AutoJudge: Programming Problem Difficulty Prediction

A Machine Learning Project Report

Submitted by:

Name: Aryan Soni

Year:2nd

Department: Chemical Engineering

Institute: IIT Roorkee

Submitted to:

ACM IIT Roorkee

Here is the demo video link(also in README.md) :

https://drive.google.com/file/d/1FXM557jdhEPgr9OhxmJlVn3WPCQrF5k5/view?usp=sharing

INTRODUCTION

With the increasing popularity of online coding platforms, programming problems are commonly grouped into difficulty levels such as Easy, Medium, and Hard. But mostly, these levels of difficulty are subjective.

This project, titled AutoJudge, aims to explore whether the textual description of a programming problem alone can provide meaningful insights into its difficulty. The goal is to build a machine learning system that predicts both a difficulty class and a numerical difficulty score using only problem statements, without relying on code, constraints, or platform-specific metadata.

The project combines Natural Language Processing algorithms with classical machine learning algorithms and offers a web interface for making predictions.

PROBLEM STATEMENT

As the goal of this project is to implement an automatic system to predict programming difficulty based on textual information, the system accomplishes two operations:

1. Categorization of problems as Easy, Medium, or Hard.

2. Regression-based prediction of a numerical difficulty score.

The main issue comes from the fact that the level of difficulty is a subjective classification and lacks any information like constraints or source code. In other words, it is a real-world machine learning issue.

DATASET DESCRIPTION

The dataset for the project is composed of programming exercises gathered from online competitive programming sites. Every entry within the dataset is related to a single programming question and has the following elements:

- Title of the problem

- Problem description

- Input description

- Output description

- Difficulty class label (Easy / Medium / Hard)

- Numerical difficulty score

The dataset is stored in JSON Lines (.jsonl) format, where each line represents one problem. Only textual fields are used for model training and evaluation.

DATA PREPROCESSING

Before we start training the machine learning models we need to make sure the data is good and consistent. This is a step, for the machine learning models. The dataset needs to be cleaned up so the machine learning models can work properly with the data.

I loaded the dataset from a file that had a lot of information in it. This file was special because it had a lot of lines and each line was about a programming problem. Sometimes the words in the lines were missing. I had to do something about that. I went through the title and the description and the input and output parts. If something was missing I just put in an empty space instead.

To make it easier to get features, from the text we put all the text parts of a problem together in one column called `full_text`. The `full_text` includes the problem title, the description, the input format and the output format. This way we can see the problem at once. The `full_text` is the thing we use to put into the NLP pipeline

No stopword removal or stemming was done at this stage. The reason is that the TF-IDF vectorization process takes care of these things on its own.

FEATURE ENGINEERING

Feature engineering is really important when we want to change text into numbers that a computer can understand. This is because computers can only work with numbers so we need to find a way to turn our text into something they can use. Feature engineering does this by taking the textual data and turning it into numerical representations. This way machine learning algorithms can work with the text.

In this project we used something called Term Frequency–Inverse Document Frequency or TF-IDF for short to turn words into numbers. Term Frequency–Inverse Document Frequency does this by looking at how words appear in one

document compared to all the others. If a word shows up a lot in one document but not much, in the others Term Frequency–Inverse Document Frequency thinks it is more important. We only looked at the 5000 most important words because using Term Frequency–Inverse Document Frequency on all of them would have taken too long and used up too much computer power but we still wanted Term Frequency–Inverse Document Frequency to work well.

We also used two numerical features along, with the TF-IDF features. These special features were made by hand. The two handcrafted numerical features were added to the TF-IDF features.

1. ***Text length***: This is the number of characters in the problem description. The text length is like a measure that shows how complicated a problem is. The more characters, in the problem description the more complicated the problem is. So the text length is a way to figure out the complexity of a problem, which is the problem description.

2. ***Keyword frequency***: The count of common competitive programming keywords such as “dp”, “graph”, “tree”, “dfs”, “bfs”, and “recursion” within the problem text.

These extra things help us get a sense of the structure that we might not see just by looking at how often words are used. We make the list of features by putting together the TF-IDF vectors and these numerical features.

We had two machine learning models that were trained separately. One model was for classification tasks. The other model was for regression tasks in this project.

We used ***Logistic Regression*** to figure out how hard something is. It can be Easy, Medium or Hard. We picked Logistic Regression because it is simple and easy to understand. It also works well with big sets of words and things like TF-IDF vectors. Logistic Regression is good at handling lots of information, from text.

For the regression task we used a ***Random Forest Regressor*** to guess the numerical difficulty score of the things we were looking at. Random Forests are really good at figuring out relationships that're not straightforward and they can handle noisy data, which makes them a good choice, for this kind of task. We chose Random Forests because they can do a job of understanding how different things are connected even when those connections are not simple.

