초급반 16 조

태깅 작업 (Tagging Task)

손한기, 김경리, 이동민

Contents

목차

#태깅 작업(Tagging Task)

Tagging Task

1. 태깅 작업이란?

각 단어가 어떤 유형에 속해 있는지를 알아내는 작업

2. 태깅 작업을 하는 이유

단어는 표기는 같지만, 품사에 따라서 단어의 의미가 달라지는 경우가 있기 때문에

Ex) Fly = 날다(동사), 파리(명사)

못 = 명사(망치를 사용해서 목재 따위를 고정하는 물건), 부사(Can't)

=> 즉, 단어의 의미를 제대로 파악하기 위해서는 '해당 단어가 어떤 품사로 쓰였는지' 가 주요 지표가 될 수 있음!

3. 태깅 작업의 분류

각 단어가 어떤 유형에 속해 있는지를 알아내는 작업

1. 각 단어의 유형이 사람, 장소, 단체 등 어떤 유형인지를 알아내는 개체명 인식
   (Named Entity Recognition)

2. 각 단어의 품사가 명사, 동사, 형용사 인지를 알아내는 품사 태깅
   (Part-of-Speech Tagging)

#케라스를 이용한 태깅 작업(Tagging Task using Keras)

Tagging Task using Keras

1. 개요

- 케라스(Keras)로 인공 신경망을 이용하여 태깅 작업을 하는 모델(개체명 인식기와 품사 태거)을 제작합니다.

- 두 작업은 RNN의 다-대-다(Many-to-Many) 작업이면서 또한 앞, 뒤 시점의 입력을 모두 참고하는 양방향 RNN(Bidirectional RNN)을 사용합니다.

2. 양방향 LSTM(Bidirectional LSTM)

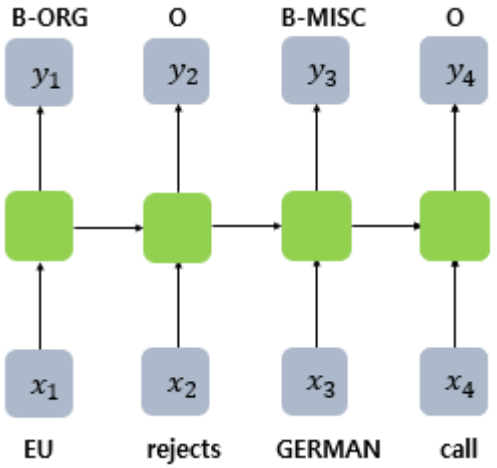
- 바닐라 RNN 보다 성능이 개선된 RNN인 LSTM을 사용

```
model.add(Bidirectional(LSTM(hidden_size, return_sequences=True)))
```

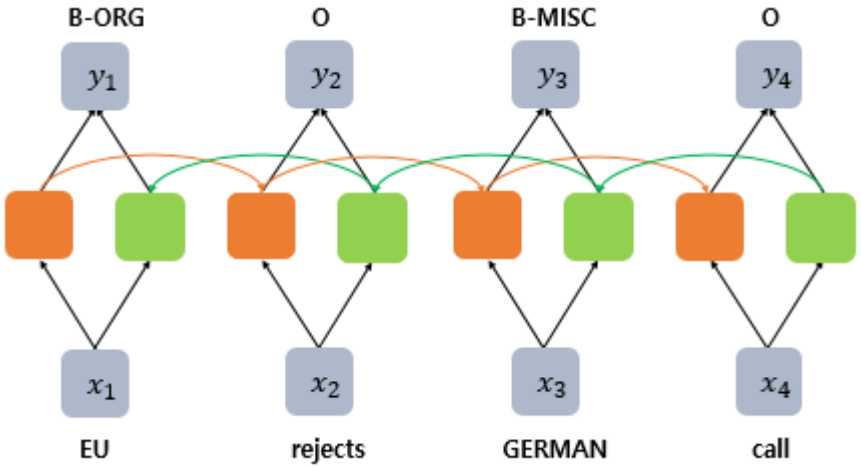
- 양방향 LSTM을 사용하기 위해 LSTM()을 Bidirectional() 안에 넣어 줍니다.

- return_sequences=True 로 설정하여 출력층에 모든 은닉 상태값을 보냅니다.

2. 양방향 LSTM(Bidirectional LSTM)



다-대-다(Many-to-Many)



양방향 RNN

3. 훈련 데이터에 대한 이해

-	X_train	y_train	lenght
0	['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'lamb']	['B-ORG', 'O', 'B-MISC', 'O', 'O', 'O', 'B-MISC', 'O']	8
1	['peter', 'blackburn']	['B-PER', 'I-PER']	2
2	['brussels', '1996-08-22']	['B-LOC', 'O']	2
3	['The', 'European', 'Commission']	['O', 'B-ORG', 'I-ORG']	3

- 태깅 작업은 텍스트 분류 작업과 동일하게 지도학습(Supervised Learning)

- 태깅을 해야하는 단어 데이터를 X, 레이블에 해당되는 태깅 정보 데이터는 y라고 이름을 붙였습니다.

- X에 대한 훈련 데이터는 X_train, 테스트 데이터는 X_test라고 명명하고
  Y에 대한 훈련 데이터는 y_train, 테스트 데이터는 y_test라고 명명합니다.

4. 시퀀스 레이블링(Sequence Labeling)

-	X_train	y_train	lenght
0	['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'lamb']	['B-ORG', 'O', 'B-MISC', 'O', 'O', 'O', 'B-MISC', 'O']	8
1	['peter', 'blackburn']	['B-PER', 'I-PER']	2
2	['brussels', '1996-08-22']	['B-LOC', 'O']	2
3	['The', 'European', 'Commission']	['O', 'B-ORG', 'I-ORG']	3

- X_train[3]의 'The'와 y_train[3]의 'O'는 하나의 쌍(pair)입니다. 또한, X_train[3]의 'European'과 y_train[3]의 'B-ORG'는 쌍의 관계를 가지며, X_train[3]의 'Commision'과 y_train[3]의 'I-ORG'는 쌍의 관계를 가집니다.

- 이처럼 입력 시퀀스 X = [x1, x2, x3, ..., xn]에 대하여 레이블 시퀀스 y = [y1, y2, y3, ..., yn]를 각각 부여하는 작업을 시퀀스 레이블링 작업(Sequence Labeling Task)이라고 합니다.

4. 시퀀스 레이블링(Sequence Labeling)

-	X_train	y_train	lenght
0	['EU', 'rejects', 'German', 'call', 'to', 'boycott', 'British', 'lamb']	['B-ORG', 'O', 'B-MISC', 'O', 'O', 'O', 'B-MISC', 'O']	8
1	['peter', 'blackburn']	['B-PER', 'I-PER']	2
2	['brussels', '1996-08-22']	['B-LOC', 'O']	2
3	['The', 'European', 'Commission']	['O', 'B-ORG', 'I-ORG']	3

- X_train[3]의 'The'와 y_train[3]의 'O'는 하나의 쌍(pair)입니다. 또한, X_train[3]의 'European'과 y_train[3]의 'B-ORG'는 쌍의 관계를 가지며, X_train[3]의 'Commision'과 y_train[3]의 'I-ORG'는 쌍의 관계를 가집니다.

- 이처럼 입력 시퀀스 X = [x1, x2, x3, ..., xn]에 대하여 레이블 시퀀스 y = [y1, y2, y3, ..., yn]를 각각 부여하는 작업을 시퀀스 레이블링 작업(Sequence Labeling Task)이라고 합니다.

양방향 LSTM를 이용한 품사 태깅 (Part-of-speech Tagging using Bi-LSTM)

Part-of-speech Tagging using Bi-LSTM

0. 개요

- chapter2 토큰화 에서 NLTK 와 KONLPy 를 이용해서 품사를 태깅하였습니다.

- 양방향 LSTM을 이용한 품사 태깅을 수행하는 모델을 직접 제작

    1. 품사 태깅 데이터에 대한 이해와 전처리

    2. 양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기

    3. 양방향 LSTM + CRF(Bi-directional LSTM + CRF)으로 POS Tagger 만들기

Part-of-speech Tagging using Bi-LSTM

1. 품사 태깅 데이터에 대한 이해와 전처리

```
import nltk
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
```

- 필요한 module 을 import 합니다.

1. 품사 태깅 데이터에 대한 이해와 전처리

```
In [2]: tagged_sentences = nltk.corpus.treebank.tagged_sents() # 토큰화에 품사 태깅이 된 데이터 받아오기

        print("품사 태깅이 된 문장 개수: ", len(tagged_sentences)) # 문장 샘플의 개수 출력

        품사 태깅이 된 문장 개수:  3914
```

- NLTK를 이용하여 영어 코퍼스에 토큰화와 품사 태깅 전처리를 진행한 문장 데이터를 받아옵니다.

- 총 3914 개의 샘플이 있는 것을 확인할 수 있습니다.

Part-of-speech Tagging using Bi-LSTM

1. 품사 태깅 데이터에 대한 이해와 전처리

```
print(tagged_sentences[0]) # 첫번째 샘플 출력
```

```
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'),
('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'), ('board',
'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'),
('Nov.', 'NNP'), ('29', 'CD'), ('.', '.')]
```

- POS Tagging 된 자료가 나옵니다.(tag 에 대한 내용은 아래를 참조해 주세요)

https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

- 훈련을 시키기 위해서 훈련 데이터에서 '단어에 해당되는 부분'과 '품사 태깅 정보에 해당되는 부분'을 분리해야 함

['Pierre', Vinken, '61', ...] ['NNP', 'NNP', 'CD' ...]

