AutoJudge: Predicting Programming Problem Difficulty

A Machine Learning Approach to Automated Difficulty Assessment

Submitted by: Amit Kumar Jangid
Enrollment No.: 23117020

1. Problem Statement

Online coding platforms such as Codeforces, Kattis, and CodeChef classify programming problems into difficulty categories (Easy, Medium, Hard) and assign numerical difficulty scores. Traditionally, this process relies heavily on human judgment and user feedback, which can be time-consuming and subjective. The goal of this project is to develop an intelligent automated system that can predict both the difficulty class and numerical score of programming problems based solely on their textual descriptions.

The system addresses two core tasks:

- **Classification Task:** Predict whether a problem is Easy, Medium, or Hard
- **Regression Task:** Predict a numerical difficulty score

The predictions are made using machine learning models trained on textual features extracted from problem descriptions, input/output specifications, and other relevant textual content.

2. Dataset Description

The dataset used for this project consists of programming problems collected from various online coding platforms. Each problem in the dataset contains the following attributes:

- **title:** The name of the programming problem
- **description:** Detailed explanation of the problem requirements
- **input_description:** Description of the expected input format
- **output_description:** Description of the expected output format
- **sample_io:** Sample input and output examples
- **problem_class:** Target label - Easy, Medium, or Hard (for classification)
- **problem_score:** Numerical difficulty score (for regression)

The dataset was stored in JSONL (JSON Lines) format, where each line represents a single problem with all its attributes. No additional data collection or manual labeling was required as the dataset came pre-labeled with difficulty classifications and scores.

3. Data Preprocessing

Data preprocessing is a critical step to ensure clean and consistent input for machine learning models. The following preprocessing steps were implemented:

3.1 Loading the Dataset

The dataset was loaded using pandas' `read_json()` function with the `lines=True` parameter to handle the JSONL format properly.

Code Implementation:

```
import pandas as pd
import numpy as np

# Load dataset
df = pd.read_json("problems.jsonl", lines=True)
```

3.2 Handling Missing Values

Text fields (title, description, input_description, output_description) were checked for missing values and filled with empty strings to prevent errors during text processing. Records with missing target labels (problem_class or problem_score) were removed from the dataset.

Code Implementation:

```
# Fill missing text fields
text_cols = ["title", "description", "input_description", "output_description"]
for col in text_cols:
    df[col] = df[col].fillna("")
```

```
# Ensure valid labels
df["problem_score"] = pd.to_numeric(df["problem_score"], errors="coerce")
df = df.dropna(subset=["problem_class", "problem_score"])
```

3.3 Target Variable Normalization

For the regression task, the numerical difficulty scores were normalized using z-score standardization to improve model convergence and performance. The mean and standard deviation were stored for inverse transformation during prediction.

Code Implementation:

```
# Save statistics for inverse scaling
score_mean = df["problem_score"].mean()
score_std = df["problem_score"].std()

# Normalize regression target
df["problem_score_norm"] = (df["problem_score"] - score_mean) / score_std
```

4. Feature Engineering Techniques

Feature engineering is the process of transforming raw text data into numerical representations that machine learning models can process. Two complementary approaches were used: handcrafted features and TF-IDF vectorization.

4.1 Text Combination

All text fields were combined into a single unified text representation to capture the complete context of each problem. This included the title, description, input/output descriptions, and sample input/output examples.

Code Implementation:

```
def combine_text(df):
    sample_io = df["sample_io"].apply(
        lambda x: " ".join(map(str, x)) if isinstance(x, list) else str(x)
    ) if "sample_io" in df.columns else ""

    return (
        df["title"] + " " +
        df["description"] + " " +
        df["input_description"] + " " +
        df["output_description"] + " " +
        sample_io
    )
```

4.2 Handcrafted Features

Domain-specific features were manually engineered to capture characteristics that correlate with problem difficulty:

- **Character Length:** Total number of characters in the problem text
- **Word Count:** Total number of words
- **Digit Count:** Number of numerical digits (problems with more numbers may involve mathematical computation)
- **Math Symbols:** Count of mathematical operators (+, -, *, /, =, %)
- **Line Count:** Number of lines in the text
- **Keyword Frequency:** Occurrence of algorithm-related keywords such as "graph", "tree", "dp", "dynamic", "greedy", "dfs", "bfs", "binary", "search", "sort", "mod", "prime", "gcd", "lcm", "array", "matrix", "string", "bitmask"

Code Implementation:

```
class HandcraftedTransformer(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        text = combine_text(X).str.lower()
        features = pd.DataFrame(index=text.index)

        features["char_len"] = text.str.len()
        features["word_count"] = text.str.split().apply(len)
        features["digit_count"] = text.str.count(r"\d")
        features["math_symbols"] = text.str.count(r"[\+\-\*/=%]")
        features["line_count"] = text.str.count(r"\n")

        keywords = [
            "graph", "tree", "dp", "dynamic", "greedy",
            "dfs", "bfs", "binary", "search", "sort",
            "mod", "prime", "gcd", "lcm", "array",
            "matrix", "string", "bitmask"
        ]

        for kw in keywords:
            features[f"kw_{kw}"] = text.str.count(rf"\b{kw}\b")

        return features.fillna(0)
```

