DDI detection

AHLT | MAI

Ramon Mateo Navarro
Benet Manzanares Salor

15/05/2022

Introduction

In this report we describe the work performed for the first two laboratories of the AHLT subject. These practical assignments consist of the construction of a Detect drug-drug interactions (DDI) system using both a rule-based and machine learning DDI approaches. In the following, we detail and justify our decisions regarding the rules and features corresponding to these systems. Specific codes and results are also provided.
In the last section, we discuss the learning outcomes and reflections that emerged during this work.

Rule-based approach

For this laboratory, we had to define a rule-based approach for the DDI task. In particular, our task was to improve a predefined baseline improving the existing rules and adding new patterns. In the following subsections our improvements and new patterns are defined.

Improvement of existing patterns

The baseline code provided the **check_LCS_svo** and **check_wib** patterns, which used in cascade (only calling the latter if the former returns none) were capable of achieving a 30.6% of macro-average F1 score in the devel set. For improving these results, we employed the **explore.py** script on the training data. This script returns the probability of each found pattern input to belonging to an DDI or null class. Additionally, the number of instances that allowed to compute this probability is provided (e.g., *P(effect|enhance_VB) = 67/77 = 0.87*). On this basis, we checked the existing rules for both patterns and searched for better rules. Our criteria was mainly founded on the probability of a rule to be correct (that determines accuracy) and the number of instances (what determines recall and overfitting). In particular, we discarded rules with a probability close to 50% and those with less than 10 instances, being considered ambiguous and potentially causes of overfitting. Exceptions were made when an improvement in the recall of a certain class was desired and few rules exist for it or when the specific rule followed the common sense, being probably good for generalization.
In addition, for both patterns we controlled the case that the word to check is preceded by “not”, since the negation completely changes the implications of the rules. The resulting code for these modifications is depicted below, including mentions to added and ambiguous rules.

```
def check_LCS_svo(tree,tkE1,tkE2):
   if tkE1 is not None and tkE2 is not None:
      lcs = tree.get_LCS(tkE1,tkE2)

      #  If it's verb and not negated
      if tree.get_tag(lcs)[0:2] == "VB" and tree.get_word(lcs-1) != "not":
         path1 = tree.get_up_path(tkE1,lcs)
         path2 = tree.get_up_path(tkE2,lcs)
         func1 = tree.get_rel(path1[-1]) if path1 else None
         func2 = tree.get_rel(path2[-1]) if path2 else None

         if (func1=='nsubj' and func2=='obj') or (func1=='obj' and func2=='nsubj'):
            lemma = tree.get_lemma(lcs).lower()
            if lemma in ['diminish','augment','exhibit','experience','counteract',
            'potentiate','enhance','reduce','antagonize']: # reduce, antagonize are
ambiguous
               return 'effect'
            if lemma in ['impair','inhibit','displace','accelerate','bind','induce',
            'decrease','elevate','delay']:   # DECREASE, inhibit, displace, induce,
elevate, delay are ambiguous
               return 'mechanism'
            if lemma in ['exceed']:
               return 'advise'
            if lemma in ['suggest']:   # SUGGEST is ambiguous
               return 'int'

   return None


def check_wib(tree,tkE1,tkE2,entities,e1,e2):
   if tkE1 is not None and tkE2 is not None:
      # get actual start/end of both entities
      l1,r1 = entities[e1]['start'],entities[e1]['end']
      l2,r2 = entities[e2]['start'],entities[e2]['end']

      for t in range(tkE1+1,tkE2):
         # get token span
         l,r = tree.get_offset_span(t)
         # if the token is in between both entities and it's not negated
         if r1 < l and r < l2 and tree.get_word(t-1) != "not":
            lemma = tree.get_lemma(t).lower()
            if lemma in ['tendency','stimulate','regulate','prostate','modification',
            'augment','accentuate','exacerbate','proliferation','rarely','secondary']: #
proliferation, rarely, secondary added
               return 'effect'
            if lemma in ['react','faster','presumably','induction','substantially',
            'minimally','delay','induce','modest','induction']: # delay, induce, modest,
induction added | DELAY, INDUCE, substantially, minimally are ambiguous
               return 'mechanism'
            if lemma in ['exceed','extreme','cautiously','should']: # should added |
SHOULD is ambiguous
               return 'advise'
            if lemma in ['interact']:  # interact is ambiguous
               return 'int'

   return None
```

