Advanced Human Languages Technologies
DDI Report

Denaldo Lapi and Francesco Aristei

November 5, 2022

1 Introduction

This report contains some possible solutions of the task 9.2 of the *SemEval-2013 challenge* (link to the paper), concerning the detection and classification of drug-drug interactions between pairs of drugs (DDI). We were provided with a dataset consisting of XML files containing sentences and entities representing various types of classes (drug, brand, group and drug_n), together with the type of interactions between entities (mechanism, effect, advise, interaction). Each sentence contains attributes of type *pair* which highlights the presence of an interaction between two entities in the sentence. The data is already splitted in three subsets (folders): Train, Devel and Test.

We were also provided with evaluation scripts to compute metrics like precision, recall and F1 score.

In order to solve the task, we focused on 2 main approaches:

- Rule-based (that was used as a baseline)
- Machine Learning based

The aim of this report is to explain how we used these two approaches to solve the DDI task, describing the main aspects, rules and decisions taken in order to otimize the performance of each method.

2 Rule-based baseline

At the beginning, we developed a simple rule-based system in order to have a baseline for our DDI task, i.e. a lower bound for the ML system that we built in the second part of our work.

The idea of this approach is to build a simple list of rules, chaining them as a cascade of statements, to decide the presence of some kind of interaction between two entities in the sentence.

2.1 Ruleset construction

In order to extract meaningful rules able to spot the interactions between entities, we performed some kind of preliminary data exploration by looking at the structure of the dependency tree of

the sentences, obtained through the *Stanford CoreNLP* dependecy parser.
We performed this activity on the *train* folder of the provided dataset, while we checked the performance of the various rules on the *devel* folder; this last step is needed, in order to choose the most meaningful rules for this approach and their right order inside the cascade of rules.
We were provided with an initial version of the code which achieves 30% macro average F1 on the devel set.

We will now make a brief digression on the relations we observed in the sentences of the *train* folder to extend and modify the initial rules.
In order to spot the candidate interactions we used the *explore.py* file, a utility script that checks for the patterns we think could be useful to detect DDI interactions. Specifically, it prints useful information to visualize the goodness of the rule defined. Below, the list of rules we tried, together with the performances obtained:

- We first looked at the *svo.check* file, after running the already provided *check_pattern_LCS_svo* method in the *explore.py* file. This rule checks when the LCS (Lowes Common Subsumer) is a verb and one entity is under its *nsubj* and the other under its *obj*. Specifically, in the *svo.check* we observed an high probability of having interactions of a certain type between the pair of entities, when the LCS was equal to specific verbs. Therefore, we modified the *check_LCS_svo* method in the *pattern.py* file, adding the said verbs. We first added *increase* in the list of verbs for the LCS that returns a *mechanism* type of interaction. The performances increased from 30.6% to 31.6% (F1 score). Then *indicate* was added for the *mechanism* list, but we didn't see any improvement. Then we continued in the *mechanism* list with *produce*, increasing to 31.7 and lastly with *have* without obtaining any better results. Then we tried to enlarge the list of verbs for the *advise* class with *mantain*, *used*, *extend* and *require*, given the high probabilities returned by the *check_pattern_LCS_svo* method, without any improvement. Lastly we focused our attention on the *effect* class, adding *include*, *block*, *prolong*, *attenuate* and *prevent*. In the first two cases, we saw an improvement passing from 31.7% to 32.2% in the first case, and from 32.2% to 32.4% in the second. While for the last three verbs, we didn't get any better result.
- We then added in the *patterns.py* the *check_should_advise* method, which checks if the LCS is a verb having as child the *should* verb, and if this is verified, it returns the class *advise* as interaction type for the pair of entities. With this simple rule, we increased drastically the performances from 32.4% to 38.6%. Given the specificity of this rule, we decided to put it as the first one.
- We now implemented a new method in the *explore.py*, to check whether two entities in a certain pair are equal to each other. What we observed in the *svo.check* file is that almost all the pairs having the two entities perfectly equal are never in some DDI relation. Therefore we added in the *pattern.py* a rule to check whether both entities in a pair are equal, and when the pattern is verified, we returned *None*. The performances increased, from the previous 38.6% to 39.6%. We added this rule as the first onenin the *baseline-DDI.py*.
- We tried to extract some hints from the scientific papers provided in the *Papers* folder. Specifically, we read through the *HerreroZazo-et-al-2013.pdf*. In the paper a set of simple rules exploiting the drug classes are written. Particularly, in the paper is suggested to look the order in which the classes appears, i.e if the first entity is for example of *brand* type and the second one is of *drug* type, then the *effect* class of interaction should be matched. Therefore we applied these following rules:
  - First entity of class *brand*, second of class *drug*: we returned *effect*

  - First entity of class *drug*, second of class *drug*: we returned *mechanism*
  - First entity of class *brand*, second of class *group*: we returned *effect*
  - First entity of class *drug*, second of class *group*: we returned *effect*
  - First entity of class *group*, second of class *drug*: we returned *advise*

  The only one who gave us actual improvements in the performance was the one defining an interaction of class *effect* when the first entity was of type *brand* and the second one was of type *group*. The performance increased from 39.6% to 41.2%.