Part-of-speech Tagging using Bi-LSTM

1. 품사 태깅 데이터에 대한 이해와 전처리

```
sentences, pos_tags = [], []
for tagged_sentence in tagged_sentences: # 3,914개의 문장 샘플을 1개씩 불러온다.
    sentence, tag_info = zip(*tagged_sentence) # 각 샘플에서 단어들은 sentence에 품사 태깅 정보들은 tag_info에 저장한다.
    sentences.append(list(sentence)) # 각 샘플에서 단어 정보만 저장한다.
    pos_tags.append(list(tag_info)) # 각 샘플에서 품사 태깅 정보만 저장한다.
```

- zip()함수를 이용해서 sentences 와 pos_tag 로 나눠줍니다. (chapter2 데이터 분리)

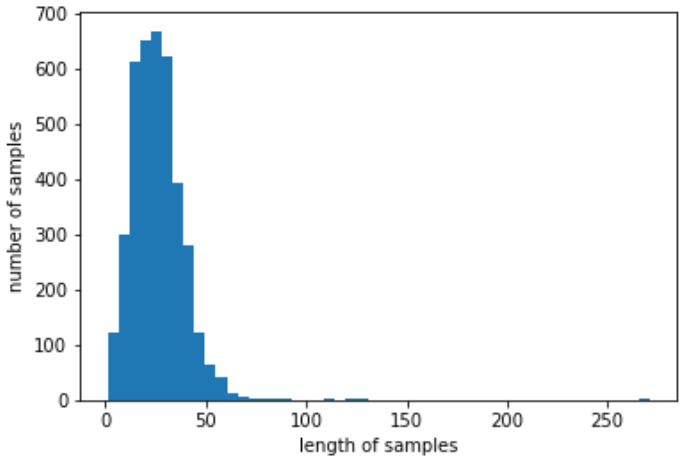
```
print(sentences[0])
print(pos_tags[0])
```

```
Sentences[0] =[
'Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutive',
'director', 'Nov.', '29', '.'
]
Pos_tags = [
'NNP', 'NNP', ',', 'CD', 'NNS', 'JJ', ',', 'MD', 'VB', 'DT', 'NN', 'IN', 'DT', 'JJ', 'NN', 'NNP', 'CD', '.'
]
```

1. 품사 태깅 데이터에 대한 이해와 전처리

```
print('샘플의 최대 길이 : %d' % max(len(l) for l in sentences))
print('샘플의 평균 길이 : %f' % (sum(map(len, sentences))/len(sentences)))
plt.hist([len(s) for s in sentences], bins=50)
plt.xlabel('length of samples')
plt.ylabel('number of samples')
plt.show()
```

샘플의 최대 길이 : 271 샘플의 평균 길이 : 25.722024



- 대부분의 샘플의 길이가 150 이내이며
- 대부분 0~50의 길이를 가지는 것을 알 수 있습니다.

1. 품사 태깅 데이터에 대한 이해와 전처리

- 정수 인코딩

```
def tokenize(samples):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(samples)
  return tokenizer
```

```
src_tokenizer = tokenize(sentences)
tar_tokenizer = tokenize(pos_tags)
```

- Sentences 를 scr_tokenizer, Pos_tags 를 tar_tokenizer

```
vocab_size = len(src_tokenizer.word_index) + 1
tag_size = len(tar_tokenizer.word_index) + 1
print('단어 집합의 크기 : {}'.format(vocab_size))
print('태깅 정보 집합의 크기 : {}'.format(tag_size))
```

단어 집합의 크기 : 11388 태깅 정보 집합의 크기 : 47

1. 품사 태깅 데이터에 대한 이해와 전처리

- 정수 인코딩

```
X_train = src_tokenizer.texts_to_sequences(sentences)
y_train = tar_tokenizer.texts_to_sequences(pos_tags)
print(X_train[:2])
print(y_train[:2])
```

```
X_train[:2] = [[5601, 3746, 1, 2024, 86, 331, 1, 46, 2405, 2, 131, 27, 6, 2025, 332, 459, 2026, 3],
               [31, 3746, 20, 177, 4, 5602, 2915, 1, 2, 2916, 637, 147, 3]]
y_train[:2] = [[3, 3, 8, 10, 6, 7, 8, 21, 13, 4, 1, 2, 4, 7, 1, 3, 10, 9],
               [3, 3, 17, 1, 2, 3, 3, 8, 4, 3, 19, 1, 9]]
```

-모든 샘플의 길이를 150 으로 맞춰줍니다.

```
max_len = 150
X_train = pad_sequences(X_train, padding='post', maxlen=max_len)
# X_train의 모든 샘플의 길이를 맞출 때 뒤의 공간에 숫자 0으로 채움.
y_train = pad_sequences(y_train, padding='post', maxlen=max_len)
# y_train의 모든 샘플의 길이를 맞출 때 뒤의 공간에 숫자 0으로 채움.
```

1. 품사 태깅 데이터에 대한 이해와 전처리

- 정수 인코딩

```
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=.2, random_state=777)
y_train = to_categorical(y_train, num_classes=tag_size)
y_test = to_categorical(y_test, num_classes=tag_size)
```

- 훈련 데이터와 테스트 데이터를 8:2의 비율로 분리
- 원-핫 인코딩을 수행

```
print('훈련 샘플 문장의 크기 : {}'.format(X_train.shape))
print('훈련 샘플 레이블의 크기 : {}'.format(y_train.shape))
print('테스트 샘플 문장의 크기 : {}'.format(X_test.shape))
print('테스트 샘플 레이블의 크기 : {}'.format(y_test.shape))
```

```
훈련 샘플 문장의 크기 : (3131, 150)
훈련 샘플 레이블의 크기 : (3131, 150, 47)
테스트 샘플 문장의 크기 : (783, 150)
테스트 샘플 레이블의 크기 : (783, 150, 47)
```

2.양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, InputLayer, Bidirectional, TimeDistributed, Embedding
from tensorflow.keras.optimizers import Adam
```

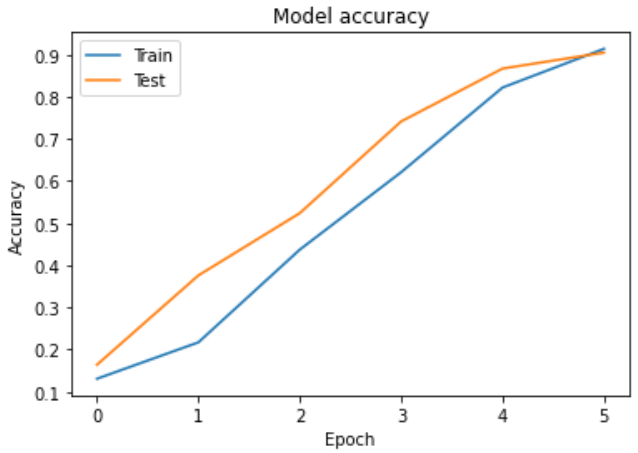
```
model = Sequential()
model.add(Embedding(vocab_size, 128, input_length=max_len, mask_zero=True))
model.add(Bidirectional(LSTM(256, return_sequences=True)))
model.add(TimeDistributed(Dense(tag_size, activation=('softmax'))))
model.compile(loss='categorical_crossentropy', optimizer=Adam(0.001), metrics=['accuracy'])
```

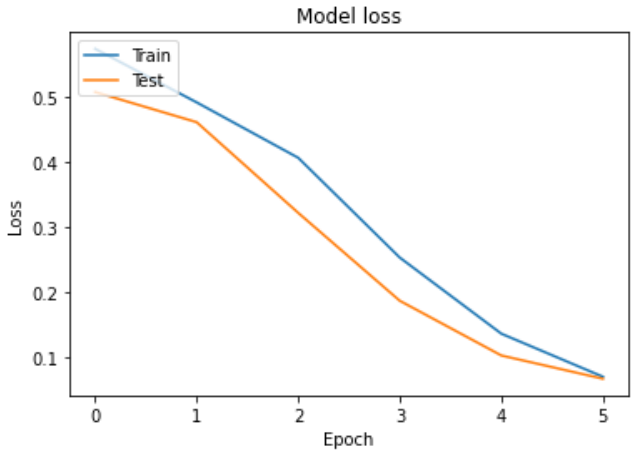
```
model.fit(X_train, y_train, batch_size=128, epochs=6,  validation_data=(X_test, y_test))
```

```
Epoch 1/6
25/25 [==============================] - 12s 153ms/step - loss: 0.5736 - accuracy: 0.1367 - val_loss: 0.5068 - val_accuracy: 0.1770
Epoch 2/6
25/25 [==============================] - 1s 60ms/step - loss: 0.4926 - accuracy: 0.2221 - val_loss: 0.4624 - val_accuracy: 0.3678
Epoch 3/6
25/25 [==============================] - 1s 59ms/step - loss: 0.4070 - accuracy: 0.4271 - val_loss: 0.3218 - val_accuracy: 0.5119
Epoch 4/6
25/25 [==============================] - 1s 59ms/step - loss: 0.2513 - accuracy: 0.6291 - val_loss: 0.1845 - val_accuracy: 0.7474
Epoch 5/6
25/25 [==============================] - 1s 58ms/step - loss: 0.1329 - accuracy: 0.8251 - val_loss: 0.1017 - val_accuracy: 0.8639
Epoch 6/6
25/25 [==============================] - 1s 59ms/step - loss: 0.0689 - accuracy: 0.9113 - val_loss: 0.0673 - val_accuracy: 0.9020
```

2.양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기

```
Epoch 1/6
25/25 [==============================] - 12s 153ms/step - loss: 0.5736 - accuracy: 0.1367 - val_loss: 0.5068 - val_accuracy: 0.1770
Epoch 2/6
25/25 [==============================] - 1s 60ms/step - loss: 0.4926 - accuracy: 0.2221 - val_loss: 0.4624 - val_accuracy: 0.3678
Epoch 3/6
25/25 [==============================] - 1s 59ms/step - loss: 0.4070 - accuracy: 0.4271 - val_loss: 0.3218 - val_accuracy: 0.5119
Epoch 4/6
25/25 [==============================] - 1s 59ms/step - loss: 0.2513 - accuracy: 0.6291 - val_loss: 0.1845 - val_accuracy: 0.7474
Epoch 5/6
25/25 [==============================] - 1s 58ms/step - loss: 0.1329 - accuracy: 0.8251 - val_loss: 0.1017 - val_accuracy: 0.8639
Epoch 6/6
25/25 [==============================] - 1s 59ms/step - loss: 0.0689 - accuracy: 0.9113 - val_loss: 0.0673 - val_accuracy: 0.9020
```





2.양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기

