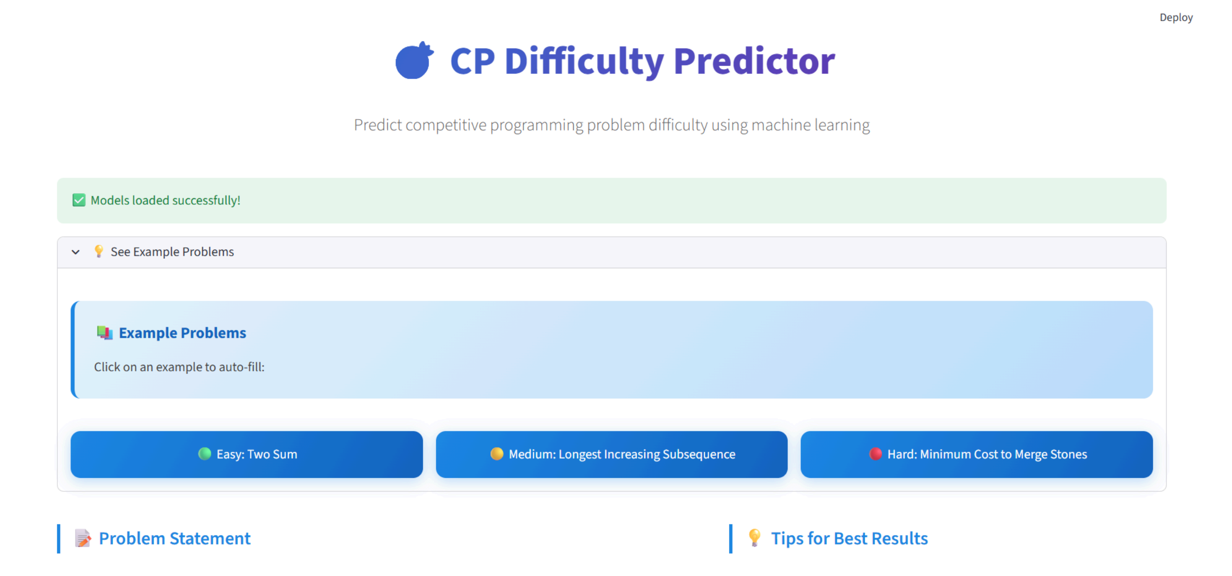
10. Conclusion

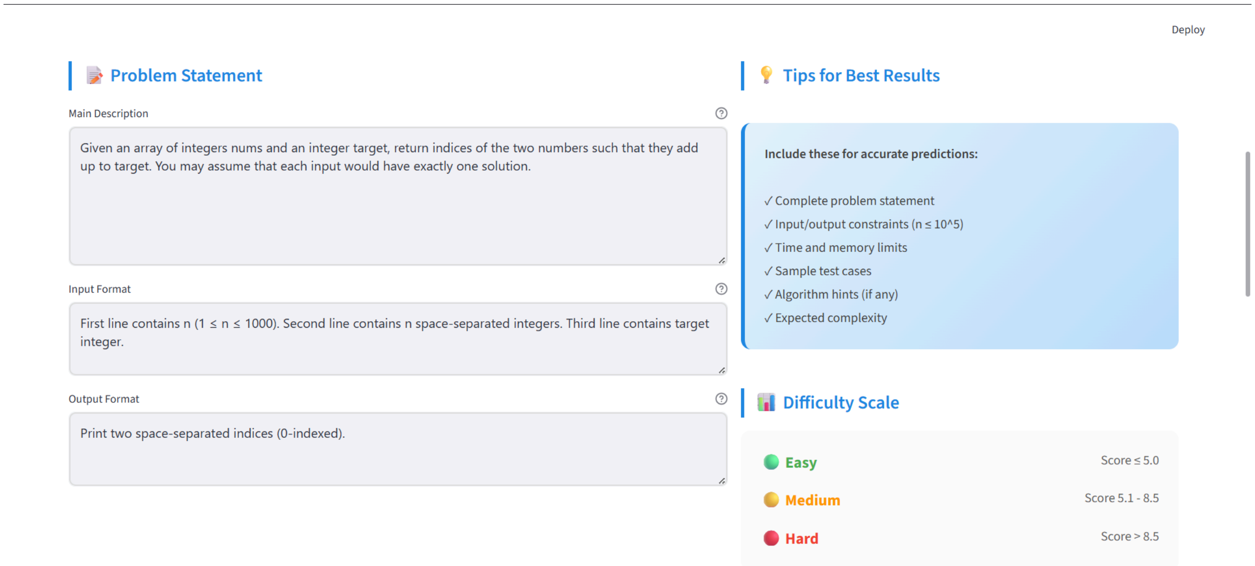
In this project, I successfully built an end-to-end machine learning system that predicts competitive programming problem difficulty using textual data. By combining NLP techniques with domain-specific feature engineering, I achieved meaningful results and created an explainable, deployable solution.