- Then, we tried to check the case in which the second entity is the parent of the first one. From the *svo.check* file we noticed that most of the time there is no DDI relationship when this happen. Therefore we tried to add this rule in the *pattern.py*. However the performances remained unchanged at 41.2%, but we still mantained the rule.

- Inspired by the rule defined above, we quickly tried to check the case in which the two entites have the same parent. By using *explore.py* we noticed that there was an high correlation bethween a parent tag of type verb and the interaction type *advise*; while in case of same parent with a tag different from a noun or a verb we almost always have no interaction. We therefore added this rule and we saw a little improvement in the F1 score, from 41.2% to 41.3%.

- At this point, we tried to exploit more the content of the *svo.check* file. Just applying a visual inspection is not enough to understand all the useful patterns. Therefore, we wrote a python script to better process the lines in the file. The first thing we tried was to extract the parent of the first entity when the probability of the type of interaction between the entities, given this parent, was above a defined threshold, and the number of time this pattern appeared (i.e the *support*) was above a certain value. Then we saved the results of the search in a list and we defined a rule in the *pattern.py* that returns the associated type of interaction when the first entity has a parent matching one of the elements in the predefined list. We tried the rule with different probabilities and supports, the best improvement was reached using a minimum probability of 0.5 and a minimum support of 2. The performance improved from 41.3% to 43.3%. We then tried the same rule for the second entity, increasing the performance from 43.3% to 44.5%.

- Lastly, we modified the already provided *check_wib* method, which checks if a word in between both entities belongs to certain list, with the exact same reasoning. We only changed the rules about the probabilities and the supports. Specifically, if the support of the lemmas in between the two entities is greater or equal to 2 and less or equal to 20, we only looked for probabilities greater than 0.75, otherwise, if the support exceed 20, we relaxed a bit the probability, which needs to be at least 0.5. With this new rule, we reached our final result of 45.7%,

The final results on the test and devel datasets will be reported in the following sections.

2.2 Code

We include below the code of the function *check_interaction* and of the python file *patterns.py* containing the methods called by the previous function. The explanation of the functions is provided in the comments reported inside the code snippets.

Below the code of the *check_interation*:

```python
## -------------------
## -- 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'])

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

   p = patterns.check_should_advise(tree,tkE1,tkE2)
   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_wib(tree,tkE1,tkE2,entities,e1,e2)
   if p is not None: return p

   p = patterns.check_group_after_brand(entities, e1, e2)
   if p is not None: return p

   p = patterns.check_e1_under_e2(tree, tkE1, tkE2)
   if p is not None: return p

   p = patterns.check_e2_under_e1(tree, tkE1, tkE2)
   if p is not None: return p

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

   p = patterns.check_parent_e1(tree, tkE1, tkE2)
   if p is not None: return p

   p = patterns.check_parent_e2(tree, tkE1, tkE2)
   if p is not None: return p

   return "null"
```

