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).

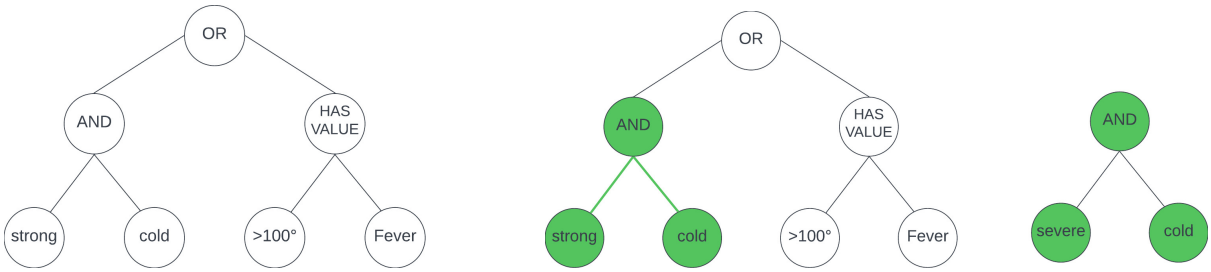
Algorithm 1 outlines the general steps required to create a tree-like Map from a text file. Note that we must use a Map since we are essentially over-writing Entity nodes if they have been modified by some sort of relation. For example, if on line 1 of the text file we encounter the entity T1: 'Flu', 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

This approach involved building a machine learning model using NLP. We leveraged the SpaCy API for Python to perform named entity recognition on unstructured clinical trial eligibility criteria and patient health data. Algorithm development in this stage can be categorized into, roughly, three stages. These are: Data pre-processing and Training data preparation, Model training, and Patient matching and evaluation.

3.1 Data Pre-processing

Our code leverages a large JSON file, which is a consolidation of all of the clinical trials from the Chia data-set. We parsed through each file individually, and formatted and aggregated each one into a structured JSON file. We found that SpaCy is picky about the way it consumes its data. For example, given that certain text fields began/ended with white space or non-alphanumeric characters, we experienced a multitude of errors during training. We had to pay very close attention to cleaning the data at this stage, and so in our GitHub we include several useful data cleaning functions that future researchers can leverage when cleaning up their CT and patient data for SpaCy. The data-set includes roughly 1000 entries for clinical trial eligibility. We set aside 50 (we did not include these in the training set) in order to test our model after training. We acknowledge that 50 is a small amount for testing, but we wanted our model to have as much input as possible (since we also think 1000 entries to be a small number when it comes to training data).

3.2 Model Training

We fine-tuned SpaCy's `en_core_web_lg model`, a large English multi-task CNN trained on OntoNotes, a corpus for text processing. We performed annotation parsing (extracting text and a corresponding

annotation) on our training data. Annotation parsing is critical for named entity recognition, which is identifying specific key information and labeling it in raw/disorganized text. We then initialize hyper-parameters for our model using a base configuration (see GitHub) and use this to begin training the NER model.

The model training had to be cut short. Due to the fact that the training was done on a local machine (an old Macbook Air), each training iteration took anywhere from 5-10 minutes. This meant that waiting for 7 iterations (as we did for this case) took roughly 1 hour in total. Thus, we believe our model has the potential to have higher levels of accuracy, and we discuss further in subsequent sections.

3.3 Patient matching and evaluation

Our final model reads the information from a clinical trial (one it hasn't seen before) to identify relevant eligibility information it may hold. Our goal then was to use this parsed information to identify the ideal patients we would want to match to a given trial. Thus, we tested our model by creating ideal patient matches for each trial in our testing dataset (the 50 values set aside from the training data). We iterated over each of the test trials, generating labels from the trials' descriptions using our model. Finally, we iterated over each patient to determine which patient was the best match for a given trial using our model's labels and the patient's data. That is, we essentially looked for an optimal matching between a trial's labels and a patient's labels. For each trial, we ranked the compatibility between our list of patients and the trial, and simply matched our trial to the patient with the highest 'compatibility score'.

