**Rasa Custom NLU Component**

**개발 가이드**

2020년 8월 28일

인턴사원 윤용선

# 목차

[목차 2](#_Toc49506614)

[문서의 목적 4](#_Toc49506615)

[**1. 데이터** 5](#_Toc49506616)

[**2. Custom Component** 6](#_Toc49506617)

[**2.1. \_\_init\_\_** 6](#_Toc49506618)

[**2.1.1. defaults** 6](#_Toc49506619)

[**2.2. train** 7](#_Toc49506620)

[**2.3. persist / load** 7](#_Toc49506621)

[**2.4. process** 8](#_Toc49506622)

[**3. Tokenizer** 9](#_Toc49506623)

[**3.1. Mecab** 9](#_Toc49506624)

[**3.2. \_\_init\_\_** 9](#_Toc49506625)

[**3.3. tokenize** 10](#_Toc49506626)

[**3.4. train** 11](#_Toc49506627)

[**3.5. persist / load** 11](#_Toc49506628)

[**4. Featurizer** 13](#_Toc49506629)

[**4.1. Word Embedding** 13](#_Toc49506630)

[**4.1.1. Word2Vec** 13](#_Toc49506631)

[**4.1.2. FastText** 14](#_Toc49506632)

[**4.1.3. Flair Embedding** 14](#_Toc49506633)

[**4.2. pretrain** 15](#_Toc49506634)

[**4.3. train** 16](#_Toc49506635)

[**4.4. persist / load** 17](#_Toc49506636)

[**5. Extractor** 19](#_Toc49506637)

[**5.1. RasaModel** 19](#_Toc49506638)

[**5.1.1. prepare\_layers** 20](#_Toc49506639)

[**5.1.2. batch\_loss** 20](#_Toc49506640)

[**5.1.3. batch\_predict** 21](#_Toc49506641)

[**6. Classifier** 22](#_Toc49506642)

[**6.1. Network** 22](#_Toc49506643)

[**6.2. train** 23](#_Toc49506644)

[**6.3. persist / load** 24](#_Toc49506645)

# 문서의 목적

본 문서는 오픈소스 챗봇 프레임워크 Rasa의 Custom Component 개발 과정에 대한 기술적인 설명을 제공합니다. 주요 내용은 아래와 같습니다.

* Rasa Custom Component의 구성과 흐름
* 주요 Component의 구성과 개발 요소
* Component 구성 딥러닝 모델에 대한 개요와 코드

# **1. 데이터**

본 프로젝트는 AI Hub가 제공하는 [한국어 대화 데이터](http://www.aihub.or.kr/aidata/85) 중 “카페” 도메인 데이터를 사용하였다. 엔티티(Entity)에 한정하여 데이터 정제를 진행하였고, 최종적으로 128개의 인텐트(Intent)와 43개의 엔티티로 구성되었다.

데이터 정제 단계에서 메뉴, 음료 종류, 빵 종류와 같이 중복되는 의미를 지닌 엔티티를 통합하였고, 인스턴스가 1 ~ 2개이거나 분석에 불필요한 엔티티를 제거하였다.

# **2. Custom Component**

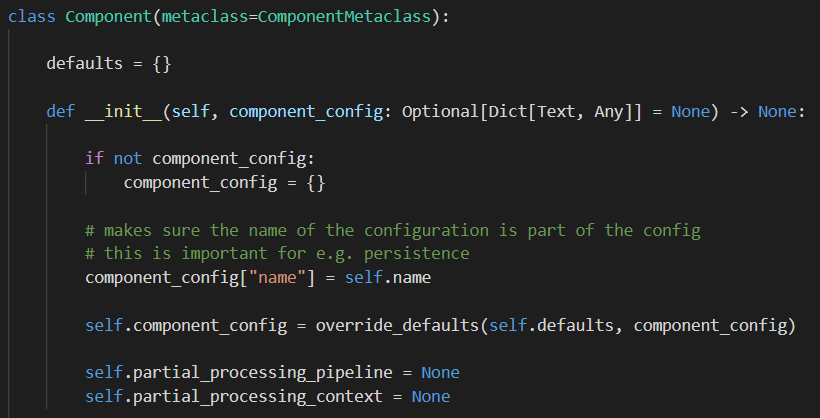
Rasa의 파이프라인은 컴포넌트들의 시퀀스로 구성된다. 컴포넌트에는 고유의 작업을 정상적으로 수행하기 위한 기능들이 메소드로 정의되어 있으며, 커스텀 컴포넌트를 개발할 때는 이러한 메소드들을 필수적으로 정의해야 한다. 이 장에서는 커스텀 컴포넌트를 구성하는데 필요한 요소를 설명한다.

## **2.1. \_\_init\_\_**

\_\_init\_\_ 메소드는 Python에서 객체를 생성할 때 초기화를 하는 내장 메소드이며, Rasa 컴포넌트에서도 같은 역할을 한다. 구체적인 기능은 컴포넌트별로 상이하지만, 주로 입력 파라미터의 인스턴스 변수화, 데이터 검증, configuration 검증 등의 기능을 한다.

### **2.1.1. defaults**

Rasa Component에서 필요한 config 파라미터의 기본값을 설정하는 기능을 한다. 딕셔너리 형태로 파라미터 이름이 key, 파라미터의 기본값이 value로 구성된다. 한가지 주의 사항은 default 변수가 메소드 밖에서 정의되어야 한다는 것이다.

