Report 2 : DDI
AHLT - Advanced Human Languages Technologies

Lackmann Simon
Piotrowski Maciej

1 Introduction

This document is a report for the DDI project of the AHLT - Advanced Human Languages Technologies course. It consists of the implementation of a simple python program that parses all XML files in the folder given as argument and classifies drug-drug interactions between pairs of drugs. It is splitted in two parts: A baseline and a machine learning version of DDI. It's inspired by SemEval 2013 Task 9: DDIExtraction. We received some documents extracted from DrugBank (Drug description leaflets database) and MedLine (abstracts of medical papers) as the initial documentation. We also received three directories containing data files that we had to use as training, validation (devel) and test sets for our experiments. For both parts we received an initial structure with some initial rules that we had to improve. Basic F1 scores were 30% for the first part and 47% for the second part. The goal was to reach respectively 35-40%and 60%. For the purpose of this project, we used Stanford CoreNLP dependency parser, which can be called via nltk, and integrates a tokenizer, a part-of-speech tagger, and a dependency parser.

2 Rule-based baseline

In this section, we describe the experiments we did with a rule-based baseline, what we observed during the data exploration and which rules we wrote according to those observations. Our baseline consists of some an arrangement of patterns which are checking whether there is an interaction between the entities and try to classify the interaction type. We didn't implement any statistical approaches for this section. The purpose was to find a lower bound for the performance of ML systems that we build in the second part of this assignment. We include our code and the results we obtained on the devel and test datasets with our best combination. We also describe all the other rule combinations we used that we think are relevant and the according results.

2.1 Ruleset construction

We improved the existing patterns by adding some words to the existing lists. We found those words using the grep command described in the section 2.3. We also added a new pattern which, given 2 entities in a sentence, checks if the words "should" or "caution" appeared in the sentence before the first entity. We obtained the best results with the following patterns (in order):

1. check_wib (A word in between both entities belongs to certain list):
   (a) effect : [tendency, stimulate, regulate, prostate, modification, augment, accentuate, exacerbate]
   (b) mechanism : [react, metabolism, faster, presumably, induction, substantially, minimally, concentration]
   (c) advise : [exceed, extreme, cautiously, should]
   (d) int : [interact]
2. check_LCS_svo (LCS is a verb, one entity is under its "nsubj" and the other under its "obj"):
   (a) effect : [diminish, augment, experience, counteract, potentiate, enhance, reduce, antagonize, include, lower]
   (b) mechanism : [impair, inhibit, displace, accelerate, bind, induce, decrease, elevate, delay, produce, prolong, cause, show]
   (c) advise : [exceed, should]

(d) int : [suggest]

3. check_wb (A word before first entity belongs to certain list):

   (a) advise : [should, caution]

Furthermore, we checked lemmas which occur after the second entity in a sentence. For this purpose, we used the command grep, in order to get all sentences and corresponding entities for a given class (mechanism,int,effect,advise). Then we filtered all sentences such that we yield the rest of a sentence after the second entity. For time complexity reasons, it was easier for us to write a simple python script. We then tested the most occuring resulting words in our method "check_wa". For example when we return 'effect' if lemma is in 'cause', at the beginning we were able to increase our F1-Score, but in our final pattern-combination it did not help, so we omit it.