```
index_to_word=src_tokenizer.index_word
index_to_tag=tar_tokenizer.index_word

i=4 # 확인하고 싶은 테스트용 샘플의 인덱스.
y_predicted = model.predict(np.array([X_test[i]])) # 입력한 테스트용 샘플에 대해서 예측 y를 리턴
print (y_predicted)
y_predicted = np.argmax(y_predicted, axis=-1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.
print (y_predicted)
true = np.argmax(y_test[i], -1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.

print("{:15}|{:5}|{}".format("단어", "실제값", "예측값"))
print(35 * "-")

for w, t, pred in zip(X_test[i], true, y_predicted[0]):
    if w != 0: # PAD값은 제외함.
        print("{:17}: {:7} {}".format(index_to_word[w], index_to_tag[t].upper(), index_to_tag[pred].upper()))
```

```
[[[1.8879127e-06 1.6558263e-03 1.7224101e-04 ... 2.8803431e-06 1.6533038e-06 1.2959171e-06]
[3.2604567e-06 1.2074580e-02 3.0342508e-05 ... 5.4572911e-06 4.4504427e-06 2.6098680e-06]
[1.3069923e-06 2.7858157e-04 2.8647974e-04 ... 1.3848685e-06 1.7156316e-06 3.6600818e-07]
...
[2.0916359e-02 2.1964937e-02 2.1704426e-02 ... 2.0960502e-02 2.0948686e-02 2.0980909e-02]
[2.0916359e-02 2.1964937e-02 2.1704426e-02 ... 2.0960502e-02 2.0948686e-02 2.0980909e-02]
[2.0916359e-02 2.1964937e-02 2.1704426e-02 ... 2.0960502e-02 2.0948686e-02 2.0980909e-02]]]


[[ 3  3 17  5  3  3 12 16 15  1  6  2  4  7  1 11  7  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
9  9]]
```

2.양방향 LSTM(Bi-directional LSTM)으로 POS Tagger 만들기

```
print("{:15}|{:5}|{}".format("단어", "실제값", "예측값"))
print(35 * "-")

for w, t, pred in zip(X_test[i], true, y_predicted[0]):
    if w != 0: # PAD값은 제외함.
        print("{:17}: {:7} {}".format(index_to_word[w], index_to_tag[t].upper(), index_to_tag[pred].upper()))
```

```
단어              |실제값  |예측값
-----------------------------------
mr.               : NNP     NNP
watson            : NNP     NNP
says              : VBZ     VBZ
0                 : -NONE-  -NONE-
mrs.              : NNP     NNP
yeargin           : NNP     NNP
never             : RB      RB
complained        : VBD     VBD
to                : TO      TO
school            : NN      NN
officials         : NNS     NNS
that              : IN      IN
the               : DT      DT
standardized      : JJ      JJ
test              : NN      NN
was               : VBD     VBD
unfair            : JJ      JJ
.                 : .       .
```

3. 양방향 LSTM + CRF(Bi-directional LSTM + CRF)으로 POS Tagger 만들기

```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Embedding, Bidirectional, LSTM, TimeDistributed, Dense, Input
from tf2crf import CRF, ModelWithCRFLoss
```

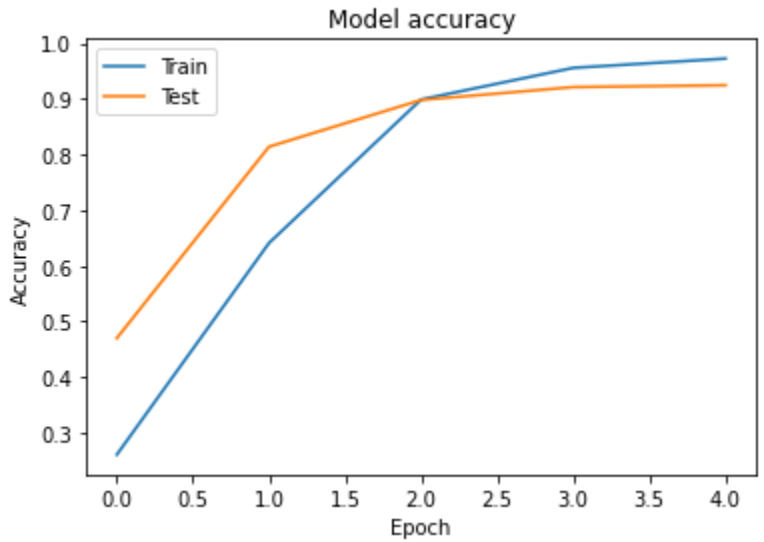
```
inputs = Input(shape=(None,), dtype='int32')
output = Embedding(vocab_size, 128, mask_zero=True)(inputs)
output = Bidirectional(LSTM(128, return_sequences=True))(output)
crf = CRF(tag_size)
output = crf(output)
base_model = Model(inputs, output)

model = ModelWithCRFLoss(base_model, sparse_target=False)
model.build(input_shape=(None, 22))
model.compile(optimizer='adam')
```

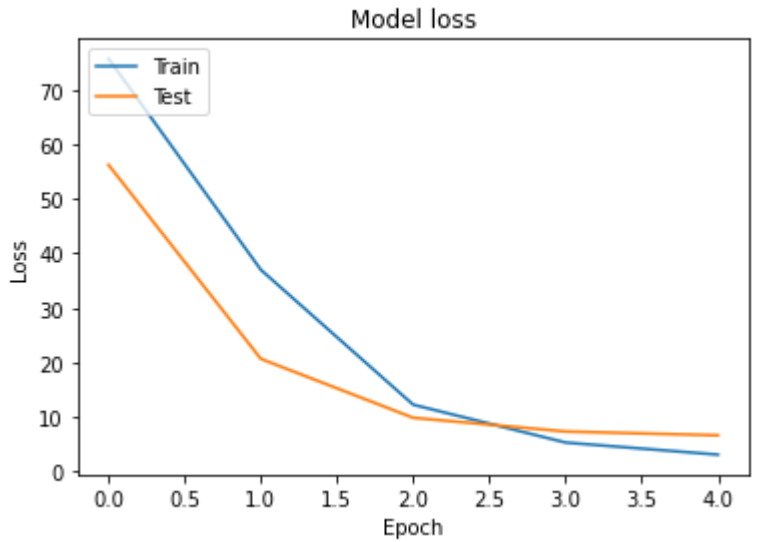
3. 양방향 LSTM + CRF(Bi-directional LSTM + CRF)으로 POS Tagger 만들기

```
history = model.fit(X_train, y_train, batch_size = 32, epochs = 5, validation_split = 0.2, verbose = 1)
```

```
Epoch 1/5 79/79 [==============================] - 70s 794ms/step-loss: 75.6673 - accuracy: 0.2608 - val_loss_val: 56.2252 - val_val_accuracy: 0.4702
Epoch 2/5 79/79 [==============================] - 61s 773ms/step-loss: 36.9858 - accuracy: 0.6412 - val_loss_val: 20.6731 - val_val_accuracy: 0.8143
Epoch 3/5 79/79 [==============================] - 64s 810ms/step-loss: 12.2725 - accuracy: 0.8992 - val_loss_val: 9.8749 - val_val_accuracy: 0.8983
Epoch 4/5 79/79 [==============================] - 60s 755ms/step-loss: 5.3434 - accuracy: 0.9560 - val_loss_val: 7.3667 - val_val_accuracy: 0.9214
Epoch 5/5 79/79 [==============================] - 67s 852ms/step-loss: 3.1276 - accuracy: 0.9726 - val_loss_val: 6.6643 - val_val_accuracy: 0.9248
```

```
25/25 [==============================] - 6s 251ms/step - loss_val: 6.6484 - val_accuracy: 0.9272 테스트 정확도: 0.9263
```





3. 양방향 LSTM + CRF(Bi-directional LSTM + CRF)으로 POS Tagger 만들기

```
index_to_word=src_tokenizer.index_word
index_to_tag=tar_tokenizer.index_word

i=2 # 확인하고 싶은 테스트용 샘플의 인덱스.
y_predicted = zmodel.predict(np.array([X_test[i]])) # 입력한 테스트용 샘플에 대해서 예측 y를 리턴
true = np.argmax(y_test[i], -1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.

print("{:15}|{:5}|{}".format("단어", "실제값", "예측값"))
print(35 * "-")

for w, t, pred in zip(X_test[i], true, y_predicted[0]):
    if w != 0: # PAD값은 제외함.
        print("{:17}: {:7} {}".format(index_to_word[w], index_to_tag[t].upper(), index_to_tag[pred].upper()))
```

```
단어              |실제값  |예측값
-----------------------------------
i                 : PRP     PRP
believe           : VBP     VBP
0                 : -NONE-  -NONE-
you               : PRP     PRP
have              : VBP     VBP
*-1               : -NONE-  -NONE-
to                : TO      TO
use               : VB      VB
the               : DT      DT
system            : NN      NN
*-2               : -NONE-  -NONE-
to                : TO      TO
change            : VB      VB
it                : PRP     PRP
.                 : .       .
```

y-predicted 값이 정수 인코딩으로 나오기 때문에 바로 적용 가능

태깅작업(Tagging Task)

3, 4 파트

3) 개체명 인식(Named Entity Recognition)

개체명 인식이란?

이름을 가진 개체를 인식하는 것

➡ 문자열을 입력으로 받아 단어별로 해당되는 태그를 내뱉게 하는 multi-class 분류작업

유정이는 **2018년**에 **골드만삭스**에 입사했다.

사람 시간 조직

개체명 인식을 위해서는 토큰화, 품사 태깅 등 전처리 과정이 필요

3) 개체명 인식(Named Entity Recognition)

NLTK를 이용한 개체명 인식

Python 라이브러리 중 NLTK에서 개체명 인식기(NER chunker)를 지원

1. 라이브러리 import & NER 대상 문자열 선언