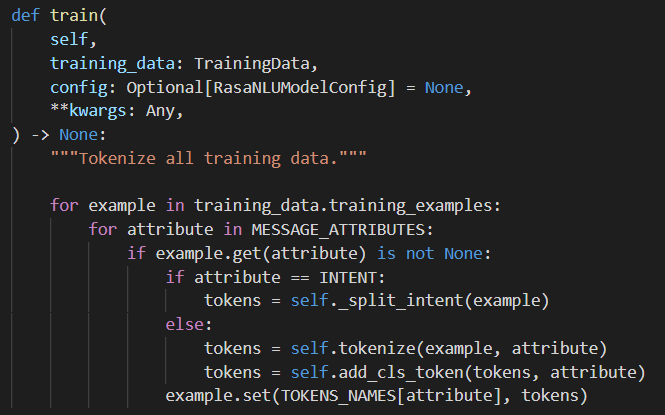


컴포넌트의 \_\_init\_\_ 메소드 구성

## **2.2. train**

train 메소드는 학습 데이터를 사용해 모델을 학습시키는 기능을 한다. TrainingData 객체를 입력받는데, TrainingData의 training\_examples는 Message 객체들로 구성된 리스트이다. Message의 attribute를 통해 원하는 정보(텍스트, 엔티티 등)을 추출할 수 있다.

Tokenizer와 Featurizer의 경우, 모델 학습 외에도 모든 학습 데이터에 자신들의 process를 진행 후, Message attribute에 추가하는 기능을 한다.



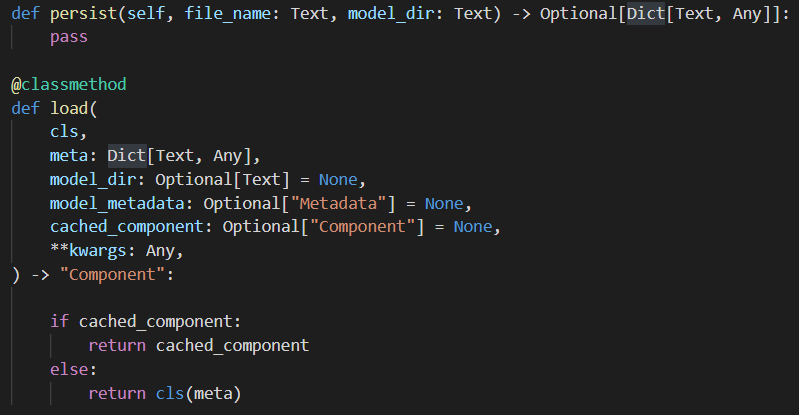
Tokenizer의 train 메소드 예시

## **2.3. persist / load**

persist 메소드는 컴포넌트의 학습이 종료된 후, 테스트와 서비스에서 필요한 데이터를 저장한다. 주로 모델 파라미터, vocabulary, index mapping dictionary 등이 이에 해당한다.

load 메소드는 persist에서 저장된 파일들을 다시 불러오는 기능을 한다. classmethod를 사용하여 필요한 파일을 불러온 후, 컴포넌트를 생성할 수 있다.

두 메소드 모두 파일이 저장될 임시 경로를 입력 받기 때문에, 해당 경로를 기준으로 파일이 저장될 구체적인 경로를 지정할 수 있다.



persist와 load 메소드 구성

## **2.4. process**

process는 Message를 입력 받아 컴포넌트 고유의 작업을 수행하는 메소드이다.

# **3. Tokenizer**

토큰화(Tokenize)는 문장 (혹은 문서)를 작은 의미 단위로 구분하는 작업이다. 한국어는 조사, 어미 등에서 많은 변형이 나타나기 때문에 적절하게 토큰화하는 것이 중요하다.

토큰화 방법은 크게 1) 띄어쓰기를 기준으로 분절하는 ‘공백 기반’, 2) 형태소를 기준으로 분절하는 ‘형태소 기반’, 3) 텍스트에서 자주 나타나는 글자조합을 기준으로 분절하는 ‘서브워드(subword) 기반’ 방법이 있다. 본 프로젝트에서는 형태소 기반 방법을 사용하여 Custom Tokenizer를 구현하였다.

## **3.1. Mecab**

본 프로젝트에서는 토크나이저로 Mecab을 사용하였고, 그 이유는 다음과 같다.

1) 한국어 모델에서 서브워드 기반 토큰화보다 형태소 기반 토큰화의 성능이 더 뛰어나다.

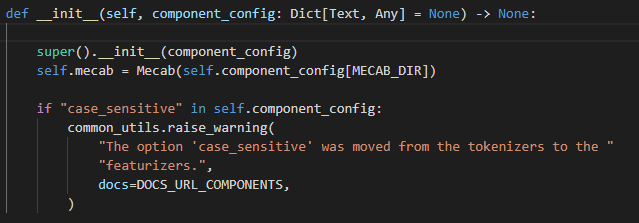
2) Mecab이 다른 형태소 분석기에 비해 처리 속도가 빠르다.