2.2 Code

```
## -------------------
## -- check if a pair has an interaction and of which type, applying a cascade of rules.

    def check_interaction(tree, entities, e1, e2) :

   # 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'])

   #check different patterns for interaction
   p = patterns.check_wib(tree,tkE1,tkE2,entities,e1,e2)
   if p is not None: return p

   p = patterns.check_LCS_svo(tree,tkE1,tkE2)
   if p is not None: return p

   p = patterns.check_wb(tree,tkE1,entities,e1)
   if p is not None: return p

   #Commented out as it decreased the result
   """p = patterns.check_wa(tree,tkE1,tkE2,entities,e1,e2)
   #if p is not None: return p"""

   return "null"

## -------------------
## -- check pattern:  LCS is a verb, one entity is under its "nsubj" and the other under its "obj"

def check_LCS_svo(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" :
         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','experience','counteract','potentiate',
                        'enhance','reduce','antagonize','include',
                        'lower'] :
              return 'effect'
           if lemma in ['impair','inhibit','displace','accelerate','bind','induce',
                        'decrease','elevate','delay','produce',
                        'prolong','cause','show'] :
              return 'mechanism'
           if lemma in ['exceed','should'] :
              return 'advise'
           if lemma in ['suggest'] :
              return 'int'

   return None

## -------------------
## -- check pattern:  A word in between both entities belongs to certain list

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']
      p = []
      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
         if r1 < l and r < l2:
            lemma = tree.get_lemma(t).lower()
            if lemma in ['tendency','stimulate','regulate','prostate',
            'modification','augment','accentuate','exacerbate'] :
               return 'effect'
            if lemma in ['react','metabolism','faster','presumably','induction',
            'substantially','minimally','concentration']:
               return 'mechanism'
            if lemma in ['exceed','extreme','cautiously','should']:
               return 'advise'
            if lemma in ['interact'] :
               return 'int'

   return None
```

```
## -------------------
## -- check pattern:  A word before first entity belongs to certain list

def check_wb(tree,tkE1,entities,e1):

   if tkE1 is not None and tkE2 is not None:
      # get actual start/end of the first entity
      l1,r1 = entities[e1]['start'],entities[e1]['end']
      p = []
      for t in range(tkE1) :
         # get token span
         l,r = tree.get_offset_span(t)
         # if the token is in before the entity
         if l1 > r:
            lemma = tree.get_lemma(t).lower()
            if lemma in ['should','caution']:
               return 'advise'

   return None

# The following did not increase the result
# But we did not comment it out for better readability

## -- check pattern:  A word after second entity belongs to certain list
def check_wa(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']
      p = []
      for t in range(tkE2,tree.get_n_nodes()):
         # get token span
         l,r = tree.get_offset_span(t)
         # if the token is in between both entities
         if r2 < l:
            lemma = tree.get_lemma(t).lower()
            # No word increased the score here
            if lemma in []:
               return 'mechanism'
            if lemma in [] :
               return 'advise'
            if lemma in [] :
               return 'int'
            if lemma in [] :
               return 'effect'
   return None
```

```
## -- check pattern:  LCS of both entities is a verb with a should child,
# return 'advise' if it is
def check_LCS_verb(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":
         children_of_lcs = tree.get_children(lcs)
         for child in children_of_lcs:
            lemma = tree.get_lemma(child).lower()
            if lemma in ['should']:
               return 'advise'
   return None

# check if the parent of both entities is the same
def check_parent(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']
      p = []
      parent1 = tree.get_parent(tkE1)
      parent2 = tree.get_parent(tkE2)
      if (parent1==parent2):
          return 'advise'
    return None

# The python script for finding most occuring words for check_wa:

import os
import collections

with open("mechanismCheck", "r") as f:
    x = f.readlines()
    lenx = len(x)
with open("mechanismCheck", "r") as f:
    all_words = []
    for i in range(lenx):
        sentence = f.readline()
        if "matches found" in sentence or "P(" in sentence:
            break
        entities = f.readline()
        _ = f.readline()
        _ = f.readline()
        _ = f.readline()
        word = entities.split("/ ")[1].split("\n")[0]
        rest = sentence.split(word)[1]
        rest = rest.split(" ")
        all_words.extend(rest)
    print(collections.Counter(all_words).most_common(100))
```

2.3 Experiments and results

With our best ruleset construction described in the section 2.1, we are able to reach a precision of 59.2%, a recall of 42.3% and a F1 score of 45.0% on the devel data. But on the test data, we obtain a precision of 38.1%, a recall of 27.8% and a F1 score of 27.7%. So our ruleset combination clearly "overfits" the devel data.