We chose to use machine learning models instead of deep learning approaches. This was done on purpose. The reason is that classical machine learning models are easier to understand and they do not need much computer power. So we can see what the classical machine learning models are doing. They are not too complicated.

EXPERIMENTAL SETUP

The data was split up into two parts: one for training and one, for testing. We used most of the data 80 percent for training and the rest 20 percent, for testing. The training data was used to teach the model what to do. The testing data was used to see how well the model worked with data that it had not seen before.

For the feature extraction part we used something called ***TF-IDF vectorization*** on all the written descriptions combined. We did this to get the information from the text.

We only took the 5000 features to stop the system from getting too good at the training data and not good at new data. This also helped the computer to do the work faster.

We also added some information, like how long the text is and how often certain keywords appear. We added this information to the TF-IDF feature matrix.

For classification, Logistic Regression was trained with a maximum of 1000 iterations to ensure convergence. For regression, a Random Forest Regressor with 200 trees was used to capture non-linear relationships in the data.All experiments were conducted using Python and the scikit-learn library. Model training and evaluation were performed locally on a standard personal computer.

RESULTS AND EVALUATION

The performance of the proposed system was looked at for two things: the classification tasks and the regression tasks. We used the ways to measure how well the proposed system did for the classification tasks and the regression tasks.

For the classification task we looked at how accurate the model was. We used a confusion matrix to see how well it did. The Logistic Regression model we trained got it right ***53 percent*** of the time. If we were just guessing we would get it right 33 percent of the time because there are three classes to choose from, which are Easy, Medium and Hard. So the fact that our model did better than that means it is actually learning something from the text descriptions.

The model does a job of finding Easy problems. It is not as good at finding Medium and Hard problems. This makes sense because Medium problems are a mix of Easy and Hard problems. They have some things in common with both so the model has a time telling them apart just by looking at the text. The model is better at identifying easy problems because they are more straightforward. The model has the trouble, with Medium problems.

For the task of figuring out how things are related we used ***Mean Absolute Error*** and ***Root Mean Squared Error*** to see how good our predictions were. The Random Forest Regressor did a good job with a Mean Absolute Error of about ***1.69*** and a Root Mean Squared Error of about ***2.03***. This means that the difficulty scores we predicted are pretty close to the values, which is what we were hoping for.

Figures illustrating the confusion matrix and sample prediction outputs are included in the report.

WEB INTERFACE AND SAMPLE PREDICTIONS

To make the system interactive and easy to use, a simple web-based interface was developed using Streamlit. The web interface allows users to input the textual details of a programming problem and obtain difficulty predictions in real time.

The interface provides separate input fields for the problem title, problem description, input description, and output description. Upon clicking the “Predict Difficulty” button, the input text is processed and converted into numerical features using the same preprocessing and feature extraction pipeline used during model training.

The trained classification and regression models are then loaded to generate predictions. The predicted difficulty class (Easy, Medium, or Hard) along with the numerical difficulty score is displayed on the interface.

The application runs entirely on local execution and does not require any external hosting or database. Screenshots of the web interface and sample prediction outputs are included in this report to demonstrate the working of the system.

