Identification of Drug-Drug Interactions

Zofia Tarant, Marcel Pons

June 4, 2021

This report encompasses the second substak of the SemEval 2013 DDI Extraction competition: extraction of drug-drug interactions. The objective of this subtask is to extract and classify drug-drug interactions[1] in biomedical texts from the DrugBank database and MedLine abstracts on the subject of drug-drug interactions (DDI corpus). Existing DDIs can be classified into four groups, according the relationship between the drugs involved:

- *mechanism*: described by their pharmacokinetic mechanism[2].
- *effect*: described by an effect or a pharmacodynamic mechanism[3].
- *advise*: used when a recommendation or advice regarding a drug interaction is given.
- *int*: when a DDI appears in the text without providing any additional information.

Two approaches are followed in order to perform the task: a rule-based and a Machine Learning approach. The first approach will serve as a baseline to compare with the other more complex models, such as using Machine Learning techniques. For both approaches the assumption that the entities (i.e. drugs) are already detected is followed, using the ground truth to extract them.

Both implementations consist in writing one or more python programs that parse all the XML files of the DDI corpus and perform a relation extraction task which consists on deciding whether a sentence is expressing a DDI between two drugs, and if so determines its type between the aforementioned interaction relationships. The performance of the models is evaluated by means of an evaluate (`evaluator.pyc`) script provided by the instructors. The evaluator compares the results with a ground truth and returns classification metrics (precision, recall , *F1*, etc.) for each type of interaction. Besides, a microaverage and a macroaverage are also provided. The macroaverage is the metric used to evaluate the rules implemented and the final performance of the models.

Standford CoreNLP

In both models, the Standford CoreNLP dependency parser will be used to tokenize, get the part-of-speech (PoS) tags and dependency trees of the sentences. The parser does not obtain the span offsets of each token, therefore they are manually extracted and included in the output of the parser.

[1]An interaction between two drugs occurs when one drug influences the level or activity of another drug.
[2]The process by which drugs are absorbed, distributed, metabolised and extracted is affected.
[3]The effects of one drug are changed by the presence of another drug at its site of action

1 Rule-based DDI

In this model, simple heuristic rules are implemented to carry out the identification and classification of drug-drug interactions, that is, a baseline model is built that will be compared with future models.

The final program consists on 2 functions: `analyze()` and `check_interaction()`.

Analyze

The `analyze()` function consists on executing the `CoreNLP` to obtain the tokens, tags, and dependency tree (a `DependencyGraph` from the `nltk` module). Once the tree is obtained, it adds the start and end offsets to each node.

Check interactions

The `check_interactions()` is the function responsible for identifying whether two entities in the same sentence present interaction between them. If there is interaction, it is classified among the four types based on simple *if-then-else* rules. With the purpose of finding rules that are right in most of the cases visual inspection and simple statistics are performed on the train dataset from which, by means of parsing all its XML files and extracting characteristics from the trees, a dataframe is created. The dataframe contains the type of interaction, the entities that are involved and their heads, with the corresponding PoS tag and the relation between them. It also contains boolean columns specifying whether the entities are under the same verb, under the same word, one entity under the other (and viceversa) and the entities being the subject or direct object of the verb[4]. Moreover, the shortest path between the entities is obtained by means of first, converting the tree into a NetworkX `MultiDiGraph`, and then using the method `nx_graph().to_undirected()` of the NetworkX API.

Figure 1 illustrates the number of occurrences of each type of interaction. As it can be observed, the *effect* is the most frequent type written in the biomedical texts, followed by *mechanism*. It is important to acknowledge this fact because the classification scores on these types will have a much greater impact on overall performance than those achieved on *advise* and *int*, which have few instances.

[4]Clue words in the sentence, between both entities and outside the entities were also analysed, however we considered that by analyzing dependencies more insights could be obtained.

