

- 과제목표: 한국어 text 에서 개체명을 인식하는 시스템을 개발하고 실험한다.
 - 이 작업은 전형적인 sequence labeling 문제이다.
 - 입력: token sequence
 - 출력: 입력의 각 token 에 대하여 NE classs 의 label 을 출력한다.
- 실험에 사용할 AI model 2가지 : 2 type 의 neural network 을 사용하는 모델에 대하여 실험하고 성능을 비교한다.
 - (1) RNN
 - (2) LSTM
 - 구조는 동일하게 한다. 소자만 다름.
- 학습데이터:
 - ETRI 가 구축한 한국어 개체명인식용 tagged corpus 를 사용한다.
 - 이 annotated corpus 로 부터 training examples 을 만들어 사용한다.
- 과제 수행 단계:
 - (1) 제공된 tagged corpus 로 부터 훈련예제 리스트를 생성한다.
 - (2) 훈련예제 리스트를 훈련용과 테스트용으로 나눈다 (80% : 20%)
 - 실제로는 검증(validation) 용으로도 할당해야 하는데 이 과제에서는 검증은 생략한다.
 - 검증 단계도 넣고 싶은 사람은 훈련 80%, 검증 10%, 테스트 10% 으로 나누어 실험한다.
 - (3) 위 두 model 에 각각에 대하여 학습 및 테스트를 시행한다.
 - 성능 measure 로 recall, precision, f-score 를 사용한다.
 - (4) 구조는 동일하고 neuron 소자만 RNN 과 LSTM으로 하는 두 모델의 성능을 비교해 본다.
 - (5) 결과 보고서를 작성하여 프로그램과 같이 제출한다.

- ETRI 에서 개발한 tagged corpus 를 이용한다.
- 원본 데이터
 - 문장 속에 annotation 을 위한 tag를 넣어 놓은 형태임
 - 개체명 부분을 각괄호("<“ , ”>“)로 둘러 싸고 “:개체명태그“ 를 마지막 부분에 넣어 준다.
 - 주의: 한 개체명 부분은 한 어절로 구성된 경우가 많으나 여러 어절로 구성된 상당히 많다.

파일명: "1_NER_tagged_corpus_ETRI_exobrain_team.txt"

```
1  특히 <김병현:PS>은 4회말에 무기력하게 6실점하면서
2  <빅비:PS>가 2루 도루를 시도하다 아웃됐고
3  <서호프:PS>와 <파사노:PS>에게 연속 안타를 내주며
4  <우리금융그룹:OG>은 <19일:DT> <지난주부터:DT> <네덜란드:LC> 프로축구 <에인트호벤:OG>의 미드필더 <박지성:PS>과 <미국:LC> 프로야구
5  <새미 소사:PS>(36.<볼티모어 오리올스:OG>)가 은퇴한 '홍천왕' <마크 맥과이어:PS>(전 <세인트루이스 카디널스:OG>)와 개인통산 홈런 타이틀
6  0...프로야구 <두산:OG>의 에이스 <박명환:PS>이 <19일:DT> 잠실구장에서 열린 <한화:OG>와의 홈경기에 양배추 더미를 모자 속에 쓰고 투구
7  <프랭크 로빈슨:PS> 감독, "<김선우:PS> 정말 잘 던졌다"
8  <김선우:PS>의 투구 내용을 칭찬하며 "승리 투수가 될 자격이 충분히 있었지만 공을 칠 수 없는 상황이었고 모험을 걸 수 없어 하는 수 없이 교체
```

- 원본 파일([NER_tagged_corpus_ETRI_exobrain_team.txt](#))에 대한 1 차 처리 작업:

- 한 어절을 한 줄에 놓는다(각 괄호를 가진 부분은 한 어절의 내부로 본다).
- 어절 내에 NE 가 있으면 태그 기호 "<" 와 ">"에서 절단하여

절단된 각 조각마다 새로운 한 줄에 놓는다.

