NER systems report

AHLT

Ramon Mateo Navarro
Benet Manzanares Salor

27/03/2022

Introduction

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

Rule-based baseline

For the first laboratory, we had to define a rule-based approach to act as baseline for the NER task. In particular, our task was to construct the rule set for single token classification and allow concatenating multiple tokens classified with the same class as a multi-token detection.

Rule set construction

For defining the rule set, first we improved the average F1 by removing the upper-case rule and suffixes rules from the initial code, keeping only that which makes usage of a dictionary based on *external* resources.
Subsequently, we implemented the code for detecting multi-token entities, using consecutive tokens belonging to the same class.
After that, we developed the explore.sh script. This script receives a class name as parameter (drug, drug_n, brand or group) and returns the tokens of this class found in the training data (using the *ner2gold.py* script) along with the amount of occurrences, sorted by this occurrences count.
Our first approach was to extract rules of this information, such as the most frequent suffixes and prefixes per class. Nevertheless, this doesn't provide good enough results, improving recall but with seriously low precision for all the classes. Under this premise, we decided to (instead of manually defining rules) create a token-to-class dictionary such as the *external* one, called *our_resources*, using the output of the explore.sh script. The output of the script required an additional step performed in the *preprocessing.py* script, that removes repeated

words between classes. The usage of *our_resources* significantly improved the average F1 by increasing the recalls, but again with low precisions. For improving precision, we performed a high pass filter in the construction of the resources dictionary based on the number of occurrences. In this way, since we only consider the most frequent tokens of each class, false positives are reduced. The percentage of tokens discarded for each class was tuned to maximize the average F1 at development set. Concretely, the 20% and 50% of the tokens belonging to the drug and drug_n classes were discarded, respectively. This implies that for these classes, many tokens not always belong to that specific class (maybe they don't belong to any class). On the other hand, a zero percent of tokens from both the brand and group classes were discarded, implying that for these most of the tokens only belong to that class. With this change we obtain the final version of our implementation.

Code

Our code can be executed with the *run.sh* script, which obtains the results for both the devel and test sets. In the following we depict the main functions of our implementation (contained in the *baseline-NER.py* script).

extract_entities function:

```
## -------------------- Entity extractor -------------------- ##
## -- Extract drug entities from given text and return them as
## -- a list of dictionaries with keys "offset", "text", and "type"
## -- EXTENDED for detecting multi-token entites
def extract_entities(stext, use_multi_token=False):

    # tokenize text
    tokens = tokenize(stext)

    result = []
    # classify each token and decide whether it is an entity.
    last_drug_type = None
    last_token_start = -1
    last_token_end = -1
    in_multi_token = False
    for (token_txt, token_start, token_end)  in tokens:
        drug_type = classify_token(token_txt)

        if drug_type != "NONE":
            if not use_multi_token:
                e = { "offset" : str(token_start)+"-"+str(token_end),
                      "text" : stext[token_start:token_end+1],
                      "type" : drug_type
                    }
                result.append(e)
            # If using multi-token
            else:
                if in_multi_token:
                    if drug_type == last_drug_type:
                        last_token_end = token_end
                    else:
                        # Add the previous multi-token
                        e = { "offset" :
str(last_token_start)+"-"+str(last_token_end),
                              "text" : stext[last_token_start:last_token_end+1],
```

```
                            "type" : last_drug_type
                           }
                    result.append(e)
                    # Prepare for the next multi-token
                    last_drug_type = drug_type
                    last_token_start = token_start
                    last_token_end = token_end
            else:
                # Prepare for the next multi-token
                last_drug_type = drug_type
                last_token_start = token_start
                last_token_end = token_end
                in_multi_token = True
        elif use_multi_token:
            if in_multi_token:
                # Multi-token stops
                in_multi_token = False
                # Add the previous multi-token
                e = { "offset" : str(last_token_start)+"-"+str(last_token_end),
                      "text" : stext[last_token_start:last_token_end+1],
                      "type" : last_drug_type
                     }
                result.append(e)

    # Add the last multi-token
    if in_multi_token:
        e = { "offset" : str(last_token_start)+"-"+str(last_token_end),
              "text" : stext[last_token_start:last_token_end+1],
              "type" : last_drug_type
             }
        result.append(e)


    return result
```

classify_token function (called from extract_entities):

```
## -------------------- Classify token ---------------------- ##
## -- check if a token is a drug part, and of which type
def classify_token(txt):
    txt_lower = txt.lower()

    if txt_lower in our_resources: return our_resources[txt_lower]
    elif txt_lower in external: return external[txt_lower]
    else: return "NONE"
```

Experiments and results

For evaluating the effect of every decision, we have performed multiple experiments during the implementation. For these experiments, only the results on the devel set were observed, avoiding “overfitting” ourselves to the test set. In the following, the most important experiments and its results are depicted. Finally, we show the results of our final implementation in the test set.