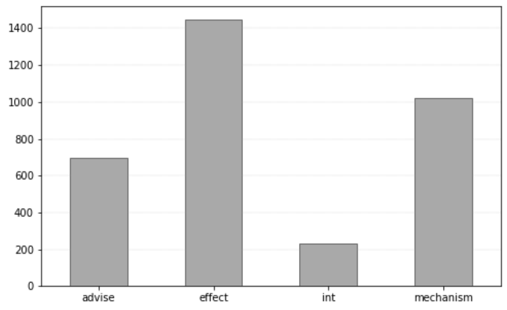


Figure 1: Nº interaction types

On the other hand, four dataframes are obtained, each one with a type of interaction. Visual inspections regarding the part-of-speech tag were carried out but no outstanding conclusions were extracted due to similar results among all the types of interactions. The same conclusions were reached regarding the PoS tags of the heads of the entities. However, when inspecting which were the most frequent heads for either entity 1 and entity 2, different results were obtained between interaction types that could help to build some rules (Annex Figure 8).

By analyzing the relationships between the entities in the dependency tree, it was observed that the *advise* interaction presented both entities under the same verb more frequently than the other types. With regard to the other evaluated relationships between entities, no significant conclusions were obtained (relative frequencies are provided in Annex Table 1).

To find the shortest path, `networkx.shortest_path()` was used. The lemmatized words appearing in the shortest paths between two entities were extracted and counted. Analysis showed that for each type of interaction, there were words that stood out and seemed to appear very frequently only for these types, but weren't present for other interactions. The most frequent words appearing in shortest paths can be found in Annex Table 2.

With these analysis, we found several characteristics, for example:

- if *effect* is in the path between e1 and e2, the interaction is effect
- if *patient* appears in the shortest path between e1 and e2, the interaction is *advise*
- if e1 is under the word *response* and e2 is under *man, alcohol* or *steroid*, the interaction is *effect*
- if e2 is under words such as *metabolism, concentration, clearance, level, absorption, dose*, the interaction is *mechanism*

From the insights that are extracted from the inspection, the different rules of the classifier are established. It is worthy to mention that the order in which the rules are implemented is important, since they are executed sequentially.

To find the node corresponding to an entity in the tree, the program iterates through the list of nodes and returns the first one which represents any word included in an entity. So effectively, when it comes to *two-word entities*, only one word is considered. This approach is not very precise, but it is close enough for a rule-based heuristic approach.

Once all the rules have been evaluated the final `check_interactions()` is implemented, which given a dependency graph object with all sentence information, the list of all entities in the sentence and the *ids* of the two entities to be checked returns the type of interaction between `e1` and `e2` expressed by the sentence, or `None` if no interaction is described.

```
def check_interaction(analysis, entities, e1, e2):

    tree = analysis.tree()

    entity1 = entities[e1]
    entity2 = entities[e2]

    e1_node = find_entity_in_tree(entity1, analysis)
    e2_node = find_entity_in_tree(entity2, analysis)

    h_e1 = e1_node['head'] if e1_node else None
    head_e1 = analysis.nodes[h_e1]['lemma'] if e1_node else None

    h_e2 = e2_node['head'] if e2_node else None
    head_e2 = analysis.nodes[h_e2]['lemma'] if e2_node else None

    nxgraph = analysis.nx_graph().to_undirected()

    shortest_path = networkx.shortest_path(nxgraph, e1_node['address'], e2_node['address']) if␣
↪(e1_node and e2_node) else []
    shortest_path = [analysis.nodes[x]['lemma'] for x in shortest_path]

    h2_effect = ['man', 'alcohol', 'steroid']
    h1_effect = ['response', 'effect', 'enhance', 'diminish']
    h1_mechanism = ['concentration', 'presence', 'dose', 'absorption', 'interfere']
    h_int = ['interact', 'interaction']
    h1_advise = ['co-administration', 'take', 'coadminister', 'treatment', 'therapy', 'tell']
    h2_effect_2 = ['effect', 'alcohol', 'action','use', 'combination', 'inhibitor']
    h2_mechanism = ['metabolism', 'concentration', 'clearance', 'level', 'absorption', 'dose']

    # --- RULES ---
    if 'effect' in shortest_path:
        return 'effect'
    if 'increase' in shortest_path or 'decrease' in shortest_path or 'concentration' in␣
↪shortest_path:
        return 'mechanism'
    if 'interact' in shortest_path:
        return 'int'
    if 'patient' in shortest_path:
        return 'advise'
    if head_e1  == 'response' and head_e2 in h2_effect:
        return 'effect'
    if head_e1 == 'administer' and head_e2 == 'administer':
        return 'advise'
    if head_e1 in h1_effect:
        return 'effect'
    if head_e1 in h1_mechanism:
        return 'mechanism'
    if head_e1 in h1_int:
        return 'int'
    if head_e1 in h1_advise:
        return 'advise'
    if head_e2 in h2_effect_2:
        return 'effect'
    if head_e2 in h2_mechanism:
        return 'mechanism'
    if head_e2 in h_int:
        return 'int'

    return None
```

