BABEŞ BOLYAI UNIVERSITY, CLUJ-NAPOCA, ROMÂNIA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
COMPUTER SCIENCE

AUTOMATED COMPETITIVE PROGRAMMING PROBLEM LABELLING USING MACHINE LEARNING

– Diploma thesis –

Supervisor
Lecturer Professor Mircea Ioan-Gabriel

Author
Pop Bogdan-Petru

2022

UNIVERSITATATEA BABEŞ BOLYAI, CLUJ-NAPOCA, ROMÂNIA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

ETICHETAREA PROBLEMELOR DE PROGRAMARE COMPETITIVĂ FOLOSIND ÎNVĂȚARE AUTOMATĂ

– Lucrare de licență –

Conducător Științific
Lector Mircea Ioan-Gabriel

Absolvent
Pop Bogdan-Petru

2022

Abstract

Complex systems have been capable of producing solutions to different reasoning problems, but without creating an explainable solution or bringing valuable information in the cases in which the solution is not correct. The methods presented in this paper are trying to provide more interpretable, but incomplete solutions for complex problems from the competitive programming space. Concretely, this paper describes 3 different algorithms - based on language transformers, k-nearest neighbours and decision trees - for labelling problem statements with tags related to their respective solution, which can serve as hints for a human solver. Benchmarking shows that each of the 3 algorithms is capable of outperfoming the others on certain target tags. The differences in limitations and strengths between the approaches display the diverse nature of automated reasoning tasks.

This paper is the result of my own work. No unauthorized assistance was received or given.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last few years, computers became able to solve difficult reasoning tasks such as writing mathematical problem proofs [38]. An interesting field of reasoning that has great potential in terms of utility is automatic code generation. A part of coding which became a usual learning strategy for programmers is competitive programming, which rose in popularity in the last decade. Competitions like IOI [21] and ICPC [12] have an increasing popularity, together with websites hosting contests such as Codeforces [4], Hackerrank [15] and Atcoder [18]. Being able to solve difficult competitive programming problems and industry difficult tasks have a certain degree of correlation. Competitive programming has also the advantage - from a learning perspective - of being a more controlled and organized set-up than programming in the industry (tasks are organized by difficulty, have sample solutions and intended target complexities). Motivated by the previously stated reasons, some recent research has focused on creating an algorithm for automatic code generation [23] in the competitive programming environment.

While the results are impressive in the current state of the art, there are still questions left regarding whether we can target solving more problems at the cost of giving away the end-to-end solution aspect and whether we can produce results which are more interpretable for humans. Trying to bridge the gap between human reasoning and artificial intelligence reasoning, this paper is proposing an algorithm which is meant to assist human problem solvers rather than replace them, targeting a better accuracy and ability to help solving harder problems. In order to achieve this goal, we are going to target predicting labels for possible solutions of competitive programming problems which would serve as hints for the human solver.

The need for exploring this kind of tasks is not only related to improving the productivity or assisting the skill of programmers. The research in this field might help understanding more about the way machine learning algorithms perceive reasoning tasks and could produce valuable results in terms

of knowing the limitations and advantages of various approaches.

1.1 Paper structure

This paper will go through existing algorithms, will propose an approach for the target problem and display an application which uses the proposed algorithm as an engine for an automated problem solving assistant.

In chapter 3 similar problems that have been recently explored are described, together with their approaches. In chapter 2, chapter 4, chapter 5 we focus on the theoretical part of the algorithm and on its implementation and performance, while section chapter 6 focuses on description of the practical application of our algorithm.

Specifically, in chapter 2 presents the problem and a formal definition of our target.Then, in chapter 3 we present 3 problems along with papers which solve the problems. We then try to derive useful insights and similarities to our problem. 3 algorithms are proposed in chapter 4, and later a detailed benchmark and analysis of their practical strengths and limitation is presented in chapter 5. A concrete application of the algorithm in a user-friendly set-up is presented in chapter 6.

Chapter 2

Problem

Previously we stated how creating an accurate algorithm for competitive programming problem statements would be beneficial for the fields of automated reasoning and code generation.Reducing the scope, we try to find a setup in which such an algorithm could be properly trained and evaluated. Note that the exact accuracy metrics will only be detailed in Chapter 5.

In this section we are going to define the concrete setup of the problem we are trying to solve, focusing on inputs, outputs and their respective characteristics.

2.1 Problem definition

Given a competitive programming problem statement, we aim to predict labels which would be hints about the solution. For both the input and the output, we use problems from the Codeforces[4] platform. A problem statement from Codeforces contains multiple sections (description, input and output for the desired program, samples for input and output). In this paper the main focus will be on the text section (statement description in natural language), as it is not as platform-dependent as the metadata (time limit, input format etc) and our current focus is on language based reasoning. Even though this focus might limit the results of this specific problem, it has more applications in reasoning tasks, being possible to adapt it to more natural language-based setups. Further research might explore more of the platform specific features. The target output which we would want to predict consists of the problem tags. Problem tags are hints related to the problem solution, which can be edited by the contest participants who solved the problems based on their solution and are from a finite subset of labels. They vary from some easy to derive from the statement without solving the problem (such as geometry, graphs) to others which have a deeper meaning and involve more understanding of the solution (dynamic programming, greedy). This difference between various tags implies our problem will

have different difficulties depending on the target label and we will try to take this into consideration when assessing our model and searching for limitations.

2.2 Goals

This paper aims to produce an algorithm which can produce useful insights in the prediction of problem tags and integrate it in a user-friendly application which can be used as a virtual assistant by a problem solver. The paper focuses on the language understanding part of the problem, our main data source being natural language. We can see predicting each tag for a problem as independent reasoning problems of varying difficulties. Side goal of our research will include providing generic insights about reasoning tasks based on natural language and understanding how the algorithms behave based on the difficulty of each individual reasoning problem.

2.3 Target definition

Having the input data as the problem statement (with only the data a contestant is allowed to see during the contest) the goal is to create an algorithm which has high accuracy in inferring all the tags. As mentioned, this can also be viewed as predicting a binary label for each possible problem tag, which means whether the tag should be attributed to the problem or not. Our algorithm should be an accurate predictor for the following function:

$$f(S,l) = \begin{cases} 0, & l \notin L(S) \\ 1, & l \in L(S) \end{cases}$$

$L(S)$ is the set of labels attributed to the statement S, l is an arbitrary label

Chapter 3

Related research

Even though the problem of labelling competitive programming statements has not been a focus for researchers, there are still tightly coupled problems (code and proof generation, scientific paper classification) which have promising results and can provide useful insights for the purpose of this paper's research. In this section we are going to explore some of the related problems from the fields of computer science and mathematical related text understanding and existing approaches.

3.1 Competition level code generation

One recently explored problem is generating code for competitive programming statements. The current state of the art manages to rank among top 54.3% competitive programmers [23]

The approach relies on deep learning - having a transformer based model. The pipeline includes pre-training the model on a larger sample of code (non-competition specific) from Github [19], finetuning and large scale sampling. Finetuning is done using GOLD[26] with tempering [8]. GOLD (Generation by Off-policy Learning of Demonstrations) is an offline reinforcement learning algorithm which is attempting to overcome the over-generalization[17] [35] of Maximum Likelihood Estimation in the use case of text generation and is able to improve transformer based models. Tempering is a regularization method aiming to adjust the token probability distribution by applying a transformation before the softmax layer.

