Language Technology - EDAN20
Assignment 4 - Fall 2020

Extracting Syntactic Groups
Using Machine Learning Techniques

Hicham Mohamad, (hi8826mo-s)

November 8, 2020

1 Introduction

In this assignment, we will work on important tasks in NLP as Part-of-speech (PoS) and Named Entity Recognition (NER) tagging. In the first part, we will write a program to detect partial syntactic structures from a text and applied to the CoNLL 2000 dataset, whereas in the second part we will deal with the principles of supervised machine learning techniques applied to language processing for training a ML model to detect noun groups.

2 Choosing a training and a test sets

For this assignment, we will consider and use the data available from **CoNLL 2000** [6], as annotated data and annotation scheme. Thus, before starting implementing our tasks we download both the training and test sets, i.e. `train.txt` and `test.txt`. To solve the following tasks in Python, the popular machine learning toolkit **Scikit-learn** will be used.

To load the datasets, we use the functions `read_sentences(file)` and `split_rows(sentences, column_names)` by which we can store the corpus in a **list of sentences** where

- each sentence is a **list of rows**,
- and each row is a **dictionary** representing a word.
- each word consists of **three columns**: the first column contains the current word, the second its part-of-speech (pos) tag and the third its chunk tag.
- Chunk have two types of chunk tags, B-CHUNK for the first word of the chunk and I-CHUNK for each other word in the chunk.

3 Baseline chunker

Most statistical algorithms for language processing start with a baseline. In this section, we need to implement the baseline proposed by the organizers of the CoNLL 2000 shared task [6]. After reading this baseline, we see that results was obtained by selecting the chunk tag which was most frequently associated with the current part-of-speech tag. Six of the eleven systems obtained an F-score between 91.5 and 92.5, and two systems performed a lot better using Support Vector Machines and Weighted Probability Distribution Voting.

Baseline implementation For implementing this baseline program, our code has these principal steps:

- We first collect all the parts of speech (pos) and we count them.
- We compute the chunk distribution for each part of speech using the train file and we store the results in a dictionary.
- For each part of speech, we select the best association, i.e. the most frequent chunk.
- Using the resulting associations, we apply our chunker to the test file.
- We store the results in an **output file** that has four columns. The three first columns will be the input columns from the test file: word, part of speech, and gold-standard chunk. We append the **predicted chunk** as the 4th column. The CoNLL 2000 evaluation script will use these two last columns, chunk and predicted chunk, to compute the performance.

performance of the system We can measure the performance of the system with the percentage of words that receive the correct tag, but this accuracy is very misleading as it is biased by the most frequent tags. Thus, we use the **conlleval.txt** evaluation program used by the CoNLL 2000 shared task.

To evaluate our results, we use the Python translation of the original Conlleval script and get the baseline score of 0.770671072299583.

4 Using Machine Learning

In this section, we need to apply and explore the machine learning program that won the CoNLL 2000 shared task (Kudoh and Matsumoto, 2000) [7].

This program used a window of five words around the chunk tag to identify, c_i. They built a feature vector consisting of:

- The values of the **five words** in this window: w_{i-2} , w_{i-1} , w_i , w_{i+1} , w_{i+2}
- The values of the **five parts-of-speech** in this window: t_{i-2} , t_{i-1} , t_i , t_{i+1} , t_{i+2}
- The values of the **two previous chunk tags** in the first part of the window: c_{i-2} , c_{i-1}

The two last parameters are said to be dynamic because the program computes them at run-time. As mentioned in [7], Kudoh and Matsumoto trained a classifier based on support vector machines.

In this task, we build our classifier using logistic regression that uses only the two first sets of features. The program structure consists of

- After extracting the features, we encode them using `DictVectorizer()`
- and then we train the `LogisticRegression()` model.
- After training, we predict the chunk labels to the test dataset and add them to the sentences.
- At the end, we save the output file for evaluating the performance using Conleval script and get the resulting **F1 score 0.915119639107748**.

5 Improving the Chunker by using the two dynamic features

The goal of this task is to come forward with a better chunker using machine learning by adding all the features from Kudoh and Matsumoto which after a training phase can recognize the chunk segmentation of the test data as well as possible. For implementing this program we need to

- Complement the feature vector used in the previous section with the two **dynamic features**, c_{i-2} , c_{i-1} , and train a new model. For this reason, we modify the `extract_features_sent()` and `predict()` functions.

- In this improved model, we use also logistic regression and the default parameters from sklearn, i.e. `linear_model.LogisticRegression()`
- Training data is used for training this dynamic chunker. However, here the feature vector **X** is considered incomplete.
- For the prediction, we need to re-inject dynamically the **two previously predicted tags** to have the full feature vector.
- With this classifier and the dynamic feature vector, we obtain an improved **F1 score 0.9231961786642086**.

Since the chunk labels are not given in the test data, but instead they are decided dynamically during the tagging of chunk labels, this technique can be regarded as a sort of **Dynamic Programming** (DP) matching, in which the best answer is searched by maximizing the total certainty score for the combination of tags.