< 와 > 기호 사이 부분을 이용하여 다음처럼 한 줄 또는 여러 줄을 만든다:

- 먼저 : 기호 다음의 NE-tag 를 준비한다.
- 각괄호 내의 맨 좌측 어절에 NE-tag 앞에 "B-" 를 붙인 label을 주면서 한 줄을 만든다,
- 각괄호 내의 다른 어절들마다 NE-tag 앞에 "I-" 를 붙인 label을 주며 새 줄을 만든다.

- 개체명 태그가 없는 각 줄은 태그 O 를 붙인다.
- 문장과 문장 사이에는 하나의 빈 줄(enter 키 하나만 가짐)을 둔다.

- 그 결과로 파일 "[2_ner_eojeol_label_per_line.txt](#)" 를 얻는다.

- 우측 그림 참고.



2_ner_eojeol_label_per_line.txt :

```

1  특히 O
2  김병현 B-PS
3  은 O
4  4회말에 O
5  무기력하게 O
6  6실점하면서 O
7
8  빅비 B-PS
9  가 O
10  2루 O
11  도루를 O
12  시도하다 O
13  아웃됐고 O
14
15  서호프 B-PS
16  와 O
17  파사노 B-PS
18  에게 O
19  연속 O
20  안타를 O
21  내주며 O
22
23  우리금융그룹 B-OG
24  은 O
25  19일 B-DT
26  지난주부터 B-DT
27  네덜란드 B-LC
28  프로축구 O
29  에인트호벤 B-OG
30  의 O
31  미드필더 O
32  박지성 B-PS
33  과 O

```

파일변환 2: 어절을 토큰 열(list)로 변환

- 각 줄(line)에서 어절 부분을 tokenize 하여 token list 로 대체한다. "3_ner_tokens_label_per_line.txt" :
 - 결과로 얻는 파일명: ner_tokens_label_per_line.txt
- 어절을 token 들로 나누는 방법:
 - etri_tokenizer 를 이용한다.
 - 프로그램 제공:  eojeol_etri_tokenizer
 - program working directiory 에 위 "eojeol_etri_tokenizer" directory를 넣는다.
 - 토큰 사전이 제공된다:
 - 제공되는 토큰사전 파일  vocab.korean.rawtext.list 을 program working directory 에 저장한다.
- tokenizer 이용 방법: 어절의 토큰 리스트 변환 및 토큰의 id 변환 : 아래 예시코드 참고

```
import time

import eojeol_etri_tokenizer.file_utils
from eojeol_etri_tokenizer.file_utils import PYTORCH_PRETRAINED_BERT_CACHE
from eojeol_etri_tokenizer.eojeol_tokenization import eojeol_BertTokenizer

eojeol_tokenizer = eojeol_BertTokenizer("./vocab.korean.rawtext.list", do_lower_case=False)
fp = open("./ner_eojeol_label_per_line.txt", "r", encoding='utf-8')
while True:
    e_sent = fp.readline()
    if len(e_sent) < 2:
        continue
    e_sent_sp = e_sent.split()
    eoj = e_sent_sp[0]
    eoj_tk = eojeol_tokenizer.tokenize(eoj)
    eoj_tkid = eojeol_tokenizer.convert_tokens_to_ids(eoj_tk) # token을 index 로 변경.
    print("eojeol=", eoj)
    for i in range(len(eoj_tk)):
        print("token=", eoj_tk[i], " id=", eoj_tkid[i])
    print("\n")
    time.sleep(3)
```

```
1  특히_ O
2  김 병 현_ B-PS
3  은_ O
4  4 회 말 에_ O
5  무 기 력 하게_ O
6  6 실 점 하면서_ O
7
8  빅 비_ B-PS
9  가_ O
10  2루_ O
11  도 루 를_ O
12  시도 하다_ O
13  아웃 됐 고_ O
14
15  서 호 프_ B-PS
16  와_ O
17  파 사 노_ B-PS
18  에게_ O
19  연속_ O
20  안타를_ O
21  내 주 며_ O
22
23  우리 금융 그룹_ B-OG
24  은_ O
```