Results of the Ruled Based Model

From these functions and the rules established, the program `baseline-DDI` is developed, which takes as arguments the path where the XML files are located and the name of the output file. The output of the program recreates the hidden gold annotation {`IdSentence|id_e1|id_e2|ddi_type`} which will be used to evaluate the performance.

The development dataset has been used in order to assess which rules are useful for the classification and which are not. In general, the macroaverage F-measure is 31%.

```
                    tp      fp       fn     #pred    #exp     P        R        F1
------------------------------------------------------------------------------
advise              37     190      101      227      138    16.3%    26.8%    20.3%
effect              79     226      236      305      315    25.9%    25.1%    25.5%
int                 31      73        4      104       35    29.8%    88.6%    44.6%
mechanism          106     257      158      363      264    29.2%    40.2%    33.8%
------------------------------------------------------------------------------
M.avg               -       -        -        -        -     25.3%    45.2%    31.0%
------------------------------------------------------------------------------
m.avg              253     746      499      999      752    25.3%    33.6%    28.9%
m.avg(no class)    370     629      382      999      752    37.0%    49.2%    42.3%
```

Figure 2: Results of rule-based model for devel data

With the test dataset, the macroaverage F-measure is 27.5%.

```
                  tp      fp       fn    #pred    #exp    P        R        F1
------------------------------------------------------------------------------
advise             56     167      156     223     212    25.1%    26.4%    25.7%
effect            110     370      173     480     283    22.9%    38.9%    28.8%
int                 7      55       11      62      18    11.3%    38.9%    17.5%
mechanism         158     337      179     495     337    31.9%    46.9%    38.0%
------------------------------------------------------------------------------
M.avg               -       -        -       -       -    22.8%    37.8%    27.5%
------------------------------------------------------------------------------
m.avg             331     929      519    1260     850    26.3%    38.9%    31.4%
m.avg(no class)   419     841      431    1260     850    33.3%    49.3%    39.7%
```

Figure 3: Results of rule-based model for test data

Rules that were tried and discarded

Different sets of words under which entities could appear were tried. For example, it was observed that including *action* and *agent* in the possible words *above* entity1 decreased the f-score by 2.7pp. Discarding the rule *if e1 and e2 are under the same word, then return mechanism or int* improved the f-score by 4.5pp. The rule *if e1 is under e2, then the interaction is effect* was removed as well, because it resulted in too many false positives.

Analysis of *whether an entity is the subject of one verb and the other is inside the direct object of the same verb* did not allow to form any useful rules because there were too few occurrences. The analysis of the entities (words) of the interactions showed that there couldn't be constructed any rules regarding the structure of the words themselves. The inspection of the words outside the interval between e1 and e2 was unsuccessful too.

2 Machine Learning DDI

For the machine learning approach, a Maximum Entropy model is implemented in order to recognize and classify interactions between drugs in the biomedical texts. It is an exponential probabilistic classifier, which chooses the model with the largest entropy.

Since in NLP applications like this one features are related to appearing words, configurations of the dependency tree, relationships between entities, etc., a feature extraction process has to be previously done.

Feature Extractor

The feature extractor takes as arguments the sentence ID, its dependency tree, the entities corresponding to drugs that appear on it, and the nodes of two entities conforming an interaction pair. The extractor calls several functions which are provided in the annex for further reading.

First, the position of both entity nodes in the dependency tree of each sentence is found by means of the `find_entity_in_tree()` function. Moreover, the position of the head nodes of the entities is also obtained with `find_head()` and the shortest path between entities is retrieved using `networkx.shortest_path()`.

Extracted features are related to the information about the head of the entities in the tree, such as lemma and PoS tag, whether the head appears in a list of verbs for each interaction type[5], whether they share the head or whether the entities themselves are related in a parental manner. Additional features include the types (*drug, drug-n, brand, group*) of both entities. Moreover, the shortest path followed to get from one entity to the other is extracted as a feature, which represents the dependencies between the nodes (by means of the `traverse_path()` function). Clue verbs appearing between the entities are also extracted (with `find_clue_verbs()` function).

