Neural network-based NERC and DDI detection

AHLT | MAI

Ramon Mateo Navarro and Benet Manzanares Salor
19/06/2022

Introduction

In this report we describe the work performed for the last two laboratories of the AHLT subject. These practical assignments consist of the construction of a Named Entity Recognition Classifier (NERC) and a Detect drug-drug interactions (DDI) detection system using Neural Networks (NN). In the following, we detail and justify our decisions regarding models' inputs, architectures and training settings 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.

NERC

For this laboratory, we had to define a neural network approach for the NERC task. In particular, our task was to improve a predefined baseline of 51% F1 score for arriving to a close to 70% F1 score in the development set. Next subsections describe our approach for this improvement and partial and final obtained results in both devel and test sets.

Our approach

For developing our approach, we incrementally tried different configurations of inputs, architectures and training settings (equivalent to training phases). In other words, architecture was only modified when the best set of inputs is defined and training settings were modified when the best architecture is determined. For evaluating each setting, the average of a minimum of three executions is computed, reducing the random-based variability of neural networks training. Additionally, maximum sequence length was reduced to 40, noting that no longer sequences are present in the data. In the following, our development decisions are detailed.

Inputs:

- Prefixes with length 5: **54.9% [NEW BEST]**
    - Suffixes and prefixes of length 3: **55.3% [NEW BEST]**
- Part-of-Speech: 23.9%
- Lowercase instead of cased words: **57.9% [NEW BEST]**
- additional_features {upper}{alpha}{digits}{dash}{brackets}: **62.1% [NEW BEST]**
  The *{upper}{alpha}{digits}{dash}{brackets} stands for a set of 5 boolean variables that are concatenated for forming a string. {upper} means if all the word is uppercase, {alpha} that only contains alphabetic characters and the rest stand for symbols.*
- additional_features {upper}{alpha}{digits}{dash}{brackets}_{drugbank_class}: **65.6% [NEW BEST]**

Drugbank data is collected and added to the additional_features string. If the word is in the drugbank, its class is concatenated; otherwise, a "NO" is appended.

- Drugbank as a separated input: **66.3% [NEW BEST]**
  Considering the success of adding the drugbank class to the additional features, it was employed as a single input, with the corresponding embedding.
- Drugbank as a separated input but with cased words: 60.9%

Architecture:

- GRU instead of LSTM: 64.2%
- Chaninging embedding and recurrent dropouts to 0.2: 64.9%
- LSTM 250 units: 66.1%
  - LSTM 100 units: 64.6%
- Embedding sizes [50, 25, 25, 25]: 65%
- LSTM 100 units and embedding sizes [50, 25, 25, 25]: **66.2% [NEW BEST]**
  Despite no F1 improvement is found, since model size is significantly reduced (divided by half), this is considered the new best. This complexity reduction should also lead to less overfitting.
- Adding {length} to additional_features: **67.8% [NEW BEST]**
  The length attribute was employed due to the success that provided in the feature-based approach from a previous laboratory. It seems that the intuition that drugs names are usually long words also benefits the neural network approach.
- {length>10} at additional_features: 67.1%
  Aiming to reduce the granularity of the approach, length was aimed to be substituted by a boolean variable using a threshold. Similar to the case of feature-based model, no benefit was observed.
- {length//2} at additional_features: 66.3%
  For a more subtle granularity reduction, length is divided into 2 and cast to integer.
- Using LSTM 200 units and embedding sizes [100, 50, 50, 50]: 67.6%

Training settings:

- Batch size = 16: **68.3% [NEW BEST]**
- Batch size = 8: 67.3%
- Adam with amsgrad: 68.2%
- L1 Regularization of Dense layer: 23.8%
- Change optimizer to RMSProp: **70.1% [NEW BEST]**
- Changing class weights at loss to = [0, 4, 10, 7, 10, 6, 5, 6, 5, 7]**:** 60.8%
  With the target of reducing the class imbalance problem, weight were adjusted manually for providing more weight to the less frequent classes manually.
- Inversely proportional to labels frequencies = [0, 1152, 7075, 1683, 2627, 56, 104876, 479, 242, 530]: 56.9%
  Following the previous approach, we employed the inversion (1/N) of labels frequencies from the training set as weights.