```
from nltk import word_tokenize, pos_tag, ne_chunk
sentence = "James is working at Disney in London"
```

2. 문자열을 단어단위로 토큰화 & 품사태깅

```
sentence=pos_tag(word_tokenize(sentence))
print(sentence) # 토큰화와 품사 태깅을 동시 수행
```

[('James', 'NNP'), ('is', 'VBZ'), ('working', 'VBG'), ('at', 'IN'), ('Disney', 'NNP'), ('in', 'IN'), ('London', 'NNP')]

3. 토큰화 및 품사태깅된 문자열에서 개체명 추출

```
sentence=ne_chunk(sentence)
print(sentence) # 개체명 인식
```

(S (PERSON James/NNP) **is**/VBZ working/VBG at/IN (ORGANIZATION Disney/NNP) **in**/IN (GPE London/NNP))

James는 PERSON(사람), Disney는 조직(ORGANIZATION), London은 위치(GPE) 정상적으로 NER 수행

4) 개체명 인식의 BIO 표현 이해하기

특정 도메인에 맞는 개체를 인식하기 위해서는 개체명인식기를 만들어 사용

BIO 표현

코퍼스에서 개체명을 인식하는 보편적인 방법

B(Begin의 약자) – 개체명이 시작하는 부분
I(Inside의 약자) – 개체명의 내부 부분
O(Outside의 약자) – 개체명이 아닌 부분

해	리	포	터		보	러		메	가	박	스		가	자
B	I	I	I		O	O		B	I	I	I		O	O
			movie							theater				

B와 I는 개체명을 위해 사용, O는 개체명이 아니라는 의미
각 개체가 어떤 종류인지도 함께 태깅

4) 개체명 인식의 BIO 표현 이해하기

양방향 LSTM을 이용한 개체명 인식기

CONLL 2003

EU NNP B-NP B-ORG
rejects VBZ B-VP O
German JJ B-NP B-MISC
call NN I-NP O
to TO B-VP O
boycott VB I-VP O
British JJ B-NP B-MISC
lamb NN I-NP O
. . O O
Peter NNP B-NP B-PER
Blackburn NNP I-NP I-PER

[단어] [품사 태깅] [청크 태깅] [개체명 태깅]

고유명사
단수형
(NNP)
3인칭 단수
동사 현재
형(VBZ)

https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

인물(PER)
기관 및 단체(ORG)
장소 및 위치(LOC)
기타(MISC)

4) 개체명 인식의 BIO 표현 이해하기

데이터 전처리 하기

1. 라이브러리 import

```
import re
%matplotlib inline
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
import numpy as np
```

2. 개체명 인식 데이터를 읽어 전처리 수행

```
f = open('train.txt의 현재 경로', 'r')
tagged_sentences = []
sentence = []

for line in f:
        if len(line)==0 or line.startswith('-DOCSTART') or line[0]=="₩n":
          if len(sentence) > 0:
                tagged_sentences.append(sentence)
                sentence = []
          continue
        splits = line.split(' ') # 공백을 기준으로 속성을 구분한다.
        splits[-1] = re.sub(r'₩n', '', splits[-1]) # 줄바꿈 표시 ₩n을 제거한다.
        word = splits[0].lower() # 단어들은 소문자로 바꿔서 저장한다.
        sentence.append([word, splits[-1]]) # 단어와 개체명 태깅만 기록한다.
```

4) 개체명 인식의 BIO 표현 이해하기

3. 전체 샘플 개수 확인 & 첫번째 샘플 출력

```
print("전체 샘플 개수: ", len(tagged_sentences)) # 전체 샘플의 개수 출력
```

전체 샘플 개수 : 14041

```
print(tagged_sentences[0]) # 첫번째 샘플 출력
```

[['eu', 'B-ORG'], ['rejects', 'O'], ['german', 'B-MISC'], ['call', 'O'], ['to', 'O'], ['boycott', 'O'], ['british', 'B-MISC'], ['lamb', 'O'], ['.', 'O']]

4. 학습을 위해 단어와 개체명 태깅 정보 분리

```
sentences, ner_tags = [], []
for tagged_sentence in tagged_sentences: # 14,041개의 문장 샘플을 1개씩 불러온다.
        sentence, tag_info = zip(*tagged_sentence) # 각 샘플에서 단어들은 sentence에 개체명 태깅 정보들은 tag_info에 저장.
        sentences.append(list(sentence)) # 각 샘플에서 단어 정보만 저장한다.
        ner_tags.append(list(tag_info)) # 각 샘플에서 개체명 태깅 정보만 저장한다.
```

```
print(sentences[0]) #단어는 예측을 위한 X
print(ner_tags[0]) #예측대상인 Y
```

['eu', 'rejects', 'german', 'call', 'to', 'boycott', 'british', 'lamb', '.']
['B-ORG', 'O', 'B-MISC', 'O', 'O', 'O', 'B-MISC', 'O', 'O']

```
print(sentences[12]) #단어는 예측을 위한 X
print(ner_tags[12]) #예측대상인 Y
```

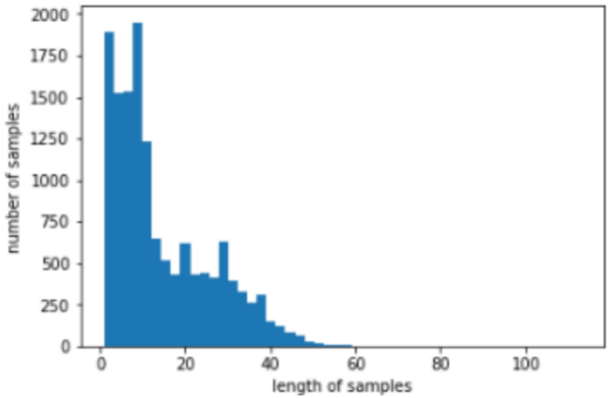
['only', 'france', 'and', 'britain', 'backed', 'fischler', "'s", 'proposal', '.']
['O', 'B-LOC', 'O', 'B-LOC', 'O', 'B-PER', 'O', 'O', 'O']

4) 개체명 인식의 BIO 표현 이해하기

5. 전체 데이터 길이 분포 확인

```
print('샘플의 최대 길이 : %d' % max(len(l) for l in sentences))
print('샘플의 평균 길이 : %f' % (sum(map(len, sentences))/len(sentences)))
plt.hist([len(s) for s in sentences], bins=50)
plt.xlabel('length of samples')
plt.ylabel('number of samples')
plt.show()
```

샘플의 최대 길이 : 113
샘플의 평균 길이 : 14.501887



샘플의 길이는 대체적으로 0~40
0~20 의 길이를 가진 샘플이 큰 비중

4) 개체명 인식의 BIO 표현 이해하기

6. 토큰화와 정수 인코딩

```
max_words = 4000 #높은 빈도수를 가진 4000개의 단어만 사용
src_tokenizer = Tokenizer(num_words=max_words, oov_token='OOV')
src_tokenizer.fit_on_texts(sentences) #문장 데이터

tar_tokenizer = Tokenizer()
tar_tokenizer.fit_on_texts(ner_tags) #개체명 태깅 정보
```

```
vocab_size = max_words
tag_size = len(tar_tokenizer.word_index) + 1
print('단어 집합의 크기 : {}'.format(vocab_size))
print('개체명 태깅 정보 집합의 크기 : {}'.format(tag_size))
```

```
단어 집합의 크기 : 4000
개체명 태깅 정보 집합의 크기 : 10
```

```
X_train = src_tokenizer.texts_to_sequences(sentences) #문장 정수인코딩
y_train = tar_tokenizer.texts_to_sequences(ner_tags) #개체명 태깅 정수인코딩
```

```
print(X_train[0])
print(y_train[0])
```

```
[989, 1, 205, 629, 7, 3939, 216, 1, 3]
[4, 1, 7, 1, 1, 1, 7, 1, 1]
```

4) 개체명 인식의 BIO 표현 이해하기

7. 디코딩을 통한 확인

```
index_to_word = src_tokenizer.index_word #정수에서 텍스트 데이터로 변환
index_to_ner = tar_tokenizer.index_word
```

```
decoded = []
for index in X_train[0] : # 첫번째 샘플 안의 인덱스들에 대해서
                decoded.append(index_to_word[index]) # 다시 단어로 변환
print('기존 문장 : {}'.format(sentences[0]))
print('빈도수가 낮은 단어가 OOV 처리된 문장 : {}'.format(decoded)) #빈도수가 4000안에 들지 않으면 OOV로 표시됨
```

```
기존 문장 : ['eu', 'rejects', 'german', 'call', 'to', 'boycott', 'british', 'lamb', '.']
빈도수가 낮은 단어가 OOV 처리된 문장 : ['eu', 'OOV', 'german', 'call', 'to', 'boycott', 'british', 'OOV', '. ' ]
```

8. 길이를 맞춘 후 Train data / Test data 로 분리

```
max_len = 70 #모든 샘플 길이를 70으로 맞춤
X_train = pad_sequences(X_train, padding='post', maxlen=max_len) # X_train의 모든 샘플들의 길이를 맞출 때 뒤의 공간에 숫자 0으로 채움.
y_train = pad_sequences(y_train, padding='post', maxlen=max_len) # y_train의 모든 샘플들의 길이를 맞출 때 뒤의 공간에 숫자0으로 채움.
```

```
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=.2, random_state=777)
#8:2 비율로 학습 데이터와 테스트데이터로 분리
```

4) 개체명 인식의 BIO 표현 이해하기

9. 태깅 정보에 대해서 원-핫인코딩 수행