The function *check_interaction* takes as input the dependency tree of the sentence together with the pair of entities and applies the cascade of rules defined in the *patterns.py* whose code is reported below: :

```python
#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) # get the LCS
```

```
        if tree.get_tag(lcs)[0:2] == "VB" :  # LCS is a verb
            path1 = tree.get_up_path(tkE1,lcs) # path from tkE1 to the LCS
            path2 = tree.get_up_path(tkE2,lcs) # path from tkE2  to the LCS
            func1 = tree.get_rel(path1[-1]) if path1 else None #last element of the path
            func2 = tree.get_rel(path2[-1]) if path2 else None

            # check condition
            if (func1=='nsubj' and func2=='obj') or (func1=='obj' and func2=='nsubj') :
                lemma = tree.get_lemma(lcs).lower() # lemma of the LCS
                if lemma in ['diminish','augment','exhibit','experience','counteract',
                'potentiate','enhance','reduce','antagonize', 'include', 'block'] :
                    return 'effect' # if the lemma is one of the above, the relationship is of type effect
                if lemma in ['impair','inhibit','displace','accelerate','bind','induce',
                'decrease','elevate','delay', 'increase', 'indicate', 'produce'] :
                    return 'mechanism'
                if lemma in ['exceed'] : # 'mantain', 'useda', 'extend', 'require'
                    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) : # for all the tokens in between
            # 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 ['phosphorylation', 'enhance', 'locomotor', 'action', 'response', 'oxidative', 'stress', 'e', 'ig', '4th', 'protection', 'tbars', 'cerebral', 'radical', 'damage', 'peroxidation', 'ie', 'man', 'bleeding', 'odds', 'retinal', 'transduction', 'counteract', 'equally', 'antagonize', 'stimulate', 'proliferation', 'epithelium', 'transferrin', 'mitogenic', 'regulate', 'prostate', 'modification', 'secondary', 'adrenocortical', 'augment', 'alcohol',
```

```
'neuron', 'central', 's.', 'c.', 'mumol', 'liter', 'rarely', 'ventricular',
↪   'fibrillation', 'asparaginase', 'antineoplastic', 'weakness', 'hyperreflexia',
↪   'incoordination', 'acetyltransferase', 'consequence', 'Abciximab',
↪   'photosensitivity', 'actinic', 'keratose', 'aggregation', 'cilostazol',
↪   'exacerbate', 'prothrom', 'bin', 'tendency', 'hypokalemic', 'antiplatelet',
↪   'exaggerate', 'syndrome', 'AKINETON', 'transdermal', 'nervous', 'mefloquine',
↪   'Parkinsons', 'antagonistic', 'antiparkinsonian', 'trihexyphenidyl', 'related',
↪   'oxygen', 'nimbex', 'sleeping', 'accentuate', 'glaucoma', 'serotoninergic',
↪   'migraine', 'Imitrex', 'bacteriostatic', 'hyperuricemic', 'hypoparathyroid',
↪   'sodation', 'halogenate', 'hydrocarbon', 'autonomic', 'irritability',
↪   'arrhythmia', 'NUROMAX', 'lengthen', 'occurrence', 'ototoxic', 'Injection',
↪   'timolol', 'rebound', 'antimuscarinic', 'weaken', 'metaraminol', 'abciximab',
↪   'GP', 'iib', 'iiia', 'tabloid']:
               return 'effect'
            if lemma in ['acute', 'biotransformation', 'statistically',
            ↪   'Lomefloxacin', 'react', 'faster', '31', 'induction', 'due',
            ↪   'presumably', 'accelerate', 'pancreatin', 'cyp', 'Cmin',
            ↪   'medicinal', 'modest', 'displace', 'Acetazolamide',
            ↪   'Acetaminophen', 'q12h', '1a2', 'malabsorption', 'ascorbic',
            ↪   'fruit', 'phosphate', 'ionized', 'species', 'ed50', 'John',
            ↪   'p450iiia4', 'elc', 'gefitinib', 'carbonate',
            ↪   'plasmaconcentration', 'triazolo', 'ingest', 'where',
            ↪   'determinant', 'indigestion', 'remedy', 'intense', 'respiratory',
            ↪   'sulfamethizole', 'sulphasalazine', 'tpmt']:
               return 'mechanism'
            if lemma in ['index', 'SSRI', 'pth', 'methysergide', 'Ophthalmic',
            ↪   'Solution', 'pulse', 'myocardial', 'SUBOXONE', 'isoenzyme',
            ↪   'management', 'nephrotoxic', 'withdraw', 'cautiously', 'exceed',
            ↪   'predominantly', 'narrow', 'window', 'supraventricular',
            ↪   'terbinafine', 'tell', 'doctor', 'buprenorphine', 'pure',
            ↪   'methylergonovine', 'antimycotic', 'adjunctive', '533', '133',
            ↪   'isrecommend', 'INVEGA']:
               return 'advise'
            if lemma in ['interact', 'Diuretics', 'Tylenol', 'cyp2b6', 'MIVACRON',
            ↪   'incompatible', 'thiamine', 'Loop']:
               return 'int'

   return None

# check pattern:  LCS is a verb with a 'should' child
def check_should_advise(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": # LCS is verb

           # check for 'should' child
            for child in tree.get_children(lcs):
                if tree.get_lemma(child) == 'should':
                    return 'advise'
    return None