파일변환 3: 각 줄은 token 하나 및 그 label 을 가지도록 한다

- 앞의 결과에서 한 줄이 2개 이상의 토큰을 가진 경우,
 맨 좌측 토큰보다 우측의 토큰들을 각각 새로운 줄에 나오게 한다.
- 이렇게 새로운 줄로 간 토큰들의 NE label 로 X 를 부여한다.
- 결국 모든 줄은 토큰 하나 및 그것의 NE 레이블을 가진다.
 - 우측의 예시 참고.
- 길이가 MSL = 128 보다 작으면 특수단어 및 레이블 [PAD] 로 채운다.
 - MSL: maximum sequence length
- 이렇게 얻은 파일:
 4_ner_token_label_per_line_padded.txt

"4_ner_token_label_per_line_padded.txt"

1	이	0
2		B-PS
3		X
4		X
5		0
6	4	0
7		X
8		X
9		X
10		0
11		X
12		X
13	하게	X
14	6	0
15		X
16		X
17	하면서	X
18	[PAD]	[PAD]
19	[PAD]	[PAD]
20	[PAD]	[PAD]
21	[PAD]	[PAD]
22	[PAD]	[PAD]
23	[PAD]	[PAD]
24	[PAD]	[PAD]
25	[PAD]	[PAD]
26	[PAD]	[PAD]
27	[PAD]	[PAD]
28	[PAD]	[PAD]
29	[PAD]	[PAD]
30	[PAD]	[PAD]
31	[PAD]	[PAD]
32	[PAD]	[PAD]
33	[PAD]	[PAD]
34	[PAD]	[PAD]

파일변환 4: 토큰 및 레이블을 id (index) 로 변환한 파일

- 각 토큰을 해당 id (token index)로, 각 label 을 id (label index) 로 변경한다.
- 그 결과로 얻는 파일명: "5_ner_token_id_label_id_per_line_padded.txt"
- 각 문장마다 128 개이 줄이 마련된다.
- 문장과 문장 사이에는 빈 줄(enter key 글자 하나만 가짐)이 하나 온다.

5_ner_token_id_label_id_per_line_padded.txt :


1	532	3
2	120	8
3	238	12
4	797	12
5	18	3
6	99	3
7	243	12
8	168	12
9	13	12
10	69	3
11	37	12
12	416	12
13	181	12
14	111	3
15	119	12
16	204	12
17	521	12
18	0	0
19	0	0
20	0	0
21	0	0
22	0	0
23	0	0
24	0	0
25	0	0
26	0	0
27	0	0
28	0	0
29	0	0
30	0	0
31	0	0
32	0	0
33	0	0
34	0	0

- 이 파일을 프로그램에게 제공한다.
 - 프로그램은 이 파일을 읽어 훈련예제 리스트를 준비한다.



```
NUM_CLASS = len(label_list)
```

```
label_list = ["[PAD]", "B-DT", "I-DT", "O", "B-LC", "I-LC", "B-OG", "I-OG", "B-PS", "I-PS", "B-TI", "I-TI", "X", "[CLS]", "[SEP]"]  
label id: 0      1      2      3      4      5      6      7      8      9     10     11     12     13     14
```

ID 

- PYTORCH 라이브러리를 사용하여 개발한다.
- 다음은 RNN/LSTM 을 사용하여 설계한 model architecture 이다.

```
class NER_model(nn.Module):
    def __init__(self, token_vocab_size, dim_embedding, num_hidden_layers, hidden_state_size):
        super(NER_model, self).__init__()
        self.embedding = nn.Embedding(token_vocab_size, dim_embedding, padding_idx=0)

        self.lstm = nn.LSTM(input_size=dim_embedding, hidden_size=hidden_state_size, \
                             num_layers=num_hidden_layers, batch_first=True, bidirectional=True)

        self.linear1 = nn.Linear(2*hidden_state_size, 512, bias=True)
        self.relu = nn.ReLU()

        self.linear2 = nn.Linear(512, 256, bias=True)
        self.relu = nn.ReLU()

        self.linear3 = nn.Linear(256, NUM_CLASS)

    def forward(self, X):
        x1 = self.embedding(X)
        x2 = self.lstm(x1)
        x2 = x2[0]
        x3 = self.linear1(x2)
        x4 = self.relu(x3)
        x5 = self.linear2(x4)
        x6 = self.relu(x5)
        x7 = self.linear3(x6)
        return x7

model = NER_model(SZ_TOKEN_VOCAB, DIM_EMBEDDING, NUM_HIDDEN_LAYERS, SZ_HIDDEN_STATE)
```