Code

For facilitating the task of adding, removing and modifying input features, we slightly modified the Codemaps class for being parameterized by the *codes_settings* dictionary (shown below, declared in the *__init__* method). The behavior of the class is exactly the same (obtaining vocabularies, indexes and encodings) but employing this dictionary. Other dictionaries such as *code_indexes* (that stores the indexes of each input) are also defined to this end.

```
self.codes_settings = {
```

```
            "WORD": lambda tk: tk['lc_form'],
            "SUFFIX": lambda tk: tk['lc_form'][-self.suflen:],
            "PREFIX": lambda tk: tk['lc_form'][:self.suflen],
            "ADDITIONAL_FEATURE": self.get_additional_features,
            "DRUGBANK": lambda tk: self.drugbank.get(tk['form'], "NO"),
        }
```

As can be observed, we also defined the *get_additional_features* method for simplifying the definition of this input.

```
 def get_additional_features(self, tk):
         upper = tk['form'].isupper()
         alpha = tk['form'].isalpha()
         digits = bool(re.search(r'\d', tk['form']))
         dash = bool(re.search(r'-|_', tk['form']))
         brackets = bool(re.search(r"(\(|\[|\{)\w+(\)|\]|\})", tk['form']))
         length = len(tk['form'])
    # sizes
         return f"{upper}_{alpha}_{digits}_{dash}_{brackets}_{length}"
```

We also added the *get_codes_sizes_dict* method, employed for building the network.

```
def get_codes_sizes_dict(self):
         return {name:len(code_indexes) + 2 for name, code_indexes in self.codes_indexes.items()}    #
+2 for padding and unknown
```

Building network

This is the final code to generate the final network. As can be observed, the *get_codes_sizes_dict* method simplifies the definition of the network embeddings, not requiring from manually changing the code each time the set of inputs is modified.

```
 def build_network(codes):
    codes_sizes = codes.get_codes_sizes_dict()
    # Declare inputs, embedding, and dropout layers
    inputs = []
    drops = []
    embeddings_dims = {name: 25 for name in codes_sizes.keys()}
    embeddings_dims["WORD"] = 50 # Exception for word
    for name, code_size in codes_sizes.items():
       inpt = Input(shape=(codes.maxlen,))
       inputs.append(inpt)
       embedding = Embedding(input_dim=code_size, output_dim=embeddings_dims[name],
                    input_length=codes.maxlen, mask_zero=True)(inpt)
       drop = Dropout(0.1)(embedding)
       drops.append(drop)
    # Concatenate drops
    drops = concatenate(drops)
    # biLSTM
    bilstm = Bidirectional(LSTM(units=100, return_sequences=True,
                           recurrent_dropout=0.1))(drops)
    # output softmax layer
    n_labels = codes.get_n_labels()
    out = TimeDistributed(Dense(n_labels, activation="softmax"))(bilstm)
    # build and compile model
    model = Model(inputs, out)
    model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

    return model
```

Results

In this section, the best results obtained for each of the development phases are depicted.

As can be noted below, the main deficiencies of the baseline are in the classification of brand and drug_n entities, the less frequent ones. For brand, recall was the main reason for this, while for drug_n it was precision (but with also bad recall).

Baseline on devel:

```
                    tp      fp       fn    #pred    #exp     P        R        F1
------------------------------------------------------------------------------
brand               45       3      329      48      374     93.8%    12.0%    21.3%
drug              1604     123      302    1727     1906     92.9%    84.2%    88.3%
drug_n              13     118       32     131       45     9.9%     28.9%    14.8%
group              543     122      144     665      687     81.7%    79.0%    80.3%
------------------------------------------------------------------------------
M.avg             -       -        -        -        -       69.6%    51.0%    51.2%
------------------------------------------------------------------------------
m.avg             2205     366      807    2571     3012     85.8%    73.2%    79.0%
m.avg(no class)   2347     224      665    2571     3012     91.3%    77.9%    84.1%
```

After finding the best sets of inputs, it's possible to observe a big increase in the F1 macro average. This is mainly argued by an improvement in brand recall and drug_n precision. With this, a decent F1 is obtained for brand, but for drug_n keeps having a reduced recall.

