# PROJECT REPORT

## 1. Dataset used

The dataset contains problem statements collected in CSV format. Each row has multiple text fields (title, description, input_description, output_description, sample_io), a categorical label (problem_class — Easy / Medium / Hard), and a numerical difficulty score (problem_score, range 0–10).

## 2. Data preprocessing

Missing values & basic cleaning:
All text fields were inspected for missing or null values. Missing entries were replaced with empty strings to avoid NaNs during concatenation and vectorization.

Text normalization steps:
1. Lowercasing: All text was converted to lowercase to collapse case variants.
2. Punctuation handling: Punctuation characters were replaced with spaces (rather than removed) to prevent merging tokens (e.g., 'a,b' → 'a b').
3. Whitespace normalization: Multiple consecutive spaces were collapsed into single spaces.

Combined input field:
For modeling simplicity, the final model input is a single combined_text column created by concatenating title, description, input_description, output_description and sample I/O examples. This promotes the learning of cross-field signals (e.g., if both description and sample I/O hint at complexity).

## 3. Feature Engineering

TF-IDF configuration:
TF-IDF was computed using scikit-learn's TfidfVectorizer class with the following important choices:
- ngram_range=(1,2) to capture both unigrams and common bigrams (e.g., 'binary search').
- stop_words='english' to remove high-frequency function words that add noise.
- max_features=5000 to control dimensionality and memory usage while preserving informative terms.
This produced a high-dimensional sparse matrix representing lexical importance.

Other numeric features related to the problem description:
- Text length (words): integer count of words in combined_text.
- Keyword frequency: sum of occurrences of curated keywords relevant to algorithmic patterns (e.g., dp, graph, dfs, bfs, greedy, recursion, matrix …). Implementation is a straightforward substring count; exact keywords can be extended.
- Mathematical symbol count: total occurrences of symbols such as +, -, *, /, =, <, >, ^, % to capture algebraic/expressive complexity in problems.

These numeric features were scaled implicitly by being appended as dense columns to the sparse TF-IDF matrix (scipy.sparse.hstack).

Feature combination and scaling considerations:
Because TF-IDF produces a sparse matrix and other numeric features are dense, the pipeline concatenates them using scipy.sparse.hstack. If desired for models sensitive to feature scale (e.g., linear models), numeric features can be standardized (e.g., StandardScaler wrapped via FunctionTransformer) prior to concatenation.

## 4. Modeling Details

Classification models evaluated:
1. Logistic Regression (multinomial): Baseline, fast to train, interpretable coefficients. Key params: solver='lbfgs', multi_class='multinomial', max_iter increased to ensure convergence.
2. Linear Support Vector Classifier with linear kernel (LinearSVC): Known to perform well in many text classification tasks; requires careful regularization (C parameter) tuning.
3. Random Forest Classifier: Used because handcrafted numeric features introduce non-linear interactions that tree ensembles exploit. Parameters adjusted in experiments: n_estimators (100–700), class_weight='balanced' to compensate for label imbalance.

Random_state=42 was used so that the evaluation can be fair for all the models.
Model selection was based on macro-F1 and overall accuracy on a held-out test set.

Regression models evaluated:
1. Linear Regression: Baseline to assess linear relationships between features and difficulty score.
2. Random Forest Regressor: Captures non-linear patterns and interactions; hyperparameters include n_estimators and max_depth.
3. Gradient Boosting Regressor: Used in exploratory runs for potential gains.
Regression was harder due to label noise; numbers reported in the Results section correspond to the final trained artifacts saved in the models/ directory.

## 5. Experimental Setup

Train/test split and stratification:
Data was split into training (80%) and test (20%). For classification, stratify=y was used to preserve class proportions. TF-IDF vectorizer was fit only on training data and reused (transform) on test data to avoid leakage.

Reproducibility and environment:
All experiments were run using scikit-learn, pandas, numpy, and joblib. Random seeds (random_state) were set on ensemble models for reproducible results. A requirements.txt file captures package versions used.

Evaluation protocol:
Classification metrics: accuracy, F1-score (macro), and confusion matrix. Regression metrics:

MAE, RMSE (root mean squared error), and $R^2$. Tables and plots in the report correspond to the evaluation scripts in notebooks/ and source code under src/.

## 6. Results of Evaluation

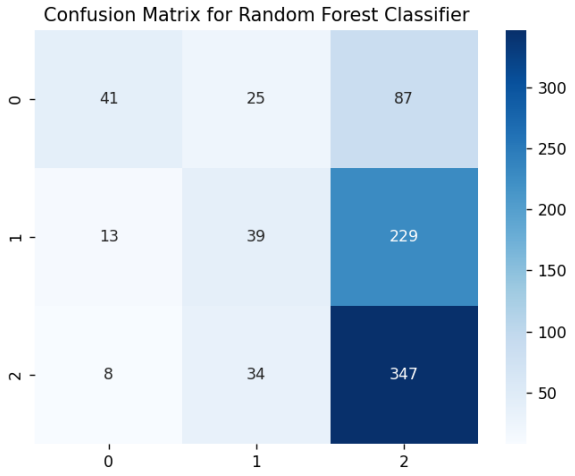
**• Table 1 — Classification results: model, accuracy, macro-F1**

| Model | Accuracy (%) | F1 Score (macro) |
|---|---|---|
| **Logistic Regression** | 47.873 | 0.465 |
| **Linear SVC** | 41.920 | 0.321 |
| **Random Forest Classifier** | 51.883 | 0.416 |