이러한 이유로 Mecab을 사용하였지만, 프로젝트의 성격과 파이프라인에 따라 다른 토크나이저를 사용할 수 있으며, 구현 방법은 크게 다르지 않다. Mecab의 설치 방법은 아래 페이지를 참고하였다. 설치 후 mecab과 mecab-ko-dic 디렉토리가 정상적으로 생성되는 것을 확인할 수 있다.

* Mecab 설치 방법: [Windows](https://cleancode-ws.tistory.com/97), [MacOS](https://buttercoconut.xyz/251/), [Linux](https://somjang.tistory.com/entry/PythonUbuntu%EC%97%90-mecab-ko-dic-%EB%9D%BC%EC%9D%B4%EB%B8%8C%EB%9F%AC%EB%A6%AC-%EC%84%A4%EC%B9%98%ED%95%98%EA%B8%B0)

## **3.2. \_\_init\_\_**

\_\_init\_\_ 단계에서는 Mecab 객체를 생성한다. Mecab 객체는 python-mecab이 아닌 konlpy.Tagger 객체를 사용하였다. 객체를 생성하는 과정에서 mecab-ko-dic 디렉토리의 경로를 요구할 때가 있다. 이 경우를 위해 mecab-ko-dic 경로를 config.yml에서 정의할 수 있도록 하였다.



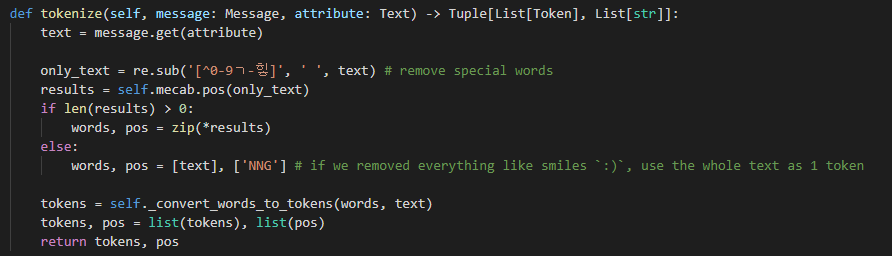
Mecab Tokenizer \_\_init\_\_ 메소드



config.yml 에서 mecab\_ko\_dic 경로 지정 예시

## **3.3. tokenize**

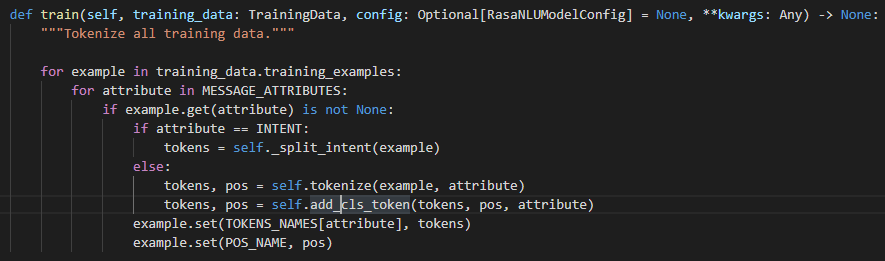
tokenize 메소드는 Message를 입력 받아 토큰화를 진행한다. 토큰화 전에 특수문자와 영어를 제거하는 전처리를 진행한다. mecab.pos로 토큰화를 할 경우, 토큰 텍스트와 토큰 텍스트의 형태소 태깅이 출력된다. 이를 각각 words와 pos 변수에 할당한다. 최종적으로 words의 토큰 텍스트를 Token 객체로 변경 후 출력한다.



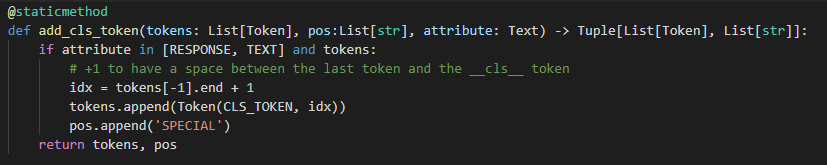
Mecab Tokenizer tokenize 메소드

## **3.4. train**

Mecab은 서브워드 기반 방법과 달리 모델의 학습을 필요로 하지 않는다. 따라서 train 메소드는 모든 학습 데이터에 대해 토큰화를 진행하는 과정만 포함한다. add\_cls\_token 메소드는 토큰의 마지막에 \_\_cls\_\_ 토큰을 추가한다. 시퀀스의 길이를 일치시키기 위해 pos 에도 ‘SPECIAL’ 이름으로 태깅을 추가한다.



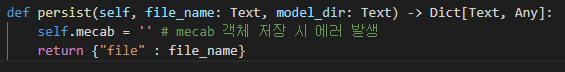
Mecab Tokenizer train 메소드



Mecab Tokenizer add\_cls\_token 메소드

## **3.5. persist / load**

형태소 기반 tokenizer를 사용할 경우 특별한 persist 와 load 처리를 필요로 하지 않는다. 한 가지 유의할 점은 mecab 객체를 지워야 한다는 점이다. Persist에는 각 컴포넌트들을 pickling 하는 과정이 포함되어 있는데, konlpy.Tagger 객체의 경우 이 과정에서 에러가 발생한다. 따라서 할당한 mecab을 지워 에러를 방지할 수 있다.



