AUTOJUDGE: PROGRAMMING PROBLEMS DIFFICULTY PREDICTION MODEL

Problem Statement

- Online coding platforms rely on manual evaluation and user feedback to assign difficulty levels to programming problems
- This project aims to automatically predict the problem class (Easy / Medium / Hard) and a numerical difficulty score using only the textual description of the problem. The system treats this as a classification and regression task based on natural language understanding.
- The textual description will contain the problem description,input and output description etc.

Dataset Used:

Dataset Link: https://huggingface.co/datasets/open-r1/codeforces

- This project uses the Codeforces dataset from Hugging Face, as Codeforces (along with CodeChef) is one of the most widely used competitive programming platforms.
- The dataset is well-curated and includes rich problem metadata such as contest details, problem ID, time and memory limits, input/output descriptions, full problem statements, and sample test cases. These comprehensive textual and structural features make it highly suitable for training an accurate difficulty classification and regression model.

Data Preprocessing and Feature Engineering Steps:

1) Dense Features :
   ➢ As the first step in preprocessing, dense statistical features are extracted, including sample_count, time_limit, memory_limit, and text_length.
   ➢ Features such as avg_in_char, avg_out_char, and avg_line measure the average number of characters and lines in the input and output across all sample test cases, while sample_count represents the number of example test cases provided with each Codeforces problem.
   ➢ Together, these features help capture the structural complexity of a problem in numerical form.

2) Complexity Based Features:

   ➢ Complexity indicator features are extracted to estimate how difficult a problem is based on its structure and content.

The index_score is derived from the problem ID, where problems with higher indices (for example, Problem D compared to Problem A) are generally more difficult in Codeforces contests.

➤ Text-based features such as avg_line_count, avg_sentence_length, and formula_symbol_count are computed by combining the problem description, input description, and output description.

Longer problems with more lines, longer sentences, and a higher number of mathematical or logical symbols usually indicate greater complexity, and these features help the model learn such difficulty patterns effectively.

3) <u>One-hot encoded features:</u>

➤ Algorithmic tags represent the main concepts required to solve a problem and are explicitly provided alongside each problem on Codeforces.

First, all unique tags appearing across the dataset (such as dp, greedy, math, graphs, geometry, and number theory) are collected to form a kind of tag vocabulary which contained all unique tags mentioned across all problems of the dataset

➤ For every problem, this vocabulary is transformed into one-hot encoded features, where a value of 1 indicates that a particular tag is mentioned for the problem and 0 indicates that it is not. This helps the model directly learn the relationship between commonly used Codeforces tags and problem difficulty.

4) <u>Textual Embeddings:</u>

To capture semantic meaning from problem statements:

➤ The NLTK library was used to tokenize each problem by splitting the problem statement, input description, and output description into individual words.

This allows the model to process the text at the word level instead of treating the entire description as a single string.

➤ Rather than using a pre-trained embedding model, a Word2Vec model was trained directly on the dataset, enabling it to better capture competitive programming–specific terminology, problem-solving language, and recurring patterns unique to Codeforces problems.

Each word is represented using a 300-dimensional embedding (emb_0 to emb_299) to encode semantic meaning. Since a problem consists of many words, mean pooling is applied to average all word embeddings, producing a single fixed-size vector that represents the overall meaning of the problem and can be easily used as input for machine learning models.

5) **Knowledge Graph Modeling:**

➤ To capture relationships between different algorithmic concepts, a Knowledge Graph (KG) is constructed where each important concept—such as dynamic programming, graphs, binary search, constraints, and other algorithmic tags—is represented as a node, and meaningful relationships between them are modeled as edges.

This structured representation enables the system to understand how algorithmic techniques are interconnected rather than treating them as independent features.

Problem metadata including algorithmic tags, categories, constraints, and technical concepts is processed and converted into (head, relation, tail) triplets, which explicitly define the connections between concepts and serve as training data for the knowledge graph.

➤ A TransE (Translation Embedding) model is then used to learn embeddings by enforcing the relationship head + relation ≈ tail, allowing the model to capture semantic relationships between algorithmic concepts.

