Patient-Clinical Trial Matching Problem

Technical Demo: A comparison of a semantic and NLP-based approach

Tomás Esber, Vikram Sharma, Marvin Pinerva-Lobos

We present two novel algorithms for addressing the Patient-Clinical Trial Matching Problem. Our approaches borrow from existing algorithmic systems and databases to determine whether a patient is eligible for a given clinical trial. The first algorithm we propose is a novel semantic-based parser. The second algorithm we propose leverages Natural Language Processing (NLP) to perform named entity recognition (NER) on unstructured text. We conclude our discussion of these algorithms by comparing their performance on Chia, a publicly available clinical trial data-set.

---

1 Introduction

Given unstructured patient health data and unstructured clinical trial eligibility data, the Patient-CT Matching Problem seeks to determine whether a patient is eligible for the trial, or not. This is a prevalent problem in the field that has, in recent years, begun to be addressed through the use of NLP techniques. In this technical demo, we will discuss two algorithms, one classical and one using machine learning, that intend to address the Patient-CT Matching Problem. We then discuss the limitations of our algorithms. Our algorithms build off of existing work done by researchers in the clinical trial space [1], [2], [3], [4]. You can find all of the code for this project via this link: https://github.com/tomas-esber-dev/cs333final.

2 Approach 1: Semantic-based parser

This is a classical (non-ML) algorithm invented by our team to understand strings as logical binary tree entities.

2.1 Input data

The intuition behind this algorithmic system stems from the idea that, largely, patient health records and clinical trial eligibility criteria can be represented as chunks of entities grouped together by a combination of logical ANDs and ORs. This algorithm takes as input labeled, unstructured patient health records and clinical trial eligibility data. We obtained the labeled data from Chia [1]. This data was labeled via a combination of manual and automated techniques. The exact specifications of labeling are out of scope for our discussion.

Table 1 demonstrates what the labeled input data for this algorithm would look like. There are more than a dozen entities labeling text as Condition, Procedure, Person, Drug, Observation, and so on. There are also relational labels such as AND, OR, Has_value, Subsumes, and more. The intuition of our algorithm lies in the fact that we represent entities as leaf nodes in a binary tree, and relational labels as parent nodes. Moreover, we modified the Chia data-set to make OR a 'special' relational label, and so it is represented with an 'O' instead of an 'R'. This is to allow the algorithm to select the 'special' relational labels as the root node of the tree.

Our algorithm begins by parsing through the labeled text file, and creating a logical binary tree representing entities and their relationships. We take advantage of the fact that relational labels have exactly two arguments in this labeled data-set, and so we are easily able to represent the relational label and its arguments as a parent node with two children.

entity_id	entity_type	arg1	arg2	text
T1	Condition	nil	nil	metastatic carcinoid tumors
T2	Value	nil	nil	proven
T3	Procedure	nil	nil	biopsy
R1	Has_value	T3	T2	nil
R2	AND	T1	T3	nil
T4	Condition	nil	nil	other neuroendocrine tumors
O1	OR	T1	T4	nil

Table 1: Example of Labeled Clinical Trial Eligibility Criteria

2.2 Tree Creation

In practice, we create a Map of `entity_id` objects such that the keys of the map are the `entity_id` and the values of the Map are objects (nodes) that store pointers to other nodes (their children). If nodes are Entity types (i.e., non-relational), then the nodes simply point to `null` (they are leaf nodes).