각 시간의 token idx 마다 token embedding vector 를 제공함

각 시간에서 입력벡터의 크기로, 이는 token embedding vtr 크기임.

Hidden 층의 memory block 의 갯수

0 인덱스의 embedding 벡터는 학습(update)에서 제외됨. 즉 embedding vector 가 갱신되지 않음.

앞층이 bid-LSTM 이므로 히든층 크기의 2 배의 출력을 가지며 이들이 입력이 됨.

입력신호의 수 = 512, 출력신호의 수(즉 뉴론수) = 256.

모델의 입력데이터임. 이것의 shape은?

모델의 출력데이터임. 이것의 shape은?

SZ_TOKEN_VOCAB = 30797
DIM_EMBEDDING = 768
NUM_HIDDEN_LAYERS = 3
SZ_HIDDEN_STATE = 512

- 한 학습예제는 (X, Y) 형태이다.
 - X: 한 문장의 token sequence(토큰열) 이다. (각 값은 id)
 - Y: X 의 각 토큰에 대하여 NE-레이블로 구성된 label sequence 이다. (각 값은 id)
 - 모델에게는 텐서 타입의 batch 로 제공하여야 한다.
 - 따라서 모든 예제가 동일한 shape 을 가져야 한다.
 - 결국, token sequence 의 길이를 특정길이(MSL) 로 통일하여야 한다.
 - 짧은 문장은 padding 이 필요하고,
 - 너무 긴 문장은 truncation 이 필요한 이유이다.

• 훈련예제 리스트 만들기

- 준비된 입력화일을 읽음
- padding/truncation 수행
- 입력과 정답을 별도의 리스트에 준비함

```
def load_X_and_Y(filename):
    ids_list = []
    with open(filename, "r", encoding="utf-8-sig") as f:
        for line in f:
            ids_list.append(line[:-1])

    X = []
    Y = []
    temp_X = []
    temp_Y = []

    for ids in ids_list:
        if len(ids) == 0: # '\n'
            assert(len(temp_X) == 128)

            X.append(temp_X)
            Y.append(temp_Y)

            temp_X = []
            temp_Y = []
        else:
            ids = ids.split('\t') # a pair of strings (string with digit sequence)

            temp_X.extend([ids[0]]) # the first string is inserted into temp_X as the last member.
            temp_Y.extend([int(ids[-1])]) # note that CrossEntropyLoss is used as loss function.
    X = np.array(X, dtype=np.int32)
    Y = np.array(Y, dtype=np.int32)
    return X, Y

all_X, all_Y = load_X_and_Y( './data/ner_token_label_per_line_padded.txt' )
```

• train, validation 데이터 준비

- 전체 예제 리스트를 적당한 비율로 나눔

```
num_examples = all_X.shape[0] # the number of examples read in from the file.
num_tra = int(0.8 * num_examples) # number of examples to be used for training.
num_val = int(0.1 * num_examples)

train_X = np.zeros((num_tra, MSL), np.int32)
train_Y = np.zeros((num_tra, MSL), np.int32)

train_X[:, :] = all_X[:num_tra, :]
train_Y[:, :] = all_Y[:num_tra, :]

dev_X, dev_Y = np.zeros((num_val, MSL), np.int32), np.zeros((num_val, MSL), np.int32)
dev_X[:, :] = all_X[num_tra:num_tra+num_val, :]
dev_Y[:, :] = all_Y[num_tra:num_tra+num_val, :]
```

dev 를 test 로 수정해야 함

- model 에게 batch 로 제공한다.
- 한 batch 의 shape: (BATCH_SIZE, MSL)
- data type: LongTensor
- 우리는 전체 예제들을 담은 두개의 리스트를 준비한다:
 - `train_X`: X 들의 리스트
 - `train_Y`: 위의 각 X 에 대응하는 Y 의 리스트
- 이를 batch 별로 모델에 제공하는 작업은 pytorch 의 DataLoader 를 이용한다.