```
y_train = to_categorical(y_train, num_classes=tag_size)
y_test = to_categorical(y_test, num_classes=tag_size
```

```
print('훈련 샘플 문장의 크기 : {}'.format(X_train.shape))
print('훈련 샘플 레이블의 크기 : {}'.format(y_train.shape))
print('테스트 샘플 문장의 크기 : {}'.format(X_test.shape))
print('테스트 샘플 레이블의 크기 : {}'.format(y_test.shape))
```

```
훈련 샘플 문장의 크기 : (11232, 70)
훈련 샘플 레이블의 크기 : (11232, 70, 10)
테스트 샘플 문장의 크기 : (2809, 70)
테스트 샘플 레이블의 크기 : (2809, 70, 10)
```

4) 개체명 인식의 BIO 표현 이해하기

양방향 LSTM(Bi-directional LSTM)으로 개체명 인식기 만들기

1. 라이브러리 import

```
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM, Bidirectional, TimeDistributed
from keras.optimizers import Adam
```

2. 모델생성

```
model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=128, input_length=max_len,
mask_zero=True)) #패딩을 통해 0이 많아졌으므로 0은 연산에서 제외
model.add(Bidirectional(LSTM(256, return_sequences=True))) #Many to Many
model.add(TimeDistributed(Dense(tag_size, activation='softmax')))
```

3. 모델학습

```
model.compile(loss='categorical_crossentropy', optimizer=Adam(0.001), metrics=['accuracy'])
model.fit(X_train, y_train, batch_size=128, epochs=8, validation_data=(X_test, y_test))
```

```
Train on 11232 samples, validate on 2809 samples
Epoch 1/8 11232/11232
[==============================] - 36s 3ms/sample -
loss: 0.1890 - acc: 0.8254 - val_loss: 0.1270 -
val_acc: 0.8334
... 중략 ...
Epoch 8/8 11232/11232
[==============================] - 34s 3ms/sample -
loss: 0.0214 - acc: 0.9689 - val_loss: 0.0318 -
val_acc: 0.9573
```

4) 개체명 인식의 BIO 표현 이해하기

4. 정확도 평가

```
print("₩n 테스트 정확도: %.4f" % (model.evaluate(X_test, y_test)[1]))
```

테스트 정확도: 0.9573

5. 실제 데이터와 비교

```
i=10 # 확인하고 싶은 테스트용 샘플의 인덱스.
y_predicted = model.predict(np.array([X_test[i]])) # 입력한 테스트용 샘플에 대해서 예측 y를 리턴
y_predicted = np.argmax(y_predicted, axis=-1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.
true = np.argmax(y_test[i], -1) # 원-핫 인코딩을 다시 정수 인코딩으로 변경함.

print("{:15}|{:5}|{}".format("단어", "실제값", "예측값"))
print(35 * "-")
for w, t, pred in zip(X_test[i], true, y_predicted[0]):
                if w != 0: # PAD값은 제외함.
                                print("{:17}: {:7} {}".format(index_to_word[w], index_to_ner[t].upper(),
index_to_ner[pred].upper()))
```

```
단어 |실제값 |예측값
-----------------------------------
sarah : B-PER B-PER
brady : I-PER I-PER
, : O O
whose : O O
republican : B-MISC B-MISC
husband : O O
was : O O
OOV : O O
OOV : O O
```

출력결과는 유사해보이지만 정확도 측정 방법에 문제가 있음 -> 전부 O로 예측해도 정확도가 높게 나오기 때문

13. 태깅 작업(Tagging Task)

5) BiLSTM을 이용한 개체명 인식(Named Entity Recognition, NER)

1. 개체명 인식 데이터에 대한 이해와 전처리

2. 양방향 LSTM을 이용한 개체명 인식

3. F1-Score 검증

13. 태깅 작업(Tagging Task)

5) BiLSTM을 이용한 개체명 인식(Named Entity Recognition, NER)

1. 개체명 인식 데이터에 대한 이해와 전처리

a) 데이터 다운로드 (https://www.kaggle.com/abhinavwalia95/entity-annotated-corpus)

b) Null 값 제거

c) 중복값 제거 (예: 대소문자)

d) "단어와 개체명 - 태깅 정보" 쌍(pair)으로 묶는 작업 수행

e) 단어 부분 / 개체명 태깅 정보 부분 분리 (학습 목적)

f) 정수 인코딩 (케라스 토크나이저 활용)

- 문장 데이터에 대해서는 src_tokenizer (x_data)
- 레이블에 해당되는 개체명 태깅 정보에 대해서는 tar_tokenizer를 사용 (y_data)

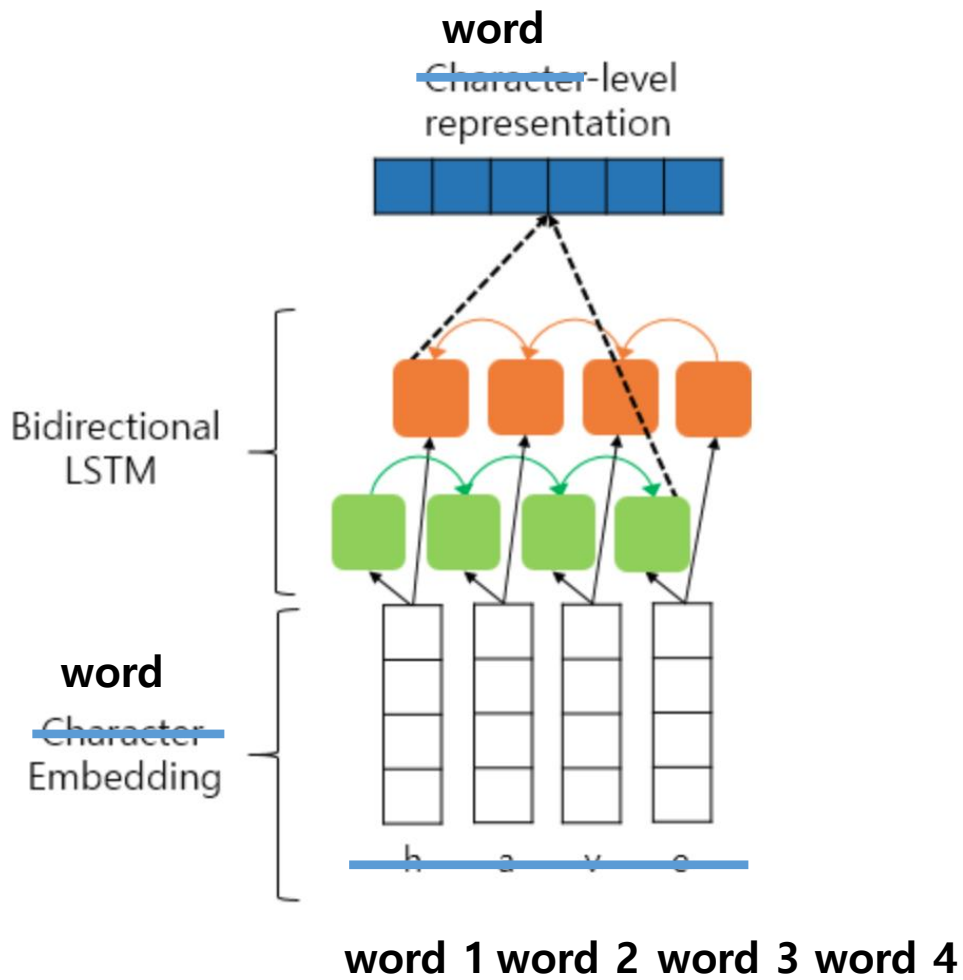
g) 패딩 작업 실시 (가장 긴 샘플 길이를 70으로 설정. max_len=70)

h) 훈련 데이터 : 테스트 데이터 = 8 : 2 로 분리

13. 태깅 작업(Tagging Task)

5) BiLSTM을 이용한 개체명 인식(Named Entity Recognition, NER)

2. 양방향 LSTM을 이용한 개체명 인식



13. 태깅 작업(Tagging Task)

5) BiLSTM을 이용한 개체명 인식(Named Entity Recognition, NER)

3. F1-Score

a) 정확도(Accuracy)는 클래스가 불균형한 문제에서는 좋은 지표가 아닐 수 있음. 이 경우, F1 Score를 고려하는 것이 필요

정확도(Accuracy) / 정밀도(Precision) / 재현율(Recall)의 개념을 잘 구분해 봅시다.

Confusion Matrix

	실제 참	실제 거짓
예측 참	True Positive	False Positive
예측 거짓	False Negative	True Negative

$$Accuray = \frac{TP+TN}{TP+FP+TN+FN}$$

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

$$F = \frac{(\beta^2+1)*Precision*Recall}{\beta^2*Precision+Recall}$$

1. 개체명 인식 데이터에 대한 이해와 전처리

a) 데이터 다운로드

```
print('데이터프레임 행의 개수 : {}'.format(len(data)))
print('데이터에 Null 값이 있는지 유무 : ' + str(data.isnull().values.any()))
print('어떤 열에 Null값이 있는지 출력')
print('==============================')
data.isnull().sum()
```

```
데이터프레임 행의 개수 : 1048575
데이터에 Null 값이 있는지 유무 : True
어떤 열에 Null값이 있는지 출력
==============================

Sentence #    1000616
Word                0
POS                 0
Tag                 0
dtype: int64
```

```
print('sentence # 열의 중복을 제거한 값의 개수 : {}'.format(data['Sentence #'].nunique()))
print('Word 열의 중복을 제거한 값의 개수 : {}'.format(data.Word.nunique()))
print('Tag 열의 중복을 제거한 값의 개수 : {}'.format(data.Tag.nunique()))
```