The following table shows how these changes significantly improved the results, achieving a 37.1% F1 score in the devel set.

	tp	fp	fn	#pred	#exp	P	R	F1	
advise	34	65	107	99	141	34.3%	24.1%	28.3%	
effect	43	22	269	65	312	66.2%	13.8%	22.8%	
int	19	3	9	22	28	86.4%	67.9%	76.0%	
mechanism	41		86	220	127	261	32.3%	15.7%	21.1%
M.avg	-	-	-	-	-	54.8%	30.4%	37.1%	
m.avg	137		176	605	313	742	43.8%	18.5%	26.0%
m.avg(no class)	170		143	572	313	742	54.3%	22.9%	32.2%

New patterns

For the creation of new patterns we aimed to manage these LCS-based rules that were ignored by the **check_LCS_svo** pattern due to being limited to verbs and a relation of nsubj<->obj between the LCS and the entities. Under this premise, we modified the **explore.py** script in different ways, what lead to new LCS-based patterns.

The first exploration considered both VB and NN part-of–speech for the LCS, all the relationships with the entities except the nsubj<->obj (since it's already considered by **check_LCS_svo**) and discarded the cases where the LCS was preceded by "not". The lemma of the LCS was returned by the exploration method. Using the results, we created **check_LCS_VB_NN** using the same paradigm based on probabilities and number of instances used for the baseline rules improvement (including the exceptions). On concept, this pattern should be placed after **check_LCS_svo**, contemplating the cases ignored by the former. The resulting code for this pattern is shown below.

```
def check_LCS_VB_NN(tree, tkE1, tkE2):
   if tkE1 is not None and tkE2 is not None:
      lcs = tree.get_LCS(tkE1,tkE2)

      # If it's verb or a noun and not negated
      if tree.get_tag(lcs)[0:2] in ["VB","NN"] and tree.get_word(lcs-1) != "not":
         lemma = tree.get_lemma(lcs).lower()
         if lemma in ['accentuate', 'counteract', 'augment', 'combine', 'diminish',
          'regulate', 'enhance', 'injection', 'potentiate']: # injection, potentiate are
ambiguous
            return 'effect'
         if lemma in ['impair', 'accelerate', 'react']:  # accelerate and react are
ambiguous
            return 'mechanism'
         if lemma in ['exceed', 'titrate', 'avoid', 'prescribe']:
            return 'advise'

   return None
```

The second exploration considered only VB part-of–speech for the LCS and the same for its parent. The lemma of the LCS join to the lemma of its parent was returned by the exploration method. Using the results, we created the **check_LCS_VB_parent** pattern, which aimed to consider very specific cases of LCS and parent pairs but that were usual (more than 70%)

and frequent (more than 15 instances) in the training set. This tried to disambiguate some case from the other patterns such as 'interact'. Due to its disambiguating aiming, this pattern probably needs to be the first called to be effective.

```
def check_LCS_VB_parent(tree,tkE1,tkE2):
   if tkE1 is not None and tkE2 is not None:
      lcs = tree.get_LCS(tkE1, tkE2)
      if tree.get_tag(lcs)[0:2] == "VB":
         parent = tree.get_parent(lcs)
         if parent is not None and tree.get_tag(parent)[0:2] == "VB":
            parent_and_lcs = tree.get_lemma(parent).lower() + "_" +
tree.get_lemma(lcs).lower()
            if parent_and_lcs in ['show_inhibit']:
               return 'mechanism'
            if parent_and_lcs in ['occur_administer', 'report_diminish',
'suggest_regulate']:
               return 'effect'
            if parent_and_lcs in ['recommend_use']:
               return 'advise'
            if parent_and_lcs in ['affect_interact']:
               return 'int'

   return None
```

We also tried other exploration approaches, for instance, we filtered the cases where the LCS is located between the entities or used the relations between a LCS and the entities for the creation of very specific rules. However, the resulting patterns negatively or very slightly affected the results and were discarded.