The system is capable of solving problems with a reasonable accuracy, having a 28.4% solve rate in a multiple submission set-up.

Despite its powerful reasoning, the model is has limitations including dead code generation, copying from training set and sensitivity to metadata. Sensitivity to metadata (difficulty rating, tags) is hard to avoid, but it highlights the importance of having a clean dataset, with a minimal number of labelling

faults. Dead code generation does not affect the solution, but makes the solutions of the model less interpretable and harder to use in a real life human-assisted set-up. Copying from training set is a significant limitation in as the model becomes very sensitive to the training data and the reasoning of the model becomes more tightly constrained by the limits of human reasoning, being hard for it to produce solutions for difficult out-of-sample problems which would appear in a real-life set-up.

Another limitation, which is more related to the performance rather than solution type, is related to the accuracy metric, loss being a poor proxy for the actual solve rate. Defining a metric for the solve rate is a difficult problem because the problem has a one-of-many solution [24], thus parameter validation becomes itself a difficult problem and a limitation.

The research produced impressive results, being able to create an AI competitive with humans, but still leaves some questions open and creates a new class of problems which can emerge from attempting to surpass the limitations with possible trade-offs

3.2 Mathematical competition proofs

Similar to the previous problem, there is also recent work in the field of formal proof generation by Polu, Han et al[30], targeting progress in extensive planning and symbolic reasoning.

Using a deep learning approach (similar to GPT-3 [3]), systems are able to create solutions for mathematical problems. State of the art algorithm relies on the expert iteration methodology, based on GPT-f [31]. GPT-3 is a few shot learning algorithm which can be applied to a wide variety of Natural Language Processing tasks. Few-shot learning is a setting in which model is given demonstrations of performed tasks without being able to update its weights. GPT-f is a deep learning transformer based algorithm specialized in theorem proving

Generating the dataset of the state of the art involved syntethic inequality generation and manually formalising a dataset consisting of problems.

Solve rate (measured for the first attempt) achieved on miniF2F dataset [40] - a formal mathematical benchmark containing contest and mathematics classes - is 33.6% with the best algorithm. A limitation of the current state of the art is the limited reasoning depth - it is the most common case for the algorithm to generate at most 3 non-trivial steps in a proof for most problems. Even though it has the ability to solve some complex problems, it is still not able to compete with the top students participating in mathematical competitions. These limitations display a possible need of human assistance required by systems in order to solve more complex problems.

3.3 Mathematical aware automated classification and similarity of papers

Another similar topic which was recently explored is classification of scientific papers. Even though it is a problem from a different domain from ours, it is more similar in terms of evaluation, the problem having a clearer output target (known MSC [36] codes, similar to the tags which serve as target labels for our problem).

Růžička and Sojka [38] proposed an unsupervised learning algorithm, aiming to add a mathematical term - aware component to STEM paper classification. Their paper proposes a few versions of the algorithm, built based on Latent Dirichlet Allocation [1] and Latent Semantic Indexing [9] with Term Frequency Inverse Document Frequency transformation [33] (LDA / TfIdf-LDA, TfIdf-LSI). Latent Dirichlet Allocation (LDA) is a generative probabilistic model capable of understanding discrete data (including text corpora), able to extract topics through hierarchical Bayesian inference. Latent Semantic Indexing (LSI) is an indexing method used in Natural Language Processing based on matrix decomposition.

The main goal in the work of Ruzicka and Sojka, besides improving the classification accuracy, is testing the value MTerms add to the process. MTerms [37] are a special compressed and generalized encoding of formulae, which facilitate mathematical expressions as input to the model. Even though MTerms produce a good score if a model is trained solely on them (0.1743 using LDA), mathematical expressions did not provide a major improvement in the accuracy of the algorithm (F_1 score of 0.3427 compared to 0.3419 for text only data or 0.3366 for text and TEX data). This displays that there is signal at the level of mathematical expressions, but with the current state of the art in mathematical processing, it is difficult to infer non redundant information when added to text. Despite of this, the experiment created useful insights about what directions could be explored when processing mathematical data in classification and exposed the possibility to improve the accuracy of similarity algorithms on scientific papers using mathematical data.

Conclusions and insights

Going through all the mentioned papers, we notice that problem-solving oriented reasoning still has limitations - it is difficult for it to produce full solutions for the hardest problems, which require many non trivial observations. State of the art for both programming and mathematical competitions is fairly competitive with a human solver, which is an impressive progress in the field of machine reasoning. Progress was made especially with the help of deep learning, proper finetuning and data engineering,

producing accurate problem solvers for competitively difficult problems. However, the systems do not seem yet capable of exceeding top human solvers, and there is a long road to that point.

Some of the technical limitations of the problems were related to the problem set-up, for which is difficult to define a solve metric (except for the third problem), this being one of the reasons why choosing a binary set-up like in our case would be an accessible starting point for our research.

Overall, we can see there is definitely information which can be perceived by today's statistical learning techniques in the competitive programming problem statements. We will aim to train and adapt a model capable of capturing this signal and using it for the target of our paper.

Chapter 4

Proposed approach

Coming back to the original problem, our main target is to produce a valuable helper in hard reasoning tasks, in the setup of competitive programming problems. Labelling is a different approach as it transforms the system into an assistant rather than a problem solver, requiring more human effort, but can be helpful in solving more complex problems.

The current issue with state of the art models 2.1 and 2.3 is that they lack interpretability, relying on deep learning and producing end-to-end statement parsing to code generation. The advantage of labelling and a human assisted approach is that the system is allowed to have a margin of error while still producing a valuable result for the user, which can help overcoming the issue of limited difficulty a system is capable of solving given its limitations in depth of reasoning.

This section is exploring various possible transformations of different ways to reason into implementable algorithms. Note that our approaches mainly rely on training on a single target variable and then combining the results. The reasoning behind this is based on the diverse nature of problem tags (some of them requiring significantly deeper reasoning than others) which causes finetuning the models to become significantly harder when trying to train a single multilabel predictor.

4.1 Reasoning based on common tokens

Decision Trees are a good knowledge representation method which can be applied to reasoning tasks. There are supervised learning algorithms [32] that are able to build decision trees based on a dataset using statistical greedy estimations for the tree's reasoning quality.

The current approach is aiming to create a humanly interpretable reasoning which is easy to check. For this, both simply processed features and an explainable algorithm are used. A decision tree is a good fit for our requirements, as it can produce complex reasoning (in multiple steps) without trading

too much explainability. Note that through the process we will assume we are targeting a single problem tag as output label, and repeat the steps 4 and 5 for all the tags. Steps 1,2 and 3 will only be done once in the beginning for all the tags.

We are going to use the implementation provided by scikit-learn [28] for the decision tree

The methodology for training consists of the following steps:

1. Find the most common words in the training dataset
2. Manually filter words which are not related to the problem definition (prepositions, common verbs, pronouns). Denote the manually filtered set as T.
3. Define the function $occurs(problemStatement, commonWord)$ which is simply checking whether a certain $commonWord$ has at least one occurrence in $problemStatement$.
4. For a problem statement S, transform it into a binary array which contains the value of $occurs(S, t)$ for all $t \in T$. We can do this by tokenizing the text and filtering to have only unique words which appear in T
5. Train a reasoning model using the binary vectors generated by the entire dataset. The reasoning model we are going to use is a decision tree, as it provides an interpretable reasoning which can be used for a better understanding of the algorithm's reasoning and can serve as an extractory data analysis step for the following approaches.