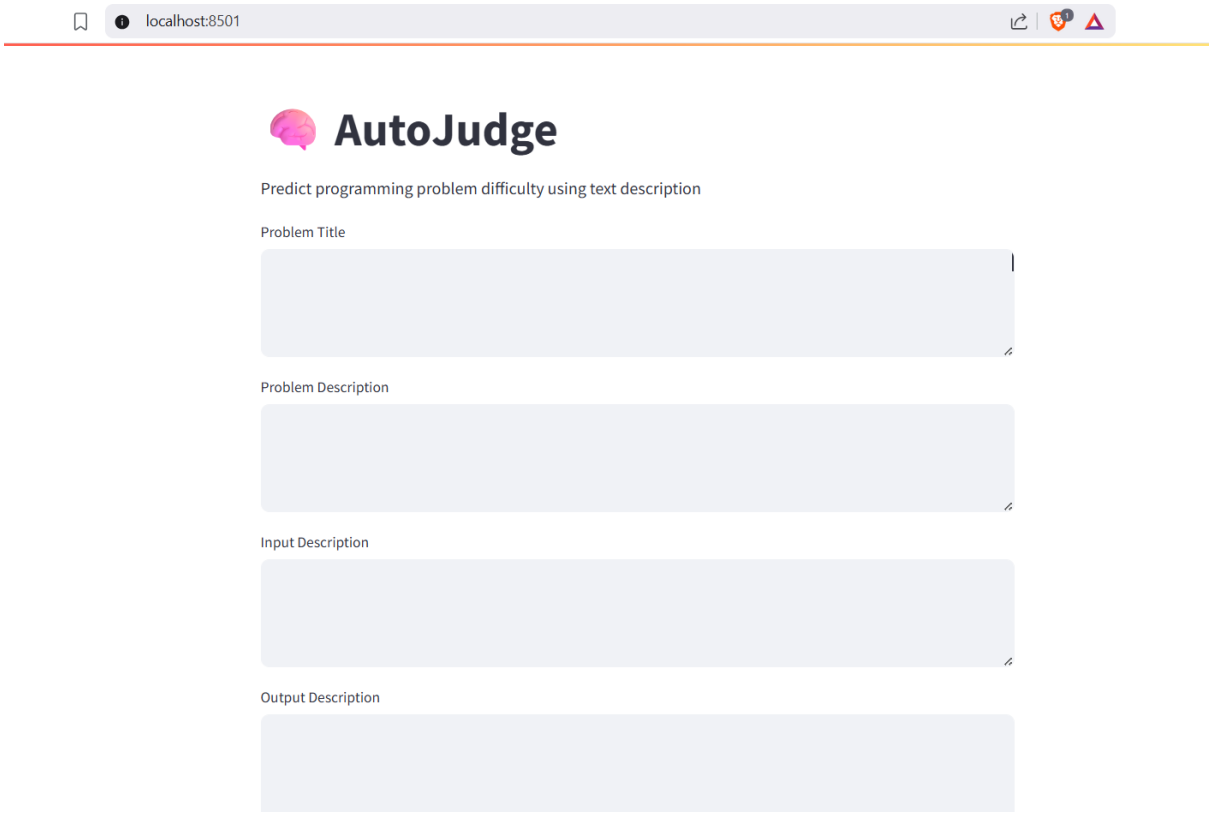

Figure 1: Home page of the AutoJudge web interface.

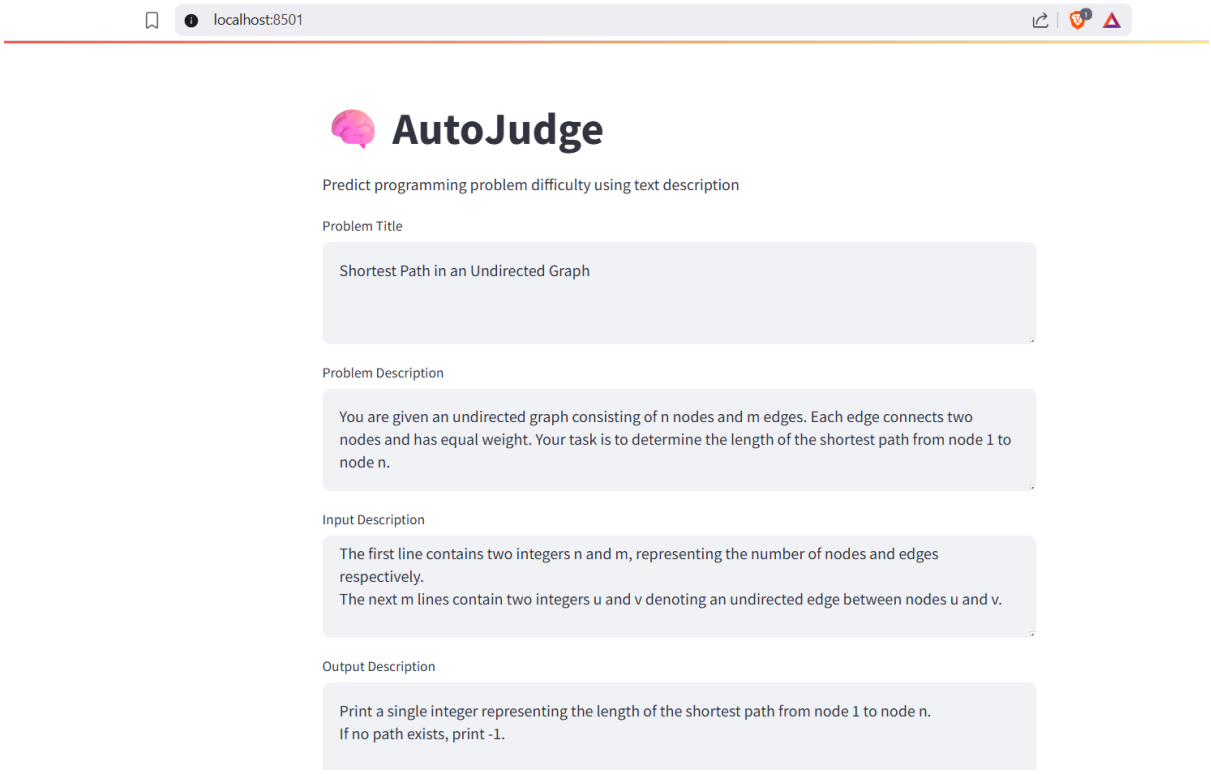

Figure 2: Web interface with sample problem details entered by the user.