The model is trained using Margin Ranking Loss, ensuring that valid relationships are placed closer together than incorrect ones in the embedding space.

After training, each concept is represented as a 128-dimensional vector (kg_0 to kg_127). These embeddings are appended to the problem's feature set before being passed to the final prediction models.

➤ The construction and integration of this Knowledge Graph is a key novelty of the project, as it combines structured relational knowledge with traditional feature extraction and classical machine learning models, leading to a more informed and interpretable difficulty prediction system.

Model Selection and Training

After extracting dense statistical features, generating Word2Vec embeddings, and learning knowledge graph (KG) embeddings, all features were concatenated to form the final feature matrix X, consisting of 9,492 rows and 475 features. A train–test split was then applied with a test size of 0.2, using stratified sampling on the class labels to preserve the original distribution of Easy, Medium, and Hard problems.

This resulted in X_train with shape (7,593, 475) and X_test with shape (1,899, 475). The same train–test indices were used for both classification and regression tasks, ensuring that X_train, y_train and X_test, y_test correspond to identical subsets of the dataset. This consistency prevents data leakage and ensures fair and reliable comparison across both prediction tasks.

Multiple Machine Learning models were evaluated for both classification and regression tasks before final selection such as LightGBM, CatBoost, Logistic/Linear Regression, SVM, RandomForest, XGBoost, GradientBoost, AdaBoost.

1) <u>Classification Results for all models are as follows:</u>

- For classification task Logistic Regression,SVC,RandomForest,XGBclassifier, Boosting classifiers and CatBoost Classifier were used

  For tree-based boosting models such as LightGBM and XGBoost a label encoder was applied to convert the categorical difficulty classes Easy, Medium, and Hard into numerical labels 0, 1, and 2, respectively.

  This encoding is required because these models operate on numerical targets and cannot directly process string-based class labels. The encoded labels ensure correct and efficient training of the classification models.
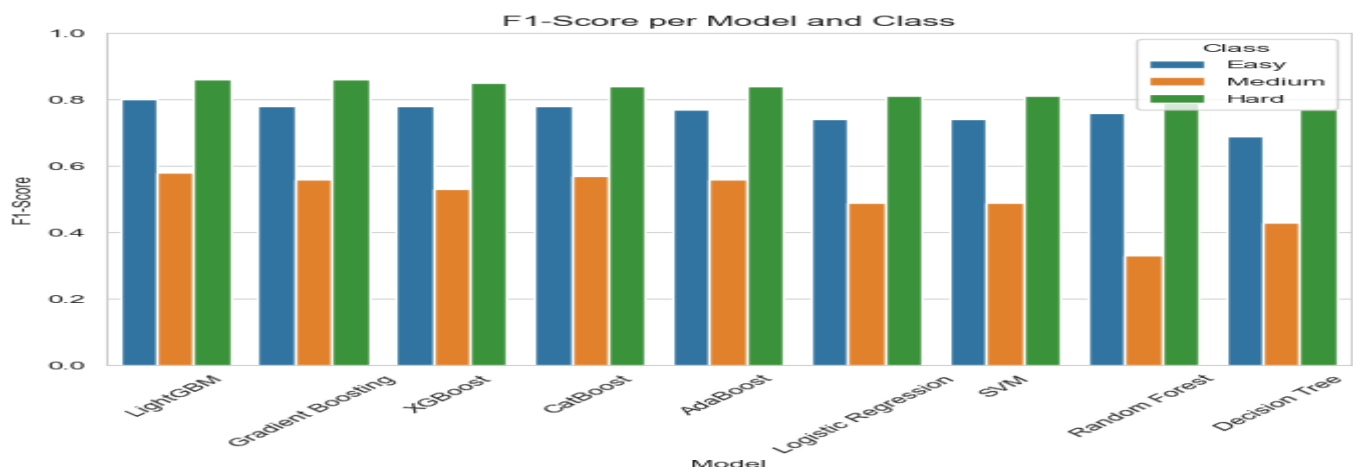
- Above shows the plots of F1 score for all three classes easy medium and hard for all models .We can see medium class has low f1 score for all models which makes it crucial to select the model which gives better f1 score for medium class along with overall metrics.