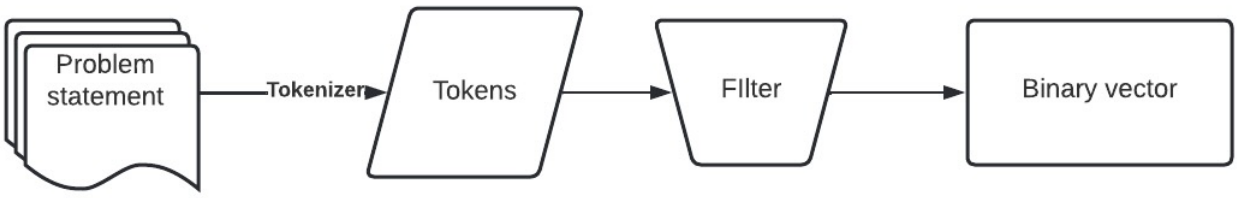

Figure 4.1: Pipeline for feature generation in the common token approach

4.2 Similarity based approach

We are shifting out attention to an approach that is meant to work in space not limited by a manually selected set of tokens, having the advantage of being easier to adapt on other datasets and taking into consideration more than a very limited set of words which involves human effort on the selection side.

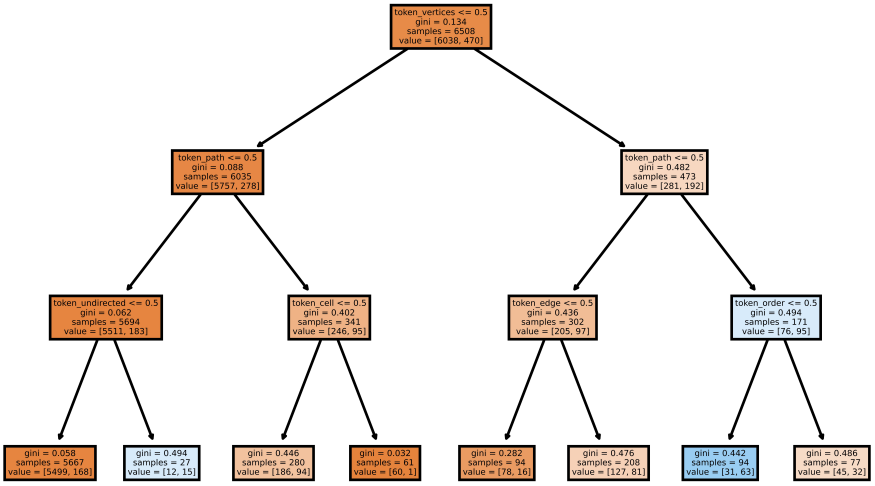

Figure 4.2: Sample decision tree representation

As we seen in the state of the art algorithms for the related problems there is learning value in embedding the text into a geometrical space, as it is one of the best performing techniques of transforming text in terms of information loss.

In order to achieve the embedding of the text, we are going to use a state of the art language transformer [10]. Having a space in which we know have a metric for distance (cosine similarity between

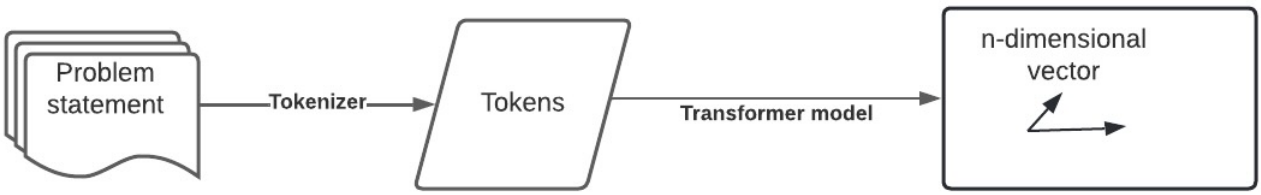

Figure 4.3: Pipeline for feature generation in the similarity based approach

two embeddings), we can use an algorithm which predicts based on the training set statements "closest" to the input statement. For this, we can use a k-Nearest-Neighbor algorithm [6] with the n-dimensional cosine of 2 embedded vectors as distance metric.

One pitfall of the k-Nearest-Neighbor majority approach is that it does not take into account the mean of the target, so we will define a weighted mean used in kNN in order to increase sensitivity of the algorithm for labels which are not common. Our prediction for the statement S to have tag T will

be:

$$p(S,T) = \begin{cases} 1, & if \frac{\sum_{S' \in kNN(S)} isLabeled(S',T)}{k} > \overline{L_T} \\ 0, & otherwise \end{cases}$$

where

$$L_T$$

is the multiset containing the values for

$$isLabeled(S'',T)$$

for a large sample of statements S'' that appear in the historical data known by the model (in our implementation the large sample is a set of neighbours for a significatly larger k threshold - ideally the whole dataset should be used but this is a performance optimisation).

4.3 Deep Learning

Noticing that the kNN approach is still quite limited as the domain specific information might not be highlighted by the embedding model, we are going to try an approach similar to the ones of the state of art algorithms 1 and 2 presented in chapter 3. The approach will rely on finetuning an embedding model and switching the objective to a target. For simplicity and flexibility of finetuning we will keep one of the ideas from the first algorithm and train a separate network for each label.

4.3.1 Finetuning

Our architecture relies on a frozen transformer [11], for which we will add more layers in order to finetune. The output cosists of 2 logits, with the probability for each class (True or False). This architecture is equivalent to a model which takes as an input the n-dimensional vector produced by the transformer and outputs the probability for the vector to correspond to a problem having a specific tag. The model architecture will be detailed in chapter 5

4.3.2 Loss function

We are going to use cross entropy [7] a loss function. It is a good fit for a binary distribution regardless of its mean. As an optimizer for the neural network, we are using Adam, a gradient based algorithm which uses estimations of lower order moments [22].

Chapter 5

Algorithm Implementation

In this section we will focus on implementing the algorithms described in Chapter 4. We are going to describe the dataset, show performance benchmarking of the algorithms and the insights provided by it and describe how the data analysis resulted in hyperparameter tuning.

5.1 Data

The dataset [5] is generated from Codeforces [4] statements, containing all statements and labels for over 6000 problems from more than 1000 contests. It has the same content as on the website, except for mathematical formatting - our algorithms do not have any specific formula handling. We are going to use the plain problem statement as input (no metadata). The sample output for each problem is a set of tags - which we are seeing as a binary vector for all the possible tags for our performance measurements. There are over 30 tags, with various meaning depth - which will be visible in the performance benchmarking. The problem tags are set and edited by community members which solved the problem.