	tp	fp	fn	#pred	#exp	P	R	F1
advise	63	116	78	179	141	35.2%	44.7%	39.4%
effect	48	13	264	61	312	78.7%	15.4%	25.7%
int	19	3	9	22	28	86.4%	67.9%	76.0%
mechanism	108	189	153	297	261	36.4%	41.4%	38.7%
M.avg	-	-	-	-	-	59.2%	42.3%	45.0%
m.avg	238	321	504	559	742	42.6%	32.1%	36.6%
m.avg(no class)	277	282	465	559	742	49.6%	37.3%	42.6%

Figure 1: Best ruleset construction scores on the devel data

	tp	fp	fn	#pred	#exp	P	R	F1
advise	103	346	106	449	209	22.9%	49.3%	31.3%
effect	78	23	208	101	286	77.2%	27.3%	40.3%
int	2	6	23	8	25	25.0%	8.0%	12.1%
mechanism	91	244	249	335	340	27.2%	26.8%	27.0%
M.avg	-	-	-	-	-	38.1%	27.8%	27.7%
m.avg	274	619	586	893	860	30.7%	31.9%	31.3%
m.avg(no class)	306	587	554	893	860	34.3%	35.6%	34.9%

Figure 2: Best ruleset construction scores on the test data

We tried many other combinations as well but unfortunately we couldn't reach a better score with them. It only decreased it. Here are the most interesting conditions that we expected to perform well (but didn't) and the corresponding **F1 scores (devel/test)** if we add it to our best ruleset:

- For the check_LCS_svo() pattern:
  1. effect : Adding to the list ['alter', 'cause'] only decreased to 44.0 % / 26.6%.

     Adding ["require", "receive", "change", "prevent"] and keeping "exhibit" made no difference.
  2. mechanism : Adding ['demonstrate', 'have', 'alter'] only decreased the F1 score to 44.5% / 27.6%.

     Adding ['modifiy', 'delay'] made no difference.
  3. int : Adding ['fluoroquinolone', 'invalidate', 'maintain'] made no difference.
- We tried to implement an additional pattern check_LCS_verb() which was supposed to classify as 'advise' if the LCS of both entities was a verb with a 'should' child. But it decreased the F1 Score to 43.6% / 27.0%. (The pattern is present in the patterns.py file but it isn't used)
- We tried to implement a pattern check_wa() for checking the words appearing after the second entity. We wrote a script to get the most common words appearing after the second entity depending on the interaction type between the entities and tried out some of the combinations but unfortunately it only decreased the score. (Ex: checking if 'cause' is appearing after the second token and classifying it as an "effect" interaction decreased the F1 score to 44.2% / 25.3%). Some examples of words that we tried:
  1. effect : ['cause', 'depression']

2. mechanism : ['metabolism', 'intravenous', 'levels', 'result']
3. int : ['salt', 'lithium']
4. advise : ['dose']

The bash command we used to find some patterns in the training data :

- ./runExplore.sh | grep -A 1 "mechanism" | grep "match" | sort | uniq -c | sort

The idea is the following:
At first, *grep -A 1 "mechanism"* returns all occurences with mechanism and also the line below. This line contains the corresponding match we are interested in. Afterwards we sort these results and remove duplicates. At the end, we can see how many matches correspond to the class "mechanism". Of course, we do it for every of the four classes. These words are then used in our pattern.py file in the check_interaction and check_wib methods. Here, we tried different word combinations in a grid_search - like manner to find the best matchings.

3 Machine learning DDI

3.1 Selected algorithm

As algorithm we used megam-64. We tried to use other classifiers but we struggled too much with the implementation so we decided to focus on the one provided. Because of this, we focused on getting better results through the feature extraction and by optimizing the megam-64 hyper-parameters. For megam-64 , we tried different combinations of the following hyper-parameters: repeat, maxi, memory, tune[lambda], norm1 and norm2. None of these did have a positive effect on the results.