  LightGBM achieves the highest overall accuracy of 0.77 consistently for Easy, Medium, and Hard classes, outperforming all other evaluated models.

  In addition to accuracy, it maintains a strong and balanced F1-score across classes (Easy: 0.80, Medium: 0.58, Hard: 0.86), indicating that the model handles class imbalance effectively and performs well in both precision and recall.

  This balance is crucial for difficulty prediction, where misclassifying Medium and Hard problems can significantly impact usefulness.

- Compared to other ensemble methods such as Gradient Boosting, XGBoost, and CatBoost, which show slightly lower accuracy (0.75–0.76) and more

variation in F1-scores—especially for the Medium class—LightGBM demonstrates more stable and consistent performance.

Simpler models like Logistic Regression, SVM, Decision Tree, and Random Forest perform notably worse, particularly in recall and F1-score for Medium and Hard classes.

- Overall, due to its highest accuracy, robust F1-scores, and consistent performance across all difficulty levels, LightGBM is selected as the final classification model for this project.
- Following is the confusion matrix for the best model LightGBM
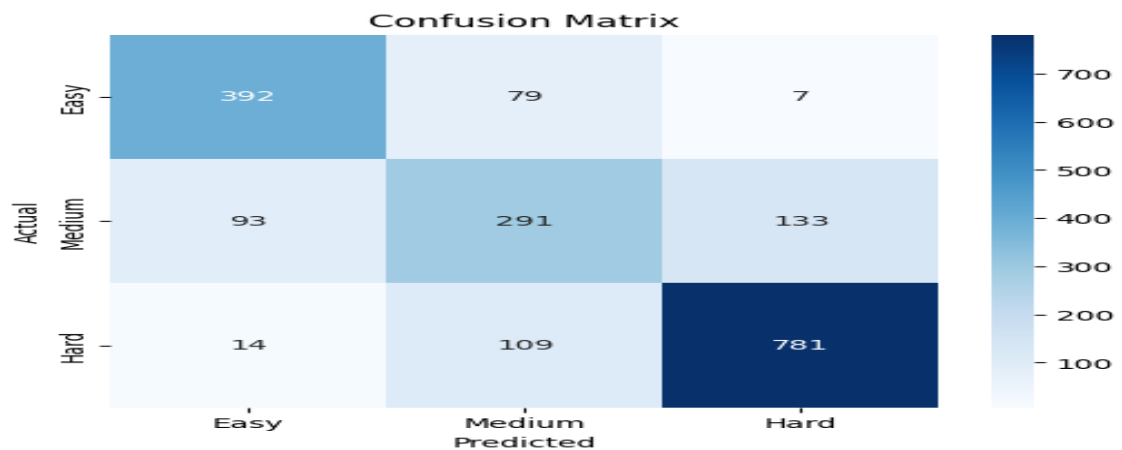

Accuracy: 0.7709 and Macro F1: 0.7476

2) Regression Results are as follows:

- Since the target difficulty score (y) lies in the range 0–10, direct regression can sometimes produce predictions outside this valid range.

  To address this, the target values are first scaled to the range [0, 1] using Min–Max scaling before training. All regression models are trained on this scaled target, which stabilizes learning and ensures consistent behavior across different models.