**• Table 2 — Regression results: model, MAE, RMSE**

| Model | MAE | RMSE |
|---|---|---|
| **Linear Regression** | 2.285 | 2.86 |
| **Gradient Boosting Regressor** | 1.696 | 2.014 |
| **Random Forest Regressor** | 1.688 | 2.006 |

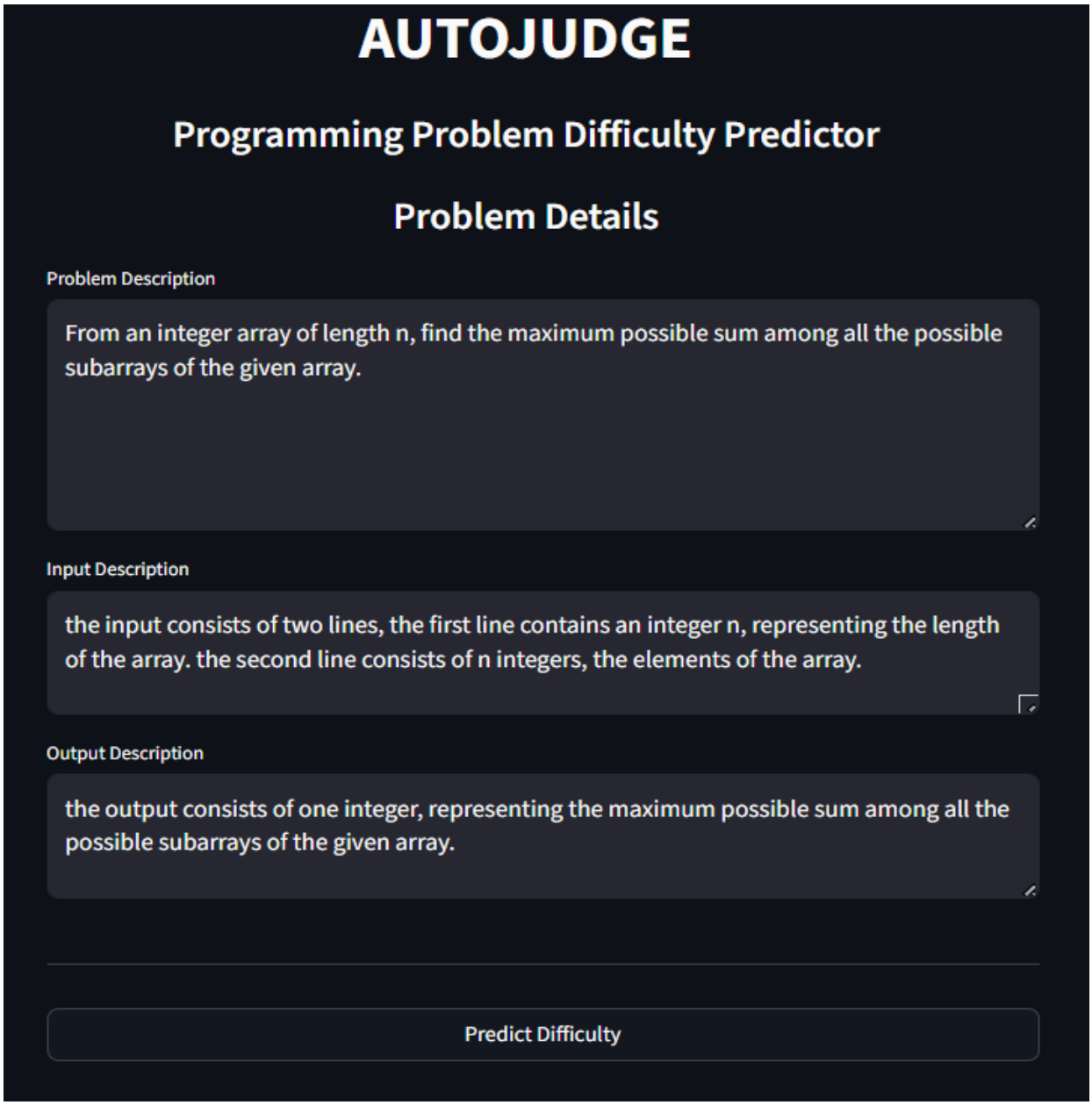
**• Figure 1 — Confusion matrix heatmap for final classifier**

