Term Project Mid-Term Report

Machine Learning과 Deep Learning을 활용한 개체명 인식
중간 보고서

Sung Jae Hyuk
Department of Computer Science
Student ID: 2019320100
okaybody10@korea.ac.kr

Shin Seung Heon
Department of Computer Science
Student ID: 2016190329
heon4077@naver.com

Choe Min Seok
Department of Computer Science
Student ID: 2021320092
goro8pyo@korea.ac.kr

2023, Fall Semester

Course: Machine Learning(COSE362(02))

1 Problem Statement

Named Entity Recognition(NER, 이하 개체명 인식)은 문장 내에서 의미를 가지는 개체들을 인식하는 것을 의미한다. 즉, 문장 속에서 각 단어가 어떤 태그에 속하게 되는지를 구분하는 문제이다. 한국어 기준 15종류의 명시된 개체명 가지고 있으며, 세분류로 150개의 개체명을 가지고 있다.[1] NER은 Word-Level에서 품사 태깅과 이루어지는 가장 기초적인 작업이며, 정보 추출(IE)을 위한 독립형 도구로 사용될 뿐만 아니라 텍스트 이해, 정보 검색, 자동 텍스트 요약, 질문 답변, 기계 번역, 지식 기반 구축 등과 같은 다양한 자연어 처리(NLP) 애플리케이션에서 필수적인 역할을 담당하고 있다.[2] 개체명 인식 작업을 수행하는 방법으로는 크게 Rule-Based, Feature-Based, Unsupervised-Based로 3가지가 존재하고, 이번 프로젝트에서는 개체명 분석 Task에서 문장의 문맥을 고려하는 것이 얼마나 많은 도움을 주는지, 유의미한 성능 발전이 있는지 알아보고자 한다. 또한 기존 모델을 단순히 활용하는 것과 추가로 Transfer Learning을 진행하였을 때의 정확률 차이에 대해서도 확인해 보고자 한다.

2 Methodology

2.1 Analysis of dataset

NER Dataset은 총 3개의 json 형태의 파일로 구성되어 있으며, 구성 방법과 주석 단계, 구축 연도 및 일련번호는 모두 동일한 양식을 가지고, 맨 처음 유형만 다르게 설정이 되어있다.
첫 번째 유형은 신문(N)으로, 신문 자료를 가공하여 만든 개체명 말뭉치 파일이고, 두 번째 유형은 일상 대화(S)로, 일상 대화 자료를 가공한 것이다. 마지막은 온라인 대화(M)으로 온라인 대화 자료를 가공하여 만든 파일이다. 내부 파일은 dictionary 형식으로 되어있으며, id, form, word, 그리고 NE로 총 4개의 key로 구성되어있다. 해당 프로젝트에서 가공해야하는 부분은 NE key로, NE는 `list[dictionary]` 형식으로 이루어져있다.
NE에서는 각 dictionary가 id, form, label, begin, end로 총 5개의 Key를 가지고 있으며 `form-label`로 개체명이 서로 연결된다.
begin과 end는 해당 단어가 문장의 몇번째 문자에서부터 시작하는지와 끝나는지를 알려주는 offset mapping이며, 현 프로젝트에서 사용할 `kobert-tokenizer`에서는 offset mapping을 지원하지 않으므로 크게 의미가 없는 key이다. label

같은 경우 150가지의 세분류를 활용하여 분류한 결과가 있으며, 지금 프로젝트에서는 15가지의 대분류만을 사용할 것이기 때문에 추후 labeling을 다시 하는 과정에서 앞의 두 글자만을 잘라서 활용할 예정이다.
아래는 국립국어원에서 제공한 공식 문서에 나와 있는 NE의 예시 중 하나에 주석을 덧붙인 것이다.

```
{
    "id": 1, // 고유번호
    "form": "캐나다", // 개체명 인식을 수행할 단어
    "label": "LCP_COUNTRY", // 위의 form의 단어의 세부 개체명
    "begin": 0, // 해당 단어가 문장의 몇번째에서 시작하는지
    "end": 3 // 해당 단어가 문장의 몇번째에서 끝이 나는지
 },
```