```

```
# check pattern: same entities in the pair
def check_pattern_same_entities(tree, tkE1, tkE2):

   if tkE1 is not None and tkE2 is not None:
      if tree.get_lemma(tkE1) == tree.get_lemma(tkE2):
         return 'null'
   return None

# check pattern: brand followed by group
def check_group_after_brand(entities, e1, e2):

    if entities[e1]['type'] == 'brand' and entities[e2]['type'] == 'group':
        return 'effect'

# check pattern: entity 2 parent of entity 1
def check_e1_under_e2(tree, tkE1, tkE2):

   if tkE1 is not None and tkE2 is not None:
      if tkE2 == tree.get_parent(tkE1):
         return 'null'
   return None

# check pattern: entity1 parent of entity 2
def check_e2_under_e1(tree, tkE1, tkE2):

   if tkE1 is not None and tkE2 is not None:
      if tkE1 == tree.get_parent(tkE2):
         return 'null'
   return None

# check pattern: same parent
def check_same_parent(tree, tkE1, tkE2):

   if tkE1 is not None and tkE2 is not None:
      p1 = tree.get_parent(tkE1)
      p2 = tree.get_parent(tkE2)

      if p1 == p2 and (p1 is not None): # same parent
         if tree.get_tag(p1)[0:2] not in ['NN', 'VB']: # parent not a verb or noun
            return 'null'
         if tree.get_tag(p1)[0:2] == 'VB': # verb parent
            return 'advise'
   return None

# check pattern: entity 1 under list of possible lemmas
def check_parent_e1(tree, tkE1, tkE2):

   if tkE1 is not None and tkE2 is not None:
      p1 = tree.get_parent(tkE1)
      if p1 is not None:
         lemma = tree.get_lemma(p1).lower()
```

```
         if lemma in ['user', 'enhance', 'attenuate', 'efficacy', 'pretreatment',
         ↪  'action', 'time', 'ethynyl', 'response', 'mediate', 'block',
         ↪  'dexamethasone', 'glucocorticoid', 'min', 'augment', 'experience',
         ↪  'prolong', 'vasopressor',
'diminish', 'chloramphenicol', 'exaggerate', 'syndrome', 'butalbital', 'quinolone',
↪  'counteract', 'butyrophenones', 'cyclopropane', 'impair', 'exacerbate',
↪  'epinephrine', 'only', 'stavudine', 'resistance', 'phosphorylation',
↪  'anticoagulants', 'capable', 'purinethol']:
            return 'effect'
         if lemma in ['presence', 'trovafloxacin', 'moxifloxacin', 'react',
         ↪  'modify', 'absorption', 'calcium', 'expect', 'exist', 'know', 'q12h',
         ↪  'compete', 'equivalent', 'videx', 'course']:
            return 'mechanism'
         if lemma in ['b1']:
           return 'int'
         if lemma in ['exert', 'start', 'dihydroergotamine', 'avoid', 'when',
         ↪  'titrate', 'stop', 'capsules', 'exhibit', 'tell', 'gland',
         ↪  'beadminister', 'while']:
            return 'advise'
   return None

# check pattern: entity 2 under list of possible lemmas
def check_parent_e2(tree, tkE1, tkE2):

   if tkE1 is not None and tkE2 is not None:
      p2 = tree.get_parent(tkE2)
      if p2 is not None:
         lemma = tree.get_lemma(p2).lower()
         if lemma in ['combine', 'effect', 'sensitivity', 'activity', 'enhance',
         ↪  'initiate', 'add', 'dasatinib', 'action', 'produce', 'egf',
         ↪  'stimulate', 'antagonize', 'those', 'neurotensin', 'reverse',
         ↪  'atracurium', 'blockade', 'proleukin', 'alfa', 'coumarin', 'worsen',
         ↪  'exacerbate', 'include', 'hydrochloride', 'catecholamine',
         ↪  'cephalosporin', 'anesthesia', 'withdraw', 'tetracycline', 'toxoid',
         ↪  'imitrex', 'dispersible', 'concomitantly', 'vasodilation', 'exhibit',
         ↪  'miconazole', 'anticholinergic', 'zyvox', 'starlix', 'pantoprazole',
         ↪  'piperazine', 'purinethol', 'thioguanine', 'vardenafil', 'sonata']:
            return 'effect'
         if lemma in ['molecule', 'metabolism', 'opiate', 'react', 'whereas',
         ↪  'media', 'secretion', 'elimination', 'form', 'colestipol', 'displace',
         ↪  'spray', 'anticipate', 'ergocalcitriol', 'find', 'auc0', 'max',
         ↪  'mean', 'malabsorption', 'ed50', 'oxyphenbutazone', 'bid', 'vitamin',
         ↪  'ester']:
            return 'mechanism'
         if lemma in ['solution', 'b1', 'diuretics']:
            return 'int'
         if lemma in ['dosing', 'initiate', 'alosetron', 'any', 'achieve',
         ↪  'isoenzyme', 'undergo', 'chlorprothixene', 'flecainide',
         ↪  'sumatriptan', 'avoid', 'start', 'indocin', 'cerezyme', 'naratriptan',
         ↪  'nevirapine', 'isrecommend', 'vioxx']:
            return 'advise'
   return None
```

2.3 Experiments and results

We report in figure 1 the statistics regarding the performance obtained on the devel and on the test datasets for what regards our best performing rule-based baseline obtained at the end of the rules building process.

	tp	fp	fn	#pred	#exp	P	R	F1
advise	55	139	86	194	141	28.4%	39.0%	32.8%
effect	130	227	182	357	312	36.4%	41.7%	38.9%
int	20	2	8	22	28	90.9%	71.4%	80.0%
mechanism	72	133	189	205	261	35.1%	27.6%	30.9%
M.avg	-	-	-	-	-	47.7%	44.9%	45.7%
m.avg	277	501	465	778	742	35.6%	37.3%	36.4%
m.avg(no class)	348	430	394	778	742	44.7%	46.9%	45.8%

	tp	fp	fn	#pred	#exp	P	R	F1
advise	87	166	122	253	209	34.4%	41.6%	37.7%
effect	123	292	163	415	286	29.6%	43.0%	35.1%
int	2	5	23	7	25	28.6%	8.0%	12.5%
mechanism	128	147	212	275	340	46.5%	37.6%	41.6%
M.avg	-	-	-	-	-	34.8%	32.6%	31.7%
m.avg	340	610	520	950	860	35.8%	39.5%	37.6%
m.avg(no class)	405	545	455	950	860	42.6%	47.1%	44.8%

Figure 1: Best rule-based model performance on the Devel set (on the left) and on the Test set (on the right

We would like to briefly comment the difference between the performance of the *int* class of interaction between devel and test sets which determines the overall drop between devel and test. Indeed, we can see that we pass from a F1 score of 80.0% on the devel to only 12.5% on the test test, for what regards *int*' interactions. What happens is that most probably our rules for what regards that class overfit too much the devel dataset, and they don't properly work in the test dataset: we basically capture too much the characteristics and the noise of that class on the devel XML files. This is also due to the fact that the dataset is very unbalanced, since we have too few samples from the *int* class.