5.1.1 Extractory data analysis

One valuable insight is understanding the distribution of our target output. All the target variables are in a binary distribution. We can observe some of the most significant means (most and least frequent tags) of the dataset:

Extractory data analysis has also been a key part of the implementation of the first algorithm, based on manually extracted tokens. A tokenization process was run on the entire dataset, after that, the set of most common word tokens was extracted. The need for selection was justified by the nature of the most common tokens - mostly prepositions and common words such as the, of, to - which are

Label	Mean
Data Structures	0.154621
Brute Force	0.140357
Constructive Algorithms	0.141556
Combinatorics	0.055256
DP	0.200408
Sortings	0.090974
DFS	0.086060
Trees	0.073954
FFT	0.007312
Ternary Search	0.006233

Table 5.1: Mean of the binary label for various targets

Token	Number of occurrences
first	19390
possible	6657
string	5957
order	4252

Table 5.2: Number of occurrences for most frequent feature tokens

unlikely to produce useful signal for the scope of our problem. The selection process could also be performed by the decision tree algorithm indirectly when computing the most relevant features on gain, but adding non filtered features would add a lot of memory usage to the model and significantly increase the execution time, given the large numbers of irrelevant features we would have if we would simply take the most frequent tokens.

Frequency of the most common relevant tokens

5.2 Methodology

We are going to evaluate the accuracy using F1 metric for measuring prediction accuracy for each label individually.

The reason behind the choice of these benchmarking methods is related to the nature of our output. True positive and false positive rates are an important consideration, as for our algorithm we are going to have multiple target labels, all having different binary distributions in the dataset (as we've seen in the extractory data analysis, some tags are dense in the dataset - such as DFS, contained by 20% or the problems, while other like FFT appear in less than 1% of the problems).

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

$$precision = \frac{TP}{TP + FP}, recall = \frac{TP}{TP + FN}$$

where: TP = true positive count

FP = false positive count

FN = false negative count

For all the methods we applied a train/validation/test split (with a ratio of roughly 80%/10%/10%). Since the contests are ordered in time, we used a time-based split, which is more accurate for simulating out-of-sample inference. Even though there are no obvious sources of data leaks, there might still be changes in elements such as problem writing style or difficulty that would be more accurately simulated by more recent problems for the test setup.

5.2.1 Model architecture

The architecture of the first 2 types of models does not go beyond the ideas stated in chapter 4. The optimizations which lead to better results will be presented in section 5.3.

Both approaches that rely on deep learning (kNN and finetuning) are based on the all-MiniLM-L6-v2 transformer model [11] with an output of size 384.

Finetuned neural network

For the finetuning approach, our optimal model architecture has the following structure:

1. Pretrained transformer model with frozen weights
2. 3 hidden layers with 400 neurons and ReLU [39] activation function
3. Output - 2 logits - probability for each class (true or false)

5.3 Hyperparameter tuning

The most important parameters for algorithms 1 and 3 which were finetuned were the k parameter for the kNN similarity approach and the tree depth for the decision tree based approach. These parameters serve both as a way to optimize their respective algorithms but also as a method to identify the optimal amount of information and the depth of reasoning required to deduce each individual tag. We can see

Token	Optimal tree depth	Optimal k
Probabilities	2	-
Math	10	5
Number Theory	-	15
Geometry	3	10
DP	10	15

Table 5.3: Optimal parameters for decision trees and kNN

in the table the optimal depth (algorithm 1) and optimal k (algorithm 3) for a selection of tags (− means that score was not significant enough for it to matter). Even though for some it might seem that bigger depth is correlated with number of neighbours, we can notice that monotony is not the general case. This is most likely due to the different nature of the features, the algorithms operating on domains which are hard to compare and contain different types of information.

For the neural network finetuning based approach, as mentioned in section 4.3, our neural network consists of 3 hidden layers added to a pretrained transformer model (with frozen weights) and it uses the Adam optimizer. The hyperparameters used for training are a learning rate of $5 * 10^{-4}$ and 100 training epochs. In order to prevent overfitting, the optimal model is set as the model with the best test score (F1 score) out of all the epochs. As the amount of information given by the pretrained frozen layers is quite large for the scope of our problem (generic transformers are powerful in a multitude of independent language tasks) it is a difficult problem to choose the hyperparameters which lead to a smooth loss curve. For many of the experiments, the model had the tendency to improve significantly in a small number of epochs. This was mainly limited by reducing the learning rate (originally 10^{-3}). We have not noticed significant improvements by reducing the size of the hidden layer (to 200) or decreasing the learning rate more (to $2 * 10^{-4}$).

5.4 Results

The results in the Table 5.4 represent F1 scores for each method and each of the training set tags.

Alternative metrics

We have also measured different metrics for the validation of the better performing models. For the decision tree models, as it is possible to extract the probability to have a prediction for each class, we can have a sensitivity based metric as an alternative measurement. The chosen metric is area under Receiving Operator Characteristic Curve (ROC-AUC) [2]. For ROC-AUC, the scores on the labels on which F1 is high validate the quality of the models. However, for the labels with very low means, the

Label	Decision trees	kNN	Deep Learning
DSU	0.04347	0.06444	0.09090
Geometry	0.13043	0.05010	0
Data structures	0.21259	0.0685	0.30817
Brute force	0.02409	0	0.18360
Constructive algorithms	0.05447	0	0.1846
Combinatorics	0	0.12291	0.13157
DP	0	0.40953	0.32242
Sortings	0.03007	0	0.168539
DFS	0.21782	0.16122	0
FFT	0	0	0
Games	0	0	0.54545
Implementation	0.068376	0.36775	0
Interactive	0.12	0	0
Schedules	0	0	0
Bitmasks	0	0	0.11764
Binary search	0.03076	0.23851	0.06666
Flows	0.08695	0	0
Number theory	0	0.14799	0.36363
Hashing	0.15384	0	0
Probabilities	0.34482	0	0
Math	0.07784	0.49958	0.41262
Divide and Conquer	0	0	0
Expression Parsing	0	0	0
Shortest paths	0	0.06652	0
Two pointers	0.02564	0	0.07058
Greedy	0	0.56642	0.43423
2-SAT	0	0	0
Matrices	0	0	0
Graphs	0	0.22637	0.38674
Strings	0	0.13650	0.54999
String suffix structures	0.363636	0	0
Ternary Search	0	0	0
Trees	0.36036	0	0.47692
Chinese Remainder Theorem	0	0	0

Table 5.4: F1 scores for the 3 algorithms

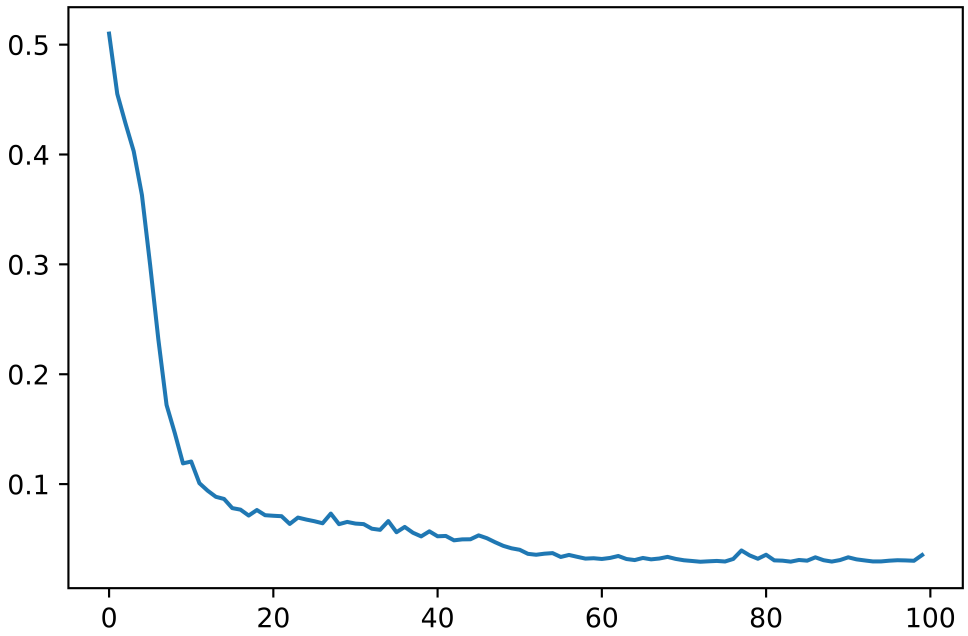

Figure 5.1: Neural network train loss for the "DP" tag over 100 epochs

Label	ROC-AUC
Trees	0.6827
String Suffix Structures	0.7315
Graphs	0.6411
Geometry	0.7548