This score could be solved as well using **beam search**, where we apply it using the *probability output* of logistic regression or the *score* if support vector machines is used.

6 Named Entity Recognition

The first step in **information extraction** is to detect the entities in the text. A named entity is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. Often, Parts of Speech (PoS) are useful features for labelling named entities like people or organizations.

Collecting the named entities

In this task, we need to collect all the named entities from the training set, defined as **NP chunks** and starting with a `NNP` (proper noun) or a `NNPS` (proper noun, plural) tag. Our program is better implemented in two-pass procedure, where for each sentence in the corpus:

- We collect the **start indices** of the noun groups which are also proper nouns.
- We collect the **segments**, starting at each index.

Resolving the entities - Web Scraping

In this section, we need to implement a method to find the collected named entities from the previous task in Wikipedia. However, to run the experiments faster, we limit the dataset to the entities starting with letter K, and call it `ne_small_set`. The implemented function to lookup entities consists of the following points:

- We run the lookup function `wikipedia_lookup(ner, base_url='https://en.wikipedia.org/wiki/')` and keep only the resolved entities. Using Python and the **Beautiful Soup library**, it is nice to illustrate the function and explain the instructions as shown in the following

```
def wikipedia_lookup(ner, base_url='https://en.wikipedia.org/wiki/'):
    try:
        # specify the website and named entities recognition ner
        url_en = base_url + ' '.join(ner)
        # get/download the webpage using Requests module
        # then read the content of the server
        html_doc = requests.get(url_en).text

        # web scraping using Beautiful Soup library
        # Parsing the page
        parse_tree = bs4.BeautifulSoup(html_doc, 'html.parser')

        # search for items using CSS selectors
        # find the first instance of 'a' tags
        # 'a' tags are links, and tell the browser to render a link to another web page.
```

```
            # The href property of the tag determines where the link goes
            entity_id = parse_tree.find("a", {"accesskey": "g"})['href']

            head_id, entity_id = os.path.split(entity_id)
            return entity_id
        except:
            pass
            # print('Not found in: ', base_url)
        entity_id = 'UNK'
        return entity_id
```

- Since entities need a confirmation, we apply the resolution with the Swedish wikipedia as well.
- At the end, we compute the intersection of the two sets and store the result in *confirmed_ne_en_sv*.

7 Reading

In this section, we need to investigate the system implemented in the article: *Contextual String Embeddings for Sequence Labeling* by Akbik et al. (2018) and compare it with ours in this assignment. To summarize, this paper proposes **contextual string embeddings**, a novel type of word embeddings based on character-level language modeling, and their use in a state-of-the-art sequence labeling architecture.

They use the LSTM variant of **recurrent neural networks** as language modeling architecture, and a subsequent **conditional random field** (CRF) decoding layer, a sort of beam search. A crucial component in a such approach are **word embeddings**, typically trained over very large collections of unlabeled data to assist learning and generalization. We can distinguish the main structure of their architecture:

- A sentence is input as a character sequence into a pre-trained bidirectional character language model.
- From this LM, for each word a **contextual embedding** is retrieved
- this contextual embedding is pass into a BiLSTM-CRF **sequence labeler**.
- Their system achieve good performance on syntactic tasks for chunking and PoS tagging. Their chunking F1-score is between 95 and 96.72 using BiLSTM-CRF for different proposed approaches.

8 Appendix

References

[1] Pierre Nugues, *Language processing with Perl and Prolog*, 2nd edition, 2014, Springer.

[2] P. Nugues, Language Technology - EDAN20, Lectures notes: `https://cs.lth.se/edan20/`

[3] D. Jurafsky, J. Martin, *Speech and Language Processing*, 3rd edition. Online: `https://web.stanford.edu/~jurafsky/slp3/`

[4] VanderPlas, A Whirlwind Tour of Python. O'Reilly 2016. Online reading: `https://jakevdp.github.io/WhirlwindTourOfPython/`

[5] Pierre Nugues, Assignment 4: `https://github.com/pnugues/edan20/blob/master/notebooks/4-chunker.ipynb`

[6] Chunking: `https://www.clips.uantwerpen.be/conll2000/chunking/`

[7] Taku Kudoh, Yuji Matsumoto, Use of Support Vector Learning for Chunk Identification: `https://www.aclweb.org/anthology/W00-0730.pdf`

[8] YamCha: Yet Another Multipurpose Chunk Annotator. `http://www.chasen.org/~taku/software/yamcha/`

[9] Akbik et al., Contextual String Embeddings for Sequence Labeling, 2018.

[10] Zalando Research, Github code and pretrained language models: `https://github.com/zalandoresearch/flair`

[11] Tutorial: Web Scraping with Python Using Beautiful Soup: `https://www.dataquest.io/blog/web-scraping-tutorial-python/`

[12] Geron, Jupyter notebook on linear algebra from Machine Learning and Deep Learning in Python using Scikit-Learn, Keras and TensorFlow: `https://github.com/ageron/handson-ml2`