3 Machine Learning DDI

After trying the rule-based system, in the second part of the task we tried to solve the same problem (DDI) using machine learning techniques. Our main objective on this part was to further improve the performance of the rule-based system by overcoming the limitations characterizing it.
From a ML point of view, the relation extraction problem can be considered as a classical classification task, were the objects to be classified are tuples of the type *(text, entity1, entity2)*, which should be encoded as feature vectors. In particular, the machine learning approach to solve the DDI task consists on the following main steps:

- Extract and define different features to be used to spot the interactions between pairs of entities and classify them in one of the four possible types (mechanism, advise, effect, int)
- Train a ML model with the obtained feature vectors
- Perform feature engineering and hyperparameter selection by using the Devel dataset
- Evaluate the final obtained model on the Test dataset

3.1 Selected algorithm

At the beginning we focused our attention on the feature engineering part by testing the "goodness" of the various features combinations on the Devel dataset and training a ML model, i.e. the *megam*, which is based on maximum entropy.

Once the features were defined, we focused our attention on the hyperparameter tuning of the *megam*. Specifically, in order to look for the best possible parameters, we decided to apply a *Grid Search* approach. To do so, we analyzed the parameters of the *megam* algorithm, which implementation was provided by the *MEGA Model optimization package* (link to the documentation). In particular, the parameters that the model allows to apply are various, but we decided to focus our attention only on those related to our problem (multiclass classification):

- *maxi int* :maximum number of iterations (default: 100).
- *dpp float* : minimum change in perplexity (default: -99999).
- *memory int* : memory size for LM-BFGS (multiclass only) (default: 5).
- *lambda float* : precision of the Gaussian prior (default: 1).
- *tune*: Tune lambda using repeated optimizations (starts with specified lambda value and drops by half each time until optimal dev error rate is achieved).
- *repeat int* : Repeat optimization int times (sometimes useful because megam thinks it converges before it actually does).

Considering the huge number of possible models we could generate combining the different parameters with different values, we decided to train the model tuning one parameter at a time, to understand which values could be useful to improve the performances. For example, we trained the model tuning the *maxi int* , leaving the other parameters with the default values. What we observed is that for certain values of the parameter, the performances of the model dropped significantly with respect to the one trained with default parameters (around 30%) and therefore we decided to not consider that value in the grid search. We applied this reasoning for all the parameters to tune, reducing the total number of models to train. For example, we did so with the *maxi* parameter, training different models. As can be observed from the table 1, for specific values of the *maxi* parameter, we observe a severe drop in the F1 score, therefore such values won't be considered when applying the grid search.

Models Comparison	
maxi	**Model Performance (F1)**
10000	**62.4%**
1000 (D)	**62.4%**
100	**61.6%**
10	**42.5%**
200	**62.2%**
20	**57.0%**
500	**62.4%**
50	**59.1%**

Table 1: Performance of models with different values for *maxi* parameter

As stated before, for *maxi* equal to 10 the F1 score drops to 42.5%, therefore it's not worth using it in the grid search.

Notice that the obtained performances exceeded significantly the ones of the default model (around 47% of F1 on devel) because we had already done some feature extraction (our default MEM model with these features had around 60% F1 score on the Devel set).

The *megam* model allows to specifiy other parameters, like *tune*, which don't assume any specific values, instead, they trigger certain functions when specified in the list of parameters. For example, in the case of the *tune* parameter, when specified, the algorithm tunes the lambda value using repeated optimizations (starts with specified -lambda value and drops by half each time until optimal dev error rate is achieved). Tuning this kind of parameters, require to double the number of models each time (either the parameter is applied or not). Therefore, to reduce the computation effort, we trained several models applying these kind of parameters, and then we trained the same models without specifying them. In the case of the *tune*, we observed an overall improvement in performances everytime it was applied. Therefore, instead of using it in the list of parameters to tune with the grid search, we directly decided to apply it by default on every model.

After this preliminary analysis, we concluded that the main parameters to focus on when performing the hyperparameter tuning were:

- *maxi* with values: 100, 500, 1000
- *dpp* with values: -99999, -9999, -999
- *memory* with values: 5, 50, 500
- *lambda_value* with values: 1, 10, 100

After having completed the feature extraction process (which will allow us to achieve a F1 score of 62.9% on the Devel, as we will see in the next section), we implemented the code for the grid search. The *MEGA Model Optimization Package* allows to specify the parameters for the model in the command line. Therefore, to apply the grid search we needed to work inside a bash file. First, we wrote a simple python script, in which we specified different lists, containing the values of the parameters we wanted to tune. The script then loops through them in a nested way and write in a text file the sequence of parameters with their respective value (in the following format *-maxi int -dpp float -memory int lambda float* changing line for each sequence. After the .txt file has been created, we extended the *run.sh* file, in order to perform the grid search. First, we read all the lines in the text file, and for each of them, we trained a model with that values for the parameters, saving the model in a specific folder. Then, each model is read and used in the *predict-mem.py* file. For each model, the script writes a different *.out* file which is finally used by the *evaluator.py* file to compute the performances for each model, which are appended one after the other in the *devel.stats* file. In the end we generated 81 different models.

The best performing model obtained with the grid search and after the feature engineering phase achieves a performance of 63.3% F1 on the devel and of 64.0% F1 on the test with the following parameters:

- *maxi*: 100
- *dpp*: -99999
- *memory*: 50
- *lambda_value*: 10

The complete statistics of our final and best performing model overall are the ones showed in figure 2.