- 데이터의 tensor 화, batch loader 준비

```

train_X = torch.LongTensor(train_X)      # 여기에서 LongTensor 대신 Tensor 로 하면 아래에서 에러 남!
train_Y = torch.LongTensor(train_Y)
train_data = TensorDataset(train_X, train_Y)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=BATCH_SIZE)

dev_X = torch.LongTensor(dev_X)          # 여기에서 LongTensor 대신 Tensor 로 하면 아래에서 에러 남!
dev_Y = torch.LongTensor(dev_Y)
dev_data = TensorDataset(dev_X, dev_Y)
dev_sampler = RandomSampler(dev_data)
dev_dataloader = DataLoader(dev_data, sampler=dev_sampler, batch_size=BATCH_SIZE)

```

dev 를 test 로 수정해야 함

- Batch loader 의 이용방법

```

for i, batch in enumerate(train_dataloader):

    batch = tuple(r.to(device) for r in batch)
    X, Y = batch
    ⋮
    preds = model(X)
    ⋮

```

```
#optimizer = optim.SGD(model.parameters(), lr=1e-5)
#optimizer = optim.Adam(model.parameters(), lr=1e-5, weight_decay=0.1)
#optimizer = optim.AdamW(model.parameters(), lr=1e-3, betas=(0.9, .999),\
                        #eps=1e-08, weight_decay=0.01)

optimizer = optim.AdamW(model.parameters(), lr=1e-4, betas=(0.9, .999),\
                        eps=1e-08, weight_decay=0.01)
```

```
CLASS torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100,  
    reduce=None, reduction='mean')
```

Parameters

- **ignore_index** (*int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets.
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`.
`'none'` : no reduction will be applied, `'mean'` : the weighted mean of the output is taken, `'sum'` : the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

The *input* is expected to contain raw, unnormalized scores for each class.

input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, \dots, d_K)$ with $K \geq 1$ for the K -dimensional case (described later).

- Input: (N, C) where $C = \text{number of classes}$, or $(N, C, d_1, d_2, \dots, d_K)$ with $K \geq 1$ in the case of K -dimensional loss.
- Target: (N) where each value is $0 \leq \text{targets}[i] \leq C - 1$, or $(N, d_1, d_2, \dots, d_K)$ with $K \geq 1$ in the case of K -dimensional loss.
- Output: scalar. If `reduction` is `'none'`, then the same size as the target: (N) , or $(N, d_1, d_2, \dots, d_K)$ with $K \geq 1$ in the case of K -dimensional loss.

- LSTM 은 RNN 의 일종으로서 각 timestep 에서의 loss 의 합으로 전체 loss 를 구한다.
- 특정 timestep (예를 들어 t) 에서의 loss 는 입력 token 에 대한 label prediction 과 target label에 의한 cross-entropy 이다.
 - 최종층은 NE-label 의 총 갯수인 num_class 개의 뉴론으로 구성됨.
 - 각 뉴론 i 는 레이블 i 를 지지하는 정도를 나타내는 출력을 내준다.
(우리 모델은 softmax 층을 적용하지 않은 값을 출력함.)
 - t 에서의 target label (index) 가 주어진다. 이 값이 d 라고 하자.
- Cross-entropy at timestep t = $-\ln(y_d^t)$ where y_d^t is the softmax result for neuron d at time t.
- 우리는 다음 pytorch 함수를 이용하여 loss 를 계산한다.

```
loss_fn = torch.nn.CrossEntropyLoss(ignore_index=0, reduction='sum')
```