```
sentence # 열의 중복을 제거한 값의 개수 : 47959
Word 열의 중복을 제거한 값의 개수 : 35178
Tag 열의 중복을 제거한 값의 개수 : 17
```

	Sentence #	Word	POS	Tag
0	Sentence: 1	Thousands	NNS	O
1	NaN	of	IN	O
2	NaN	demonstrators	NNS	O
3	NaN	have	VBP	O
4	NaN	marched	VBN	O
5	NaN	through	IN	O
6	NaN	London	NNP	B-geo
7	NaN	to	TO	O
8	NaN	protest	VB	O
9	NaN	the	DT	O
10	NaN	war	NN	O
		•		
		•		
		•		
22	NaN	country	NN	O
23	NaN	.	.	O
24	Sentence: 2	Families	NNS	O
25	NaN	of	IN	O

1. 개체명 인식 데이터에 대한 이해와 전처리

b) Null 값 제거

```
data = data.fillna(method="ffill")
```

```
print(data.tail())
```

```
              Sentence #       Word  POS Tag
1048570  Sentence: 47959       they  PRP   O
1048571  Sentence: 47959  responded  VBD   O
1048572  Sentence: 47959         to   TO   O
1048573  Sentence: 47959        the   DT   O
1048574  Sentence: 47959     attack   NN   O
```

```
print('데이터에 Null 값이 있는지 유무 : ' + str(data.isnull().values.any()))
```

```
데이터에 Null 값이 있는지 유무 : False
```

1. 개체명 인식 데이터에 대한 이해와 전처리

c) 중복값 제거 (소문자로 통일)

```
data['Word'] = data['Word'].str.lower()
print('Word 열의 중복을 제거한 값의 개수 : {}'.format(data.Word.nunique()))
```

```
Word 열의 중복을 제거한 값의 개수 : 31817
```

```
print(data[:5])
```

```
     Sentence #           Word  POS Tag
0  Sentence: 1      thousands  NNS   O
1  Sentence: 1             of   IN   O
2  Sentence: 1  demonstrators  NNS   O
3  Sentence: 1           have  VBP   O
4  Sentence: 1        marched  VBN   O
```

```
sentence # 열의 중복을 제거한 값의 개수 : 47959
Word 열의 중복을 제거한 값의 개수 : 35178
Tag 열의 중복을 제거한 값의 개수 : 17
```

1. 개체명 인식 데이터에 대한 이해와 전처리

d) "단어와 개체명 - 태깅 정보" 쌍(pair)으로 묶는 작업 수행

```
func = lambda temp: [(w, t) for w, t in zip(temp["Word"].values.tolist(), temp["Tag"].values.tolist())]
tagged_sentences=[t for t in data.groupby("Sentence #").apply(func)]
print("전체 샘플 개수: {}".format(len(tagged_sentences)))
```

```
전체 샘플 개수: 47959
```

```
print(tagged_sentences[0]) # 첫번째 샘플 출력
```

```
[('thousands', 'O'), ('of', 'O'), ('demonstrators', 'O'), ('have', 'O'), ('marched', 'O'), ('through', 'O'), ('londo
n', 'B-geo'), ('to', 'O'), ('protest', 'O'), ('the', 'O'), ('war', 'O'), ('in', 'O'), ('iraq', 'B-geo'), ('and',
'O'), ('demand', 'O'), ('the', 'O'), ('withdrawal', 'O'), ('of', 'O'), ('british', 'B-gpe'), ('troops', 'O'), ('fro
m', 'O'), ('that', 'O'), ('country', 'O'), ('.', 'O')]
```

```
sentence # 열의 중복을 제거한 값의 개수 : 47959
Word 열의 중복을 제거한 값의 개수 : 35178
Tag 열의 중복을 제거한 값의 개수 : 17
```

1. 개체명 인식 데이터에 대한 이해와 전처리

e) 단어 부분 / 개체명 태깅 정보 부분 분리 (학습 목적)

```
print(tagged_sentences[0]) # 첫번째 샘플 출력
```

```
[('thousands', 'O'), ('of', 'O'), ('demonstrators', 'O'), ('have', 'O'), ('marched', 'O'), ('through', 'O'), ('londo
n', 'B-geo'), ('to', 'O'), ('protest', 'O'), ('the', 'O'), ('war', 'O'), ('in', 'O'), ('iraq', 'B-geo'), ('and',
'O'), ('demand', 'O'), ('the', 'O'), ('withdrawal', 'O'), ('of', 'O'), ('british', 'B-gpe'), ('troops', 'O'), ('fro
m', 'O'), ('that', 'O'), ('country', 'O'), ('.', 'O')]
```

```
sentences, ner_tags = [], []
for tagged_sentence in tagged_sentences: # 47,959개의 문장 샘플을 1개씩 불러온다.
    sentence, tag_info = zip(*tagged_sentence) # 각 샘플에서 단어들은 sentence에 개체명 태깅 정보들은 tag_info에 저장.
    sentences.append(list(sentence)) # 각 샘플에서 단어 정보만 저장한다.
    ner_tags.append(list(tag_info)) # 각 샘플에서 개체명 태깅 정보만 저장한다.
```

" zip 함수 사용 "

```
print(sentences[0])
print(ner_tags[0])
```

```
['thousands', 'of', 'demonstrators', 'have', 'marched', 'through', 'london', 'to', 'protest', 'the', 'war', 'in', 'ir
aq', 'and', 'demand', 'the', 'withdrawal', 'of', 'british', 'troops', 'from', 'that', 'country', '.']
['O', 'O', 'O', 'O', 'O', 'O', 'B-geo', 'O', 'O', 'O', 'O', 'O', 'B-geo', 'O', 'O', 'O', 'O', 'O', 'B-gpe', 'O', 'O',
'O', 'O', 'O']
```

1. 개체명 인식 데이터에 대한 이해와 전처리

f) 정수 인코딩 (케라스 토크나이저 활용)

- 문장 데이터에 대해서는 src_tokenizer (x_data)
- 레이블에 해당되는 개체명 태깅 정보에 대해서는 tar_tokenizer를 사용 (y_data)

```
src_tokenizer = Tokenizer(oov_token='OOV') # 모든 단어를 사용하지만 인덱스 1에는 단어 'OOV'를 할당한다.
src_tokenizer.fit_on_texts(sentences)
tar_tokenizer = Tokenizer(lower=False) # 태깅 정보들은 내부적으로 대문자를 유지한채로 저장
tar_tokenizer.fit_on_texts(ner_tags)
```

```
vocab_size = len(src_tokenizer.word_index) + 1
tag_size = len(tar_tokenizer.word_index) + 1
print('단어 집합의 크기 : {}'.format(vocab_size))
print('개체명 태깅 정보 집합의 크기 : {}'.format(tag_size))
```

```
단어 집합의 크기 : 31819
개체명 태깅 정보 집합의 크기 : 18
```

```
data['Word'] = data['Word'].str.lower()
print('Word 열의 중복을 제거한 값의 개수 : {}'.format(data.Word.nunique()))
```

```
Word 열의 중복을 제거한 값의 개수 : 31817
```

1. 개체명 인식 데이터에 대한 이해와 전처리

f) 정수 인코딩 (케라스 토크나이저 활용)

- 문장 데이터에 대해서는 src_tokenizer (x_data)
- 레이블에 해당되는 개체명 태깅 정보에 대해서는 tar_tokenizer를 사용 (y_data)

```
src_tokenizer = Tokenizer(oov_token='OOV') # 모든 단어를 사용하지만 인덱스 1에는 단어 'OOV'를 할당한다.
src_tokenizer.fit_on_texts(sentences)
tar_tokenizer = Tokenizer(lower=False) # 태깅 정보들은 내부적으로 대문자를 유지한채로 저장
tar_tokenizer.fit_on_texts(ner_tags)
```

```
vocab_size = len(src_tokenizer.word_index) + 1
tag_size = len(tar_tokenizer.word_index) + 1
print('단어 집합의 크기 : {}'.format(vocab_size))
print('개체명 태깅 정보 집합의 크기 : {}'.format(tag_size))
```

```
X_data = src_tokenizer.texts_to_sequences(sentences)
y_data = tar_tokenizer.texts_to_sequences(ner_tags)
```

"정수 인코딩 작업 실시"

```
print(X_data[0])
print(y_data[0])
```

```
[254, 6, 967, 16, 1795, 238, 468, 7, 523, 2, 129, 5, 61, 9, 571, 2, 833, 6, 186, 90, 22, 15, 56, 3]
[1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 8, 1, 1, 1, 1, 1]
```

1. 개체명 인식 데이터에 대한 이해와 전처리

g) 패딩 작업 실시 (가장 긴 샘플 길이를 70으로 설정. max_len=70)

```
print(sentences[0])
print(ner_tags[0])
```

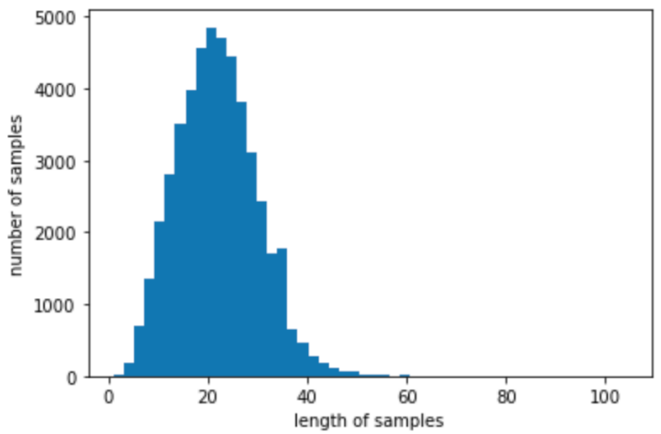
```
['thousands', 'of', 'demonstrators', 'have', 'marched', 'through', 'london', 'to', 'protest', 'the', 'war', 'in', 'ir
aq', 'and', 'demand', 'the', 'withdrawal', 'of', 'british', 'troops', 'from', 'that', 'country', '.']
['O', 'O', 'O', 'O', 'O', 'O', 'B-geo', 'O', 'O', 'O', 'O', 'O', 'B-geo', 'O', 'O', 'O', 'O', 'O', 'B-gpe', 'O', 'O',
'O', 'O', 'O']
```