What we can see in the figure is that the performance in test is very very high and this clearly shows that the obtained model generalizes very very well.

advise	109	82	32	191	141	57.1%	77.3%	65.7%
effect	162	61	150	223	312	72.6%	51.9%	60.6%
int	16	3	12	19	28	84.2%	57.1%	68.1%
mechanism	150	100	111	250	261	60.0%	57.5%	58.7%
M.avg	-	-	-	-	-	68.5%	61.0%	63.3%
m.avg	437	246	305	683	742	64.0%	58.9%	61.3%
m.avg(no class)	479	204	263	683	742	70.1%	64.6%	67.2%

	tp	fp	fn	#pred	#exp	P	R	F1
advise	115	61	94	176	209	65.3%	55.0%	59.7%
effect	169	90	117	259	286	65.3%	59.1%	62.0%
int	17	0	8	17	25	100.0%	68.0%	81.0%
mechanism	195	199	145	394	340	49.5%	57.4%	53.1%
M.avg	-	-	-	-	-	70.0%	59.9%	64.0%
m.avg	496	350	364	846	860	58.6%	57.7%	58.1%
m.avg(no class)	543	303	317	846	860	64.2%	63.1%	63.7%

Figure 2: Best performing model obtained with grid search (after feature selection) on the Devel set (on the left) and on the Test set (on the right)

3.2 Feature extraction

As briefly explained before, we first focused our attention on the feature engineering task: we tried to build several features able to encode the information contained in each pair and in the corresponding dependency tree. In this first phase we evaluated the goodness of the various extracted features mainly focusing on the F1 score obtained in the Devel dataset.

3.2.1 Tried features

We list here the various features we tried and the performance of the classifier trained over these features (on the Train dataset) evaluated on the Devel dataset (in terms of F1 score, which was around 47% at the beginning).
We tried the following features, in order:

- We firstly modified the already provided features regarding the tokens in between the 2 entities, by adding features regarding the POS tag and the REL, but we didn't obtain any improvement.
- Feature checking whether both entities are the same lemma (taken from *patterns.py*).
  F1: 49.8%
- We then added (incrementally) features checking whether the entities in the pair are parents of each other; the best result was obtained after adding both features.
  F1: 50.5%
- Feature checking whether the 2 entities have the same parent. We tried several encodings for this feature: the best one is the one encoding also the tag of the parent. We also tried to add a feature checking if the parent is a verb or not, but it didn't bring any further improvement.
  F1: 50.5%
- Presence of an entity in between the pair.
  F1: 51.7%
- Lemma, tag, word, rel of the parents of the entities in the pair: no improvement
- Number of tokens before tkE1 and after tkE2 (when added singularly there is no improvement, so we used both of them).
  F1: 52.7%
- Word, lemma, tag for the 2 tokens in the pair. We tried also a feature encoding the length of the word and the lemma. No improvement

- Word, lemma and tag for the tokens before tkE1 and for tokens following tkE2.
  F1: 58.5%
- Features encoding the presence of enities before tkE1 and after tkE2. We also tried to encode the lemma of the identified entity but this didn't improve the performance.
  F1: 58.9%
- Word, lemma, tag, rel of the LCS: no improvement.
- Check if LCS is entity: no improvement
- Check if LCS is root of the dependency tree.
  F1: 59.1%
- Added feature regarding the *check_LCS_svo* rule from *patterns.py*.
  F1: 59.6%
- *check_wib* from *patterns.py*: no improvement
- Added feature about the paths from tkE1 to the LCS and from tkE2 to LCS, containing only tags of the encountered tokens: no improvement
- Check presence of entities in path from tkE1 to LCS and from LCS to tkE2: no improvement
- Added features encoding the type of entities in the pair: no improvement
- We added a feature encoding the type of the entity also in case of found entities in between the 2 tokens, after tkE1, before tkE1, in the path from tkE1 to LCS, in the path from LCS to tkE2: no improvement
- Number of tokens in between the 2 entities: no improvement
- Features encoding the pair of lemmas, tags and words of the 2 entities of the pair.
  F1: 60.0%
- Added feature encoding the condensed path from tkE1 to LCS and from LCS to tkE2 (containing only the rel of the last elements and the lemma of the lcs).
  F1: 60.2%
- Length of the paths from tkE1 to LCS and from LCS to tkE2 and total length: no improvement
- *check_should_advise* rule from *patterns.py*.
  F1: 60.5%
- *check_parent_e1* and *check_parent_e2* from *patterns.py*: no improvement
- Path encoding the eventual entities from tkE1 to LCS and from LCS to tkE1: no improvement
- *check_group_after_brand* from *patterns.py*: no improvement
- We then added a list of *clue* verbs for each type of interaction. The considered verbs are the ones we found in the rule-based approach in the *check_LCS_svo* rule. After that, we added features checking the presence of clue verbs in between tkE1 and tkE2, before tkE1, after tkE2, in the paths from tkE1 to LCS and from LCS to tkE2.
  F1: 60.6%

We would like to notice that the results produced by the *Megam* optimizer were affected by a lot of fluctuations: the effect of adding a feature was therefore difficult to be interpreted with a single training of the algorithm. For that reason, at the very end we tried to add the features that didn't work and we tried also several combinations of features. The final list of features we selected is shown in the section below, and they allowed us to further improve the F1 score of the model up to 62.9% (without hyperparameter tuning), The final results on devel and test of the best performing model will be shown in the following sections.