Mecab Tokenizer persist 메소드

# **4. Featurizer**

Featurizer는 토큰으로부터 특징을 추출하는 과정이다. Rasa의 Featurizer는 다음과 같이 구분된다.

1) Sparse Featurizer: character level n-gram 등

2) Dense Featurizer: Word Embedding model 등

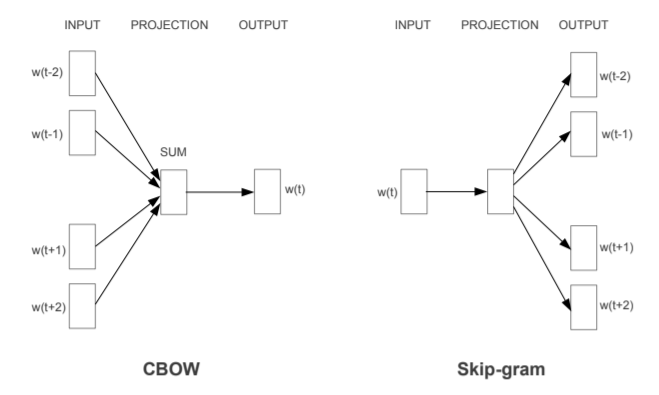
Sparse Featurizer는 언어에 상관없이 사용할 수 있는 반면, Dense Featurizer는 한국어 모델을 따로 개발을 해야 한다. 따라서 본 프로젝트에서는 한국어 Dense Featurizer를 개발하였다.

## **4.1. Word Embedding**

Featurizer 모델로는 대표적인 Word Embedding 모델인 Word2Vec과 FastText를 사용하였다. 또한 추가적으로 character-level LM 기반의 Flair Embedding 모델을 사용하였다.

### **4.1.1. Word2Vec**

[Word2Vec](https://arxiv.org/pdf/1301.3781.pdf) 모델은 특정 맥락의 토큰들이 입력되었을 때, 마스킹된 토큰을 예측하도록 학습된다. 주변 단어들이 입력된 후, 목적 단어를 예측하는 CBOW 방법과 목적 단어가 입력된 후 주변 단어를 예측하는 Skip-gram으로 구분된다.



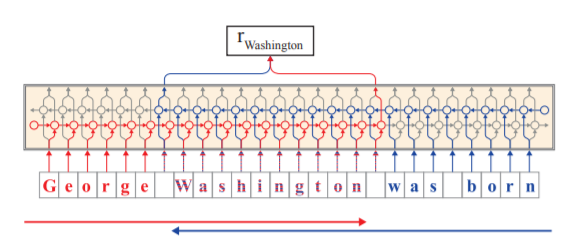
Word2Vec 구조

### **4.1.2. FastText**

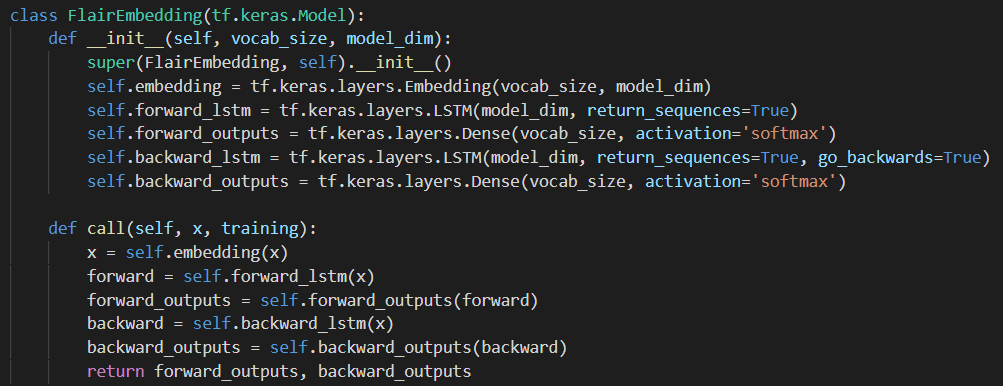
[FastText](https://arxiv.org/pdf/1607.04606.pdf) 모델은 토큰을 Character n-gram으로 구성한 후, 해당 n-gram들의 벡터를 합하여 토큰의 벡터를 산출한다. 단어의 벡터는 Word2Vec과 같은 방식으로 학습된다. 단어의 벡터에 글자 단위 정보가 포함되기 때문에 더 풍부한 특징을 학습할 수 있고, Word2Vec 모델에 비해 OOV 문제에서도 강건한 특징이 있다.

### **4.1.3. Flair Embedding**

[Flair Embedding](https://alanakbik.github.io/papers/coling2018.pdf) 모델은 Character level LM을 기반으로 NER 분야에서 좋은 성능을 보였다. Forward LSTM과 Backward LSTM을 학습시켜 토큰의 벡터를 추출한다. 아래의 그림으로 예를 들면, Washington의 벡터는 Forward LSTM에서 n이 끝난 지점과 Backward LSTM에서 W가 시작되는 지점의 hidden state vector를 병합하여 추출한다.