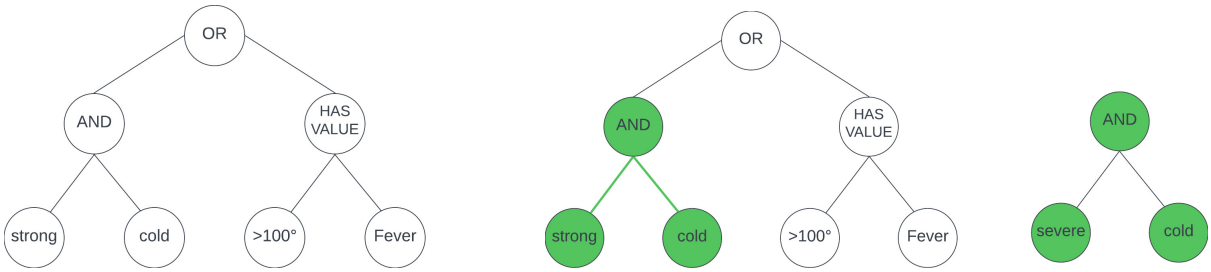
Algorithm 1 outlines the general steps required to create a tree-like Map from a text file. Note that we must use a Map since we are essentially over-writing Entity nodes if they have been modified by some sort of relation. For example, if on line 1 of the text file we encounter the entity T1: 'Flu', and on line 2 we encounter the entity T2: 'severe', we introduce those as separate key-value pairs into our map. That is, the key T1 points to a node containing the word 'Flu' (along with other metadata), and the key T2 points to a node containing the word 'severe'. Now, on line 3, suppose we find a relationship of 'R1, Has_value: T1, T2'. We introduce the key R1 and assign it a node with the associated metadata. But, we also want to over-write the node of T1 to include this new information. So T1 now points to the node held by R1. For any future node that wishes to point to T1, it will now point to R1, since R1 represents a modification of T1. Of course, this assumes that the text file is ordered. That is, the first line in this case can never be R1. The information for T1 and T2 must always come first in the text file before R1 appears.

Algorithm 1: Tree-like Map Creation

```
Data: Txt file
Result: Map<String, Node>
Map ⟶ {};
for line in text do
    if line is Entity then
        entity_id = line[0];
        Map.put(entity_id, new EntityNode(line[1:]));
    end
    if line is Relation then
        entity_id = line[0];
        arg_one = line[1];
        arg_two = line[2];
        Map.put(entity_id, new RelationNode(arg_one, arg_two, line[3:]);
        Map.put(arg_one, Map.get(entity_id));
    end
end
```

2.3 Tree Comparison

So, we perform Algorithm 1 on both labeled CT eligibility text and a patient's labeled health text. Now, we have two binary tree representations of the text records. Our task is to compare the two trees to determine if the patient is eligible for the CT. We propose a tree comparison algorithm for doing so. This algorithm first compares the entire tree structures of both trees. If they are not equal, then it checks all possible sub-trees of the clinical trial tree to see if the patient tree is a sub-tree of the clinical trial tree.



(a) Clinical Trial Logical Tree

(b) Tree comparison via sub-tree search

Figure 1: Tree Comparison between Trial and Patient Trees

As part of our comparison algorithm, we developed a recursive methodology to perform the necessary tree and sub-tree search. Algorithm 2 is the main recursive loop that makes calls to a sub-process. Essentially, Algorithm 2 ensures that each possible sub-tree of **Node a** is being compared with each possible sub-tree of **Node b**. The sub-process, Algorithm 3, performs the actual comparison to see if the sub-tree of **Node a** is equal to any sub-trees of **Node b**. Algorithm 2 is simply just passing Algorithm 3 a new root in **Node a** which Algorithm 3 can use to compare the sub-tree to the sub-trees of **Node b**.

Algorithm 2: Tree Comparison

Data: Node a, Node b, Double threshold
Result: Boolean
if *a is nil or b is nil* **then**
  Return False;
end
if *subTreesEquivalent(a, b, threshold)* **then**
  Return True;
end
Return **TreeComparison**(a.argOne, b, threshold) ∨ **TreeComparison**(a.argTwo, b, threshold);

Algorithm 3 contains a few subtleties that we think are worth noting. First, we note that the first two if-statements, while seemingly the same, handle two distinct cases. The first if-statement determines if both trees have reached a nil child of a leaf node at the same time. Since the trees are both being traversed in the same order, then if they reach a nil child at the same time, it must be that they have reached the same nil child. In the second if-statement, we are checking to see if one tree has reached a nil child, but another tree hasn't. This would mean the trees have distinct structures, and so it is impossible for them to be the same.

Note also how the recursive calls are moving both the pointers of **Node a** and **Node b**: we are traversing through the trees simultaneously and checking to see if their nodes are equal. The concept of equality warrants its own section, since this is where 'subjectivity', if we may use that term, plays a role in our algorithm.