First results correspond to the version of the system only using the external resources and without multi-token.

Results on devel using external resources, without multi-token:

	tp	fp	fn	#pred	#exp	P	R	F1
brand	319	252	55	571	374	55.9%	85.3%	67.5%
drug	1588	305	318	1893	1906	83.9%	83.3%	83.6%
drug_n	0	0	45	0	45	0.0%	0.0%	0.0%
group	137	282	550	419	687	32.7%	19.9%	24.8%
M.avg	-	-	-	-	-	43.1%	47.1%	44.0%
m.avg	2044	839	968	2883	3012	70.9%	67.9%	69.3%
m.avg(no class)	2191	692	821	2883	3012	76.0%	72.7%	74.3%

Afterwards, we add our resources without filtering. As can be noted, it improved significantly the average F1 score, even being capable of detecting some tokens for the drug_n class. Nevertheless, the precision for drug_n is very low, and the precision for drug and group classes decreased more than a 5%.

Results on devel using external and our resources, without multi-token:

	tp	fp	fn	#pred	#exp	P	R	F1
brand	320	102	54	422	374	75.8%	85.6%	80.4%
drug	1651	590	255	2241	1906	73.7%	86.6%	79.6%
drug_n	6	51	39	57	45	10.5%	13.3%	11.8%
group	309	923	378	1232	687	25.1%	45.0%	32.2%
M.avg	-	-	-	-	-	46.3%	57.6%	51.0%
m.avg	2286	1666	726	3952	3012	57.8%	75.9%	65.7%
m.avg(no class)	2383	1569	629	3952	3012	60.3%	79.1%	68.4%

Under this premise, we performed a filtering of our resources searching to maximize the average F1. This filtering was only effective for drug and drug_n classes, not being possible to filter tokens from the brand and group classes without reducing the average F1.

Results on devel using external and our filtered resources, without multi-token:

	tp	fp	fn	#pred	#exp	P	R	F1
brand	320	118	54	438	374	73.1%	85.6%	78.8%
drug	1642	258	264	1900	1906	86.4%	86.1%	86.3%
drug_n	6	18	39	24	45	25.0%	13.3%	17.4%
group	309	926	378	1235	687	25.0%	45.0%	32.2%
M.avg	-	-	-	-	-	52.4%	57.5%	53.7%
m.avg	2277	1320	735	3597	3012	63.3%	75.6%	68.9%
m.avg(no class)	2370	1227	642	3597	3012	65.9%	78.7%	71.7%

Finally, below we show the results for the complete system in the devel and test set. The difference with that employed at the previous result is the detection of multi-tokens. This causes a major improvement in both precision and recall for the group class.

Final results on devel set:

	tp	fp	fn	#pred	#exp	P	R	F1
brand	320	117	54	437	374	73.2%	85.6%	78.9%
drug	1638	243	268	1881	1906	87.1%	85.9%	86.5%
drug_n	6	18	39	24	45	25.0%	13.3%	17.4%

```
group             386       719    301   1105    687   34.9%  56.2%  43.1%
-------------------------------------------------------------------------------
M.avg             -  -      -      -      -      55.1%  60.3%  56.5%
-------------------------------------------------------------------------------
m.avg            2350      1097    662   3447   3012   68.2%  78.0%  72.8%
m.avg(no class)  2448       999    564   3447   3012   71.0%  81.3%  75.8%
```

Final results on test set:

```
                   tp        fp     fn   #pred  #exp   P      R      F1
-------------------------------------------------------------------------------
brand              257       131     17    388    274   66.2%  93.8%  77.6%
drug              1644       234    483   1878   2127   87.5%  77.3%  82.1%
drug_n               0        31     72     31     72   0.0%   0.0%   0.0%
group              424       882    269   1306    693   32.5%  61.2%  42.4%
-------------------------------------------------------------------------------
M.avg             -  -      -      -      -      46.6%  58.1%  50.5%
-------------------------------------------------------------------------------
m.avg            2325      1278    841   3603   3166   64.5%  73.4%  68.7%
m.avg(no class)  2511      1092    655   3603   3166   69.7%  79.3%  74.2%
```

As expected, the results on the test set are worse than for the development set. One of the main reasons is that none drug_n token has been properly detected, not being present in *our_resources* dictionary. In addition, significant decreases in precision and recall are found for the brand and drug classes, respectively. Oppositely, the precision and recall for these classes is better in the test set. In summary, there is an important difference in data distribution between the devel and test sets.

Machine learning NERC

In the second laboratory, we had to implement the NERC system using a feature-based approach and a machine learning algorithm. More concretely, we had to determine the machine learning algorithm and feature to use. Better results than for the rule-based baseline were expected.