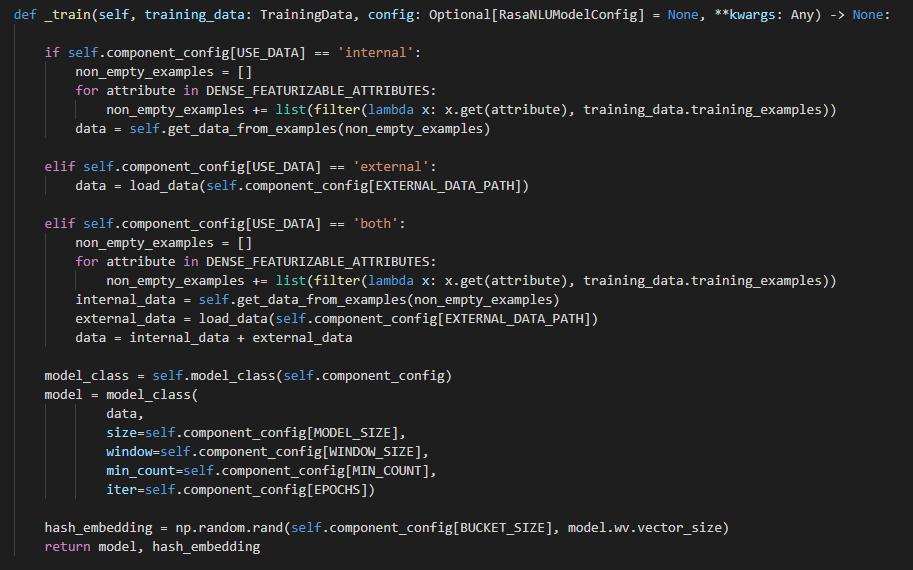
Flair Embedding 구조



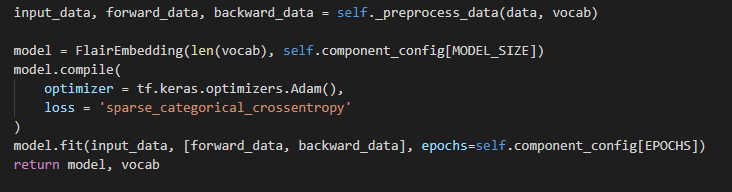
Flair Embedding 코드

## **4.2. pretrain**

pretrain은 Featurizer의 모델을 학습하는 단계이다. 학습 데이터는 대화 데이터만 사용하는 ‘internal’, 외부 수집 데이터만 사용하는 ‘external’, 두 데이터를 모두 사용하는 ‘both’에 따라 처리된다. Word Embedding Featurizer (Word2Vec, FastText)의 경우 genism 라이브러리를 사용하여 모델을 생성하였고, Flair Embedding Featurizer은 tensorflow 라이브러리를 사용하였다.



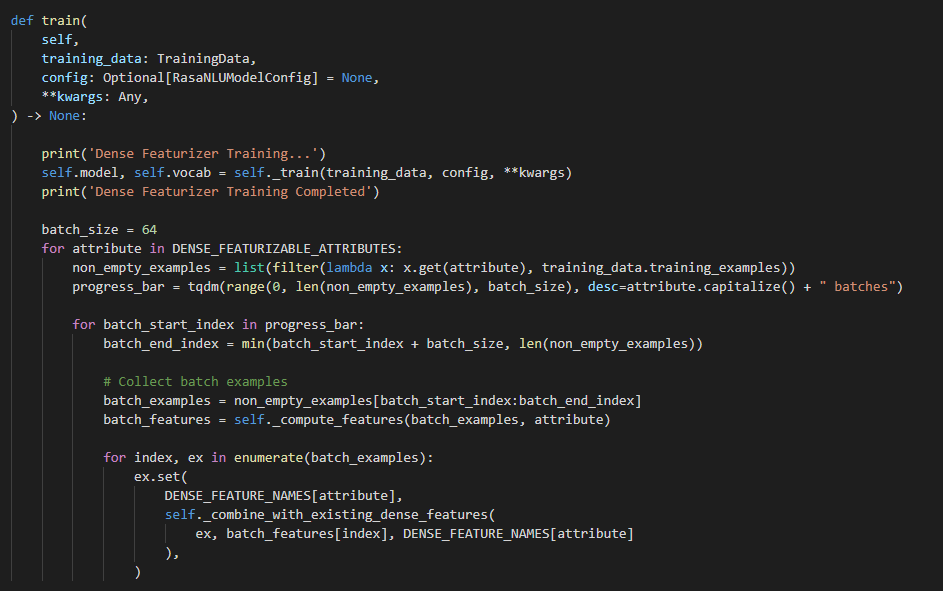
Word Embedding(Word2Vec/FastText) Featurizer \_train 메소드



Flair Embedding Featurizer \_train 메소드

## **4.3. train**

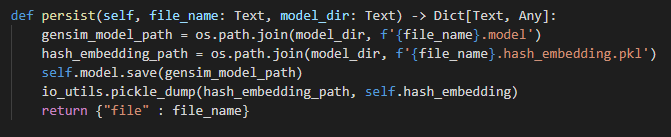
train 메소드에서는 가장 먼저 4.2. pretrain을 실행한다. 그 후 Tokenizer와 같이 모든 학습 데이터에 대해 Feature을 추출 후, Message 객체에 저장한다. 모델과 컴퓨터 사양에 따라 한번에 처리하기 어려울 수 있기 때문에 mini-batch로 나누어서 처리한다.