Algorithm 3: Sub-Tree Comparison

Data: Node a, Node b, Double threshold
Result: Boolean
NodesAreEqual ⟶ False;
if *a is nil and b is nil* **then**
  Return True;
end
if *a is nil or b is nil* **then**
  Return False;
end
if *a is Entity and b is Entity* **then**
  NodesAreEqual = findSimilarityScore(a, b) ≥ threshold
end
else
  NodesAreEqual = areEqual(a, b)
end
Return NodesAreEqual ∧ **SubTreeComparison**(a.argOne, b.argOne, threshold) ∧
SubTreeComparison(a.argTwo, b.argTwo, threshold);

2.4 Node Similarity Algorithm

In a sense, our algorithm is incomplete since we believe that more node similarity measures should be incorporated into our code. For the time being, we use two canonical string similarity algorithms to score the text of our leaf nodes. As you may have noticed in Figure 1(b), the matched sub-trees are not exactly identical. One contains the word 'strong' to describe a cold, the other contains the word 'severe'. Given the context, these are synonyms, and should be treated as equal. Yet, lexicographically, they are different strings. So, how do we encode flexibility into our algorithm (without machine learning) so that it treats lexicographically different strings (with similar semantic meaning) as the same.

We propose a 'bag of algorithms' approach. Here, we calculate several similarity scores between two strings using a variety of different canonical string similarity scoring algorithms, and then aggregate the scores into a single total score that can be measured against a user-defined threshold. In this way, we are encoding for flexibility on two ends: on our end, and on the user's end. On our end, we create a weighted 'bag of algorithms' aggregate score that accounts for similarity scores across a multitude of string matching algorithms. On the user end, we yield them the flexibility of determining a threshold. The higher the threshold, the stricter the algorithm (the user does not want many false positives). The lower the threshold, the more lenient the algorithm (the opportunity cost of inadvertently neglecting an eligible participant is high for this user).

We omit specific algorithmic details of our similarity algorithms here since they are largely based on canonical string matching algorithms in the literature. At the time of writing, the string matching algorithms used are: N-gram, and Sørensen-Dice. There is a multiplier (extra weight) on the latter since it considers entire words (we are interested in presence or absence of key words), rather than N-grams (we are less interested in character sequences). We encourage readers to take a look at our GitHub repository (linked above) for more information on these algorithms.

3 Approach 2: Named Entity Recognition

Adonis changes the *article*'s front page to make a better first-impression. The title is no longer centred nor justified, and in the two-column layout, it occupies only one column. Moreover, to give the title more prominence, the template shrinks secondary information and moves some of it to the bottom of the page. The template thus splits the front-matter into two parts, the main and secondary details.

4 Conclusion

I designed *Adonis* to be as simple to use as possible. The optional commands, for example, mean that you do not have to define everything at once; you can simply start writing. To make the template easier to use, *Adonis* also comes with a separate file, `quickstart.tex`, without text, commented-out commands and space to write.

I hope that you find this template to elevate both form and function, and that it proves it possible for the two to co-exist. If you find any issues in *Adonis*, or if you have suggestions to make it better, you can reach out to me at the email on the first page, or by opening an issue on the template's repository [**?**].

References

[1] Kury, F., Butler, A., Yuan, C. et al. Chia, a large annotated corpus of clinical trial eligibility criteria. Sci Data 7, 281 (2020). https://doi.org/10.1038/s41597-020-00620-0

[2] Meystre, S.M., Heider, P.M., Cates, A. et al. Piloting an automated clinical trial eligibility surveillance and provider alert system based on artificial intelligence and standard data models. BMC Med Res Methodol 23, 88 (2023). https://doi.org/10.1186/s12874-023-01916-6

[3] Meystre, S. M., Heider, P. M., Kim, Y., Aruch, D. B., & Britten, C. D. (2019). Automatic trial eligibility surveillance based on unstructured clinical data. International journal of medical informatics, 129, 13–19. https://doi.org/10.1016/j.ijmedinf.2019.05.018

[4] Zeng, K., Xu, Y., Lin, G. et al. Automated classification of clinical trial eligibility criteria text based on ensemble learning and metric learning. BMC Med Inform Decis Mak 21 (Suppl 2), 129 (2021). https://doi.org/10.1186/s12911-021-01492-z