Besides the aforementioned features that seek for relationships between the entities, we considered other features involving characteristics of the sentence, for instance, finding clue words before and after the pair (`find_words_outside_path()`) and finding other entities appearing in the same sentence (`find_other_entities()`).

```
def extract_features(tree, entities, e1, e2, sid):

    e1_node = find_entity_in_tree(e1, entities, tree)
    e2_node = find_entity_in_tree(e2, entities, tree)

    e1_head = find_head(tree, e1_node) if e1_node else None
    e2_head = find_head(tree, e2_node) if e2_node else None

    h1_lemma = e1_head['lemma'] if e1_node else None
    h2_lemma = e2_head['lemma'] if e2_node else None

    tag_head_e1 = e1_head['tag'] if e1_head else None
    tag_head_e2 = e2_head['tag'] if e2_head else None

    nxgraph = tree.nx_graph().to_undirected()
    shortest_path = networkx.shortest_path(nxgraph, e1_node['address'], e2_node['address']) if␣
 ↪(e1_node and e2_node) else []
    path_with_word, path_with_tag = traverse_path(shortest_path, tree)
    find_clue_verbs(shortest_path, tree)

    # --- FEATURES ---
    features = ['h1_lemma=%s' %h1_lemma,
                'h2_lemma=%s' %h2_lemma,
                'h1_tag=%s' %tag_head_e1,
                'h2_tag=%s' %tag_head_e2,
                'tagpath=%s' % path_with_tag,
                'neg_words_s=%s' %count_neg_s,
                'e1_type=%s' % entities[e1]['type'],
                'e2_type=%s' % entities[e2]['type'],
                ] + find_clue_verbs(shortest_path, tree)

    if (e1_head and e2_head):
        if h1_lemma == h2_lemma:
```

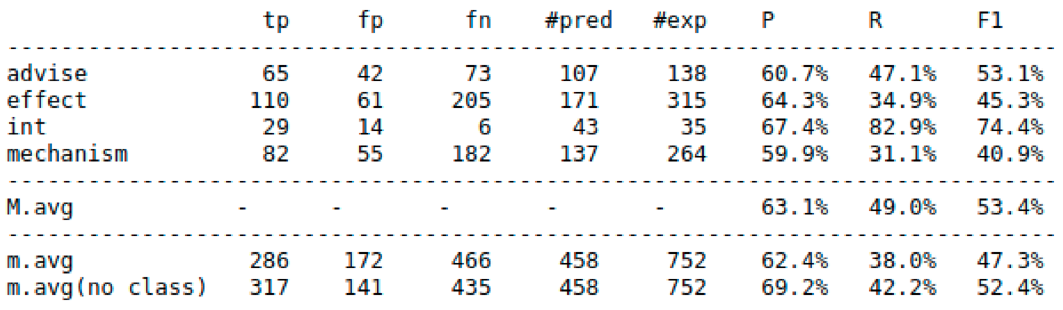
[5]The list of verbs for each type of interaction is determined through the same exploratory data analysis done for the rule based model.

```
            features.append('under_same=True')
            if tag_head_e1[0].lower() == 'v':
                features.append('under_same_verb=True')
            else:
                features.append('under_same_verb=False')
        else:
            features.append('under_same=False')
            features.append('under_same_verb=False')

        if h1_lemma == e2_node['lemma']:
            features.append('1under2=True')
        else:
            features.append('1under2=False')

        if h2_lemma == e1_node['lemma']:
            features.append('2under1=True')
        else:
            features.append('2under1=False')
    else:
        None

    words_before, words_after = find_words_outside_path(shortest_path, tree)
    for word in words_before:
        features.append(f'lemmabefore={word}')
        features.append(f'tagbefore={pos_tag(word)[0][1]}')
    for word in words_after:
        features.append(f'lemmaafter={word}')
        features.append(f'tagafter={pos_tag(word)[0][1]}')

    other_entities = find_other_entities(e1, e2, sid, entities, tree)
    for e_node, e_type in other_entities:
        features.append('typeother=%s' % e_type)

    return features
```

Maximum Entropy Modeling

Once the features are extracted, two programs are developed: `megam-learner.py`, which creates the trained model, and `megam-classifier.py`, which identifies and classifies interactions using that model. The programs use the `MaxentClassifier` from the `nltk` module and the `MEGAM` optimization package to carry out the classification task.