Algorithm 4: Patient-Trial Compatibility

```
Data: List test_data, List patient_data
Result: List Assignments
Assignments ⟶ [];
for trial in test_data do
    List<(Text, Label)> labels = Perform_NER(trial);
    Max ⟶ 0.0;
    Max_Index ⟶ ∞;
    for index in range(patient_data) do
        Score = Generate_Score(Trial, Patient);
        if Score > Max then
            Max_Index = index;
            Max = score ;
        end
    end
    Assignments.add((Trial ⟶ Max_Index));
end
```

Algorithm 4 describes the compatibility algorithm we perform. Essentially, we get labeled text of each clinical trial's criteria. We create an 'ideal' patient for each of the trials (an ideal patient is just a set of all the trial's text criteria with their associated labels, let's call this set **P**). We then run our NER model on that same clinical trial's criteria. For each patient, we compare the labeled data generated by the NER (we'll call this set **T**) to the set of the ideal patient's labels. Note that, for a given trial, $\mathbf{T} \subseteq \mathbf{P}$. In practice,

this might not be true since **T** might contain mislabeled data. But, the intuition behind it is worth noting. We calculate a score (just checking to see the intersection of labels, $\mathbf{T} \cap \mathbf{P}$, between NER trial labels and ideal patient labels), and match the patient that has the largest intersection of labels with the trial. The scoring algorithm is not quite a set intersection (there is a scoring methodology), but in theory, this is what is being done.

Notably, the model was able to identify the ideal patient for each clinical trial in the test dataset with 90% accuracy.

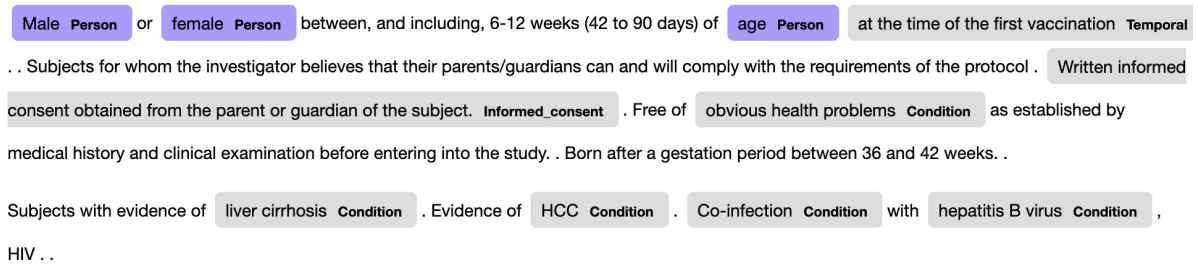


Figure 2: Clinical Trial Eligibility Criteria Labeled by Our Team's NER Model

4 Comparison of Approaches

4.1 Accuracy

The classical semantic tree-matching algorithm performed well overall, though it was not as robustly tested as the NER algorithm was. To test the tree-matching algorithm, we selected a very small set of trials from the Chia dataset. Of these trials, we borrowed a sub-set of the criteria for each trial in order to create a corresponding pseudo-patient. That is, for a trial with a set of criteria **X**, we selected a set of criteria **Y** such that $\mathbf{Y} \subseteq \mathbf{X}$, and where **Y** represents the patient's health records.

We tested a total of 42 patient-CT matching combinations. Of those 42, we labeled 7 as eligible matches, and 35 as non-eligible matches. The reason for the skew towards non-eligible matches is simply because it was a tedious process to re-write patient sentences that had the same semantics and logical structure as their trial counterparts. The algorithm achieved an accuracy of 95.2%. While this is an impressive figure at first glance, it is important to remember that most of our trees were not meant to be matched (only 7 eligible matches existed). Think of a algorithm that, input agnostic, simply spits out False. This algorithm would achieve a roughly 83% accuracy rate. So, we rely less on this figure as a gauge on performance, and instead look at the accuracy on the sub-set (size 7) of test data that we wanted the algorithm to predict a correct eligibility. It was able to accurately determine that the patient is in fact eligible for the trial 5 out of the 7 times, or a 71.4% accuracy. We suspect that this is more telling of the matching algorithm's ability to match patients to trials.