Our approach on devel after input selection:

```
                    tp      fp       fn    #pred    #exp     P        R        F1
------------------------------------------------------------------------------
brand              234      31      140     265      374     88.3%    62.6%    73.2%
drug              1520     196      386    1716     1906     88.6%    79.7%    83.9%
drug_n               9       5       36      14       45     64.3%    20.0%    30.5%
group              531      95      156     626      687     84.8%    77.3%    80.9%
------------------------------------------------------------------------------
M.avg             -       -        -        -        -       81.5%    59.9%    67.1%
------------------------------------------------------------------------------
m.avg             2294     327      718    2621     3012     87.5%    76.2%    81.4%
m.avg(no class)   2341     280      671    2621     3012     89.3%    77.7%    83.1%
```

Architecture selection provided a more subtle but noticeable improvement. Precision and recall for the brand class are the main benefits from these changes, while these values for drug_n are significantly decreased.

Our approach on devel after architecture selection:

```
                    tp      fp       fn    #pred    #exp     P        R        F1
------------------------------------------------------------------------------
brand              286      27       88     313      374     91.4%    76.5%    83.3%
drug              1559     208      347    1767     1906     88.2%    81.8%    84.9%
drug_n               8       9       37      17       45     47.1%    17.8%    25.8%
group              542     116      145     658      687     82.4%    78.9%    80.6%
------------------------------------------------------------------------------
M.avg             -       -        -        -        -       77.3%    63.7%    68.6%
------------------------------------------------------------------------------
m.avg             2395     360      617    2755     3012     86.9%    79.5%    83.1%
m.avg(no class)   2446     309      566    2755     3012     88.8%    81.2%    84.8%
```

The upgrade provided by training settings selection principally affects the drug_n, improving its recall but reducing the precision.

Our approach on devel after training settings selection:

```
                    tp      fp       fn     #pred    #exp     P        R        F1
------------------------------------------------------------------------------------
brand              291      20       83      311      374     93.6%    77.8%    85.0%
drug              1579     192      327     1771     1906     89.2%    82.8%    85.9%
drug_n              10      15       35       25       45     40.0%    22.2%    28.6%
group              539     103      148      642      687     84.0%    78.5%    81.1%
------------------------------------------------------------------------------------
M.avg             -       -        -        -        -        76.7%    65.3%    70.1%
------------------------------------------------------------------------------------
m.avg             2419     330      593     2749     3012     88.0%    80.3%    84.0%
m.avg(no class)   2474     275      538     2749     3012     90.0%    82.1%    85.9%
```

Our approach on test:

```
                    tp      fp       fn     #pred    #exp     P        R        F1
------------------------------------------------------------------------------------
brand              226      57       48      283      274     79.9%    82.5%    81.1%
drug              1699     241      428     1940     2127     87.6%    79.9%    83.6%
drug_n               4      16       68       20       72     20.0%    5.6%     8.7%
group              505     212      188      717      693     70.4%    72.9%    71.6%
------------------------------------------------------------------------------------
M.avg             -       -        -        -        -        64.5%    60.2%    61.3%
------------------------------------------------------------------------------------
m.avg             2434     526      732     2960     3166     82.2%    76.9%    79.5%
m.avg(no class)   2552     408      614     2960     3166     86.2%    80.6%    83.3%
```

The finally obtained model has a significant low performance on test than on devel, reducing F1 score from 70.1% to 61.3%. This is probably caused to the usage of devel set for our decisions (overfitting our approach) and the employment of DrugBank, which was proved on previous laboratories to not be as useful for test that it was for devel. The main affected from this overfitting is the drug_n class, with half the precision and a recall of only 5.6%. Reason can be important differences between the cases present in both sets, not existing them in the DrugBank.

It's important to note that our comments above are mainly limited to explain the differences with the corresponding previous development phase because the effect of every change in neural networks is very difficult to predict.

DDI

In this laboratory, we had to implement the DDI detection system using a neural network model. Concretely, a predefined baseline need to be improved for obtaining a F1 score close to 65%. Next subsections describe our approach, code and results obtained.

Our approach

Similarly to the previous laboratory, we incrementally tried different configurations of inputs, architectures and training settings and the average of a minimum of three executions is computed for evaluating each decision. In the following, our development decisions are detailed.

Inputs:

- Using DistilBERT for pretrained embeddings: 0%
  As first step for inputs explorations, we tried to use DistilBERT as pretrained embedding

model (the following layer were identical to the baseline model). To this end, we employed the HuggingFace transformers library, and changed all the code to PyTorch (due to configuration problems, this was required for running on GPU locally). Nonetheless, class imbalance caused the model to be uncapable of nothing better than overfit to the no-interaction class, classifying all sentences as these.