Listing 1: Example of Named Entity Dataset

2.2 Tokenizing and re-labeling

컴퓨터에서는 자연어를 받아들이지 못하기 때문에 이를 의미 있는 숫자들로 바꿔주어야 할 필요가 있는데, 이를 Embedding 과정이라고 한다. 앞의 Embedding을 하기 전에, 단어(혹은 문장)을 들고 와서 모델이 사전에 약속한 embedding matrix에 넣을 수 있게 정수화하는 과정이 필요하고 이를 Tokenizing이라 한다.
해당 프로젝트에서는 모든 word embedding을 문장 내의 관계를(Self-attention, Positional encoding) 고려하지 않은 bert embedding을 활용할 예정이며, 그러기 위해서는 Bert에서 사용하는 tokenizer으로 모두 규격을 맞춰야 할 필요가 있다.
이때 위의 Example을 참고하면 form 같은 경우 Bert tokenizer에서 사용하는 wordpiece token이 아니라 단어 자체랑 개체명이랑 연결되어 있기 때문에 bert tokenizer한 후 이를 기준으로 BIO-tagging을 해줄 필요가 있다.
현재 문서에서 각 단어의 offset을 제공하고 있지만 사용할 `KoBert tokenizer`에서는 각 단어에 따른 offset mapping을 제공해주지 않고 있기 때문에, 수동으로 단어를 보며 맞춰줄 필요가 있다.
먼저, wordpiece로 변하게 되면 subword임을 알려주기 위하여 토큰의 맨 앞에 ## 혹은 _와 같은 추가적인 기호들이 붙는데, 토큰이 현재 문장의 어디에 있는지 판단하기 위해서는 이를 모두 제거할 필요가 있다.
Bert에서는 ##을, Kobert에서는 추가로 _를 사용하고 있기 때문에 두개 모두 활용하여 replace 연산을 수행한다.
이후 BIO-tagging과 소분류에서 대분류를 바꾸는 작업을 동시에 처리한다. B tag 같은 경우에는 현재 단어가 어떤 단어의 첫 시작인지를 봐야하므로 현재 문장의 Named entity가 되어 있는 dictionary를 반복문으로 순환하면서 단어의 첫 시작이 현재 토큰과 같은지 비교한다.
이때 태깅이 역방향으로 이루어지지는 않으므로 현재까지 본 단어의 위치를 기억한 후 비교를 시작하고, 만약 문장의 끝까지 갔음에도 찾을 수 없으면 이는 O Tag로 판단한다.
I tag 같은 경우 문장의 중간에 나와야하므로, 직전 단어의 Tag가 B나 I로 시작하는지 봐야한다. 또한, 다음 단어에서 시작될 일이 없기 때문에 현재 단어에서 보지 않은 부분의 길이가 얼마인지, 보지 않은 부분의 시작부분이랑 토큰의 시작부분이랑 같은지를 모두 비교하여 동일하면 I tag를 부여한다.
BI 이후에 들어갈 두 글자는 세부 개체명의 앞의 두 글자만 잘라서 붙이는 식으로 라벨링을 마무리한다.
마지막으로 OOV, 즉 tokenizer에서 `[UNK]`라고 판별한 단어는 이 단어가 B인지 I인지 O인지 확실히 알 수 없기 때문에 무조건 O로 체크하고, 이후 나오는 단어들도 B, I, O를 확실히 알 수 없기 때문에 B가 나오기 전까지는 무조건 O로 태깅한다.