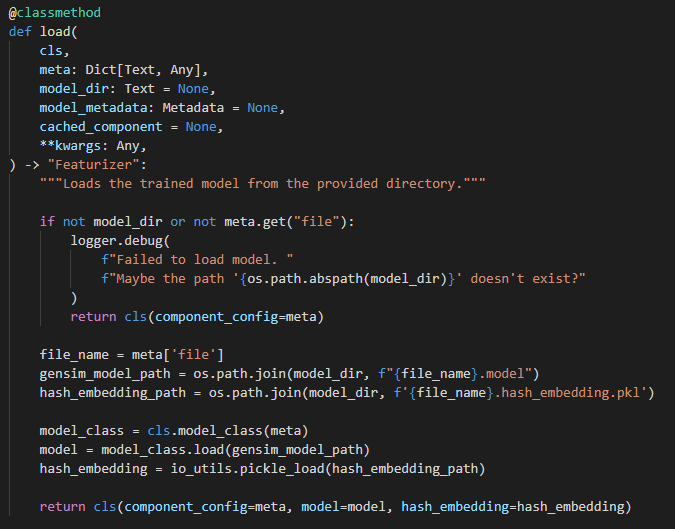
Featurizer train 메소드

## **4.4. persist / load**

train이 종료되면 persist 메소드에서 학습시킨 모델과 필요 파일(hash embedding, vocab 등)을 저장한다.



Word Embedding Featurizer persist 메소드



Word Embedding Featurizer load 메소드

# **5. Extractor**

Extractor는 Featurizer가 생성한 특징들을 입력받아 텍스트의 엔티티를 추출하는 기능을 한다. 주로 토큰별 엔티티 태그를 예측하며, 태깅 스킴(Tagging scheme)은 IO, BIO, BILOU 등 다양하지만, Rasa에서는 기본적으로 BILOU를 사용한다. BILOU는 Beginning, Inside, Last, Outside, Unit으로 구성되며, 각각 엔티티의 시작, 중간, 마지막, 엔티티 아님, 한 토큰으로 이루어진 엔티티를 의미한다.

본 프로젝트에서 Extractor component는 [DIET (Dual Intent and Entity Transformer)](https://arxiv.org/pdf/2004.09936.pdf)의 구조를 바탕으로 수정하였다. DIET는 Entity Extractor와 Intent classifier를 동시에 처리할 수 있기 때문에 아래의 코드에서도 그러한 특징을 확인할 수 있다. 하지만 Intent classifier의 사용여부는 사용자의 판단에 따라 변경할 수 있다.

## **5.1. RasaModel**

Rasa에서는 tensorflow 기반의 딥러닝 모델을 RasaModel 객체로 관리한다. 따라서 모델의 입력과 출력이 동일하고, 모델의 아키텍처만 변경하고 싶을 경우, RasaModel 객체만 수정하면 복잡한 수정없이 Custom component를 개발할 수 있다. RasaModel에서 바꿔야 할 메소드는 다음과 같다.

1) prepare\_layer : RasaModel은 dictionary에서 레이어별로 모델을 정의하는데, 이는 모델의 입력과 출력을 유동적으로 하기 위함이다. 따라서 같은 방식으로 레이어를 정의하는 과정이 필요하다.

2) batch\_loss : batch 데이터를 입력 받아 오차를 출력하는 메소드이다. batch loss만 정의하면 RasaModel에서 정의된 학습 메소드에 따라 학습이 진행된다. 한 가지 유의할 점은, 학습 메소드에서 tf.function 데코레이터를 사용하기 때문에 batch\_loss에서 이루어지는 모든 연산은 tf.function 연산 형식을 따라야 한다는 것이다.

3) batch\_predict : batch 데이터를 입력 받아 예측 결과를 출력하는 메소드이다. 학습이 완료된 후 process에서 실행된다. Entity Extractor의 경우 예측 entity의 인덱스 값을, Intent Classifier의 경우 intent별 확률 값을 Dictionary 형태로 출력한다.

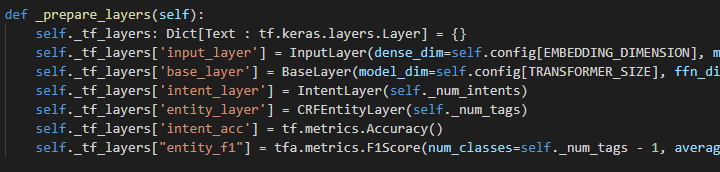
### **5.1.1. prepare\_layers**

- input\_layer: sparse feature와 dense feature를 처리 후 병합한다.

- base\_layer: transformer layer로 구성되며 encoder의 기능을 한다.

- intent\_layer: Softmax layer로, intent 별 확률값을 출력한다.

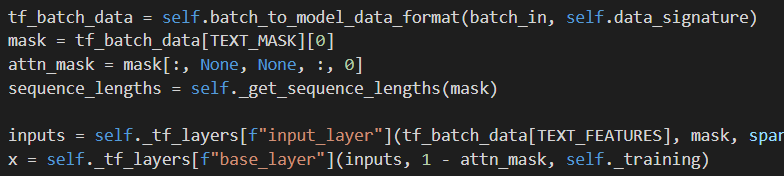
- entity\_layer: CRF와 Softmax 중 선택할 수 있다.