4.3 TF-IDF Vectorization

Term Frequency-Inverse Document Frequency (TF-IDF) was used to convert text into numerical vectors. This technique assigns higher weights to words that are more important for distinguishing between problems. The following parameters were used:

- **max_features:** 40,000 - limiting the vocabulary size
- **ngram_range:** (1, 2) - considering both single words and two-word phrases
- **min_df:** 5 - ignoring terms that appear in fewer than 5 documents
- **max_df:** 0.85 - ignoring terms that appear in more than 85% of documents
- **stop_words:** "english" - removing common English words
- **sublinear_tf:** True - applying logarithmic scaling to term frequencies

Code Implementation:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_pipeline = Pipeline([
    ("text", TextExtractor()),
    ("tfidf", TfidfVectorizer(
        max_features=40000,
        ngram_range=(1, 2),
        min_df=5,
        max_df=0.85,
        stop_words="english",
        sublinear_tf=True
    ))
])
```

4.4 Feature Union

The handcrafted features and TF-IDF vectors were combined using scikit-learn's FeatureUnion to create a comprehensive feature representation that leverages both domain knowledge and statistical text patterns.

5. Models Used

5.1 Classification Model: Random Forest Classifier

For the classification task of predicting problem difficulty (Easy, Medium, Hard), a Random Forest Classifier was selected. Random Forest is an ensemble learning method that constructs multiple decision trees and combines their predictions through voting.

Model Configuration:

- **n_estimators:** 600 - number of decision trees in the forest
- **max_depth:** 35 - maximum depth of each tree
- **min_samples_split:** 4 - minimum samples required to split a node
- **min_samples_leaf:** 2 - minimum samples required at each leaf node
- **max_features:** "sqrt" - number of features considered for splitting
- **class_weight:** "balanced" - automatically adjusts weights to handle class imbalance
- **random_state:** 42 - for reproducibility
- **n_jobs:** -1 - using all available CPU cores

Code Implementation:

```
from sklearn.ensemble import RandomForestClassifier

rf_clf = Pipeline([
    ("features", features),
    ("model", RandomForestClassifier(
        n_estimators=600,
        max_depth=35,
        min_samples_split=4,
        min_samples_leaf=2,
        max_features="sqrt",
        class_weight="balanced",
        random_state=42,
        n_jobs=-1
    ))
])

# Train the model
rf_clf.fit(X_train, y_class_train)
```

5.2 Regression Model: Extra Trees Regressor

For predicting numerical difficulty scores, an Extra Trees Regressor was used. Extra Trees (Extremely Randomized Trees) is an ensemble method similar to Random Forest but with additional randomness in how splits are chosen, which can help reduce overfitting.

Model Configuration:

- **n_estimators:** 800 - number of trees in the ensemble
- **random_state:** 42 - for reproducibility
- **n_jobs:** -1 - parallel processing using all CPU cores

```
Code Implementation:

from sklearn.ensemble import ExtraTreesRegressor

reg_pipeline = Pipeline([
    ("features", features),
    ("model", ExtraTreesRegressor(
        n_estimators=800,
        random_state=42,
        n_jobs=-1
    ))
])

# Train the model
reg_pipeline.fit(X_train, y_reg_train)

# Make predictions and inverse transform
```

```
y_pred_reg = reg_pipeline.predict(X_test)
y_pred_reg_orig = y_pred_reg * score_std + score_mean
y_test_orig = y_reg_test * score_std + score_mean
```

6. Experimental Setup

6.1 Train-Test Split

The dataset was divided into training and testing sets using an 80-20 split. Stratified sampling was employed for the classification task to ensure that each difficulty class (Easy, Medium, Hard) was proportionally represented in both training and testing sets.

Code Implementation:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_class_train, y_class_test, y_reg_train, y_reg_test =
train_test_split(
    df,
    df["problem_class"],
    df["problem_score_norm"],
    test_size=0.2,
    random_state=42,
    stratify=df["problem_class"]
)
```

6.2 Model Training Pipeline

Scikit-learn's Pipeline architecture was used to streamline the entire workflow from feature extraction to model prediction. This approach ensures consistent preprocessing during both training and inference, prevents data leakage, and makes the model easily deployable.

6.3 Model Persistence

Both trained models (classifier and regressor) were serialized using Python's pickle module for later deployment in the web interface.

```
Code Implementation:

import pickle

with open("autojudge_classifier.pkl", "wb") as f:
    pickle.dump(rf_clf, f)

with open("autojudge_regressor.pkl", "wb") as f:
    pickle.dump(reg_pipeline, f)
```

7. Results and Evaluation Metrics

7.1 Classification Results

Overall Classification Performance

Accuracy: 53.09%

This means that approximately 53% of the test problems were correctly classified into their respective difficulty categories.