Table 5.5: ROC AUC for Decision Trees on various labels

AUC is quite sensitive and easy to increase by simple models which have low accuracy, justifying the choice for another metric as the main benchmarking method.

For the deep learning approach, one additional validation measurement was evaluating the evolution of cross entropy loss in order to avoid overfitting.

5.5 Discussion

Going through the benchmark data, we notice that there is no clear winner for all the categories between the decision tree reasoning and embedding based approaches. The common token based approach seems to perform better in tokens which provide hints about rather specific domains (Probabilities, String suffix structures), while the transformer based approaches are better at identifying broader domains (Dynamic Programming, Greedy). This is explainable by the nature of the algorithms, the first one relying on human prefiltering it can reach more depth of insight in understanding some tokens, while being stuck into the narrow domains which can be described by a few common words. The other ones

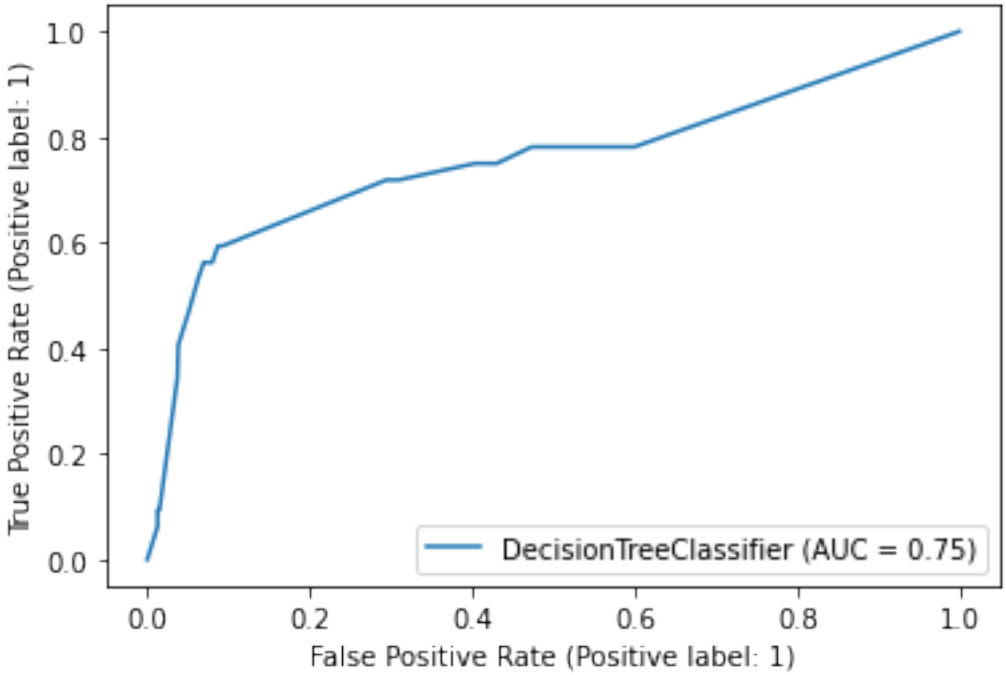

Figure 5.2: AUC plot for the prediction of "geometry" label - decision tree approach

however, lacking some of the domain specific knowledge which is rather difficult to incorporate into a text-only input model, are not performing well in the narrow domains, but have a better understanding of the broader domains, where word similarity can matter much more (e.g. for a dynamic programming problem, goals like *optimize* and *maximize* can be perceived as similar in the statement by a similarity based model, while for the other model, while for the first model, only if both were introduced to it initially as a feature might catch both).

The kNN and finetuning approaches are better than decision based approach on similar topics, as they both rely on the same type of information. There are still tags for which kNN is better than finetuning. There is no obvious indicator for a way to improve the finetuning approach, but since kNN is relying on linear operations (dot product, maximum), an ideal neural network should be able to replicate the process. However, as the progress would not be significant for many of the tags and such a network might become too large for our resources, we view the problem of incorporating the entire kNN reasoning into a neural network as outside of the scope of this paper.

One aspect which is clear in the case of all algorithms is that tags which are humanly difficult to spot are also limiting the algorithm accuracy. For instance FFT - Fast Fourier Transform - is a tag for which none of the algorithms manage to produce significantly many correct predictions - is often difficult to spot in a problem, as it is in the most cases decided by a human solver after producing multiple observations which transform the initial problem into a polynomial multiplication

(the most frequent application of FFT in competitive programming). This is a mathematically difficult reasoning problem, outside the scope of language understanding, and might require multiple non-trivial mathematical observation - which are difficult to achieve even for state of art models, as presented in the first two papers of chapter 3.

Even though there is no clear winner over the all the subproblems, it seems that the approach which is most consistent in producing non-trivial F1 scores is the deep learning finetuning. It is indeed convenient as it is easier to adapt to new datasets, requiring a small amount of feature preprocessing. In order to achieve better results, hyperparameter tuning is still difficult as the model relies on a very complex transformer that lacks interpretability, as opposed to the other features. Another limitation for the deep learning approach is the significantly larger amount of time required for training - the other methods converge significantly earlier. However, as the improvement for the latter epochs was not large in none of our experiments a fair amount of accuracy could be traded for a better training time.

Overall, the strong and weak points of each algorithm seem to be easy to interpret and correlate with the nature. Even though there are some advantages of each method (which are fuzzier between methods 2 and 3, but they still outperform each other on certain tasks), there is no clear winner for the overall problem understanding.

Chapter 6

Application Integration

In this chapter we are going to describe the objectives and implementation of an application which integrates our algorithm. We will focus on both the user experience and engineering details which describe the concrete implementation of our application.

6.1 Motivation

Coming back to one of our initial goals, we would like to have a digital helper in using difficult reasoning tasks which is easily usable. We will try to wrap our algorithm in an application which requires no knowledge of the internals for usage. For this purpose, a web app experience is a convenient choice, as it is similar to the interface of competitive programming websites which contestants use for training and is facile for new users as it requires no installation.