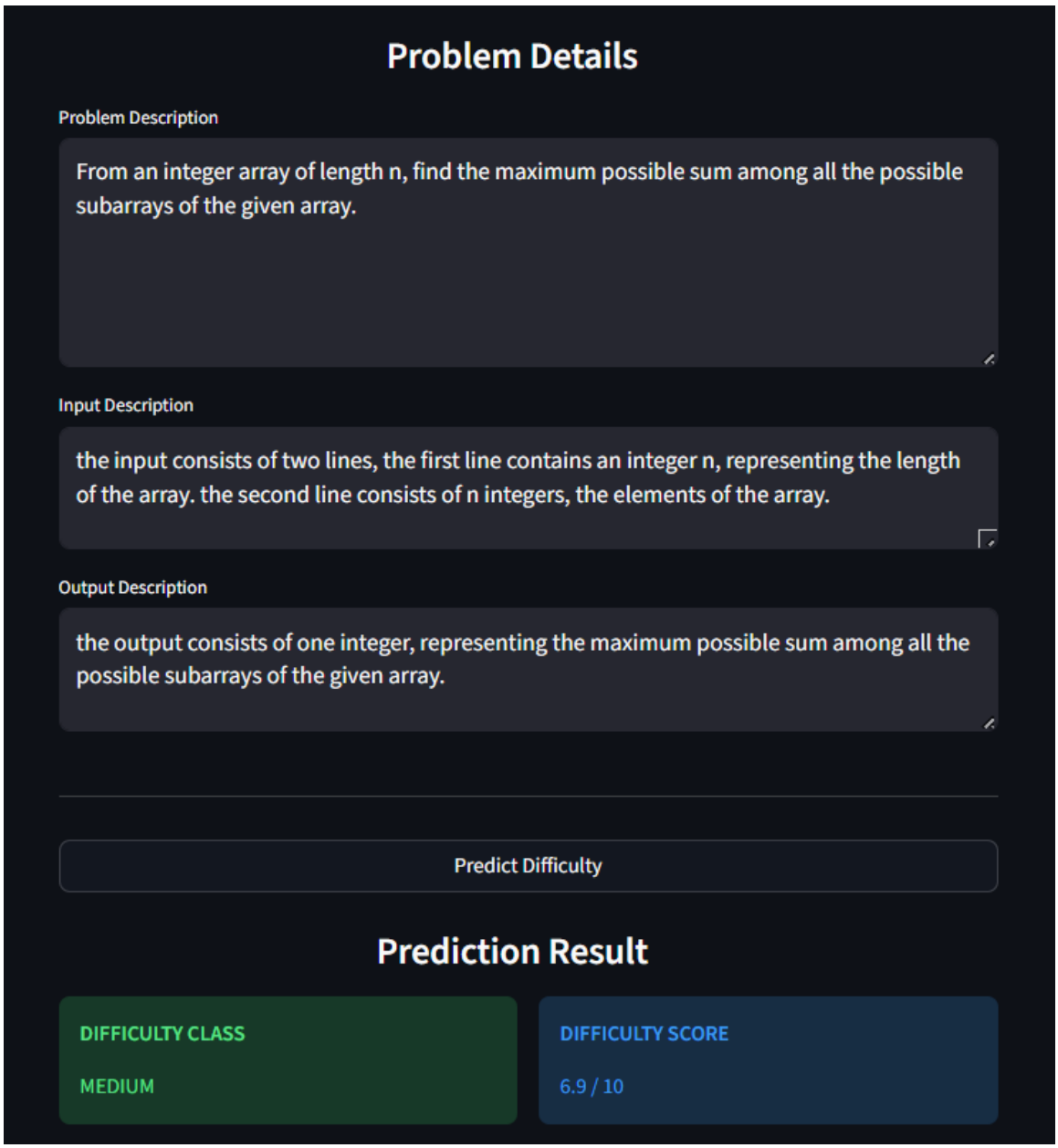



## 7. Web Interface: Implementation & Usage

Implementation details:
The web UI is built using Streamlit. Key points:
- The app loads saved artifacts from models/ (vectoriser.pkl, RFC.pkl, RFR.pkl) at startup.
- Inputs: problem description, input description, output description (each a text box).
- On button click, the app constructs combined_text, transforms it with the loaded TF-IDF vectoriser, computes handcrafted features, concatenates them, and passes the final matrix to the classifier/regressor.

Design choices & UX:
- Predictions are displayed in two columns: left column shows the predicted class box, right column shows the estimated score box. The score is derived from the predicted class.
- The app uses caching for model loading to reduce startup latency. Error handling notifies the user if model files are missing and instructs how to regenerate them.






**Pipeline:**

Data Preprocessing → Feature Extraction → Model Training → Evaluation and selection of best Model → Web Interface

## 8. Limitations

Limitations:
- Difficulty score labels have inherent subjectivity leading to noise; hence regression is expected to be less accurate than classification.
- TF-IDF based approaches are bag-of-words methods and cannot capture deeper contextual semantics as transformer models do.
- Large models (e.g., transformer fine-tuning) were not used due to compute and time constraints.

## 9. Conclusion

AutoJudge provides a reproducible pipeline from raw textual problem statements to difficulty predictions. It demonstrates solid baseline performance for classification, and provides a framework to iteratively improve regression and modeling choices. The entire codebase is organized for reproducibility and evaluation, and the Streamlit UI makes the system demonstrable.