3.3 Code

We include here the code of the *extract_features()* function, which converts a pair of drugs and their context into a feature vector, according to the features we described above.

```
# clue verbs of each interaction
effect_list = ['diminish','augment','exhibit','experience','counteract','potentiate','enhance','reduce','antagonize', 'include', 'block']
mechanism_list = ['impair','inhibit','displace','accelerate','bind','induce','decrease','elevate','delay', 'increase', 'indicate', 'produce']
advise_list = ['exceed']
int_list = ['suggest']
clue_verbs = effect_list+mechanism_list+advise_list+int_list

## -------------------
## -- 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'])

   if tkE1 is not None and tkE2 is not None:

      # num of tokens in between
      feats.add("ntokens_in_bt="+str(tkE2 - tkE1))

      # features for tkE1
      feats.add('tkE1_word='+tree.get_word(tkE1))
      feats.add('tkE1_lemma='+tree.get_lemma(tkE1).lower())
      feats.add('tkE1_tag='+tree.get_tag(tkE1))
      feats.add('tkE1_word_lenght'+str(len(tree.get_word(tkE1))))
      feats.add('tkE1_lemma_lenght'+str(len(tree.get_lemma(tkE1))))

      # features for tkE2
      feats.add('tkE2_word='+tree.get_word(tkE2))
      feats.add('tkE2_lemma='+tree.get_lemma(tkE2).lower())
      feats.add('tkE2_tag='+tree.get_tag(tkE2))
      feats.add('tkE2_word_lenght'+str(len(tree.get_word(tkE2))))
      feats.add('tkE2_lemma_lenght'+str(len(tree.get_lemma(tkE2))))
```

```

    # 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)
          rel = tree.get_rel(tk)
          feats.add("lib=" + lemma)
          feats.add("wib=" + word)
          feats.add("rib=" + rel)
          feats.add("tib=" + tag)
          feats.add("lpib=" + lemma + "_" + tag)

          # check clue verb in between
          if tag == "VB" and lemma in clue_verbs:
             feats.add("clue_verb_ib="+lemma)

          # entity in between tkE1 and tkE2
          if tree.is_entity(tk, entities) :
             feats.add("eib")

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

          # clue verb before tkE1
          if tag == "VB" and lemma in clue_verbs:
             feats.add("clue_verb_before="+lemma)

          # entity before tkE1
          if tree.is_entity(tk, entities):
             feats.add("ebf")

    # features for tokens after tkE2
    for tk in range(tkE2, tree.get_n_nodes()):
       if not tree.is_stopword(tk):
          word  = tree.get_word(tk)
          lemma = tree.get_lemma(tk).lower()
          tag = tree.get_tag(tk)
          feats.add("l_after=" + lemma)
          feats.add("w_after=" + word)
          feats.add("lp_after=" + lemma + "_" + tag)

          # clue verb after tkE2
          if tag == "VB" and lemma in clue_verbs:
```

```
                feats.add("clue_verb_after="+lemma)

            # entity after E2
            if tree.is_entity(tk, entities):
                feats.add("eaf")

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

    word  = tree.get_word(lcs)
    lemma = tree.get_lemma(lcs).lower()
    tag = tree.get_tag(lcs)
    rel = tree.get_rel(lcs)
    feats.add("LCS_w=" + word)
    feats.add("LCS_l=" + lemma)
    feats.add("LCS_tag=" + tag)
    feats.add("LCS_rel=" + rel)
    feats.add("LCS_ l_t=" + lemma + "_" + tag)

    # LCS is entity
    if tree.is_entity(lcs, entities):
        feats.add("lcs_entity")

    # LCS is ROOT
    if lcs == "ROOT":
        feats.add("lcs_root")

    # features for both entities
    feats.add("lemma_pair="+"_".join
        (sorted([tree.get_lemma(tkE1).lower(),tree.get_lemma(tkE2).lower()])))
    feats.add("tag_pair="+"_".join
        (sorted([tree.get_tag(tkE1),tree.get_tag(tkE2)])))
    feats.add("word_pair="+"_".join
        (sorted([tree.get_word(tkE1),tree.get_word(tkE2)])))

    # features about PATHS in the tree
    path1 = tree.get_up_path(tkE1,lcs)
    str_path1 = "<".join([tree.get_lemma(x).lower()+"_"+tree.get_rel(x) for x in path1])
    # path containing only tags
    str_path1_tags = "<".join([tree.get_tag(x) for x in path1])
    feats.add("path1="+str_path1)
    feats.add("path1_tags="+str_path1_tags)

    path2 = tree.get_down_path(lcs,tkE2)
    str_path2 = ">".join([tree.get_lemma(x).lower()+"_"+tree.get_rel(x) for x in path2])
    str_path2_tags = ">".join([tree.get_tag(x) for x in path2])
    feats.add("path2="+str_path2)
    feats.add("pat2_tags="+str_path2_tags)

    path = str_path1+"<"+tree.get_lemma(lcs)+"_"+tree.get_rel(lcs)+">"+str_path2
```

```
        feats.add("path="+path)
        path_tags = str_path1_tags+"<"+tree.get_tag(lcs)+">"+str_path2_tags
        feats.add("path_tags="+path_tags)

        # entity inside path
        for tk in path1 + path2:
            if tree.is_entity(tk, entities):
                feats.add("entity_in_path="+tree.get_lemma(tk).lower())

        #condensed path (seen in class)
        if len(path1) > 0 and len(path2) > 0:
            condensed_path = tree.get_rel(path1[-1])+"*<"
            condensed_path += tree.get_lemma(lcs).lower()+">*"+tree.get_rel(path2[0])
            feats.add("condensed_path="+condensed_path)

        # path wiht entities inside
        path_Entity = ""
        for tk in path1:
            path_Entity += ("Entity/"+tree.get_rel(tk) if tree.is_entity(tk, entities) else tree.get_rel(tk))+"<"
        path_Entity += tree.get_rel(lcs)
        for tk in path2:
            path_Entity+= ">"+("Entity/"+tree.get_rel(tk) if tree.is_entity(tk, entities) else tree.get_rel(tk))
        feats.add("pathEntity="+path_Entity)

        # lengths of the paths
        feats.add("path1_len="+str(len(path1)))
        feats.add("path2_len="+str(len(path2)))
        feats.add("tot_len="+str(len(path1)+len(path2)+(1 if lcs != "ROOT" else 0)))

        # features about parent entities
        p1 = tree.get_parent(tkE1)
        p2 = tree.get_parent(tkE2)

        # parent of E1
        if p1 is not None:
            lemma = tree.get_lemma(p1).lower()
            tag = tree.get_tag(p1)
            feats.add("lp1=" + lemma)
            feats.add("tp1=" + tag)

        # parent of E2
        if p2 is not None:
            lemma = tree.get_lemma(p2).lower()
            tag = tree.get_tag(p2)
            feats.add("lp2=" + lemma)
            feats.add("tp2=" + tag)

        # same entities in the pair
        if tree.get_lemma(tkE1) == tree.get_lemma(tkE2):
            feats.add("same_lemma")
```

```python

        # e1 under e2
        if tkE2 == tree.get_parent(tkE1):
           feats.add("e1_under_e2")

        # e2 under e1
        if tkE1 == tree.get_parent(tkE2):
           feats.add("e2_under_e1")

        # same parent
        if p1 == p2 and (p1 is not None):
           tag = tree.get_tag(p1)
           feats.add("same_parent_tag=" + tag)

        # LCS is a verb with a 'should' child
        should_advise_rule = patterns.check_should_advise(tree, tkE1, tkE2)
        if should_advise_rule:
           feats.add("should_advise_rule")


    return feats
```