- ignore_index: 무시할 target label. padding 할 때 0 을 target label 로 공급하였음. 이들은 loss 계산에 참여시키지 않음.
- 이 함수의 특징은 softmax 를 적용하지 않은 최종층 출력을 이용한다.
- 이 함수로의 입력 데이터는 (batch, class-label, timestep)의 shape 이어야 함.
- 우리 모델의 출력의 shape 과 순서가 다름.
 - transpose 가 필요한 이유임

```
preds = model(X)
preds_tr = torch.transpose(preds, 1, 2)
loss = loss_fn(preds_tr, Y)
```

- backward pass : loss 텐서의 backward 메소드의 호출
 - 모든 parameter 들의 gradient 계산
- parameter update: optimizer 객체의 메소트 step 의 호출
 - 모든 parameter 들이 새로운 값으로 갱신됨

```
loss.backward()
```

```
optimizer.step()
```



```

for e in range(EPOCHS):
    model.train()
    total_loss = 0.0
    for i, batch in enumerate(train_dataloader):
        batch = tuple(r.to(device) for r in batch)
        X, Y = batch
        optimizer.zero_grad()
        model.zero_grad()
        preds = model(X)
        preds_tr = torch.transpose(preds, 1, 2)
        loss = loss_fn(preds_tr, Y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / (Total_num_batches*BATCH_SIZE)
    print("Epoch: ", e, " is finished. Avg_loss= ", avg_loss)
    
```

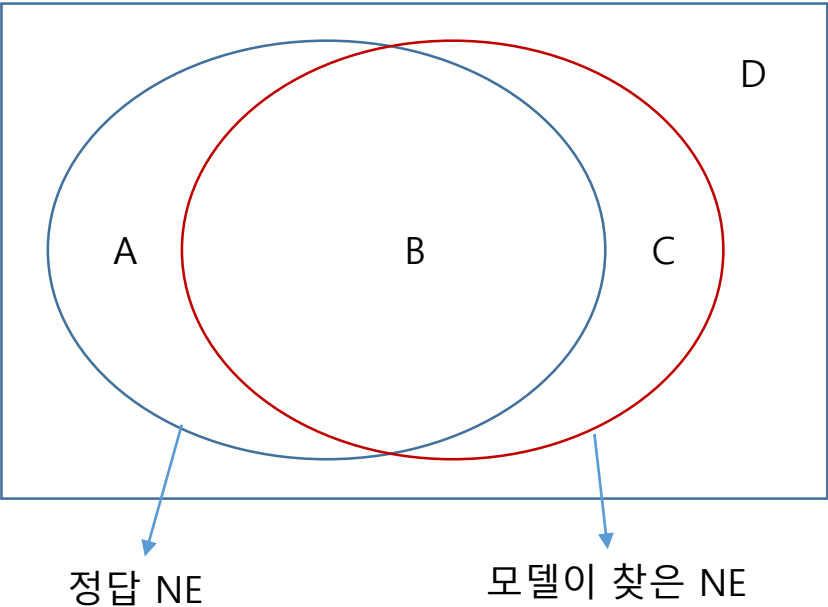
필요한 이유는?

- training 을 마친 후, 모델의 성능을 측정하는 test 을 수행한다.
- test 에 필요한 데이터는 학습에 이용된 데이터와 같은 형태를 가진다.
 - 전체 학습 데이터는 사전에 3 등분하여 놓는다 (형태는 동일):
 - train 용,
 - test 용,
 - 비율은 약 8:2 정도로 한다 (자율적으로 정하여 사용함.)
- test 필요성
 - 모델의 성능을 측정하여 본다.