Selected algorithm

As machine learning algorithm, the initial codebase provided a ready to use implementation of both CRF (Conditional Random Fields) and ME (Maximum Entropy) models. Orientative results for both methods were provided in the statement of work, with always CRF outperforming ME. Additionally, in our initial tests the ME model training was significantly slower than that of CRF. On this basis, we decided to use the CRF model , which provided good enough results and seems superior to ME model in performance and speed. Hyperparameters search was tried for the CRF training (changing values such as min. frequency, learning rate and regularization value), but no better results than those obtained with the initial/default values were achieved.
No other machine learning algorithms has been tested, focusing our work on the more interesting task of feature definition.

Feature extraction

The initial code for feature extraction included a set of simple features for each token, including token form and some information related with the previous and next tokens (such as if current token was the BoS or EoS). We consider appropriate this previous-current-next tokens features approach, focusing on the implementation of new features for each token. To this end, we implemented the *features_from_token* method, which defines context-free features for a specific token. This method is called for the three related tokens (previous, current and next) and can be configured to use only some types of features (such as that related with suffixes and prefixes). All features are obtained using this method except for the BoS and EoS. We started defining a simple set of features (such as token form, form lower-case, suffixes and prefixes of length 3) and added more features while testing its effect (such as suffixes and prefixes of length 5, usage of capitals, occurrence of symbols and token length). The final set of features can be found in the Code subsection. Most of the tested features helped to improve the average F1, highlighting the occurrence of symbols and token length. For the latter, we hypotetize that the model is learning the cultural
On this basis, we experiment with a feature that determines if the token is long (if length is greater than a threshold such as 7) but results were worse than for the length feature. We suppose that the length feature allows the model to learn what the threshold is and/or more complex relationships.
Our last try of improvement was to add a feature which informs to what class the token belongs based on the *external* and *our_resources* from the first lab. For maximizing recall, only the 20% most frequent tokens of each class are used for building *our_resources*. If the token is not found in the resources, this feature is not used. However, we found that these approach decremented the average F1, so we discarded it for the final version.
We also tried to discard some features for the previous and next token (supposing that maybe they are not necessary), but best results were obtained using all of them for the 3 considered tokens (previous, current and next).

Code

Our code can be executed with the run.sh script (slightly modified from the provided), which extracts the features, trains the CRF model and obtains the results for both the devel and test sets (storing them in the results folder). The same code of Lab1 for *our_resources* reading and filtering is used. In the following we depict the main functions of our implementation (contained in the *extract-features.py* script).

extract_features function:

```
def extract_features(tokens):
    # for each token, generate list of features and add it to the result
    result = []
    for k in range(0, len(tokens)):
        tokenFeatures = []
        token = tokens[k][0]

        # Current token
        tokenFeatures += features_from_token(token)

        # Previous token
        if k > 0:
```

```
            prev_token = tokens[k - 1][0]
            tokenFeatures += features_from_token(prev_token, prefix="prev")
        else:
            tokenFeatures.append("BoS")

        # Next token
        if k < len(tokens) - 1:
            next_token = tokens[k + 1][0]
            tokenFeatures += features_from_token(next_token, prefix="next")
        else:
            tokenFeatures.append("EoS")

        result.append(tokenFeatures)

    return result
```

features_from_token function (called from extract_features):

```
def features_from_token(token, prefix="", use_form=True, use_capitals=True,
                        use_non_chars=True, use_suf_and_pref=True,
use_resources=True):
    features = []

    # Form
    if use_form:
        features.append(f"form={token}")
        features.append(f"formLC={token.lower()}")
        features.append(f"form_length={len(token)}")

    # Capitals
    if use_capitals:
        num_capitals = len(re.findall(r'[A-Z]', token))
        features.append(f"num_capitals={num_capitals}")
        if token.islower():
            features.append("is_lower")
        elif token.isupper():
            features.append("is_upper")
        elif token.istitle():
            features.append("is_title")

    # Non-characters
    if use_non_chars:
        if bool(re.search(r'\d', token)):
            features.append("has_digits")
        if bool(re.search(r'-', token)):
            features.append("has_minus")
        if bool(re.search(r'\.', token)):
            features.append("has_points")
        if bool(re.search(r'\(|\)', token)):
            features.append("has_parenthesis")
        if bool(re.search(r'\[|\]', token)):
            features.append("has_brackets")
        if bool(re.search(r'\{|\}', token)):
            features.append("has_braces")

    # Suffixes and prefixes
    if use_suf_and_pref:
        features.append(f"suf3={token[-3:]}")
        features.append(f"suf5={token[-5:]}")
        features.append(f"pref3={token[:3]}")
```

```
        features.append(f"pref5={token[:5]}")

    # External resources (not used)
    """if use_resources:
        if token in our_resources:
            features.append(f"{our_resources[token]}_in_our_resources")
        if token in external:
            features.append(f"{external[token]}_in_external_resources")"""

    # Add prefix (if required)
    if prefix != "":
        features = [prefix + feature for feature in features]

    return features
```

Experiments and results

For evaluating the effect of every decision, we have performed multiple experiments during the implementation. For these experiments only the results on the devel set were observed, avoiding “overfitting” ourselves to the test set. In the following, the most important experiments and its results are depicted. Finally, we show the results of our final implementation in the test set.