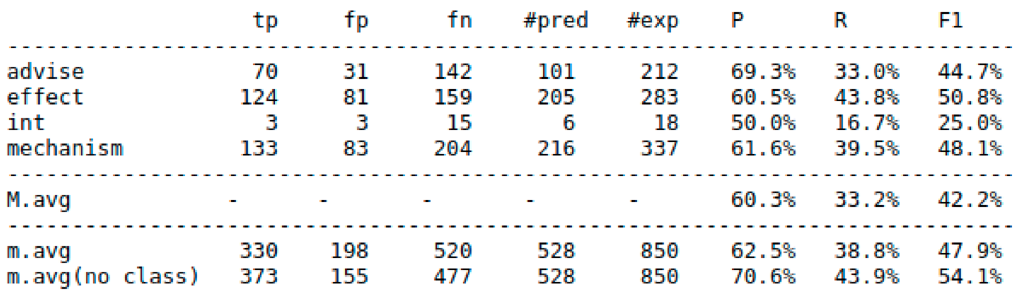
For the purpose of passing the classification results through the `evaluator.pyc`, the classifier returns a text file with the desired structure `{sid|e1|e2|prediction}`. The scores for the devel dataset are used to assess the performance of the chosen features, helping on the decision about whether they should be kept or removed from the extractor. The best combination of features resulted on a 53.4% macroaverage on the devel dataset.



```
                    tp     fp      fn    #pred   #exp     P        R        F1
------------------------------------------------------------------------------
advise              65     42      73     107     138    60.7%    47.1%    53.1%
effect             110     61     205     171     315    64.3%    34.9%    45.3%
int                 29     14       6      43      35    67.4%    82.9%    74.4%
mechanism           82     55     182     137     264    59.9%    31.1%    40.9%
------------------------------------------------------------------------------
M.avg               -       -       -       -       -    63.1%    49.0%    53.4%
------------------------------------------------------------------------------
m.avg              286    172     466     458     752    62.4%    38.0%    47.3%
m.avg(no class)    317    141     435     458     752    69.2%    42.2%    52.4%
```

Figure 4: Results of the maximum entropy model for devel data

With regard to the test dataset, the macroaverage F-measure is 42.2%.



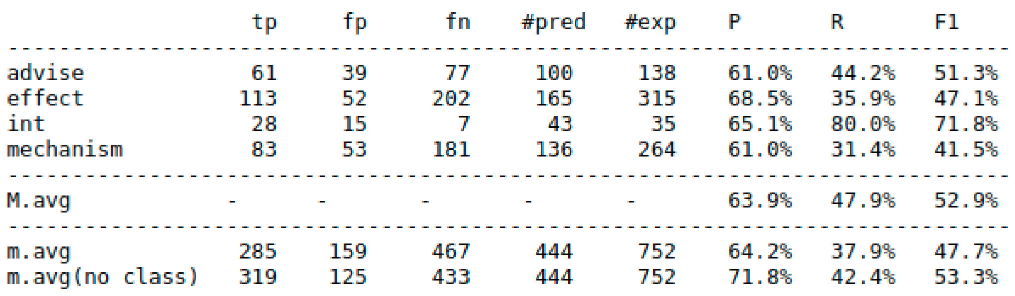
```
                    tp     fp      fn    #pred   #exp     P        R        F1
------------------------------------------------------------------------------
advise              70     31     142     101     212    69.3%    33.0%    44.7%
effect             124     81     159     205     283    60.5%    43.8%    50.8%
int                  3      3      15       6      18    50.0%    16.7%    25.0%
mechanism          133     83     204     216     337    61.6%    39.5%    48.1%
------------------------------------------------------------------------------
M.avg               -       -       -       -       -    60.3%    33.2%    42.2%
------------------------------------------------------------------------------
m.avg              330    198     520     528     850    62.5%    38.8%    47.9%
m.avg(no class)    373    155     477     528     850    70.6%    43.9%    54.1%
```