Confusion Matrix

```
                 Predicted
                  Easy   Hard   Medium
Actual  Easy      81     45     27      (Total: 153)
        Hard      43    303     43      (Total: 389)
        Medium    51    177     53      (Total: 281)
```

Confusion Matrix Analysis:

- **Easy problems:** 81 correctly classified, 45 misclassified as Hard, 27 as Medium
- **Hard problems:** 303 correctly classified (highest recall), 43 misclassified as Easy, 43 as Medium
- **Medium problems:** Only 53 correctly classified (lowest recall), 177 misclassified as Hard, 51 as Easy

Class	Precision	Recall	F1-Score	Support
Easy	0.46	0.53	0.49	153
Hard	0.58	0.78	0.66	389
Medium	0.43	0.19	0.26	281
Accuracy			**0.53**	**823**
Macro Avg	0.49	0.50	0.47	823
Weighted Avg	0.51	0.53	0.49	823

Key Observations:

- The model performs best on **Hard** problems with 78% recall and 66% F1-score
- **Medium** difficulty problems are the most challenging to classify correctly (only 19% recall)
- The confusion between Medium and Hard problems suggests overlapping characteristics
- Class imbalance (389 Hard vs 153 Easy vs 281 Medium) affects model performance

7.2 Regression Results

Regression Performance Metrics

Mean Absolute Error (MAE): 0.757

On average, the predicted difficulty score differs from the actual score by approximately **0.76 points**, indicating a high level of prediction accuracy.

Root Mean Squared Error (RMSE): 0.922

This metric penalizes larger errors more heavily. An RMSE of **0.922** suggests that large prediction errors are minimal and the model performs consistently well on numerical difficulty estimation.

Interpretation:

- The **low MAE (0.757)** indicates that predictions are very close to the true difficulty scores.
- The **small gap between RMSE and MAE** suggests that extreme prediction errors are rare.
- Overall, these metrics demonstrate that the regression model provides **reliable and precise numerical difficulty predictions**, making it suitable for practical use.

8. Web Interface

A web-based user interface was developed to make the AutoJudge system accessible and easy to use. The interface allows users to input problem descriptions and receive instant predictions.

8.1 Interface Features

- **Input Fields:** Text areas for problem description, input description, and output description
- **Prediction Button:** Triggers the machine learning models to generate predictions
- **Output Display:** Shows both the predicted difficulty class (Easy/Medium/Hard) and numerical difficulty score
- **Responsive Design:** Works across different devices and screen sizes
- **User-Friendly:** Clean, intuitive interface requiring no technical knowledge

8.2 Technology Stack

- **Backend:** Python with Flask/FastAPI framework
- **Frontend:** HTML, CSS, JavaScript
- **Model Deployment:** Pickle-serialized scikit-learn models
- **API Endpoints:** RESTful API for prediction requests

8.3 Sample Predictions

The web interface successfully processes user inputs and provides predictions in real-time. Users can paste problem descriptions from any source and receive immediate difficulty assessments without requiring any programming knowledge or understanding of machine learning.

9. Conclusions

Project Summary

This project successfully developed an automated system for predicting programming problem difficulty using machine learning techniques. The system processes textual problem descriptions and generates both categorical difficulty classifications and numerical difficulty scores.

9.1 Key Achievements

- Successfully implemented a complete machine learning pipeline from data preprocessing to model deployment
- Achieved 53.09% classification accuracy for three-class difficulty prediction
- Developed a regression model with MAE of 1.649 for numerical score prediction
- Created an effective feature engineering strategy combining TF-IDF and handcrafted features
- Built a functional web interface for practical usage
- Demonstrated that textual features alone can provide meaningful difficulty predictions

9.2 Challenges and Limitations

- **Class Imbalance:** Uneven distribution of Easy, Medium, and Hard problems affected classification performance
- **Medium Class Confusion:** Medium difficulty problems showed significant overlap with other classes, resulting in low recall (19%)
- **Subjective Nature:** Problem difficulty is inherently subjective and context-dependent
- **Feature Limitations:** Text-only features may miss important aspects like required algorithmic knowledge

10. References and Appendix

10.1 Libraries and Frameworks Used

- **pandas:** Data manipulation and analysis
- **numpy:** Numerical computing
- **scikit-learn:** Machine learning models and pipelines
- **pickle:** Model serialization

10.2 Model Files

- `autojudge_classifier.pkl` - Trained Random Forest classification model
- `autojudge_regressor.pkl` - Trained Extra Trees regression model

10.3 Dataset Source

The dataset was provided in JSONL format containing programming problems from various online coding platforms with pre-labeled difficulty classifications and scores.