4.2 Efficiency

For both algorithms, our inputs are the same. We take a patient health record with M lines, and clinical trial eligibility criteria with K lines.

Our semantic parser traverses once linearly through both records to create the tree. This is $O(M+N)$ time. We know that our patient tree contains M nodes and our trial tree contains N nodes. We check N sub-trees of our trial tree against our patient tree. Thus, our tree traversal makes $min(M,N)$ comparisons, N times. The tree-matching algorithm is then $O(N \cdot min(M,N))$. In practice, the patient's health record will be vastly larger than the eligibility criteria: $M >> N$. So, we would actually be checking the sub-trees

of M, and our realistic complexity would be: $O(M \cdot N) + O(M + N) = O(M \cdot N) = O(M)$ for sufficiently large M (we'll imagine $10,000$ lines of EHR), and treating N as a constant (on average, 10-15 lines of criteria).

These calculations would need to be performed up-front. On the other hand, the machine-learning model would take a long time to train ($\sim$ 1 hour on CPU with $1,000$ trials), but would be able to perform equally or better than the semantic parser up-front.

4.3 Limitations

The semantic-based parser in a sense is "over-fitting" to the logic of the sentences it builds on. That is, while we have encoded flexibility at the node level, there is no flexibility at the tree structure level. Think of two sentences: "The cat ran over the hill", "Over the mountain, the cat ran". At the node level, we can encode flexibility through our string similarity algorithms: "mound" is similar to "mountain", both lexicographically and in meaning. Yet, while both sentences mean the same thing, there is a chance their tree structures might not be identical simply because of the way the individual sentences are formatted. Now, imagine an eligibility criteria with dozens of sentences. Tree structures might be reflections of each other, or simply have a few different children here and there, and our algorithm would not be able to accommodate for these differences. Encoding tree flexibility via more complex tree-search methods would be a step towards a potent and adaptable semantic parser.

To our delight, the NER model performed exceptionally well at matching trials to their 'ideal' patients. It is of note, though, that the actual similarity scores between patient NER labels and trial NER labels were quite low (averaging at around 20-30% similarity). Our testing data, while diverse, was not many, and we suspect that increasing our testing data would introduce greater error into our matching performance. The reason for this is that likely there will begin to arise a great many patient-trial scores all with low scores of an equal degree. So, matching will begin to become arbitrary. We believe, though, that this is simply due to a lack of training data with which to train our model on. The Chia data-set comes with $1,000$ labeled trials. We think this is low for an NLP algorithm. We include in our GitHub a data scraper that scrapes data from the ClinicalTrials.gov site, which lists over $50,000$ trials along with their eligibility criteria. We leave this data scraper for future researchers (armed with greater computing capacity than us) to train our model on this data.

5 Takeaways

In general, we believe that the NER algorithm has more potential to be used at scale when compared to the semantic parser. Eligibility criteria for a trial is really just a set of inclusion and exclusion criteria that a patient has to meet. As a result, we think that intuitively, matching algorithms should be focused on the absence and presence of key terms that are telling of whether the patient is able to partake in the trial. Remember that as part of our societal discussion, we emphasized the fact that when it comes to trial matching, it is best to err on the side of false positives. That is, the opportunity cost of inadvertently neglecting an eligible patient is high. This is not only because generally not many people nation-wide are willing to participate in CTs, but also because diversity is key in a trial, and so having all of your options on the proverbial table is crucial to achieve a diverse representation of participants. Thus, while the semantic parser focuses a lot on sentence-level logic, we believe that simply identifying the absence and presence of key terms is telling enough of whether a patient is eligible for the trial or not.

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