- Recall : 재현률
 - 정답 NE (즉 target 가 되는 NE) 들 중에서 모델이 찾아낸 것들의 비율
 - $R = \frac{B}{A+B}$ (여기서 A: 모델이 찾지 못한 정답들의 수, B: 모델이 찾은 정답들의 수)
- Precision : 정확률
 - 모델이 찾아 낸 NE 들 중에서 정답 NE (즉 target 가 되는 NE) 들의 비율
 - $P = \frac{B}{B+C}$ (여기서 C: 모델이 찾은 NE 중에서 정답이 아닌 것들의 수)
- F1-score (F1-measure)
 - 정의: R의 역수와 P의 역수의 평균의 역수
 - $F1 = \frac{1}{\frac{1}{R} + \frac{1}{P}} = \frac{2RP}{R+P}$

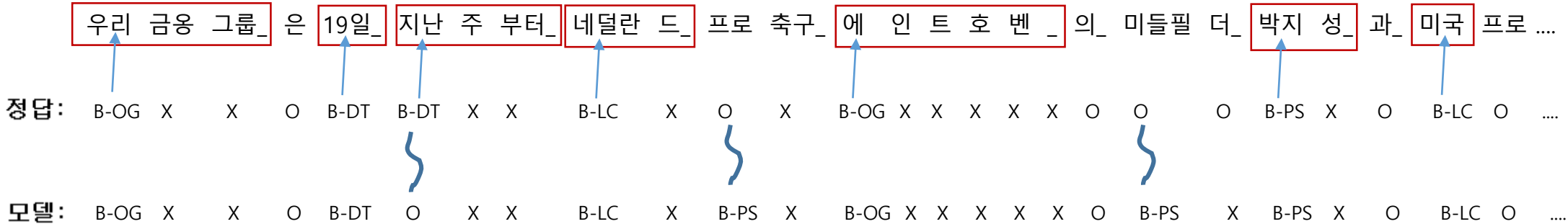
Confusion matrix:

		Real:	
		True	False
System:	True	B	C
	False	A	D



토큰 레벨의 처리 결과의 해석

- 모델은 token 레벨에서 처리를 수행한다.
 - 토큰 마다 결정을 내린다.
 - 토큰마다 정답(target)이 준비되어 있다.
- 사람은 단어(어절) 단위에서 NE 를 찾는다.
- 이 둘 사이에 mismatch 가 발생한다.
- 훈련:
 - 모든 토큰의 정답을 배우도록 훈련된다.
 - 즉 X, O 레이블을 가진 토큰들도 훈련에 참여한다.
- 예측:
 - total match: 한 NE 를 찾을 때, 이 NE 를 구성하는 모든 토큰 즉 B-, I-, X 레이블을 가진 토큰들도 모두 성공적으로 예측해야 이 NE 를 성공적으로 찾은 것으로 간주한다.
 - partial match: NE 의 첫 토큰 즉 B- 레이블을 타겟으로 가진 토큰만 찾으면 이 NE 를 성공적으로 찾은 것으로 한다.
 - 단, 찾은 NE 의 시작 위치 즉 B- 레이블의 위치는 정답의 시작 위치와 같아야 성공한 것으로 간주한다.



```
softmax_fn = torch.nn.Softmax(dim=2)
```

```
total_ne_cnt, total_match_cnt = 0, 0
with torch.no_grad():
    for k, batch in enumerate(dev_dataloader):

        batch = tuple(r.to(device) for r in batch)
        X, Y = batch
        model.eval()
        preds = model(X)
        preds = softmax_fn(preds)
        pred_label = torch.argmax(preds, dim=2)

        for i in range(len(Y)):
            target_label_seq = Y[i]
            pred_label_seq = pred_label[i]
            leng = get_leng_seq(target_label_seq) # the length of the sequence of the input.
            match_cnt, ne_cnt = 0, 0
            for j in range(leng):
                if target_label_seq[j] in [1, 4, 6, 8, 10]: # if it is one of labels
                                                            # with prefix "B-"
                    ne_cnt += 1
                    if pred_label_seq[j] == target_label_seq[j]:
                        match_cnt += 1

            total_ne_cnt += ne_cnt
            total_match_cnt += match_cnt

recall = total_match_cnt / total_ne_cnt
print("Recall of this epoch = ", recall)
```