The first experiment was an ablation study in which we remove all symbol-related/non-chars features (such as these related with the existence of brackets, parenthesis or points). This allow us to measure its effectiveness, since the results were significantly lower than with them (those at the final version).

Results on devel set without symbol-related features:

	tp	fp	fn	#pred	#exp	P	R	F1
brand	305	18	69	323	374	94.4%	81.6%	87.5%
drug	1701	126	205	1827	1906	93.1%	89.2%	91.1%
drug_n	9	7	36	16	45	56.2%	20.0%	29.5%
group	575	79	112	654	687	87.9%	83.7%	85.8%
M.avg	--	-	-	-		82.9%	68.6%	73.5%
m.avg	2590	230	422	2820	3012	91.8%	86.0%	88.8%
m.avg(no class)	2650		170	362	2820	3012	94.0%	88.0%	90.9%

We also performed this ablation study for the usage of external and our filtered resources. These proved ineffective independently of the amount of filtering, decreasing average F1.

Results on devel set using external and our filtered resources:

	tp	fp	fn	#pred	#exp	P	R	F1
brand	309	16	65	325	374	95.1%	82.6%	88.4%
drug	1728	123	178	1851	1906	93.4%	90.7%	92.0%
drug_n	11	8	34	19	45	57.9%	24.4%	34.4%
group	560	73	127	633	687	88.5%	81.5%	84.8%

M.avg	-	-	-	-	-	83.7%	69.8%	74.9%
m.avg	2608	220	404	2828	3012	92.2%	86.6%	89.3%
m.avg(no class)	2666	162	346	2828	3012	94.3%	88.5%	91.3%

In this way we obtained our final system, for which results on devel and test sets we show below.

Final results on devel set:

	tp	fp	fn	#pred	#exp	P	R	F1
brand	307	17	67	324	374	94.8%	82.1%	88.0%
drug	1702	122	204	1824	1906	93.3%	89.3%	91.3%
drug_n	12	7	33	19	45	63.2%	26.7%	37.5%
group	574	75	113	649	687	88.4%	83.6%	85.9%
M.avg	--	-	-	-		84.9%	70.4%	75.7%
m.avg	2595	221	417	2816	3012	92.2%	86.2%	89.1%
m.avg(no class)	2653		163	359	2816	3012	94.2%	88.1%	91.0%

Final results on test set:

	tp	fp	fn	#pred	#exp	P	R	F1
brand	234	47	40	281	274	83.3%	85.4%	84.3%
drug	1827	117	300	1944	2127	94.0%	85.9%	89.8%
drug_n	4	12	68	16	72	25.0%	5.6%	9.1%
group	564	131	129	695	693	81.2%	81.4%	81.3%
M.avg	--	-	-	-		70.9%	64.6%	66.1%
m.avg	2629	307	537	2936	3166	89.5%	83.0%	86.2%
m.avg(no class)	2735		201	431	2936	3166	93.2%	86.4%	89.6%

As can be noted, the results are much better than for the rule-based system. Major improvements are found at the precision for the brand and drug classes (almost perfect) and the extremely improved F1 (approximately the double) for the drugn_n and group classes.

On the other hand, as expected, the average F1 on test set was seriously less than the obtained on devel. Equivalently to the case of the rule-based system, the most affected class is drug_n, indicating that probably there is an important difference between the data distribution of devel and test sets. For example, with many drug_n tokens of test not existing in devel (or train).

Conclusions

During the development of this deliverable, we have learned about two of the most classical approaches for the NERC and other NLP tasks. Both methods are based on hand-crafted

rules or features, requiring us from analyze the data searching properties or heuristics to exploit in the classifier. Even though this exploration can be enriching for the person who does it (learning about the data), most of the used rules and features could be automatically discovered using statistics. The closer approach to this is the machine learning-based method of the second laboratory, for which many fast-designed features can be defined expecting the model to discover the useful ones. However, differences between training and development or test sets and the limited amount of training data causes the model to fail when too many features are used.

In conclusion, the use of these methods has been important to give us a historical perspective, but we expect to use much more automated, actual and advanced approaches over the AHLT course.