Figure 5: Results of the maximum entropy model for test data

Discarded models

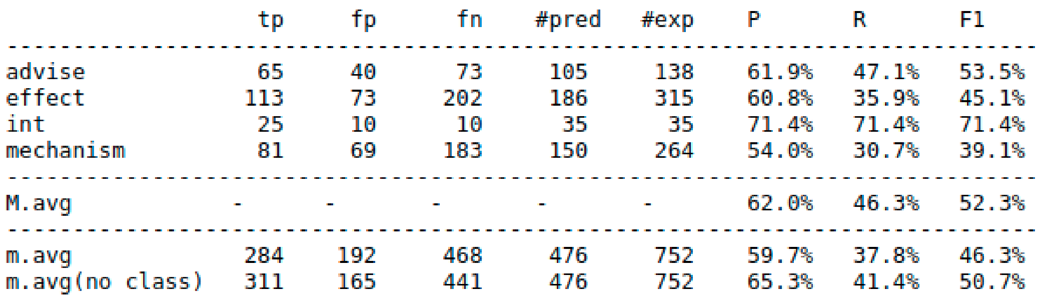
We tried different combinations of features. For instance, Figure 6 shows the metrics for the devel set where the shortest path was represented in a different way, with the lemmatized word at the top of the path instead of its tag. Figure 7 represents the result for the model where we tried creating separate features for the head of e1 and head of e2 representing the presence of clue words associated with drug types. Moreover, observed no improvement when checking for presence of words related to each type of interaction.

Removing "False" features, such as `under_same=False`, decreased the result by 0.8% and removing "stop words" from `lemma_in_between` worsened it by 2%. Also, we tried two features that counted the occurrence of negative words such as *not*, *neither*, *without*, *lack* in both the entire sentence and the shortest path, however no significant results were obtained.



```
                      tp      fp       fn     #pred    #exp     P        R        F1
------------------------------------------------------------------------------------
advise                61      39       77      100      138     61.0%    44.2%    51.3%
effect               113      52      202      165      315     68.5%    35.9%    47.1%
int                   28      15        7       43       35     65.1%    80.0%    71.8%
mechanism             83      53      181      136      264     61.0%    31.4%    41.5%
------------------------------------------------------------------------------------
M.avg                  -       -        -        -        -     63.9%    47.9%    52.9%
------------------------------------------------------------------------------------
m.avg                285     159      467      444      752     64.2%    37.9%    47.7%
m.avg(no class)      319     125      433      444      752     71.8%    42.4%    53.3%
```

Figure 6: Results of a discarded model for devel data



```
                      tp      fp       fn     #pred    #exp     P        R        F1
------------------------------------------------------------------------------------
advise                65      40       73      105      138     61.9%    47.1%    53.5%
effect               113      73      202      186      315     60.8%    35.9%    45.1%
int                   25      10       10       35       35     71.4%    71.4%    71.4%
mechanism             81      69      183      150      264     54.0%    30.7%    39.1%
------------------------------------------------------------------------------------
M.avg                  -       -        -        -        -     62.0%    46.3%    52.3%
------------------------------------------------------------------------------------
m.avg                284     192      468      476      752     59.7%    37.8%    46.3%
m.avg(no class)      311     165      441      476      752     65.3%    41.4%    50.7%
```

Figure 7: Results of a discarded model for devel data

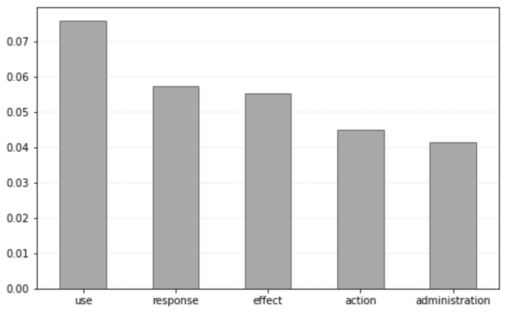
3 Conclusions

The exploratory analysis of the train dataset allowed us to distinguish several rules in the dependency trees of each type of interaction that allowed us to build a baseline solution. Mainly, some verbs were more characteristic of a type of drug-drug interaction, either in the shortest path between both entities or in the entity-head relation. Initially we thought that rules regarding part of speech tags and syntactic relations between entities and heads (like subject, direct object, etc) would be relevant to the final score. However, when we tried them on the devel dataset, no improvement was noticed. What is more, rules that focused on the direct relationship of the entities in the dependency tree, such as `e1_under_e2` and vice-versa, happened not to be useful either.