```
print(sentences[98])
print(ner_tags[98])
```

```
['she', 'had', 'once', 'received', 'a', 'kidney', 'transplant', '.']
['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
```

```
print('샘플의 최대 길이 : %d' % max(len(l) for l in sentences))
print('샘플의 평균 길이 : %f' % (sum(map(len, sentences))/len(sentences)))
plt.hist([len(s) for s in sentences], bins=50)
plt.xlabel('length of samples')
plt.ylabel('number of samples')
plt.show()
```

```
샘플의 최대 길이 : 104
샘플의 평균 길이 : 21.863988
```



1. 개체명 인식 데이터에 대한 이해와 전처리

g) 패딩 작업 실시 (가장 긴 샘플 길이를 70으로 설정. max_len=70)

h) 훈련 데이터 : 테스트 데이터 = 8 : 2 로 분리

```
max_len = 70
X_data = pad_sequences(X_data, padding='post', maxlen=max_len)
y_data = pad_sequences(y_data, padding='post', maxlen=max_len)
```

```
X_train, X_test, y_train_int, y_test_int = train_test_split(X_data, y_data, test_size=.2, random_state=777)
```

```
y_train = to_categorical(y_train_int, num_classes=tag_size)
y_test = to_categorical(y_test_int, num_classes=tag_size)
```

"태깅 정보에 대한 원-핫 인코딩"

```
print('훈련 샘플 문장의 크기 : {}'.format(X_train.shape))
print('훈련 샘플 레이블(정수 인코딩)의 크기 : {}'.format(y_train_int.shape))
print('훈련 샘플 레이블(원-핫 인코딩)의 크기 : {}'.format(y_train.shape))
print('테스트 샘플 문장의 크기 : {}'.format(X_test.shape))
print('테스트 샘플 레이블(정수 인코딩)의 크기 : {}'.format(y_test_int.shape))
print('테스트 샘플 레이블(원-핫 인코딩)의 크기 : {}'.format(y_test.shape))
```

```
훈련 샘플 문장의 크기 : (38367, 70)
훈련 샘플 레이블(정수 인코딩)의 크기 : (38367, 70)
훈련 샘플 레이블(원-핫 인코딩)의 크기 : (38367, 70, 18)
테스트 샘플 문장의 크기 : (9592, 70)
테스트 샘플 레이블(정수 인코딩)의 크기 : (9592, 70)
테스트 샘플 레이블(원-핫 인코딩)의 크기 : (9592, 70, 18)
```

2. 양방향 LSTM을 이용한 개체명 인식

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, InputLayer, Bidirectional, TimeDistributed, Embedding
from tensorflow.keras.optimizers import Adam
```

```
model = Sequential()
model.add(Embedding(vocab_size, 128, mask_zero=True))
model.add(Bidirectional(LSTM(256, return_sequences=True)))
model.add(Dense(tag_size, activation=('softmax')))
model.compile(loss='categorical_crossentropy', optimizer=Adam(0.001), metrics=['accuracy'])
```

```
history = model.fit(X_train, y_train, batch_size=128, epochs=6, validation_split=0.1)
```

```
Epoch 1/6
270/270 [==============================] - 232s 839ms/step - loss: 0.1779 - accuracy: 0.8735 - val_loss:
_accuracy: 0.9307
Epoch 2/6
270/270 [==============================] - 249s 923ms/step - loss: 0.0541 - accuracy: 0.9505 - val_loss:
_accuracy: 0.9550
Epoch 3/6
270/270 [==============================] - 270s 1s/step - loss: 0.0363 - accuracy: 0.9652 - val_loss: 0.0
curacy: 0.9572
Epoch 4/6
270/270 [==============================] - 277s 1s/step - loss: 0.0295 - accuracy: 0.9708 - val_loss: 0.0
curacy: 0.9587
Epoch 5/6
270/270 [==============================] - 269s 998ms/step - loss: 0.0254 - accuracy: 0.9743 - val_loss:
_accuracy: 0.9584
```

정확도(Accuracy) 약 95%

하.지.만...

3. F1-Score

a) 정확도(Accuracy)는 클래스가 불균형한 문제에서는 좋은 지표가 아닐 수 있음. 이 경우, F1 Score를 고려하는 것이 필요

```
print('Tag 열의 각각의 값의 개수 카운트')
print('================================')
print(data.groupby('Tag').size().reset_index(name='count'))
```

```
Tag 열의 각각의 값의 개수 카운트
================================
      Tag   count
0   B-art     402
1   B-eve     308
2   B-geo   37644
3   B-gpe   15870
4   B-nat     201
5   B-org   20143
6   B-per   16990
7   B-tim   20333
8   I-art     297
9   I-eve     253
10  I-geo    7414
11  I-gpe     198
12  I-nat      51
13  I-org   16784
14  I-per   17251
15  I-tim    6528
16      O  887908
```

3. F1-Score

a) 정확도(Accuracy)는 클래스가 불균형한 문제에서는 좋은 지표가 아닐 수 있음. 이 경우, F1 Score를 고려하는 것이 필요

```
true=['B-PER', 'I-PER', 'O', 'O', 'B-MISC', 'O','O','O','O','O','O','O','O','O','O','B-PER','I-PER','O','O','O','O','O'
# 실제값
predicted=['O'] * len(true) #실제값의 길이만큼 전부 'O'로 채워진 리스트 생성. 예측값으로 사용.
print(predicted)
```

```
['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
```

```
hit = 0 # 정답 개수
for t, p in zip(true, predicted):
    if t == p:
        hit +=1 # 정답인 경우에만 +1
accuracy = hit/len(true) # 정답 개수를 총 개수로 나눈다.
print("정확도: {:.1%}".format(accuracy))
```

```
정확도: 74.4%
```

3. F1-Score

a) 정확도(Accuracy)는 클래스가 불균형한 문제에서는 좋은 지표가 아닐 수 있음. 이 경우, F1 Score를 고려하는 것이 필요

정확도(Accuracy) / 정밀도(Precision) / 재현율(Recall)의 개념을 잘 구분해 봅시다.

Confusion Matrix

	실제 참	실제 거짓
예측 참	True Positive	False Positive
예측 거짓	False Negative	True Negative

정확도 $Accuray = \frac{TP+TN}{TP+FP+TN+FN}$

정밀도 $Precision = \frac{TP}{TP+FP}$ 잘못된 Positive 줄이는데 초점 맞춤

재현율 $Recall = \frac{TP}{TP+FN}$ 잘못된 Negaitive 줄이는데 초점 맞춤

$F = \frac{(\beta^2+1)*Precision*Recall}{\beta^2*Precision+Recall}$

3. F1-Score

a) 정확도(Accuracy)는 클래스가 불균형한 문제에서는 좋은 지표가 아닐 수 있음. 이 경우, F1 Score를 고려하는 것이 필요

정확도(Accuracy) / 정밀도(Precision) / 재현율(Recall)의 개념을 잘 구분해 봅시다.

정확도 $Accuray = \frac{TP+TN}{TP+FP+TN+FN}$

정밀도 $Precision = \frac{TP}{TP+FP}$ 잘못된 Positive 줄이는데 초점 맞춤

재현율 $Recall = \frac{TP}{TP+FN}$ 잘못된 Negaitive 줄이는데 초점 맞춤

$F = \frac{(\beta^2+1)*Precision*Recall}{\beta^2*Precision+Recall}$

스팸메일 분류 사례)

스팸메일이 아닌 것을

스팸메일로 분류하면(FP) 업무 차질 발생

악성코드 분류 사례)

악성코드인데

악성코드가 아닌 것으로 분류하면(FN) 위험 노출

3. F1-Score

a) 정확도(Accuracy)는 클래스가 불균형한 문제에서는 좋은 지표가 아닐 수 있음. 이 경우, F1 Score를 고려하는 것이 필요

정확도(Accuracy) / 정밀도(Precision) / 재현율(Recall)의 개념을 잘 구분해 봅시다.

$$F = \frac{(\beta^2+1)*Precision*Recall}{\beta^2*Precision+Recall}$$

f1-score란?

정밀도와 재현률로부터 조화 평균(harmonic mean)을 구한 값

> 정밀도, 재현율에 동일한 가중치를 주고 싶은 경우) β 값 1

> 정밀도에 더 가중치를 주고 싶은 경우) β값 1 이상

> 재현률에 더 가중치를 주고 싶은 경우) β값 0~1

참고 : https://itwiki.kr/w/혼동_행렬

3. F1-Score

a) 정확도(Accuracy)는 클래스가 불균형한 문제에서는 좋은 지표가 아닐 수 있음. 이 경우, F1 Score를 고려하는 것이 필요

```
true=['B-PER', 'I-PER', 'O', 'O', 'B-MISC', 'O','O','O','O','O','O','O','O','O','O','B-PER','I-PER','O','O','O','O','O'
# 실제값
predicted=['O'] * len(true) #실제값의 길이만큼 전부 'O'로 채워진 리스트 생성. 예측값으로 사용.
print(predicted)
```

```
['O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
```

```
hit = 0 # 정답 개수
for t, p in zip(true, predicted):
    if t == p:
        hit +=1 # 정답인 경우에만 +1
accuracy = hit/len(true) # 정답 개수를 총 개수로 나눈
print("정확도: {:.1%}".format(accuracy))
```