  - Adding <DRUG1>, <DRUG2> and <DRUG_OTHER> tokens and unfreezing: 0%
    We tried to add these tokens and unfreezing the DistilBERT model, but difference was limited to a significanlty increased training time.
  - More complex and simple architecture: 0%
    Considering that maybe was the rest of the architecture the responsable of this problem, we tried to reduce and increase the number of layers, filters and kernel sizes, without any effect.
  - Class weighting: 20%
    Changing the lass weights in the loss function (with a 10 and 100 times lower weight for the no-interaction class) provided a 20% improvement, but by classifying all the sentences to the group class, the more frequent class after no-interaction. After that, we decided to dismiss the pretrained embeddings approach.
- Single inputs:
  - Lowercased: **50.4% [NEW BEST]**
  - Lemma: **51.0% [NEW BEST]**
  - PoS: 35.6%
- Multiple inputs:
  - Lowercased + Lemmas: **52.1% [NEW BEST]**
  - Lowercased + Lemmas + PoS: 46.7%
  - Lowercased + Lemmas + EntityType: 48.1%
    The entity type attribute found in the tokens dictionaries was employed. When no entity type was available, the sentinel "NO" was returned.
  - Lowercased + Lemmas + Prefixes(len=3) + Suffixes(len=3): 49.5%
  - Cased + Lowercased + Lemmas + PoS: 51.3%

Architecture:

- Embeddings sizes to the half (50, 25s): 50.3%
  We tried this approach based on its the effectiveness the previous laboratory, but it was not as effective.
- Layer 1 kernel size:
  Incrementing the kernel size of the first layer is more or less equivalent to incrementing the maximum range of dependencies/context that can be noted by the model. As can be observed, this provides a benefit up to a certain point. Odd values were used for observing the same context at the left and right side of the words.
  - Kernel size = 3: **53.2% [NEW BEST]**
  - Kernel size = 5: **56.0%, [NEW BEST]**
  - Kernel size = 7: 52.2%
- Inception-like convolutions:
  Inspired by Google's Inception model, we tried to perform convolutions with different kernel sizes at first layer and concatenate the obtained results. In this way, context/dependencies at multiple scales can be considered.
  - Combo sizes 3 and 5 (Inception-like): **57.8% [NEW BEST]**
  - Combo sizes 2, 3 and 5 (Inception-like): 56.3%

- Second layer:
    - Filters = 60: **59.7% [NEW BEST]**
    - Filters = 30: 57.4%
    - Filters = 20: 57.8%
    - Maxpooling after second layer: **60.8% [NEW BEST]**
    - Second layer combo sizes 3 and 5 (Inception-like): 57.8%
- Third layer + max pooling: 56.8%
- Layers filters inverted (l1=64, l2=32): 57.7%
- Adding dropout 0.1 before last dense: **61.2% [NEW BEST]**
- Dropout 0.1 at embeddings: **61.5% [NEW BEST]**
- Double the model size: **63.5% [NEW BEST]**
  For this scaling, all embeddings and convolutional filters dimensions were doubled.
- Triplicate model size: 55.6%
- BiLSTM:
  Multiple combinations of BiLSTM and CNN layers were tried, not obtaining improvement in any case.
    - BiLSTM 100 units after embeddings: 53.8%
        - Without second convolutional layer: 55.0%
        - BiLSTM 200 units: 56.3%
    - BiLSTM 100 units after first conv layer, without the second one: 58.4%
      For this approach, the outputs of the BiLSTM for each token were returned, flattened and introduced to last dense layer.
        - Only with last BiLSTM outputs: 58.7%
          In contrast to the previous approach, in this the BiLSTM layer only returned a single value, corresponding to the outputs for the last tokens (in both directions).
        - BiLSTM 200 units: 56.2%
- Attention:
  Considering the *nature of the task, we contemplated Attention mechanisms as a suitable method, placing an attention layer after the embeddings. However, none of the tried approaches lead to an increment of devel macroaverage F1 score.*
    - Directly using embeddings as query and values: 45.8%
    - Two CNN layers for obtaining the query and values from embeddings (kernel size=1, filters=100): 49.1%
        - Changing these layers settings (kernel size and filters): No major improvements
    - Previous, but concatenating attention results with original embeddings: 60.1%
      We hypotetized that *the attention was mainly destroying the embeddings. Concatenating its results to them, we keep the original embeddings but enable to use attention results when considers. Results comparable to those from the previous best model were obtained, but without further improvement.*
        - Changing embedding, query-value and posterior CNN layers settings: No improvements
    - Additional embedding layer for positional embedding: 55.3%
      As a way of enabling *the attention mechanism to consider positions, we defined an embedding layer receiving inputs positions and summed its output to the embeddings. This is inspired in the positional encoding from the well-known transformers, but allowing the embedding layer to customize this positional information.*