6.2 Use cases

We will define the scope and the functionalities of the application wrapping our algorithm. The application is mainly meant to produce an easily usable digital assistant for solving problems, providing an intuitive interface for our algorithm. The use cases will all be centered around getting prediction data for statements. A user will be able to input a problem statement and get the basic tags (having a similar interface to the source data, Codeforces, in which the use simply sees text tags) or, for a previously queried statement to get a more detailed report of the predictions - probabilities for both previously visible (positive prediction) and invisible (negative prediction) labels. The input source for the statements can be both a plain text statement or a programming judge statement link (currently the only supported judge is Codeforces, but the architecture allows easily adding a new platform).

Concretely, a user of the web app should have the option to:

- input a problem statement and receive information about most likely tags for the problem
- get more information about specific tags (probability for the tag to be a good fit for the problem)
- input a link to a specific problem from an accepted website and receive the same information as for a statement

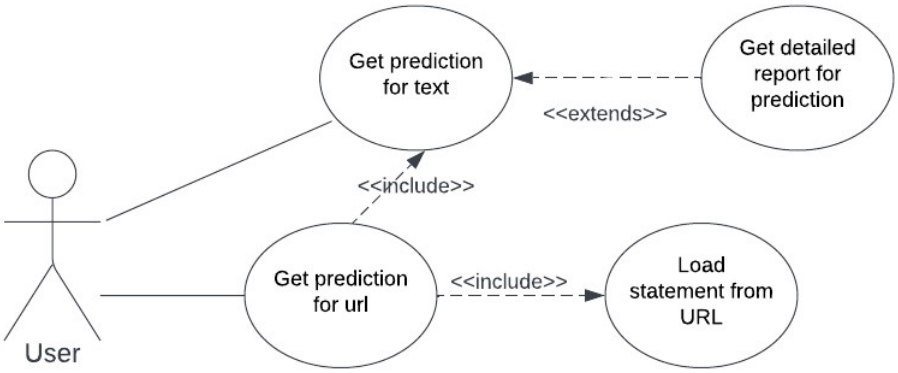

Figure 6.1: Use case diagram

6.3 API Specification

All of the following endpoints work with *POST*

- */problem* Given a statement S returns a unique integer identifier *statementId* and a list of predicted tags sorted by probability of being correct in descending order
- */report* Given an integer *id* returns a detailed report of the problem with the corresponding id, or an exception if no such problem exists
- */url* Given an URL from a predefined list of accepted websites, parses the statement found at that URL returns a the statement, which can be further adjusted and passed to the *problem* endpoint in order to get the predictions. If the URL is not a valid URL from an accepted website returns an exception

6.4 Design

Our application will follow a client server architecture. We are going to create a web server and client for simplicity and portability of the application, as the responsibilities of the client will be rather reduced.

For the client, we have a two tier architecture, based on a User Interface layer and a service layer. The UI layer will consist of components, having a functional architecture, which is defined by having multiple functions which define visual components and their states. The service layer will be divided into a storage module which is responsible for persisting local data (problem identifiers, session data) and an API module, which is responsible for fetching the data from the server through HTTP.

For the server, we are going to have an architecture which is loosely coupling the prediction algorithm and the rest of the business logic (storing predictions, parsing pages). The prediction module module will provide an universal interface which is aiming to make switching between models easy (e.g from decision tree models to similarity based model). The internals of the module will be divided between a prediction submodule and a feature submodule, which will provide a processed input for the prediction module.

The storage module will also have a universal interface independent of the underlying persistence structure, but for now will contain only a SQL based implementation.

The utility module will be responsible of handling and validating other data sources (e.g. parsing external links and assuring they are from the allowed domains). It will be used as a first pipeline step in processing inputs before passing them to the prediction module, which will perform the model specific transformations depending on the configured model strategy.

A service will be connecting the 3 main business modules and will provide an interface for configuring the strategies used for predictions, input processing and persistent storage. An API module will provide an interface for the service module and expose endpoints corresponding to the main functionalities of the service.

6.4.1 Class design

The classes will fit the abstract module design and will be based on the design patterns specified in **??**. The UML class diagram Figure 6.3 contains the classes and their respective relationships.