Experiments and results

We have tested different permutations and coalitions for the patterns in cascade defined in the **baseline-DDI.py** script, searching for the combination that provides the maximum F1 score. The best results were obtained with the combination: **check_LCS_VB_parent, check_LCS_svo**, **check_wib** and **check_LCS_VB_NN**. This followed the intuitions stated during the new patterns' development, which all the patterns improving the results. The most contributing new pattern is **check_LCS_VB_NN**, improving F1 from 37.1% to 39.0%. See the following table for details.

	tp	fp	fn	#pred	#exp	P	R	F1	
advise	35	64	106	99	141	35.4%	24.8%	29.2%	
effect	57	31	255	88	312	64.8%	18.3%	28.5%	
int	19	3	9	22	28	86.4%	67.9%	76.0%	
mechanism	43		82	218	125	261	34.4%	16.5%	22.3%
M.avg	-	-	-	-	-	55.2%	31.9%	39.0%	
m.avg	154		180	588	334	742	46.1%	20.8%	28.6%
m.avg(no class)	186		148	556	334	742	55.7%	25.1%	34.6%

Oppositely, the **check_LCS_VB_parent** pattern only offers a slight improvement, with a macro-average F1 of 39.3% instead of 39.0%. This is probably because the defined rules were too specific and not many instances activate them in the devel set.

Once the optimal patterns' combination was defined, we performed the final experiments on the devel and test sets, obtaining the following tables.

Final results on devel set:

	tp	fp	fn	#pred	#exp	P	R	F1	
advise	36	65	105	101	141	35.6%	25.5%	29.8%	
effect	60	37	252	97	312	61.9%	19.2%	29.3%	
int	19	3	9	22	28	86.4%	67.9%	76.0%	
mechanism	43		86	218	129	261	33.3%	16.5%	22.1%
M.avg	-	-	-	-	-	54.3%	32.3%	39.3%	
m.avg	158		191	584	349	742	45.3%	21.3%	29.0%
m.avg(no class)	191		158	551	349	742	54.7%	25.7%	35.0%

Final results on test set:

	tp	fp	fn	#pred	#exp	P	R	F1	
advise	70	127	139	197	209	35.5%	33.5%	34.5%	
effect	77	41	209	118	286	65.3%	26.9%	38.1%	
int	2	6	23	8	25	25.0%	8.0%	12.1%	
mechanism	41		53	299	94	340	43.6%	12.1%	18.9%
M.avg	-	-	-	-	-	42.4%	20.1%	25.9%	
m.avg	190		227	670	417	860	45.6%	22.1%	29.8%
m.avg(no class)	219		198	641	417	860	52.5%	25.5%	34.3%

As can be observed, the F1 score on the test set is significantly worst than on devel set. This is due to main differences for the “int” (interaction) class, which changes it’s F1 from 76% to 12%, with also a low 25% of precision and very low recall of 8%. On this basis, it seems that our approach overfits to the devel set. The rest of classes don’t suffer from this overfitting, with a similar or even better F1 score. Consequently, we consider that not enough nor general rules have been found for this class, since we have a reduced amount of instance variety for finding them. We expect the machine learning approach to be capable of detecting more complex relationships that allow to better detect this class (and the rest).

Machine learning approach

In the second laboratory, we had to implement the DDI detection system using a feature-based approach join to a machine learning algorithm. More concretely, we had to define new features in order to improve the F1 score of a predefined baseline.

