Project Overview & Strategic Plan

AutoJudge: AI-Powered Complexity Assessment for Programming Tasks

1. Executive Summary

In the realm of competitive programming and educational technology, classifying the difficulty of coding problems is traditionally a manual, subjective process performed by human setters. This project, **AutoJudge**, aims to automate this process using Natural Language Processing (NLP) and Machine Learning. By analyzing the textual description of a problem (including input/output specifications), the system predicts both a categorical difficulty label (**Easy, Medium, Hard**) and a granular numerical complexity score.

2. Problem Statement

Manual tagging of programming problems suffers from:

- **Subjectivity:** One person's "Easy" might be another's "Medium."
- **Scalability Issues:** Coding platforms with thousands of user-generated problems cannot manually verify every tag efficiently.
- **Inconsistency:** Difficulty scores often lack a standardized metric.

AutoJudge resolves this by training models on a dataset of 4,000+ problems to learn the linguistic patterns associated with different difficulty levels.

3. Project Objectives

- **Data Processing:** To clean and normalize raw text data containing mathematical symbols (LaTeX) and technical jargon.
- **Classification:** To build a model capable of categorizing problems into distinct difficulty tiers.
- **Regression:** To predict a precise difficulty score (floating-point number) for fine-grained ranking.
- **Deployment:** To create a user-friendly Web Interface where administrators can input raw text and receive instant predictions.

4. Strategic Plan & Methodology

The project follows a strict Machine Learning pipeline:

1. **Data Ingestion:** Loading the problems_data.jsonl dataset containing titles, descriptions, and metadata.
2. **Exploratory Data Analysis (EDA):** Visualizing class imbalances and text length distributions to understand dataset characteristics.
3. **Preprocessing & Feature Engineering:**
    - Cleaning text by removing LaTeX formatting (e.g., $N \le 100$) and special characters.
    - Combining Title, Description, Input, and Output into a single context string.
    - Converting text to numerical vectors using **TF-IDF (Term Frequency-Inverse Document Frequency)**, limiting to the top 5,000 features.
4. **Model Selection & Tuning (GridSearch):**
    - We do not rely on a single algorithm. We utilize GridSearchCV to train multiple models (Logistic Regression, Random Forest, SVM, Gradient Boosting) simultaneously.
    - The system compares performance metrics (Accuracy for classification, MAE for regression) and automatically saves the best-performing models.
5. **Deployment:** Using **Streamlit** to build a frontend that loads the saved models for real-time inference.

Exploratory Data Analysis (EDA) Summary

Understanding the Data Landscape

Before training, extensive Exploratory Data Analysis (EDA) was conducted to understand the nature of the programming problems. The dataset consists of approximately 4,112 tasks.

1. Target Variable Distribution

We analyzed the problem_class column to check for class balance.

- **Observation:** The dataset contains three classes: **Easy, Medium, and Hard**.
- **Visualization:** A Count Plot (sns.countplot) was generated.
- **Insight:** If the classes are imbalanced (e.g., significantly more "Medium" problems than "Hard"), the model might become biased. To mitigate this, we employed class_weight='balanced' in our linear models (like Logistic Regression) to penalize the misclassification of minority classes more heavily.

2. Complexity Score Distribution

We analyzed the problem_score column, which is a continuous numerical value representing difficulty.

- **Visualization:** A Histogram with a Kernel Density Estimate (sns.histplot).
- **Insight:** This visualization helped us understand if the difficulty scores follow a Normal Distribution (Bell Curve) or if they are skewed. This informed our choice of regression metrics; we chose **Mean Absolute Error (MAE)** over MSE because MAE is more robust to outliers in difficulty scoring.

3. Text Length vs. Difficulty

A hypothesis was formed: *"Are harder problems longer and more wordy?"*

- **Analysis:** We calculated the character length of the problem descriptions and plotted them against the difficulty class using a Box Plot (sns.boxplot).
- **Insight:**
    - **Easy Problems:** Often have shorter, direct descriptions.
    - **Hard Problems:** Tend to have longer descriptions, often involving complex storylines or detailed edge-case specifications.
    - **Conclusion:** Text length is a valuable feature. By using TF-IDF, the model inherently captures the "density" of information, which correlates with length.

4. Text Quality and Noise

Upon inspecting the raw text, we discovered significant "noise":

- **LaTeX Math:** Expressions like $1 \le N \le 10^5$ appear frequently. While mathematically important, they confuse standard NLP tokenizers.
- **Formatting:** Newline characters (\n) and tabs (\t) were prevalent.