3.4 Experiments and results

We report below the statistics regarding the performance obtained on the devel and on test datasets for what regards our best performing model obtained after doing feature engineering, before performing the grid search (figure 3).

	tp	fp	fn	#pred	#exp	P	R	F1
advise	106	74	35	180	141	58.9%	75.2%	66.0%
effect	160	69	152	229	312	69.9%	51.3%	59.1%
int	16	3	12	19	28	84.2%	57.1%	68.1%
mechanism	147	97	114	244	261	60.2%	56.3%	58.2%
M.avg	-	-	-	-	-	68.3%	60.0%	62.9%
m.avg	429	243	313	672	742	63.8%	57.8%	60.7%
m.avg(no class)	474	198	268	672	742	70.5%	63.9%	67.0%

	tp	fp	fn	#pred	#exp	P	R	F1
advise	119	72	90	191	209	62.3%	56.9%	59.5%
effect	182	111	104	293	286	62.1%	63.6%	62.9%
int	18	10	7	28	25	64.3%	72.0%	67.9%
mechanism	201	241	139	442	340	45.5%	59.1%	51.4%
M.avg	-	-	-	-	-	58.5%	62.9%	60.4%
m.avg	520	434	340	954	860	54.5%	60.5%	57.3%
m.avg(no class)	574	380	286	954	860	60.2%	66.7%	63.3%

Figure 3: Best model performance on the Devel set (on the left) and on the Test set (on the right) after feature engineering

With respect to the rule-based approach, we can see that the learned model generalizes pretty well on the test data, indeed we have around 2.5% drop on the F1 performance, which is a very good result.

4 Conclusions

During this assignment we faced the DDI task by using 2 different types of approach, a rule-based one and a ML one. The rule-based approach, based on simple heuristic rules, allowed us to build a sort of baseline to be used to assess the performance of the ML algorithm we built in the 2nd phase of the task. By using a list of simple rules, we reached a quite low performance on the devel set and an even worse performance on the test set , but still the 45.7% of F1 score on the devel set represents a robust baseline performance, from which starting the development of ML models.

We then moved to a ML learning approach for doing classification: this approach required us quite more effort in feature engineering, which is fundamental in order to build useful features for the classification task at hand. We dedicated most of our effort to this phase, since we know that feature engineering is a fundamental and difficult step in a ML pipeline. We know that this problematic phase is nowaday overcomed by a data-driven feature engineering, done by means of DL which allows an end-to-end learning, i.e. the DL algorithm learns not only the classifier function built upon the features, but it also learns the right features in order to classify. In conclusion, we clearly saw that the ML solution significantly outperformed the baseline and it also generalizes very well on the test dataset, without any drop on the F1 score (maybe the train data is better represented by the test set than the devel set). In particular, the improvement can be seen on the DDI of type *int* which was the most difficult one on the rule-based system.