Our approach

For developing our approach, we have iteratively tried different sets of features, maximizing the improvement for each set before advancing to the next. We also experimented with different settings for the ME training (e.g., -nobias or -repeat). Only a slight improvement was obtained by increasing the number of repetitions, which was finally set to 6 as a balance between training time and improvement. On this basis, the sets of features explored, variations for each one, best variation and improvement obtained are listed below. Explanations and reasonings are provided using italics font.

- **Features not between the entities: 55.9% (+8.7%)**
  Usage of the same features used for tokens in between the entities but for other tokens.
    - Features previous to E1: 47.5% (+0.3%)
    - Features before E2: 52.4% (+5.2%)
    - Both: 54.9% (+7.7%)
        - + Removing redundant features (lemma and word, only lemma_tag): 55.9% (+8.7%) ← ***NEW BEST***
          The lemma and word features of each token are considered redundant or uninformative due to the existence of lemma_tag, so they are removed.
- **More specific and general path features: 57.0% (+1.1%)**
  Usage of other features (more than lemma and relation) for each token of the paths from E1 to LCS, LCS to E2 and the concatenation of them with the LCS.
    - Tag_Relation: 57.0% (+1.1%) ← ***NEW BEST***
    - Lemma_Tag: 54.9% (-1%)
    - Both: 56.9% (+1%)
- **Information about LCS: 58.8% (+1.8%)**
  Features only based on the LCS.
    - Lemma_Tag: 58.8% (+1.8%) ← ***NEW BEST***
        - + If VB, previous lemma in the sentence: 57.9% (+0.9%)
- **Check if tokens share LCS: 57.5% (-1.3%, discarded)**
  When obtaining features from tokens previous, in between and after the entities have the same LCS with E1 than E1 and E2.
    - If share LCS with them: 57.5% (-1.3%)
- **Rule-based patterns from Lab3: 59.5% (+0.7%)**
  Use the output of a pattern from our rule-based system as a feature (if isn't None). The sentence "less ambiguous" means that the rules considered ambiguous (but were added for improving recall) are removed from the pattern for improving the precision, providing a more reliable feature.
    - check_wib: 57.5% (-1.3%)
        - check_wib less ambiguous: 58.5% (best-0.3% but prev+1%)

    - check_LCS_svo: 58.6% (-0.2%)
        - check_LCS_svo less ambiguous: 59.5% (best+0.7% and prev+1%) ← ***NEW BEST***
    - (check_LCS_svo + check_wib) less ambiguous: 58.5% (-0.3%)
        - (check_LCS_svo + check_wib) less ambiguous and in cascade: 59.2% (-0.4%)
    - (check_LCS_svo + check_LCS_VB_NN) less ambiguous: 59.1% (-0.5%)
        - (check_LCS_svo + check_LCS_VB_NN) less ambiguous and in cascade: 59.2% (-0.4%)
- **Don't consider stopwords when creating paths: 59.3% (-0.2%, discarded)**
  When creating the paths, ignore the tokens corresponding to stopwords.
    - For all paths: 59.3% (-0.2%)
- **Consider ancestors and children: 60.8% (+1.3%)**
  For E1, E2 and/or LCS, consider the ancestors, children or children of ancestors.
  In the case of the LCS, these allow for a more reliable understanding of what the LCS means.
    - Lemma_Tag from ancestors of E1 and E2: 59.2% (-0.3%)
    - Lemma_Tag from children of ancestors (bros) of LCS: 60.3% (+0.8%)
        - Remove features from tokens previous to E1: 59.0% (-1.3%)
        - Also add ancestors Lemma_Tag: 60.8 (+1.3%) ← ***NEW BEST***