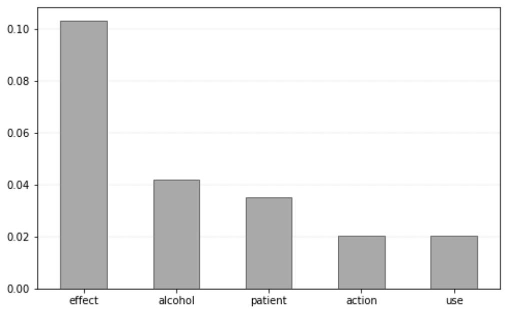
For the machine learning task, even though better results were achieved with respect to the baseline, we consider that our model did not behave as good as expected. We found that the features that carried the most information were related to the shortest path between entities, such as the dependencies between the words and the occurrence of certain clue verbs. We also discovered that features similar to the rules that we had used for the baseline solution – checking if the entities are under a word characteristic for each type of interaction – did not improve the result.

4 Annex

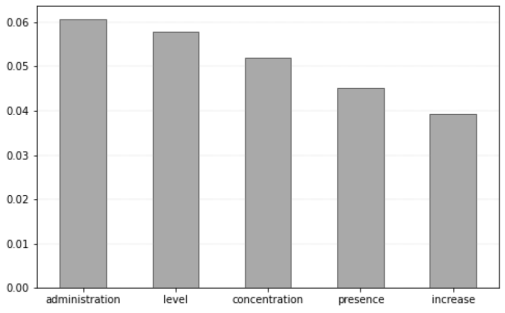
Rule Based Model



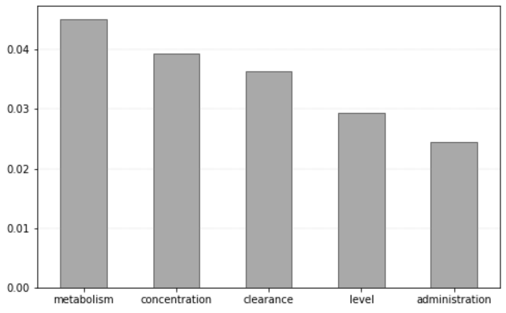
(a) Effect head_e1



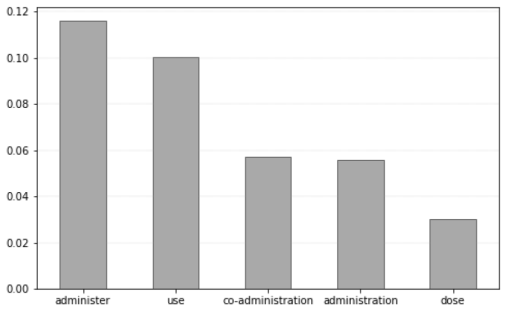
(b) Effect head_e2



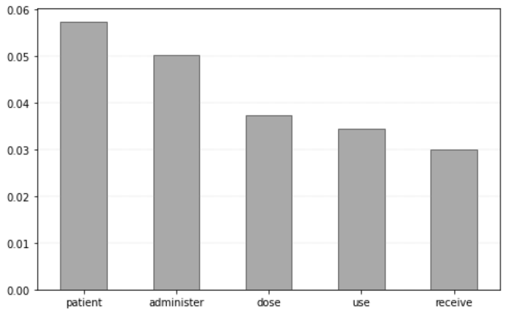
(c) Mechanism head_e1



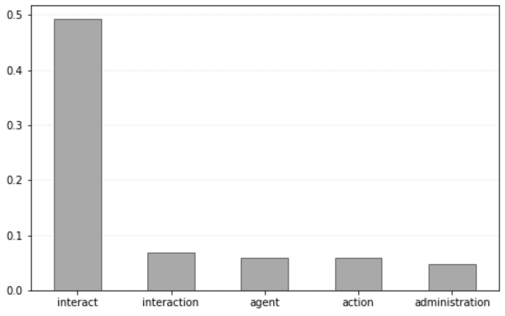
(d) Mechanism head_e2



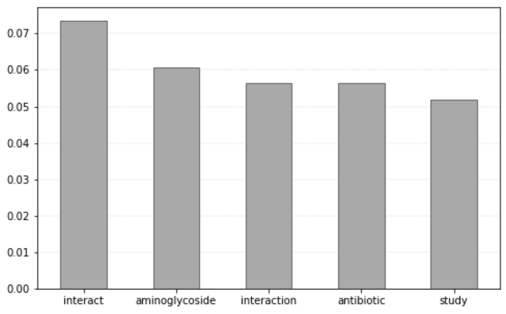
(e) Advise head_e1



(f) Advise head_e2



(g) Int head_e1



(h) Int head_e2

Figure 8: Most frequent head words on each interaction type, for entity 1 and entity 2

	Effect	Mechanism	Advise	Int
e1_under_e2	0.0034	0.0009	0.0043	0
e2_under_e1	0.0496	0.0352	0.0774	0.0519
under_same_word	0.0848	0.1362	0.1276	0.1212
under_same_verb	0	0.0774	0.1276	0

Table 1: Relative frequency of dependency configurations between entities.

effect		mechanism		advise		int	
word	count	word	count	word	count	word	count
effect	320	increase	219	patient	109	interact	123
drug	185	decrease	124	administer	106	antibiotic	72
agent	149	concentration	107	use	100	acetaminophen	52
action	143	level	92	ketoconazole	63	medication	45
use	135	phenytoin	8	inhibitor	59	Etonogestrel	44

Table 2: Top 5 most frequent words appearing in the shortest path between entity 1 and entity 2, represented as counts.

Machine Learning Model

```python
def find_entity_in_tree(eid, entities, tree):
    start_e1 = int(entities[eid]['offsets'][0])
    end_e1 = int(entities[eid]['offsets'][1].split(';')[0])

    for n in tree.nodes.items():
        node = n[1]
        if node['word'] and (node['start'] == start_e1 or node['end'] == end_e1):
            return node
```

```python
def find_head(tree, entity):