2.3 Embedding

개체명 인식은 문장을 입력으로 받아 개체명이 존재하는 부분과 개체 분류를 출력하는 과제로 볼 수 있다. 입력으로 들어오는 값이 자연어이기 때문에, SVM이나 CRF 등의 접근을 위해 입력을 벡터 공간으로 매핑하는 적절한 방법이 필요하다. 또한 이 때 비슷한 단어들은 벡터공간에서의 거리가 가깝게 하는 것이 중요한데, 만약 모든 단어들이 각각 랜덤하게 분포하도록 매핑한다면 의미적으로 적절한 거리함수를 정의할 수 없어 단어의 분포에서 학습이 매우 어렵게 된다.
이를 위해 문장을 단어, 또는 토큰 별로 분리하고 각 토큰을 One-hot vector가 아닌 의미 있는 벡터로 나타낼 수 있도록 Tokenizer와 Word Embeding을 이용하였다.한국어 NLP를 위해 Bert 모델을 기반으로 만들어진 KoBert를 이용하여, 이 과제를 위해 KoBert tokenizer 및 KoBert embedding을 사용하기로 결정했다.

2.4 LSTM

자연어를 처리하기 위해서는 언어 데이터가 가지는 특성을 고려해야 한다. 언어 데이터의 한 가지 특징은 시계열 적이라는 것인데, 기계학습에서 언어는 순환 신경망을 통해 접근되어지고 있다.
LSTM은 순환 신경망(RNN)의 한 종류로, 기존의 순환 신경망의 문제를 해결하기 위해 만들어진 네트워크이다. RNN은 글이나 시계열 데이터와 같이 연속적인 성질이 있는 데이터를 다루는데에 특화되어있다. 이는 RNN의 구조가 출력의 일부를 입력으로 재사용하는 순환 구조를 가지고 있기 때문이다. 이로 인해 이전 순서의 정보가 다음 번에 재사용되게 된다. 기본적인 RNN의 수식은 다음과 같이 표시할 수 있다.

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$
$$y_t = W_{hy}h_t + b_y$$

이 때 h_t는 t번째 단계의 누적 정보이고, x_t는 t번째 단계의 입력이다. 그러나 이런 전통적인 방식의 RNN은 σ의 기하적 누적으로 인해 Vanishing Gradient 문제가 발생했으며, 이를 해결하기 위해 LSTM 등 새로운 방법론이 등장하였다. LSTM은 셀 상태라는 개념을 도입하여 이 문제를 해결하였다.

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$h_t = o_t \odot \tanh(c_t)$$

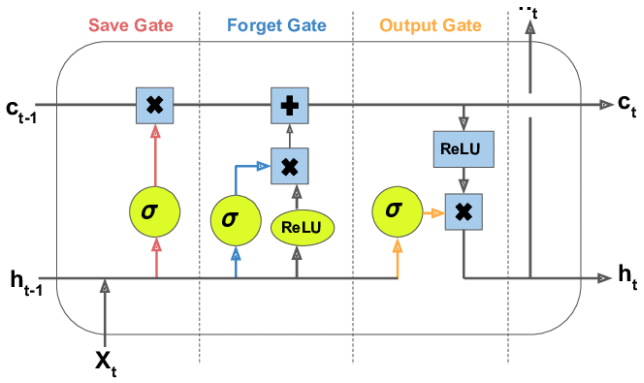

Figure 1: LSTM의 모델 구조를 나타내는 그림[3] LSTM은 정보를 얼마나 잊을지 결정하는 Forget Gate, 현 시점의 정보를 얼마나 저장할지 결정하는 Save Gate, 계산된 기존의 게이트를 바탕으로 다음 정보를 결정하는 Output 게이트로 구성되어 있다.

2.5 Bert

Bert는 기존의 LSTM을 개선한 Transformer 모델을 기반으로 등장하였다. 기존의 LSTM에서는 시계열 데이터들의 관계가 일방향 적이지만, Transformer를 기반으로 하는 Bert는 순차 데이터 간의 관계를 양방향으로 고려한다. 또한 Bert는 masking된 위치에 context를 이용해 단어를 추정하는 방식으로 학습했는데, 별도의 라벨링이 되어 있지 않아도 된다는 점 때문에 학습에 필요한 데이터를 구하기가 쉬우며 하나의 텍스트 데이터도 수많은 곳에 빈칸을 뚫어 학습에 여러번 활용할 수 있다는 점 때문에 매우 강력한 성능을 보이며 자연어 처리에 있어 강력한 모델이 되었다.
이번 과제에서는 앞서 서술한 바와 같이 다양한 Approach 및 hyperparamter에 의한 성능 차이를 비교하는 과정이 필요하기 때문에, Word Embedding으로 Bert, 특히 한국어 자료로 더 많이 학습해 한국어 성능이 강화된 KoBert의 tokenizer 및 embedding[4]을 사용하였다.