In the following we depict the **extract_features** function resulting from our work. The **tokens_features** function was defined for the sake of simplicity.

```
def tokens_features(tree, entities, start_tk, end_tk, suffix, feats):
    for tk in range(start_tk, end_tk):
        if not tree.is_stopword(tk):
            word = tree.get_word(tk)
            lemma = tree.get_lemma(tk).lower()
            tag = tree.get_tag(tk)
            #feats.add(f"l{suffix}={lemma}")
            #feats.add(f"w{suffix}={word}")
            feats.add(f"lt{suffix}={lemma}_{tag}")
            # Feature indicating the presence of an entity in previous to E1
            if tree.is_entity(tk, entities):
                feats.add(f"e{suffix}")


def extract_features(tree, entities, e1, e2):
    feats = set()
    # get head token for each gold entity
    tkE1 = tree.get_fragment_head(entities[e1]['start'], entities[e1]['end'])
    tkE2 = tree.get_fragment_head(entities[e2]['start'], entities[e2]['end'])
    if tkE1 is not None and tkE2 is not None:
        # Least Common Subsumer (LCS)
        lcs = tree.get_LCS(tkE1, tkE2)
        lcs_lemma = tree.get_lemma(lcs)
        lcs_tag = tree.get_tag(lcs)
        feats.add("lcs_lemma_tag=" + lcs_lemma + "_" + lcs_tag)
        # Get info from LCS ancestors and brothers
        lcs_ancestors = tree.get_ancestors(lcs)
        for ancestor in lcs_ancestors:
            # Add ancestor
```

```
            ancestor_lemma = tree.get_lemma(ancestor)
            ancestor_tag = tree.get_tag(ancestor)
            feats.add(f"lcs_ancestor_lemma_tag={ancestor_lemma}_{ancestor_tag}")
            # Get brothers of LCS
            brothers = tree.get_children(ancestor)
            for bro in brothers:
                if bro != lcs:
                    bro_lemma = tree.get_lemma(bro)
                    bro_tag = tree.get_tag(bro)
                    feats.add(f"lcs_bro_lemma_tag={bro_lemma}_{bro_tag}")
        # Features for tokens previous to(PT) E1
        tokens_features(tree, entities, 0, tkE1, "pt", feats)
        # Features for tokens in between(IB) E1 and E2
        tokens_features(tree, entities, tkE1 + 1, tkE2, "ib", feats)
        # Features after(A) tokens E1
        tokens_features(tree, entities, tkE2+1, tree.get_n_nodes(), "a", feats)
        # Features about paths in the tree
        path1 = tree.get_up_path(tkE1, lcs)
        path1 = "<".join([tree.get_lemma(x) + "_" + tree.get_rel(x) for x in path1])
        feats.add("path1=" + path1)
        path2 = tree.get_down_path(lcs, tkE2)
        path2 = ">".join([tree.get_lemma(x) + "_" + tree.get_rel(x) for x in path2])
        feats.add("path2=" + path2)
        path = path1 + "<" + tree.get_lemma(lcs) + "_" + tree.get_rel(lcs) + ">" + path2
        feats.add("path=" + path)
        # Our custom path features with tag and relationship
        path1 = tree.get_up_path(tkE1, lcs)
        path1 = "<".join([tree.get_tag(x) + "_" + tree.get_rel(x) for x in path1])
        feats.add("path1_tag_rel=" + path1)
        path2 = tree.get_down_path(lcs, tkE2)
        path2 = ">".join([tree.get_tag(x) + "_" + tree.get_rel(x) for x in path2])
        feats.add("path2_tag_rel=" + path2)
        path = path1 + "<" + tree.get_tag(lcs) + "_" + tree.get_rel(lcs) + ">" + path2
        feats.add("path_tag_rel=" + path)
        # Rule-based patterns
        lcs_svo = check_LCS_svo(tree, tkE1, tkE2)
        if lcs_svo is not None:
            feats.add("pattern_lcs_svo=" + lcs_svo)
    return feats
```

Experiments and results

For evaluating the effect of every decision, we have performed multiple experiments during the development. In the following, the most important results are shown.