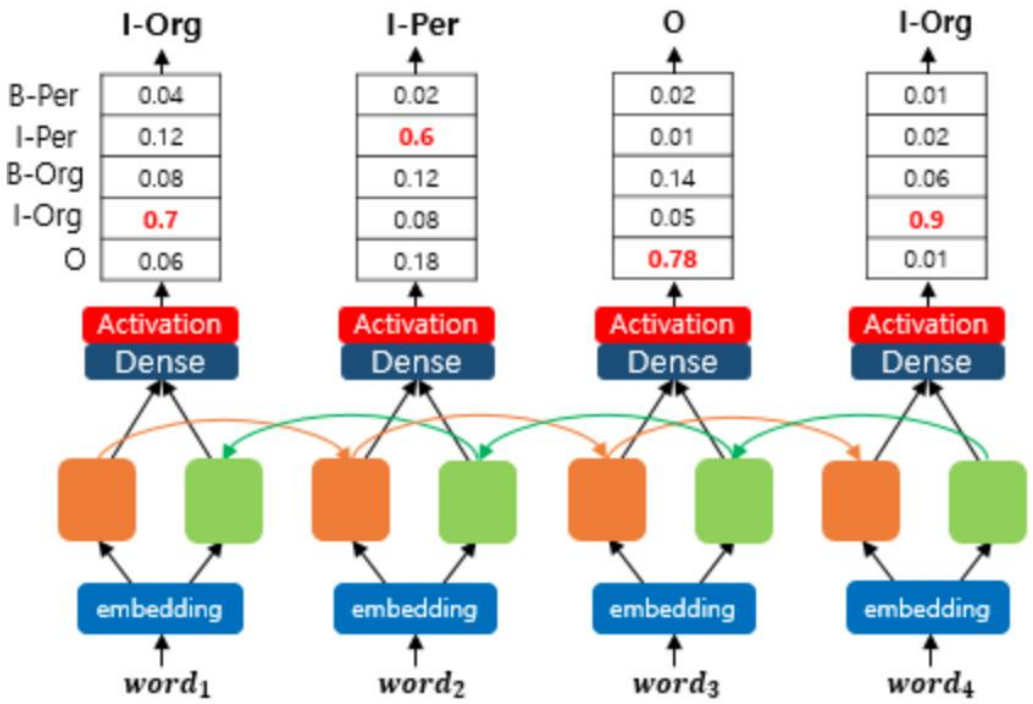
```
정확도: 74.4%
```

```
from seqeval.metrics import classification_report
print(classification_report([true], [predicted]))
```

```
              precision    recall  f1-score   support

        MISC       0.00      0.00      0.00         2
         PER       0.00      0.00      0.00         3

   micro avg       0.00      0.00      0.00         5
   macro avg       0.00      0.00      0.00         5
weighted avg       0.00      0.00      0.00         5
```

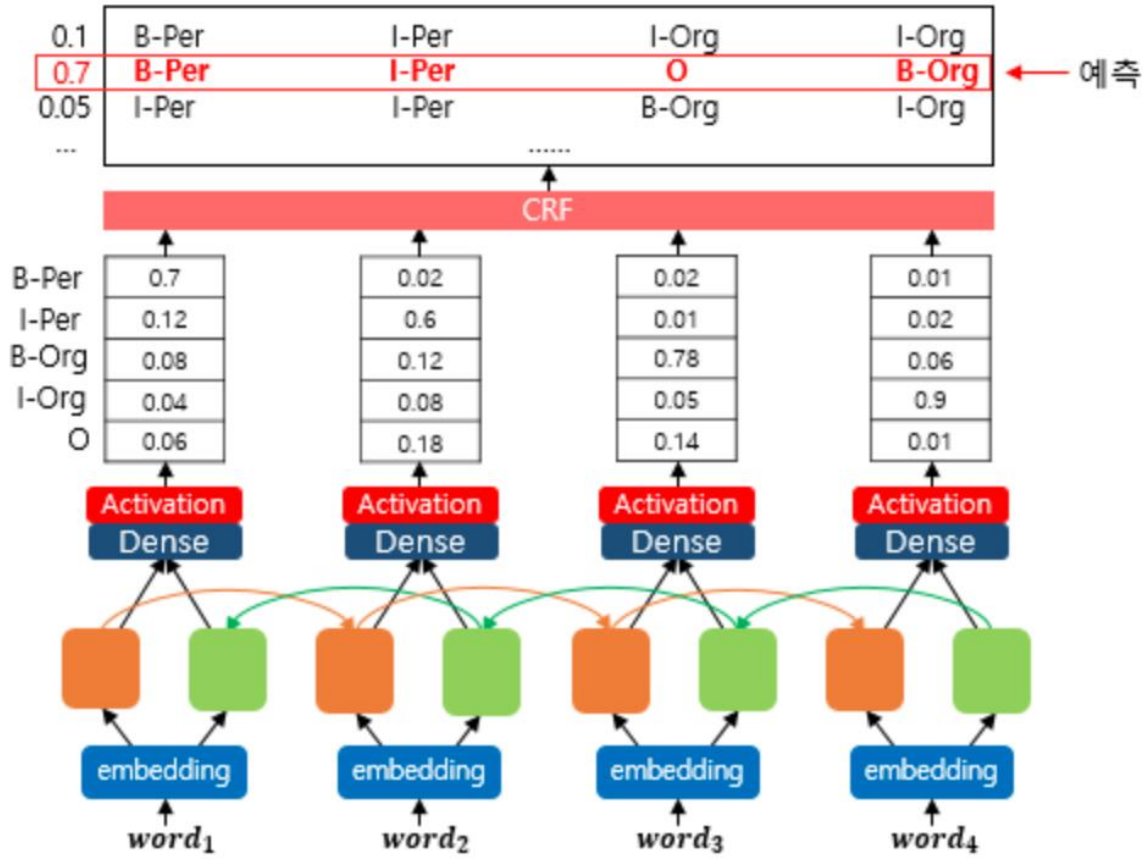
13. 태깅 작업(Tagging Task)

6) BiLSTM-CRF를 이용한 개체명 인식



BiLSTM

F1 score : 78.5%



BiLSTM-CRM

F1 score : 79.1%

13. 태깅 작업(Tagging Task)

7) 글자 임베딩(Character Embedding) 활용하기

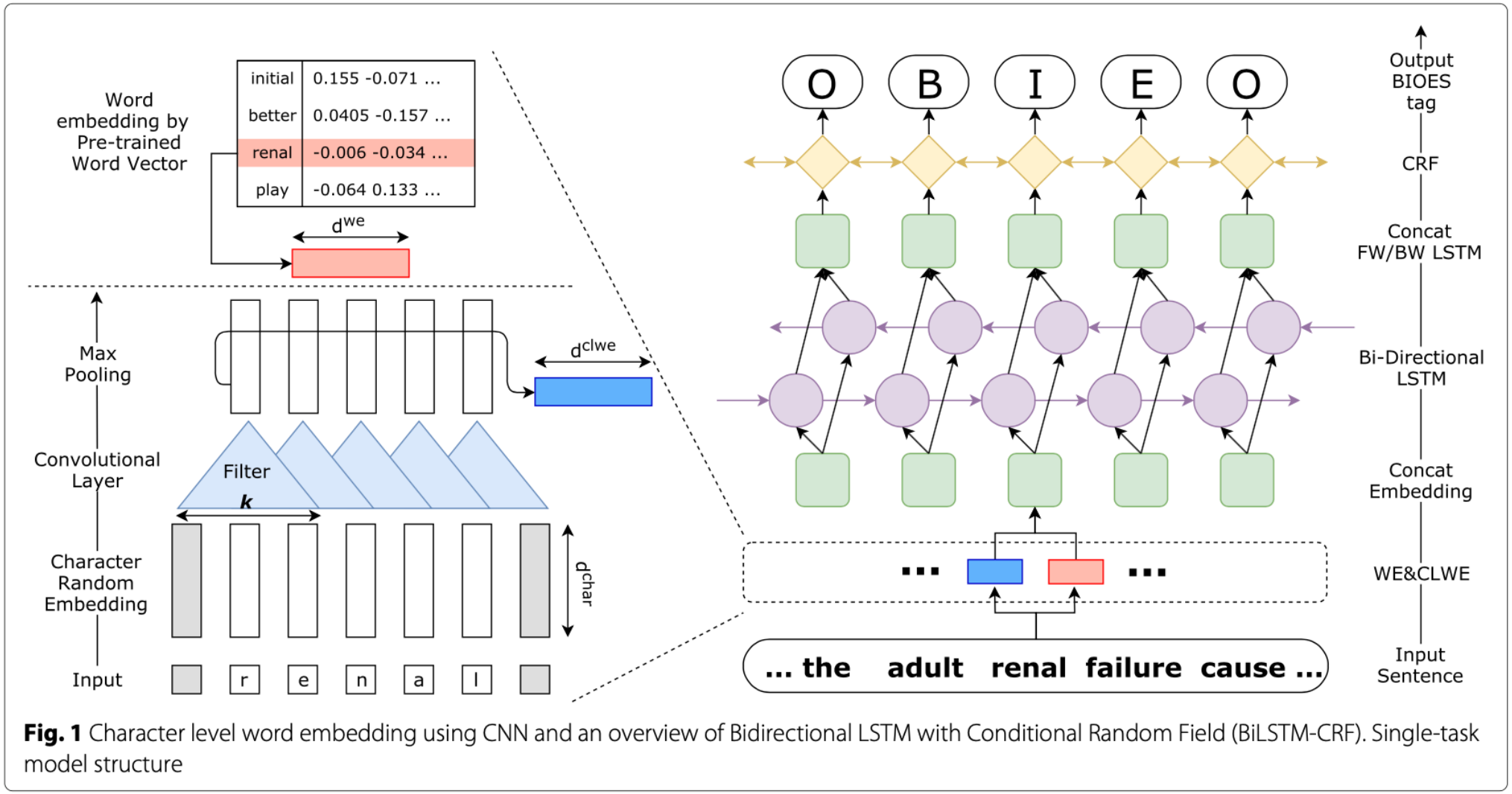


Fig. 1 Character level word embedding using CNN and an overview of Bidirectional LSTM with Conditional Random Field (BiLSTM-CRF). Single-task model structure

CollaboNet: collaboration of deep neural networks for biomedical named entity recognition

챕터 12- 2) 자연어 처리를 위한 1D CNN(1D Convolutional Neural Networks) 참조