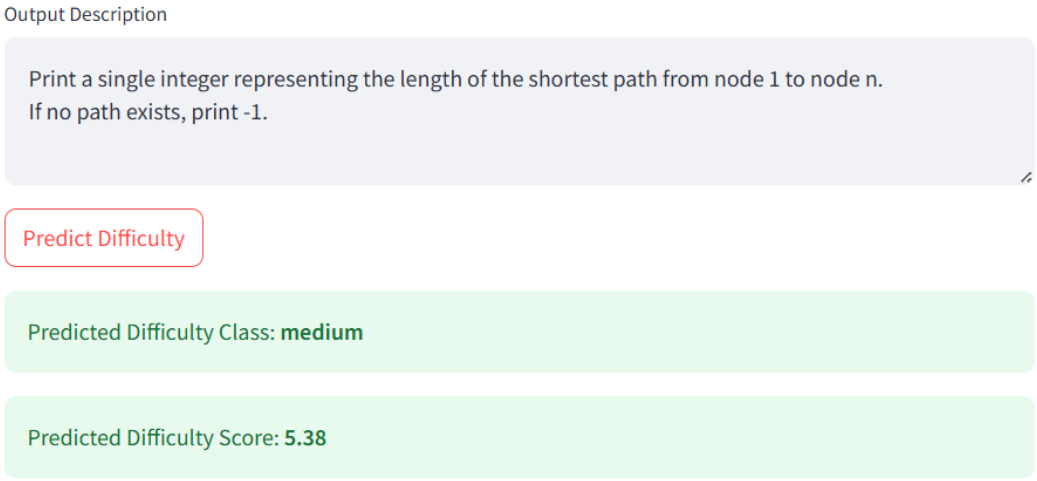

Figure 3: Predicted difficulty class and score displayed on the web interface.

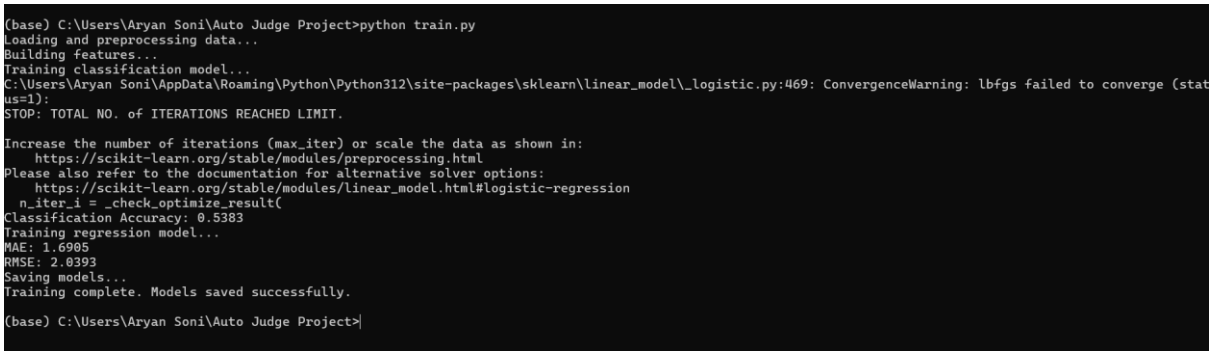

Figure 4: Terminal output showing model training and evaluation results including accuracy, MAE, and RMSE.

CONCLUSION

This project is about a system that uses machine learning to figure out how hard a programming problem is. It looks at the words used to describe the problem. Then makes a prediction. The system uses computer programs that understand human language and some basic machine learning models. This helps the system to predict two things: how hard a problem is in general and exactly how hard it is on a scale. The system is pretty good, at doing this.

The project on machine learning applications shows that we can get ideas about how hard a problem is just, by looking at what is written about it. This is true even when it is hard to say exactly how hard a problem is and when problems sound similar. Machine learning applications also need people to carefully choose what information to use and to test their ideas in a real world setting. The project really shows that machine learning applications need this planning to work well.

Future improvements may include the use of advanced transformer-based language models, incorporation of problem constraints, and platform-specific training to further enhance prediction accuracy.