Training settings:

- Batch size:
  - Batch size = 16: 42.2%
    Catastrophic results of this change are probably due to class imbalance. With a reduced batch size is more probable to only have samples of the no-interaction class, leading to updates that overfit to this majority class.
  - Batch size = 64: 57.6%
- RMSprop instead of Adam: 53.4%
- Adam learning rate:
  - 0.0001: 51.7%
  - 0.01: 36.4%

As can be noted, no improvement was obtained with many of the tried architecture changes and with none of the modifications to training settings, sometimes having catastrophic effects. Subsequently, the *"Double the model size"* approach is the best founded.

Code

For the Codemaps class, the same utility-focused modification of the previous laboratory has been applied. Under this premise, the *codes_settings* dictionary is depicted below, including the not used (so commented) inputs.

```
self.codes_settings = {
           #"WORD": lambda tk: tk['form'],
           "LC_WORD": lambda tk: tk['lc_form'],
           "LEMMA": lambda tk: tk['lemma'],
           #"POS": lambda tk: tk['pos'],
           #"ETYPE": lambda tk: tk.get('etype', "NO"),
           #"PREF": lambda tk: tk['lc_form'][:3],
           #"SUFF": lambda tk: tk['lc_form'][-3:],
       }
```

In the following, the code of the build_network method is provided. Method *get_codes_sizes_dict* is employed identically to the case of the previous laboratory.

```
def build_network(codes):
   codes_sizes = codes.get_codes_sizes_dict()
   # Declare input and embedding layers
   inputs = []
   embeddings = []
   embeddings_dims = {name: 50 for name in codes_sizes.keys()}
   embeddings_dims["LC_WORD"] = 100 # Exception for lowercase word
   for name, code_size in codes_sizes.items():
      inpt = Input(shape=(codes.maxlen,))
      inputs.append(inpt)
      embedding = Embedding(input_dim=code_size, output_dim=embeddings_dims[name],
                            input_length=codes.maxlen, mask_zero=False)(inpt)
      embeddings.append(embedding)
   # Concatenate embeddings
   if len(embeddings) == 1:
      embeddings = embeddings[0]
   else:
      embeddings = concatenate(embeddings)
   # Embeddings dropout
   embeddings = Dropout(0.1)(embeddings)
```

```
    # First convolutional layer (inception-like)
    l1_conv1 = Conv1D(filters=30, kernel_size=5, strides=1, activation='relu',
 padding='same')(embeddings)
    l1_conv2 = Conv1D(filters=30, kernel_size=3, strides=1, activation='relu',
 padding='same')(embeddings)
    l1_conv = concatenate([l1_conv1, l1_conv2])
    l1_max = MaxPool1D(pool_size=2, strides=1)(l1_conv)
    # Second convolutional layer
    l2_conv1 = Conv1D(filters=60, kernel_size=5, strides=1, activation='relu',
 padding='same')(l1_max)
    l2_max = MaxPool1D(pool_size=2, strides=1)(l2_conv1)
    # Flatten and dropout
    flat = Flatten()(l2_max)
    flat = Dropout(0.1)(flat)
    # Output softmax layers
    n_labels = codes.get_n_labels()
    out = Dense(n_labels, activation='softmax')(flat)
    # Declare and compile model
    model = Model(inputs, out)
    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Results

In the following, the best results for each development phase are depicted.

Result on devel with baseline approach:

```
                   tp     fp      fn   #pred    #exp    P       R       F1
------------------------------------------------------------------------------
advise             82     60      59     142     141    57.7%   58.2%   58.0%
effect            143     69     169     212     312    67.5%   45.8%   54.6%
int                 8      0      20       8      28    100.0%  28.6%   44.4%
mechanism          75    195     186     270     261    27.8%   28.7%   28.2%
------------------------------------------------------------------------------
M.avg            -      -       -       -       -       63.2%   40.3%   46.3%
------------------------------------------------------------------------------
m.avg             308    324     434     632     742    48.7%   41.5%   44.8%
m.avg(no class)   332    300     410     632     742    52.5%   44.7%   48.3%
```

Baseline shortages are mainly for the mechanism and int sections. As an anomaly, int precision is 100% in devel, probably due to the reduced amount of samples and/or its clear differentiation from the rest of classes.

Results on devel after inputs selection:

```
                   tp     fp      fn   #pred    #exp    P       R       F1
------------------------------------------------------------------------------
advise             80     55      61     135     141    59.3%   56.7%   58.0%
effect            137     70     175     207     312    66.2%   43.9%   52.8%
int                13      0      15      13      28    100.0%  46.4%   63.4%
mechanism          79    120     182     199     261    39.7%   30.3%   34.3%
------------------------------------------------------------------------------
M.avg            -      -       -       -       -       66.3%   44.3%   52.1%
------------------------------------------------------------------------------
m.avg             309    245     433     554     742    55.8%   41.6%   47.7%
m.avg(no class)   341    213     401     554     742    61.6%   46.0%   52.6%
```

Inputs selection provided an improvement in mechanism recall and precision and int recall.

Results on devel after architecture selection:

```
                    tp      fp       fn     #pred    #exp     P        R        F1
------------------------------------------------------------------------------
advise              83      57       58      140      141     59.3%    58.9%    59.1%
effect             205     123      107      328      312     62.5%    65.7%    64.1%
int                 19       0        9       19       28     100.0%   67.9%    80.9%
mechanism          149     188      112      337      261     44.2%    57.1%    49.8%
------------------------------------------------------------------------------
M.avg             -       -        -        -        -        66.5%    62.4%    63.5%
------------------------------------------------------------------------------
m.avg              456     368      286      824      742     55.3%    61.5%    58.2%
m.avg(no class)    521     303      221      824      742     63.2%    70.2%    66.5%
```

Changing the architecture were capable of improving the recall for all the classes, but not the precision. Concretely, except advise, the rest obtained an increment greater than 20% in recall.

Results on test after architecture selection:

```
                    tp      fp       fn     #pred    #exp     P        R        F1
------------------------------------------------------------------------------
advise             101      39      108      140      209     72.1%    48.3%    57.9%
effect             180     175      106      355      286     50.7%    62.9%    56.2%
int                  3       0       22        3       25     100.0%   12.0%    21.4%
mechanism          161     190      179      351      340     45.9%    47.4%    46.6%
------------------------------------------------------------------------------
M.avg             -       -        -        -        -        67.2%    42.7%    45.5%
------------------------------------------------------------------------------
m.avg              445     404      415      849      860     52.4%    51.7%    52.1%
m.avg(no class)    513     336      347      849      860     60.4%    59.7%    60.0%
```

As can be noted, model performance on test set is much worse than in devel, with a reduction of recall specially notorious for int. This is coherent with its low frequency, since only a small amount of possible cases has been observed during traing. However, these cases should be clearly distinguished, since again a 100% of precision is obtained. The precision for all the rest of classes is reduced with exception for advice, that increases more than a 10%.

Conclusions

In this assignment, we have explored both the NERC and DDI detection tasks using neural networks. Different architectures based on BiLSTM and CNNs have been proposed, and pretrained embeddings from DistilBERT and attention mechanisms have been explored. Development difficulties have been mainly due to neural network peculiarities. On one hand, execution-based performance variations required the repetition for evaluating model's capabilities, leading to non-negligible runtimes. On the other hand, effects of setting changes are almost unpredictable. This makes development process very frustrating, especially considering the class imbalance problem of the presented tasks, that causes many changes to have catastrophic outcomes. As a consequence, macro-average F1 score target on devel was only achieved for the NERC task (obtaining a 70.1%), failing to achieve the desired 65% for the DDI detection problem (reaching a maximum of 63.5%). Most of the work has been dedicated to the latter problem, trying different techniques without success. On test set, obtained models show an important overfitting, with a 61.3% and 45.5% F1 score for NERC and DDI detection, respectively. These results are outperformed by the simpler feature-based approaches designed by us in previous laboratories (66.1% for NERC and 60% for DDI detection). On this basis, the feature-based method can be considered a suitable approach for obtaining good results (or at least a baseline) for the presented tasks in a fast way; prior to application of the more complex neural networks that may have complications obtaining better results.