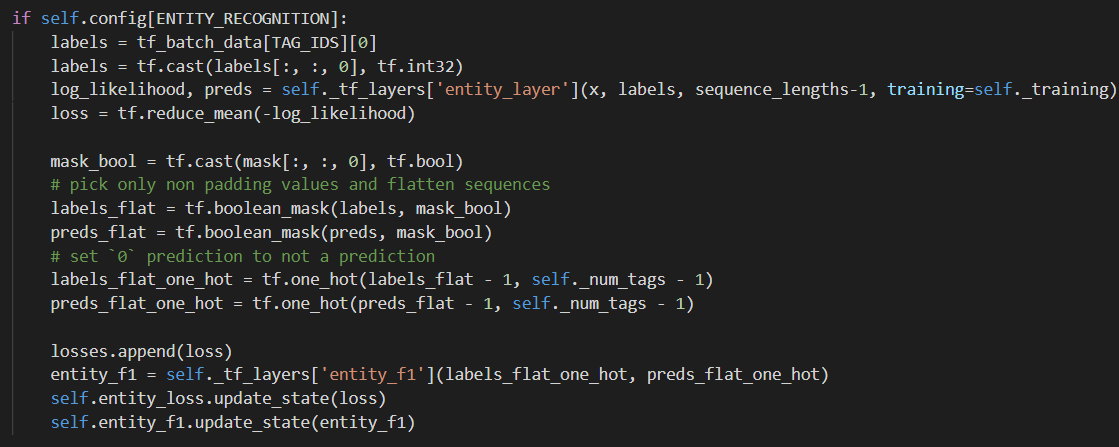


Extractor prepare\_layer 메소드

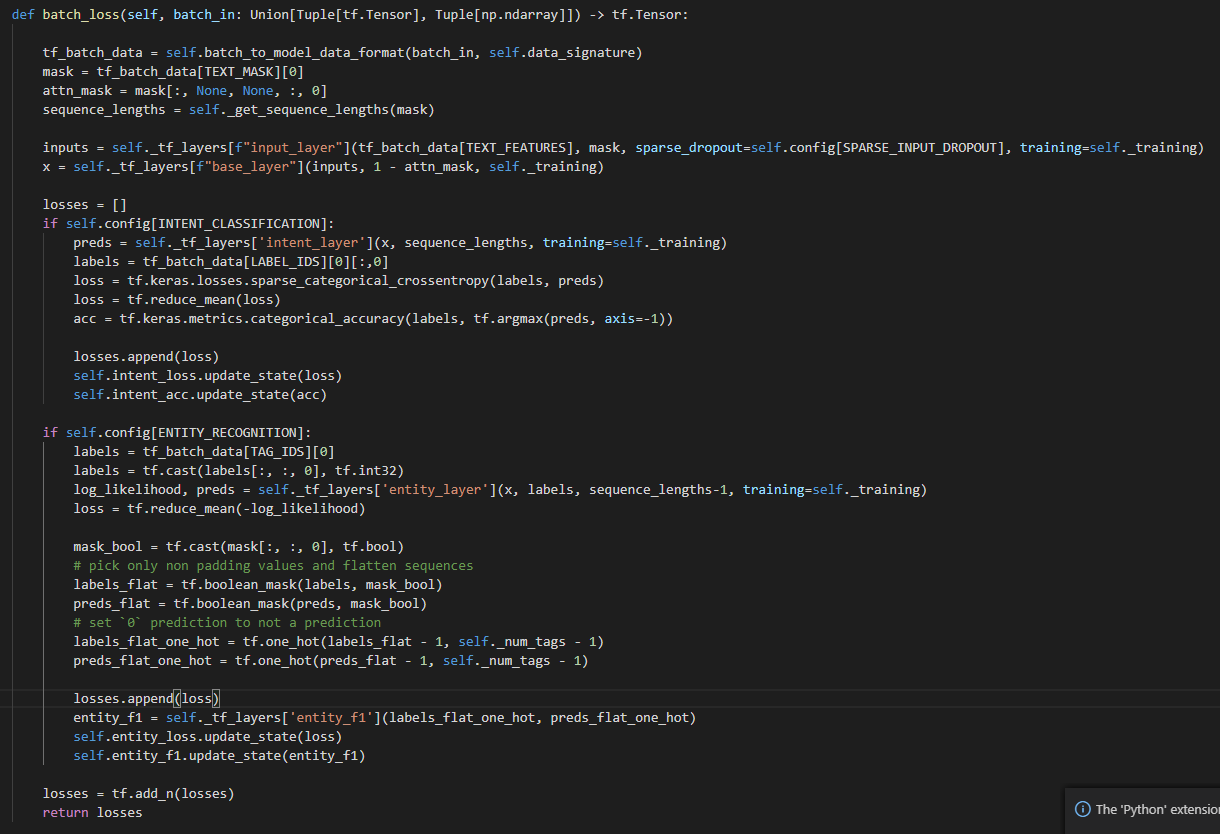
## **5.1.2. batch\_loss**

배치 데이터가 입력되면 데이터 포맷으로 변경한다. 변경된 데이터를 input\_layer와 base\_layer에 입력해 토큰별 벡터를 추출한다. config에 따라 인텐트와 엔티티 오차를 계산 후 합한 값이 최종 오차가 된다. 엔티티 오차는 엔티티 라벨의 likelihood의 음수값으로 계산한다.