Result on devel with baseline approach:

	tp	fp	fn	#pred	#exp	P	R	F1	
advise	57	89	84	146	141	39.0%	40.4%	39.7%	
effect	127	60	185	187	312	67.9%	40.7%	50.9%	
int	16	8	12	24	28	66.7%	57.1%	61.5%	
mechanism	84	119	177	203	261	41.4%	32.2%	36.2%	
M.avg	-	-	-	-	-	53.8%	42.6%	47.1%	

m.avg	284	276	458	560	742	50.7%	38.3%	43.6%
m.avg(no class)	346	214	396	560	742	61.8%	46.6%	53.1%

Results on devel with tag relation, features, previous to E1 and after E2 and removed redundant features:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	90	55	51	145	141	62.1%	63.8%	62.9%
effect	129	40	183	169	312	76.3%	41.3%	53.6%
int	15	3	13	18	28	83.3%	53.6%	65.2%
mechanism	89	58	172	147	261	60.5%	34.1%	43.6%
M.avg	-	-	-	-	-	70.6%	48.2%	56.4%
m.avg	323	156	419	479	742	67.4%	43.5%	52.9%
m.avg(no class)	346	133	396	479	742	72.2%	46.6%	56.7%

Results on devel adding check_LCS_svo without ambiguous rules:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	90	49	51	139	141	64.7%	63.8%	64.3%
effect	143	42	169	185	312	77.3%	45.8%	57.5%
int	16	4	12	20	28	80.0%	57.1%	66.7%
mechanism	103	56	158	159	261	64.8%	39.5%	49.0%
M.avg	-	-	-	-	-	71.7%	51.6%	59.4%
m.avg	352	151	390	503	742	70.0%	47.4%	56.5%
m.avg(no class)	372	131	370	503	742	74.0%	50.1%	59.8%

Final results on devel set:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	91	43	50	134	141	67.9%	64.5%	66.2%
effect	149	48	163	197	312	75.6%	47.8%	58.5%
int	16	4	12	20	28	80.0%	57.1%	66.7%
mechanism	119	72	142	191	261	62.3%	45.6%	52.7%
M.avg	-	-	-	-	-	71.5%	53.8%	61.0%
m.avg	375	167	367	542	742	69.2%	50.5%	58.4%
m.avg(no class)	394	148	348	542	742	72.7%	53.1%	61.4%

Final results on test set:

	tp	fp	fn	#pred	#exp	P	R	F1
advise	111	91	98	202	209	55.0%	53.1%	54.0%
effect	150	77	136	227	286	66.1%	52.4%	58.5%
int	16	1	9	17	25	94.1%	64.0%	76.2%
mechanism	156	113	184	269	340	58.0%	45.9%	51.2%
M.avg	-	-	-	-	-	68.3%	53.9%	60.0%
m.avg	433	282	427	715	860	60.6%	50.3%	55.0%
m.avg(no class)	466	249	394	715	860	65.2%	54.2%	59.2%

As can be seen, with this approach we have not only significantly outperformed the rule-based approach (with an F1 score of 61.0% on the devel set instead of 39%), but we have also solved the overfitting problems (with a difference between devel and test of only 1%). The main reason for this is the class “int”, for which a surprisingly good F1 has been obtained considering the reduced variety of instances in training. Moreover, the most important improvements compared with the baseline has been for this “int” class (and for “advise”), proving that we have designed not only effective features but also ones with good generalization capabilities.

Conclusions

In this assignment we have explored the task of classifying interactions between words in the drug to drug context. The two approaches employed were far from perfect in solving the task, with a macro-average F1 score on the test set of 25.9% for the rule-based and 60.0% for the machine learning-based. However, we consider the machine learning approach superior not only in terms of performance, but also easier to use and less prone to human errors and biases than the rule-based method. In other words, with the machine learning-based approach we obtained a good enough set of features with negligible overfitting without excessive efforts. For obtaining similar results with the rule-based approach, expert language knowledge joined to many analysis and exploration hours would be required for finding the appropriate patterns and rules.