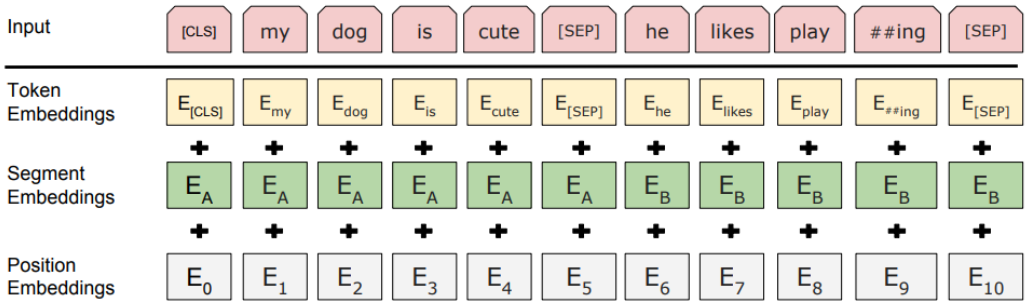

Figure 2: Bert의 모델 구조를 나타내는 그림[5] 이때 Bert에서 Contextual하게 단어들이 embedding되는 이유는 segment embedding과 position embedding 때문이다. Segment Embedding에서는 문장이 [SEP]으로 끊길 때 뒤의 문장과 앞의 문장에 대한 차이를 두기 위해 있는 vector이고, position embedding 같은 경우 단어의 위치에 따라 의미가 달라질 수 있기 때문에 들어가 있다. 해당 프로젝트에서는 단어 자체가 가지고 있는 뜻, 즉 word embedding에 의미를 두기로 하였으므로 뒤의 두 Embedding을 제외한 Token embedding만 활용한다.

3 Intermediate results

3.1 Preprocess

현재 Tokenizing하여 labeling을 다시 하는 코드는 구현이 완료된 상황이고, 전체 json 파일을 load하여 이를 전처리하는 과정은 현재 구현중에 있다.
Tokenizing하여 다시 labeling을 하는 부분 같은 경우 예제를 돌려봤을 때 다음과 같이 나온다.

```
sentence = "태안군의회, 2019년‘군민중심’의정성과 빛났다!"
ne = [
        {
            "id": 1,
            "form": "태안군의회",
            "label": "OGG_POLITICS",
            "begin": 0,
            "end": 5
        },
        {
            "id": 2,
            "form": "2019년",
            "label": "DT_YEAR",
            "begin": 7,
            "end": 12
        }
]
```

Listing 2: 아래 코드를 돌리기 위한 예제, sentence는 전체 문장이며 ne는 sentence의 단어들에 대해 개체명을 정리해놓은 사전이다.

```
tokenizer = get_tokenizer()
tok = tokenizer(sentence, padding=True, truncation=True) # Has Input_ids, token_type_ids, attention_mask
tokens_word = tokenizer.convert_ids_to_tokens(tok['input_ids'])
print(tokens_word, tagging(tokens_word, ne), sep='\n')
✓ 1.5s
['[CLS]', '_태', '안', '군의', '회', ',', '_20', '19', '년', '‘', '군', '민', '중심', '’', '의', '정', '성과', '_빛', '났다', '!', '[SEP]']
['[CLS]', 'B-OG', 'I-OG', 'I-OG', 'I-OG', 'O', 'B-DT', 'I-DT', 'I-DT', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', '[SEP]']
```

Figure 3: Labeling 결과, 원래 문장은 “태안군의회, 2019년 ‘군민중심’의정성과 빛났다!”에서 태안군의회가 OG, 2019년이 DT를 객체명으로 가지고 있으며 Tokenizing을 거쳤을 때 _태는 B 태그를, 안, 군의, 회는 I 태그를 가져야한다. 같은 방법으로 _20은 B 태그를, 19, 년 태그는 I 태그를, 이외는 모두 O 태그를 주어야한다. 현재는 [UNK] Token이 없는 상황이다.