    for n in tree.nodes.items():
            node = n[1]
            if  node['address'] == entity['head']:
                return node
```

```python
def traverse_path(path, tree):
    if len(path) == 0:
        return None, None
    path_nodes = [tree.nodes[x] for x in path]
    str_path = ""
    # traverse from e1 up
    current_node = path_nodes[0]
    while (current_node['head'] in path):
        rel = current_node['rel']
        current_node = tree.nodes[current_node['head']]
        str_path += (rel + '<')
    tag_path = str_path + current_node['tag']
    str_path += current_node['lemma']
    # traverse from e2 up
    current_node = path_nodes[-1]
    while(current_node['head'] in path):
        rel = current_node['rel']
        current_node = tree.nodes[current_node['head']]
        str_path += ('>' + rel)
        tag_path += ('>' + rel)
```

```
def find_clue_verbs(path, tree):
    CLUE_VERBS = ['administer', 'enhance', 'interact', 'coadminister', 'increase', 'decrease']
    path_nodes = [tree.nodes[x]['lemma'] for x in path]
    feats = []
    for pn in path_nodes:
        if pn in CLUE_VERBS:
            feats.append('lemmainbetween=%s' % pn)
    return feats
```

```
def find_other_entities(eid1, eid2, sid, entities, tree):
    other_entities = [(entity['eid'], entity['type']) for _, entity in entities.items() if␣
 ↪entity['sid'] == sid and entity['eid'] not in [eid1, eid2]]
    return [(find_entity_in_tree(eid, entities, tree),e_type) for eid, e_type in other_entities]
```

```
def find_words_outside_path(path, tree):
    if len(path) < 1:
        return [], []
    words_before = []
    words_after = []
    nodes_before = [node[1] for node in tree.nodes.items()][:path[0]]
    nodes_after = [node[1] for node in tree.nodes.items()][path[-1]:]

    for node in nodes_before:
        if node['address'] not in path and node['lemma'] and node['lemma'] not in string.
 ↪punctuation and not node['lemma'].isdigit():
            words_before.append(node['lemma'])
    for node in nodes_after:
        if node['address'] not in path and node['lemma'] and node['lemma'] not in string.
 ↪punctuation and not node['lemma'].isdigit():
            words_after.append(node['lemma'])
    return words_before, words_after
```