3.2 Feature extraction

For the features extraction we added the patterns.py file from the baseline. Then, we extended the extract-features.py as follows: Add new features for tokens in between e1 and e2 and try different combinations of these features. Examine features before e1 and also features after e2, in the same way as for tokens between e1 and e2. Furthermore, for features about paths in the tree, we added features for the edge label and additionally the syntactic function for both entity indices. Next, we build a feature for the length of our entities list. Finally, we take the type of enties e1 and e2 into consideration. For this purpose, we added "'type':e.attributes['type'].value" to the 'start' and 'end' value of an entity's id when initializing them.
We tried out some more features, but they did not increase/only decrease our results. Because of that, we commented them out in the provided code.

3.3 Code

```
## --------------------
## -- Convert a pair of drugs and their context in a feature vector

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'])

    sentence_len = len(tree.get_nodes())
    if tkE1 is not None and tkE2 is not None:
```

```
#Patterns from Session3
p = patterns.check_wib(tree,tkE1,tkE2,entities,e1,e2)
if p is not None: feats.add("type_wib=" + p)

p = patterns.check_LCS_svo(tree,tkE1,tkE2)
if p is not None: feats.add("type_LCS=" + p)

p = patterns.check_wb(tree,tkE1,entities,e1)
if p is not None: feats.add("type_wb=" + p)


# features for tokens in between E1 and E2
for tk in range(tkE1+1, tkE2) :
   if not tree.is_stopword(tk):
      word  = tree.get_word(tk)
      lemma = tree.get_lemma(tk).lower()
      tag = tree.get_tag(tk)
      feats.add("lib=" + lemma)
      feats.add("wib=" + word)
      feats.add("lpib=" + lemma + "_" + tag)
      feats.add("tib="+tag)
      feats.add("lwib="+lemma+"_"+word)
      feats.add("wtib="+word+"_"+tag)
      feats.add("lwtib="+lemma+"_"+word+"_"+tag)

      # feature indicating the presence of an entity in between E1 and E2
      if tree.is_entity(tk, entities) :
         feats.add("eib")

# features before E1
for tk in range(tkE1):
   if not tree.is_stopword(tk):
      word  = tree.get_word(tk)
      tag = tree.get_tag(tk)
      lemma = tree.get_lemma(tk).lower()
      feats.add("libBefore=" + lemma)

      # feature indicating the presence of an entity before E1
      if tree.is_entity(tk, entities) :
         feats.add("eibBefore")


# features after E2
for tk in range(tkE2,sentence_len):
   if not tree.is_stopword(tk):
      word  = tree.get_word(tk)
      lemma = tree.get_lemma(tk).lower()
      tag = tree.get_tag(tk)
      feats.add("libAfter=" + lemma)
```

```
            feats.add("wibAfter=" + word)
            feats.add("ta="+tag)
            feats.add("lwa="+lemma+"_"+word)
            feats.add("wta="+word+"_"+tag)
            feats.add("lwta="+lemma+"_"+word+"_"+tag)
            # feature indicating the presence of an entity after E2
            if tree.is_entity(tk, entities) :
               feats.add("eibAfter")


    # features about paths in the tree
    lcs = tree.get_LCS(tkE1,tkE2)

    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)

    #features about edge labels
    feats.add("left_edge="+(path.split("<")[0]).split("_")[-1])
    feats.add("right_edge="+(path.split(">")[-1]).split("_")[-1])

    #features about entities labels
    feats.add("rel1="+tree.get_rel(tkE1))
    feats.add("rel2="+tree.get_rel(tkE2))

    #feature about the number of entities in the sentence
    feats.add("entities_number="+str(len(entities)))

    #features about the type of the entities
    feats.add("type1="+entities[e1]['type'])
    feats.add("type2="+entities[e2]['type'])

    # The following did not increase the result
    # But we did not comment it out for better readability

    # check if path connecting e1 to e2 (over lcs) has third entity
    pathE1 = tree.get_up_path(tkE1,lcs)
    pathE2 = tree.get_up_path(tkE2,lcs)
    pathE1_E2 = pathE1 + pathE2 + [lcs]
    # remove tkE1 and tkE2 from path as we want to look for a third  entity
    if tkE1 != tkE2:
      pathE1_E2.remove(tkE1)
      pathE1_E2.remove(tkE2)
```

```
        for elem in pathE1_E2:
            if tree.is_entity(elem,entities):
                feats.add("ThirdEntity_in_path")
                break

    # features about lcs (PoS, lemma..)
    lcs = tree.get_LCS(tkE1,tkE2)
    feats.add("lcsPoS="+tree.get_tag(lcs))
    feats.add("lcsLemma="+tree.get_lemma(lcs))
    feats.add("lcsWord="+tree.get_word(lcs))
return feats
```