3.2 Word2Vec

초기엔 embedding을 word2vec으로 할 계획으로 세웠다. word2vec를 위해 pre-trained된 word vector을 해당 github에서 다운받아서 사용하고자 하였다. 허나, 해당 프로젝트에서 마지막으로 구현하고자 하는 부분은 KoBert 모델을 Fine tuning하는 과정이며, 이때는 Bert 모델의 강점을 살리기 위해서 Bert Embedding을 활용해야하기 때문에 tokenizing 결과 자체가 달라진다.
즉, 초기 문장을 제외하고는 전처리부터 완전히 달라지기 때문에 결과를 완전히 신뢰할 수 없다고 판단하였고, 토크나이저와 특성 기반의 word embedding한 결과는 동일한 상황에서 모델 성능차이를 보기 위해 아래의 Bert Embedding을 활용하기로 최종결정하였다.

3.3 Bert Embedding

Bert Embedding을 활용하기 위하여 KoBert[4]를 활용하고자 하였고, Bert모델을 finetuning할 때는 Hugging face에서 제공하는 Accelerate를 사용하기 위해서 토크나이저와 KoBert 모두 Huggingface 내에 구현되어 있는 것을 활용하고자 하였다.
이를 위해 Huggingface의 API를 지원하며 SKT사에서 개발한 것과 동일한 KoBert가 있는 해당 github의 모델을 활용하기로 하였고, embedding 같은 경우 `model.embeddings.word_embeddings`를 사용하여 tensor를 얻어낸다.

3.4 Metric

SVM, CRF, LSTM 모델 간의 비교를 위한 Metric으로는 F1-score, recall, precision을 사용할 예정이다.

- **Precision**: 예측된 Entity 중에서 정확하게 예측된 Entity의 비율이다.

$$\textbf{Precision} = \frac{\#\text{ of true positives}}{\#\text{ of true positives} + \#\text{ of false positives}}$$

$$= \frac{\text{Model이 정확하게 예측한 Entity의 수}}{\text{예측된 Entity 수}}$$

- **Recall**: 실제 Named Entity 중에서 모델이 올바르게 분류한 Named Entity의 비율이다.

$$\textbf{Recall} = \frac{\#\text{ of true positives}}{\#\text{ of true positives} + \#\text{ of false negatives}}$$

$$= \frac{\text{Model이 정확하게 예측한 Entity의 수}}{\text{실제 Entity 수}}$$

- **F1-score**: Precision과 Recall의 조화 평균으로 정의된다.

$$\textbf{F1} = 2 \times \frac{\textbf{Precision} \times \textbf{Recall}}{\textbf{Precision} + \textbf{Recall}}$$

기본적으로 precision(정밀도)와 recall(재현율)은 true/false, positive/negative plane와 관련이 있다.
precision과 recall은 trade-off가 있다. 모델이 정확해질수록 예측이 엄격해져 recall은 감소할 수 있다. 반면 재현율이 증가한다면, 재현율은 모든 경우에 positive를 주면 증가하게 되기 때문에 precision은 감소할 수 있다. F1-score는 precision과 recall의 조화평균인데, 조화평균은 둘 중 한 값이라도 작으면 더 급격하게 작아지는 경향을 다음 그래프에서 확인할 수 있다. 이 조화평균의 특징은 F1-score가 recall과 precision 사이의 균형을 이루도록 한다. 또한 precision과 recall의

Metric을 하나로 합쳐 하나의 Metric만으로 모델간의 성능을 비교할 수 있다는 장점이 있다. 아래 Figure 4는 Recall, Precision, F1-scroe를 3차원의 그래프로 표현한 것이다.