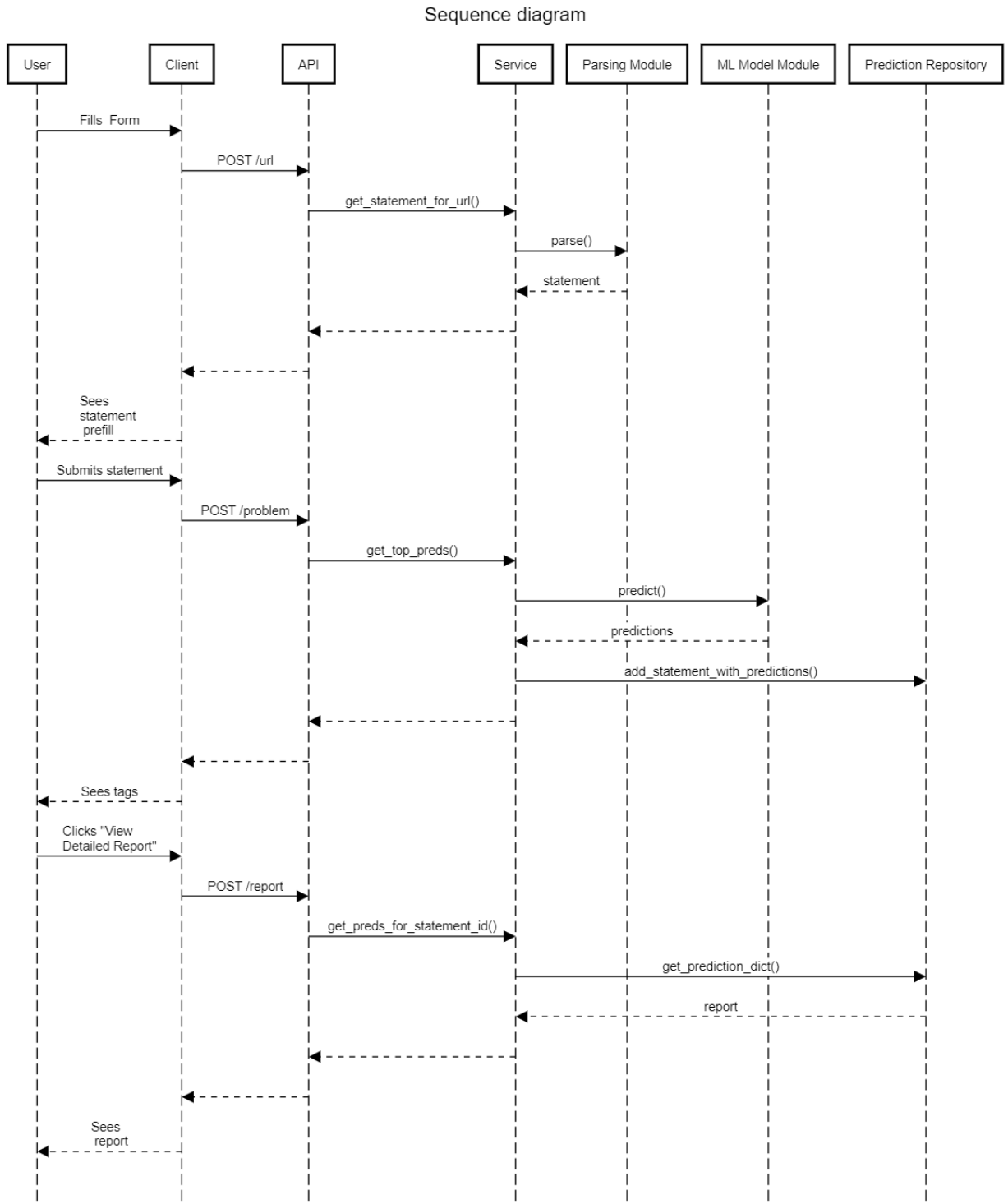

Figure 6.2: Sequence diagram

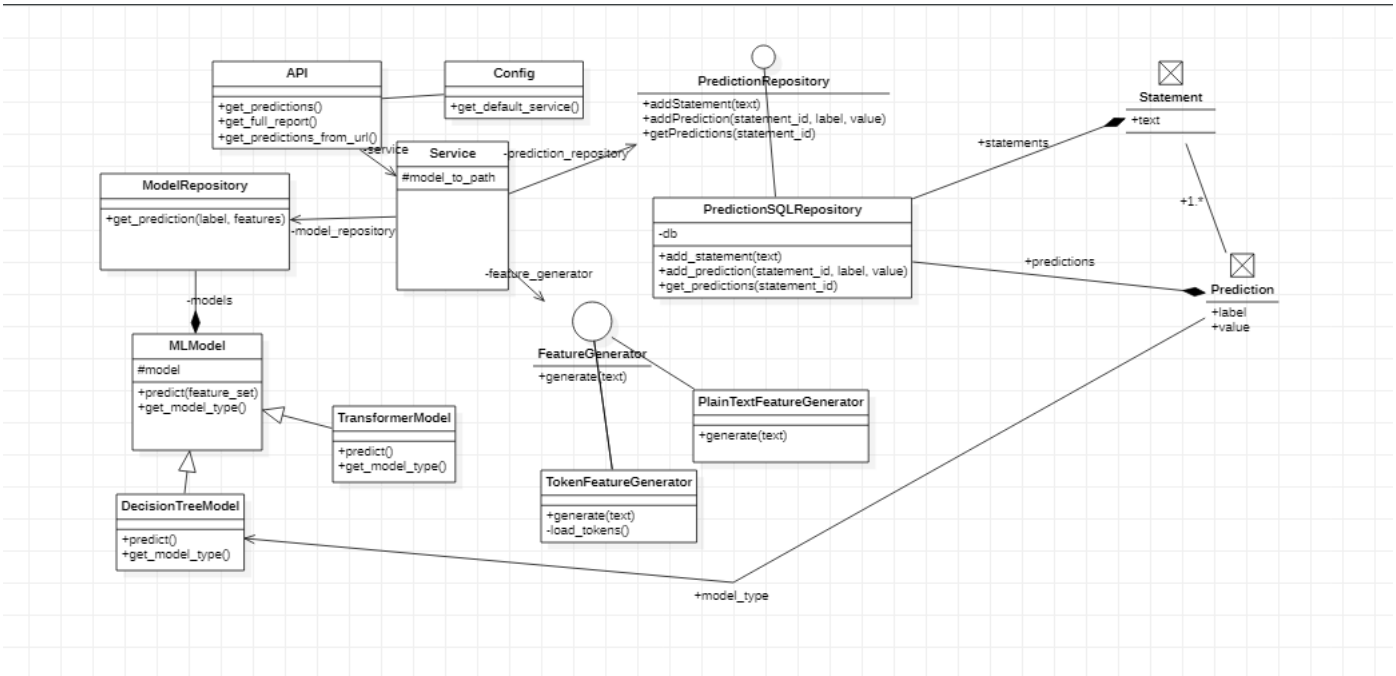

Figure 6.3: UML Class Diagram for the API implementation

6.4.2 Database design

We are going to use an relational database in order to store our predictions. The choice is justified by having a simple structure, with only 2 main entities, between which we would want to define a relation in order to easily represent and link the instances. The entities of our domain will be the problem statements and predictions.

There will be a one-to-many relationship between statement and prediction.

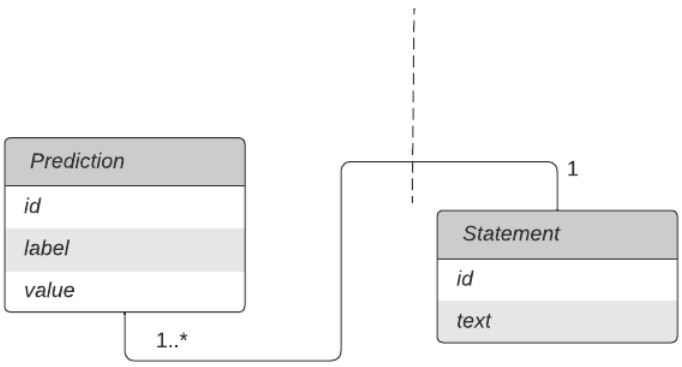

Figure 6.4: Database diagram

6.5 Implementation

In order to implement the system in a low coupling high cohesion way, we are going to use some design patterns specified by Gamma, Helm, Johnson and Vlissides [14]. For the prediction models and feature sets we are going to use strategy pattern, consisting of the base interfaces $MLModel$ and $FeatureGenerator$ and algorithm specific implementations (e.g. $DecisionTreeModel$ for $MLModel$ and $TokenFeatureGenerator$ for $FeatureGenerator$). A similar approach will be provided for $PredictionReposito$ for easier compatibility with future changes in the storage structure, but for now will have only one implementation ($PredictionSQLRepository$). The relationship between $ModelRepository$ and $MLModel$ can be seen as a factory pattern, the repository being able to create instances of model based on specifications.

6.5.1 Testing

For testing the system, we will first rely on the model performance measurements performed at the training step. Whenever we are going to test the integration of a feature, we will try to abstract the usage of the model, as many options are not deterministic and subject to sensitive changes. We intend to apply a black box testing approach for input and output processing, using a driver which replaces the actual model predictions with a test version which will deterministically produce the same predictions. The main focus of our test will be having a correct way of storing the predictions and parsing the inputs (2 out of the 3 main modules of business logic).

We also planned a white box testing strategy for the feature extraction submodule of the prediction module, making sure that the feature order is compliant with the one requested by the model part of the module.

Some integration manual testing will be added on the top of existing tests, using Postman [20] as an API client for simulating a production setup for the backend interface calls.

6.5.2 Technologies

We are going to implement the server in Python [13], using the Flask [25] framework for providing an API. One important aspect of the choice is having the same language as in model training, being easier to port code for feature generation and also for using the scikit-learn [28] and Pytorch models [27]. We are opting for SQLite [16] as a database engine for storage persistence, as it provides a lightweight binary and a simple SQL interface. For an easier implementation of the */url* endpoint, we are also going to use BeautifulSoup [34] as a web parser.

For the web client, the implementation use React [29] for both component rendering and state management.

6.5.3 User experience

The user experience is designed to fit the previously described use-cases. We can see the steps described by Figure 6.2 in Figure 6.5, Figure 6.6, Figure 6.7 and Figure 6.8.