3.4 Experiments and results

Here we present results we get for the devel and test datasets. As the feature extraction is not deterministic, it is possible that the results may slightly differ for each execution. We iterated it a few times and show the highest result that we could get. Each of our used features increased the F1-Score a bit. As we can see in Figure 3, with the combination of all our features we used, we nearly reached 60% for the F1-Score. We were able to significantly increase the low recall for our classes. Moreover, our model generalizes well, as we can see in Figure 4. We achieve an F1-Score of up to 53.3% on the test set.

	tp	fp	fn	#pred	#exp	P	R	F1
advise	87	50	54	137	141	63.5%	61.7%	62.6%
effect	169	131	143	300	312	56.3%	54.2%	55.2%
int	16	1	12	17	28	94.1%	57.1%	71.1%
mechanism	111	85	150	196	261	56.6%	42.5%	48.6%
M.avg	-	-	-	-	-	67.6%	53.9%	59.4%
m.avg	383	267	359	650	742	58.9%	51.6%	55.0%
m.avg(no class)	415	235	327	650	742	63.8%	55.9%	59.6%

Figure 3: Best scores for the ML part on the devel data

	tp	fp	fn	#pred	#exp	P	R	F1
advise	88	49	121	137	209	64.2%	42.1%	50.9%
effect	165	217	121	382	286	43.2%	57.7%	49.4%
int	13	0	12	13	25	100.0%	52.0%	68.4%
mechanism	142	157	198	299	340	47.5%	41.8%	44.4%
M.avg	-	-	-	-	-	63.7%	48.4%	53.3%
m.avg	408	423	452	831	860	49.1%	47.4%	48.3%
m.avg(no class)	481	350	379	831	860	57.9%	55.9%	56.9%

Figure 4: Best scores for the ML part on the test data

Some extra features that we tried but that didn't improve our scores in a statically relevant way:

- Checking whether there is a third entity in the path connecting e1 and e2. For this we made use of the lcs and examined words on the path between e1 - lcs - e2.
- Some features about the lcs, where we tried to add the Pos, lemma or word of the lcs.
- The edges directions

4 Conclusions

We reached the recommended percentage of F1-Score for the baseline and also tried to improve it even more. For the second part, we orbit around the recommended percentage. For the first task, we reached a F1-Score of 45.0%. We tried a lot of rules/patterns to reach the best possible score and different combinations of them. Some of them improved the score and some decreased. Surprisingly, some of the patterns that we added, couldn't improve the score even though we were thinking that they were relevant for the algorithm. One explanation for that might be that we improved our previous patterns so much that it was too effective for the other patterns to make a difference. Despite that, it was really enriching to work on them and try them out. With that, we had a really solid baseline for the next part.
Moreover, for the second part, we wanted to test some different algorithms, which are popular machine learning classifiers, but we couldn't adapt the implementation. We've spent some time on it and as it wasn't working, we decided to drop the other algorithms and focused on megam as the rest of the task seemed pretty time-consuming. With the megam64 algorithm, we could reach a score about 59.4%. Working on the different improvements of the algorithm allowed us to better understand the concepts seen in class and have a better visualisation of the different tree structures and how to handle them. We are very curious if we could reach better results using some other algorithms and are very disappointed that we couldn't try them out and compare their results with the results of megam.
The whole project was very interesting, it allowed us to discover even more the field related to NLP tasks. It allowed us to observe all the advantages and disadvantages of the different patterns. Of course, there's still a possibility to improve even more the models to obtain a better F1-Score (by finding some better rules or tuning even more the parameters) but we are happy with the scores we reached. Also, the project allowed us to put into practice the material we saw during the lectures and to get a better understanding of it.