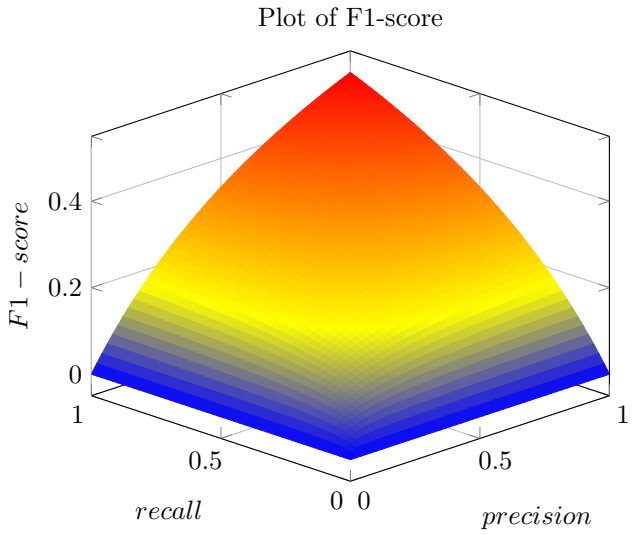

Figure 4: Recall과 Precision, 그리고 F1-Score를 3차원의 그래프로 나타낸 것이다. Recall과 Precision은 어느 정도의 상관관계를 가지고 있음을 알 수 있다.

4 Working Process

4.1 SVM & CRF

CRF는 Pytorch에서 제공하는 `pytorch-crf`를 활용한다. 이때 SVM 같은 경우 Batch화를 위해 Linear model을 활용하여 OVA와 함께 사용한다.
단순 SVM 모델을 활용하는 경우에는 `torch.nn`의 Linear을 이용하여 배치화한 이후 `Multilabelmarginloss`를 이용하여 Support Vector Machine 훈련을 진행한다.

4.2 DataLoader & Split data

먼저 Train set과 Validation set(Test set)은 8:2의 비율로 나눌 예정이며, pytorch의 util.data에서 제공하는 `DataLoader`를 이용하여 split 이후 각각 적절한 batch size를 선정하여 불러올 예정이다.
DataLoader에서는 전처리를 이미 다 완료한 후로 들고오며, 가장 처음에 불러온 data를 모두 model의 input으로 넣을 예정이다.
기본 batch size는 32로 설정하여 데이터들을 불러온다.

4.3 High-level api(Huggingface Accelerate)

Huggingface에서 제공하는 Accelerate는 단 네 줄의 코드만 추가하면 모든 분산 구성에서 동일한 PyTorch 코드를 실행할 수 있는 라이브러리이다.
간단히 말해, 대규모 학습과 추론이 간단하고 효율적이며 적응력이 뛰어난 라이브러리이다.[6]
기존에 torch와 Huggingface에서 학습하였던 LSTM계열과 Bert계열의 모델 4개는 모두 해당 Accelerate를 이용하여 학습할 예정이다.

5 Plans & Timeline

- (∼ 10/14) [First Compare] 단순 Support Vector Machine과 CRF 모델 훈련 및 Inference
- (∼ 10/19) [Second Compare] BiLSTM Model 설계 및 SVM 및 CRF 모델 훈련 및 Inference
- (∼ 10/23) [Third Compare] Pretrained된 Bert Model을 이용한 SVM / CRF Transfer Learning

6 References

[1] National Institue of Korean Language (2023). Nikl named entity corpus 2022 (v.1.0).

[2] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. A survey on deep learning for named entity recognition. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):50–70, 2020.

[3] Harsh Grover, Vinay Chamola, Dheerendra Singh, Kim-Kwang Raymond, and Tejasvi Alladi. Edge computing and deep learning enabled secure multi-tier network for internet of vehicles. *IEEE Internet of Things Journal*, PP, 04 2021.

[4] SKTBrain. Korean bert (bidirectional encoder representations from transformers), 2021.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.