Action Taken: A robust clean_text() function was implemented to strip LaTeX delimiters ($), remove non-alphanumeric characters, and lowercase all text prior to vectorization.

Results, Evaluation & Conclusion

Model Performance & Selection

The project utilized a "Tournament" approach to model selection. We trained multiple algorithms and selected the winners based on validation metrics.

1. Classification Results

We tested Logistic Regression, Random Forest, and Support Vector Machines (SVM).

- **Metric Used:** Accuracy (Percentage of correctly classified problems).
- **The Winner: Random Forest Classifier**.
- **Performance:** ~53.9% Accuracy on the test set.
- **Analysis:** While 54% might seem moderate, it is significantly better than random guessing (33%) in a 3-class problem. Text classification on technical content is notoriously difficult because "Easy" and "Medium" problems often share very similar vocabulary (e.g., arrays, strings). Random Forest performed best because it handles non-linear relationships in high-dimensional text data better than Logistic Regression.

2. Regression Results

We tested Ridge Regression, Random Forest Regressor, and Gradient Boosting.

- **Metric Used:** Mean Absolute Error (MAE).
- **The Winner: Random Forest Regressor**.
- **Performance:** MAE of **1.7228**.
- **Analysis:** An MAE of 1.72 means that, on average, the model's prediction is within 1.7 points of the actual difficulty score. Given the subjectivity of the original scores, this indicates the model has successfully learned to approximate human judgment.

3. Key Challenges Encountered

- **LaTeX Parsing:** Removing math symbols improves NLP readability but removes context (e.g., knowing constraints are $N < 10$ vs $N < 10^9$ is crucial for difficulty). Future versions could parse numbers before removing symbols.
- **Class Overlap:** The boundary between "Medium" and "Hard" is fuzzy. The TF-IDF approach relies on keywords (e.g., "Dynamic Programming" usually implies Hard), but not all hard problems use specific keywords.

4. Conclusion & Future Scope

AutoJudge successfully demonstrates the feasibility of using NLP to assess programming problem complexity. By automating the tagging process, we save time for contest setters and ensure consistency.

Future Improvements:

1. **Deep Learning:** Implementing BERT or Longformer models could capture the *context* of the problem description better than the current TF-IDF (keyword counting) approach.
2. **Code Analysis:** If the dataset included sample solutions, we could analyze the *Code Cyclomatic Complexity* alongside the text to improve prediction accuracy.
3. **Feedback Loop:** Allowing users to correct the prediction in the App to retrain the model continuously.

The Application Build

AutoJudge Web Interface

To make the machine learning models accessible to end-users, a web application was built using the **Streamlit** framework. This section details the architecture and functionality of the app.

1. Technical Architecture

The application (app.py) is designed as a stateless micro-service that interacts with the trained model artifacts.

- **Model Loading (@st.cache_resource):** Loading machine learning models (deserializing .pkl files) is computationally expensive. We utilized Streamlit's caching mechanism. The models (best_classifier.pkl and best_regressor.pkl) are loaded into memory *once* when the app starts, rather than reloading them every time a user clicks "Predict." This ensures the app is responsive.

2. User Inputs

The UI mimics the structure of the training data. We provide three distinct text areas:

1. **Problem Description:** The main body of the question.
2. **Input Description:** Constraints and format (e.g., "The first line contains an integer N").
3. **Output Description:** Expected result format.

Why split them? During training, we concatenated these fields. The app collects them separately for user convenience but concatenates them in the background to match the exact format the model was trained on.

3. The Prediction Pipeline

When the user clicks the **"Predict Complexity"** button, the following logic triggers:

1. **Concatenation:** raw_text = description + input + output

2. **Preprocessing:** The clean_text() function is applied to raw_text. This removes special characters and normalizes the string. **Crucially**, this function is identical to the one used in the training phase to ensure data consistency.
3. **Inference:**
    - clf_model.predict() determines if it is Easy, Medium, or Hard.
    - reg_model.predict() calculates the specific score (e.g., 9.75).

4. User Interface (UI) Design

The design focuses on clarity and immediate feedback:

- **Color Coding:** The application uses conditional logic to style the output.
    - "Easy" renders in **Green**.
    - "Medium" renders in **Orange**.
    - "Hard" renders in **Red**.
- **Metric Display:** The numerical score is displayed using st.metric for a dashboard-like appearance.

This interface allows non-technical users (like contest organizers) to evaluate problem difficulty without interacting with Python code.