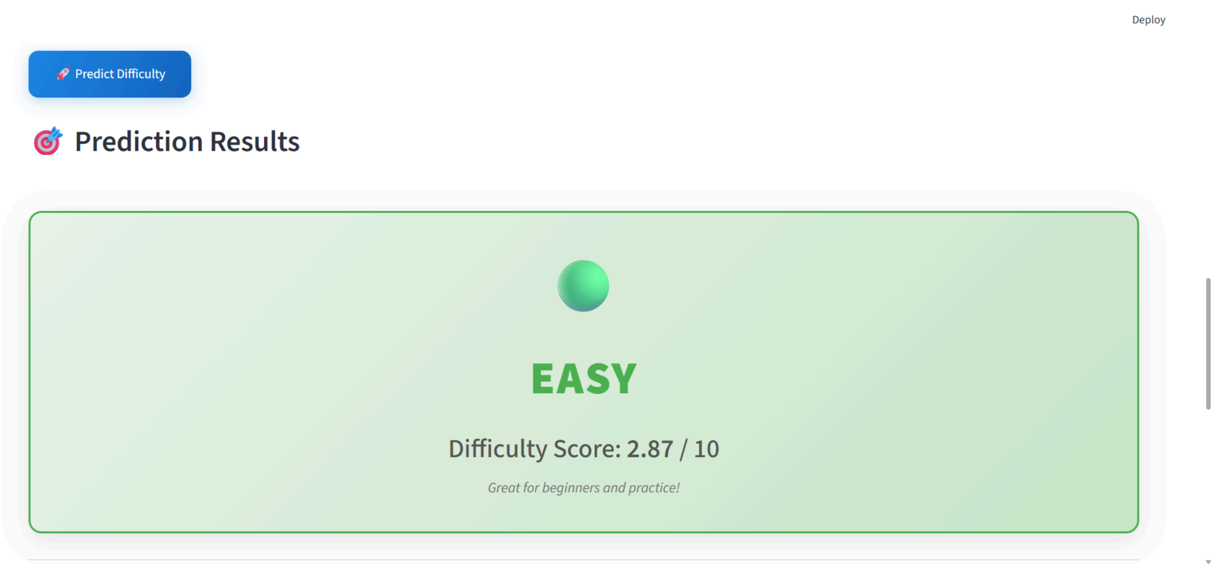
This project strengthened my understanding of:

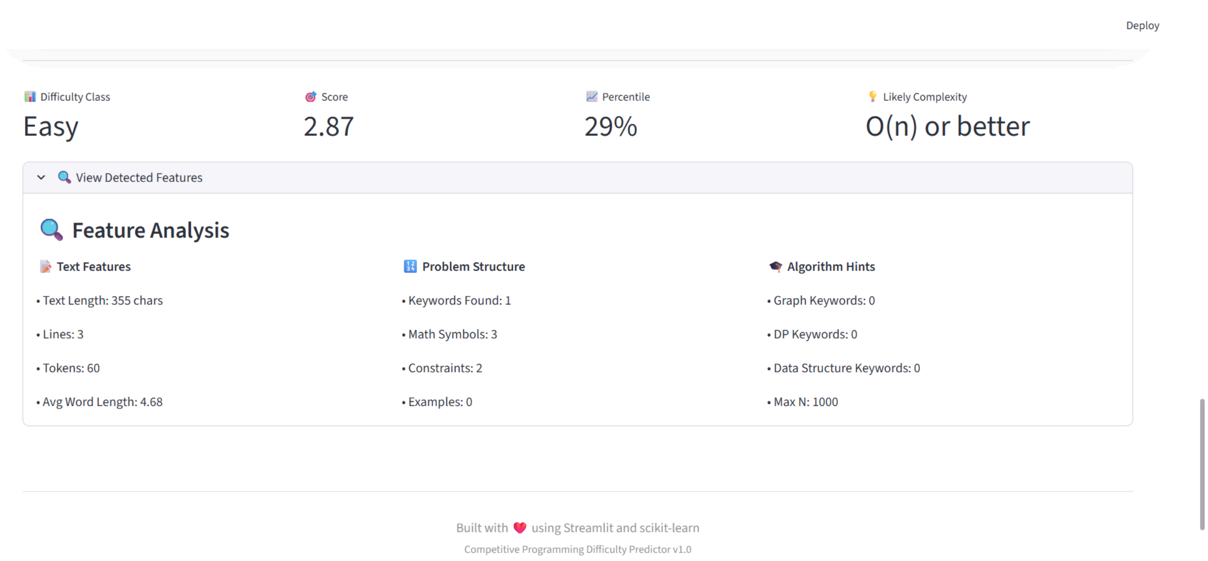
- NLP pipelines
- Feature engineering
- Model evaluation
- Deployment using Streamlit

Overall, this system demonstrates how machine learning can assist learners and educators in competitive programming ecosystems.









14. Tools & Technologies Used

- Python
- scikit-learn
- pandas, numpy

- matplotlib, seaborn
- Streamlit
- Joblib