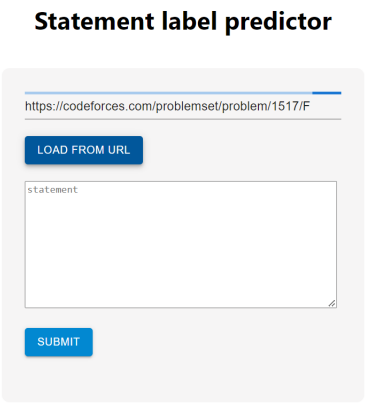

Figure 6.5: User inputs a Codeforces URL

One limitation of the application is that answers cannot be delivered instantly. Delivering instant results for URL parsing is difficult, as the platforms tend to have limitations on the requests in order to avoid attacks and limit web crawling. The prediction part, as it relies on the prediction of multiple models which are quite complex and also has a text tokenization part, it is also difficult to bring to an instant delivery state. A loading bar was introduced in order to optimize the experience under this limitations. A future version might use a caching mechanism or more preprocessing for the popular websites' data in order to avoid request latency.

The implementation of the web client makes it possible to share and revisit previously seen reports using an URL scheme which takes the id of the problem as a parameter ($\{domain\}/report/\{id\}$)

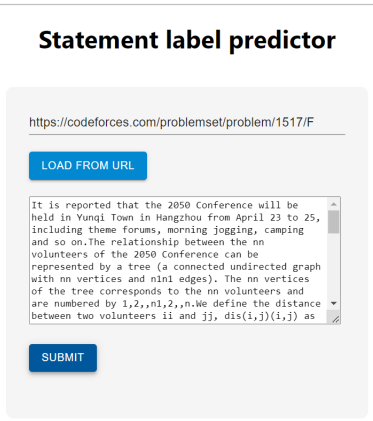

Figure 6.6: The statement is loaded and can be modified by the user

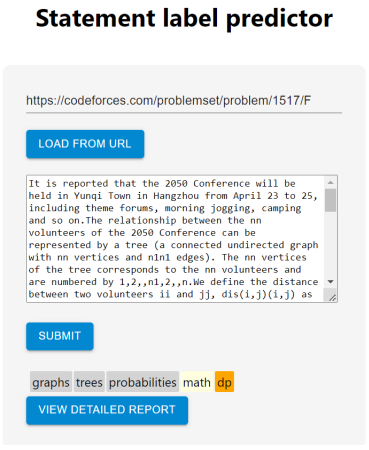

Figure 6.7: After submission, tags are generated for the problem

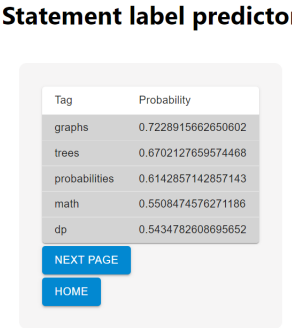

Figure 6.8: After tags are generated the user can view the detailed report

Chapter 7

Conclusions and future work

Overall, we have compared quite different approaches with different strengths and limitations. Benchmarking proved that all of our 3 methods are able to outperform the other ones on certain tags. One was better with identifying very specific problems (the decision tree approach), while the other ones proved to be better on more general fields. This displays the diverse nature of different automated reasoning problems, and also the difficulty in creating a general approach capable of easily switching between set-ups. Given the nature of the specific subproblems in each algorithm was the best performer, we can observe the type of tasks each of them are better at - whether it is a reduced, well defined but difficult scope, or a wide domain, but for which the knowledge requirements can be more relaxed. We have also noticed the advantages and disadvantages of more human assistance in an algorithm, which was definitely a helper for the first category, but hard to introduce to the second one. There is definitely a trade-off between specialization and generalization, and finding the right balance can be difficult. A combination of all the approaches presented in this paper would be our current best solution.

Future work

As all of the approaches have proven to have stronger points, exploring a combination of them might be the easiest to achieve way an improvement in our problem. However, combining the models is a problem itself, as the approach for a an aggregate algorithm might take multiple forms, from using different granular models for different tags based on experiment scores, to extracting preprocessed data from the models and creating a more complex pipeline.

Beyond creating an algorithm able to integrate and marginally improve the performance, treating the limitation regarding difficult tags is a problem which is still open, but it is highly correlated with

the problems that state of the art problem solving models face and it might require more progress in the general field of machine reasoning, which is difficult to achieve.

While our research focused on a clearly defined set-up with a binary target label, opposing to the previous work in the field which focused on creating full solutions, there are still unexplored paths on the spectrum of possible set-ups, such as providing partial solutions or hints which are not predefined by a binary label.

Beyond the machine learning part of the paper, there are also improvements to be made in the application area. Even though it works well as a proof of concept for our algorithm and provides useful insights for a set of problems, its user experience could be improved by a better integration of the programming training web platforms and a better performance.

Bibliography

[1] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[2] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[4] Codeforces. Codeforces website. Platform, 2018.

[5] Kaggle Community. Codeforces Dataset. link.

[6] Pá draig Cunningham and Sarah Jane Delany. k-nearest neighbour classifiers - a tutorial. *ACM Computing Surveys*, 54(6):1–25, jul 2022.

[7] G. Cybenko, D.P. O'Leary, and J. Rissanen. *The Mathematics of Information Coding, Extraction and Distribution*. The IMA Volumes in Mathematics and its Applications. Springer New York, 1998.

[8] Raj Dabre and Atsushi Fujita. Softmax tempering for training neural machine translation models, 2020.

[9] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[11] Hugging face. sentence-transformers/all-MiniLM-L6-v2. Documentation.

[12] ICPC Foundation. Internation Collegiate Programming Contest. Official website.

[13] Python Software Foundation. Python. link, 2010.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[15] Hackerrank. Hackerrank. Platform.

[16] Richard D Hipp. SQLite, 2020.

[17] Ferenc HuszÃ¡r. How (not) to train your generative model: Scheduled sampling, likelihood, adversary?, 2015.

[18] AtCoder Inc. Atcoder. Platform.

[19] Github Inc. Github. link, 2022.

[20] Postman Inc. Postman. link, 2022.

[21] IOI. International Olympiad in Informatics. Official website.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[23] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, RÃ©mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode, 2022.

[24] Yatin Nandwani, Deepanshu Jindal, Mausam, and Parag Singla. Neural learning of one-of-many solutions for combinatorial problems in structured output spaces, 2020.

[25] Pallets. Flask. link, 2010.

[26] Richard Yuanzhe Pang and He He. Text generation by learning from off-policy demonstrations. *CoRR*, abs/2009.07839, 2020.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[29] Meta Platforms. React. link.

[30] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

[31] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[32] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[33] Anand Rajaraman and Jeffrey David Ullman. *Data Mining*, page 1â17. Cambridge University Press, 2011.

[34] Leonard Richardson. Beautiful soup documentation. *April*, 2004-2020.

[35] Loic Simon, Ryan Webster, and Julien Rabin. Revisiting precision recall definition for generative modeling. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5799–5808. PMLR, 09–15 Jun 2019.

[36] American Mathematical Society. Mathematics subject classification. link.

[37] Petr Sojka and Martin Líška. Indexing and searching mathematics in digital libraries. In James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors, *Intelligent Computer Mathematics*, pages 228–243, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[38] Michal Růžička and Petr Sojka. Towards math-aware automated classification and similarity search of scientific publications: Methods of mathematical content representations, 2021.

[39] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015.

[40] Kunhao Zheng, Jesse Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics, 08 2021.