- During inference, predictions are generated in the scaled space, then clipped to the [0, 1] range to prevent invalid values. These clipped predictions are subsequently inverse-transformed back to the original 0–10 scale, ensuring that final outputs remain interpretable and valid.
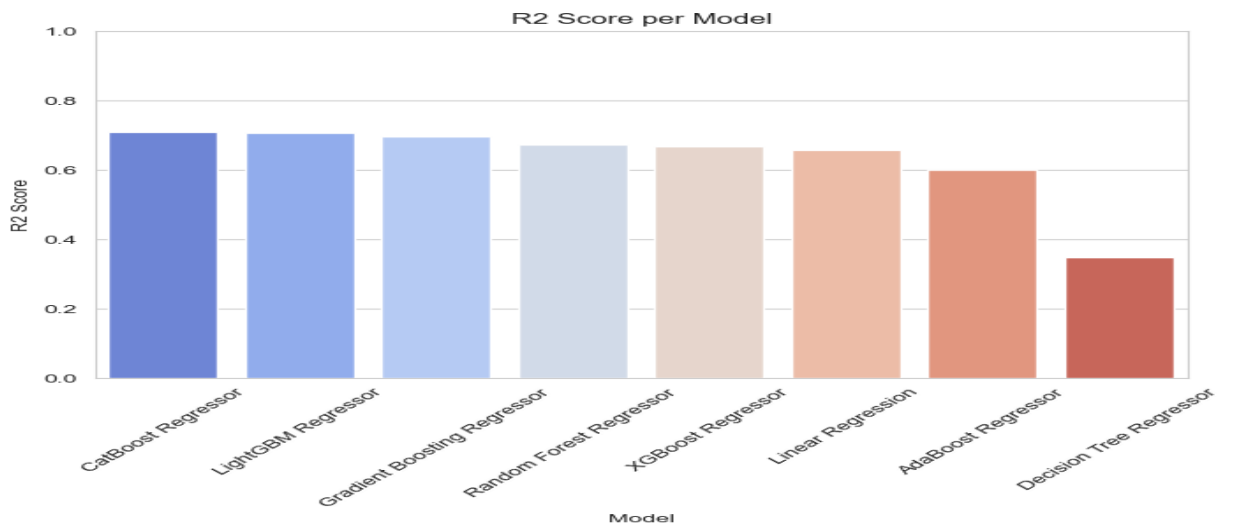
  Performance metrics such as MAE, RMSE, and R^2 score are computed on this original scale, allowing fair comparison between models while maintaining numerical stability during training.
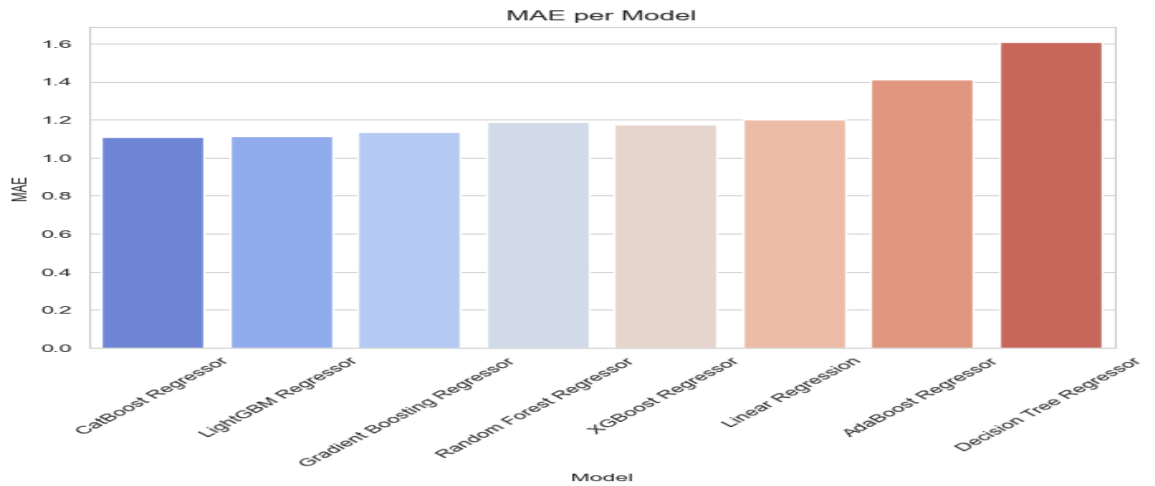
MAE across all models(lower is better)
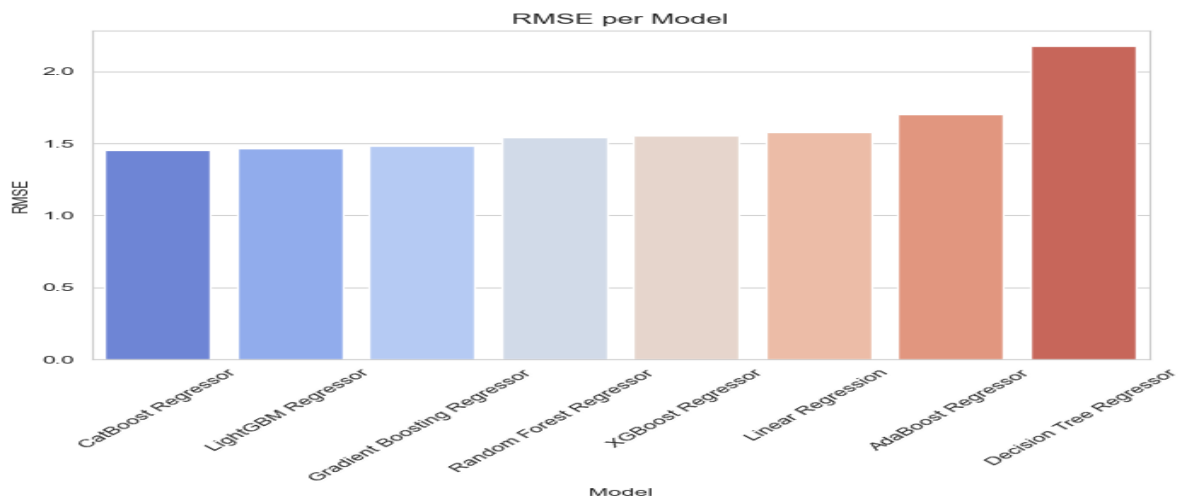
RMSE across all models(lower is better)

R2 scores across models (higher is better)

- Linear Regression performs relatively weaker compared to non-linear models, with a higher MAE (1.2001) and RMSE (1.5746), and a lower R^2 score (0.6582).

  This indicates that linear models struggle to capture the complex, non-linear relationships present in problem difficulty prediction, especially when combining textual, statistical, and knowledge-graph–based features.

- In contrast, ensemble and tree-based models such as Decision Tree, Random Forest, Gradient Boosting, AdaBoost, and XGBoost show noticeable improvements. Random Forest and Gradient Boosting reduce both MAE and RMSE by leveraging multiple trees, while boosting-based methods further improve performance by focusing on harder-to-predict samples.

  However, simpler tree models like Decision Tree Regressor still perform poorly, with the highest errors and a low R^2 score (0.3492), highlighting the importance of ensemble learning.

- Among all models, LightGBM Regressor and CatBoost Regressor achieve the best overall performance.

  CatBoost Regressor performs the best with the lowest MAE (1.1086) and RMSE (1.4523), along with the highest R^2 score (0.7092), indicating strong predictive accuracy and robustness.

  LightGBM Regressor follows closely with comparable error values (MAE 1.1124, RMSE 1.4615) and a high R^2 score (0.7055).

- Their superior performance demonstrates that gradient boosting–based models are most effective at learning complex patterns in the data, making CatBoost and LightGBM the preferred choices for the difficulty score regression task.

Metric	CatBoost (Best)	LightGBM
MAE	1.1086	1.1124
RMSE	1.4523	1.4615
R^2 Score	0.7092	0.7055

Web Interface and Sample Predictions:

The complete application is hosted on Hugging Face Spaces and is built using Streamlit, allowing users to interactively predict problem difficulty. You can access the live application [here](#)

To run the application locally, follow these steps:

1. Clone the repository : git clone
   https://huggingface.co/spaces/1404Samyak/AutoJudge

2. Navigate to the project directory : cd AutoJudge

3. Install the required dependencies : pip install -r requirements.txt

4. Run the Streamlit application : python streamlit_app.py

The application interface allows users to input a problem statement, input/output descriptions, and sample test cases, and then outputs both the predicted difficulty class (Easy, Medium, Hard) and a numerical difficulty score.

For example when this problem of codeforces as given as input to the model along with problem,input and output description ,algorithm tags and test case examples in input output format ,we can see the predicted class is "Hard" and predicted score is "7.97/10" as also confirmed from actual problem.

This end-to-end system integrates dense features, Word2Vec embeddings, knowledge graph embeddings, and classical ML models to provide accurate and interpretable predictions.