Extractor batch\_loss 메소드: 토큰 벡터 추출

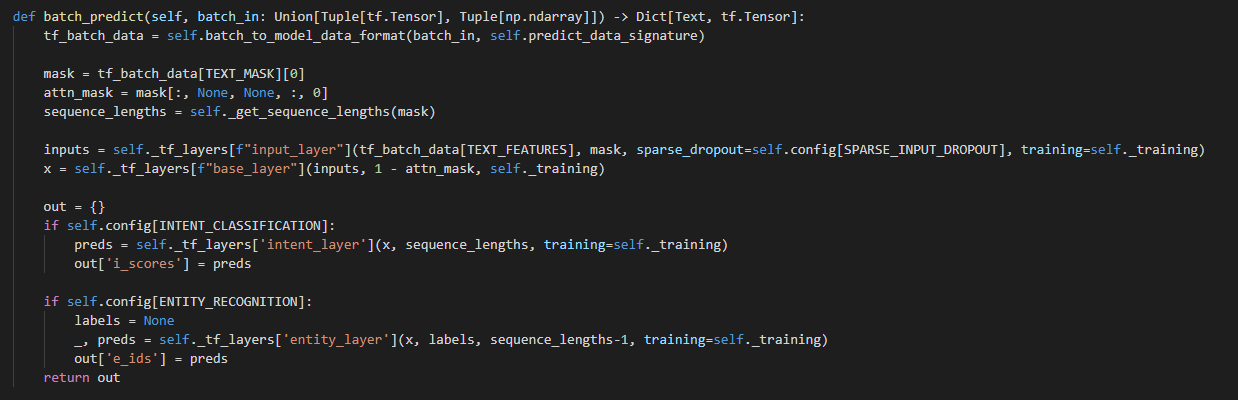
Extractor batch\_loss 메소드: Entity loss



Extractor batch\_loss 메소드

## **5.1.3. batch\_predic**t

batch\_loss와 같은 방식으로 데이터 포맷 변경 후, 모델에 입력해 예측 값을 출력한다. 인텐트의 경우 각 인텐트별 확률 값을, 엔티티의 경우 엔티티 라벨을 출력하면 된다.



Extractor batch\_predict 메소드

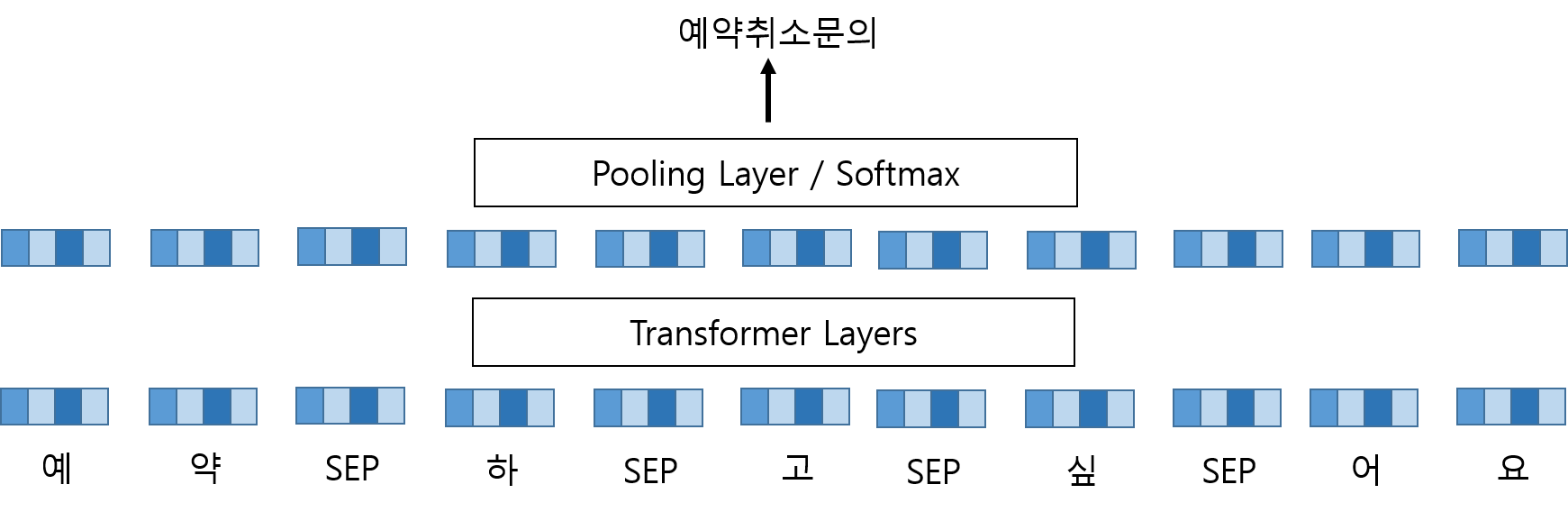
# **6. Classifier**

Classifier는 텍스트의 발화 의도(Intent)를 에측하는 기능을 한다.. 본 프로젝트에서는 기존의 토큰 단위 분류기의 문제점을 해결하고자 글자 단위 분류기를 개발하였다.

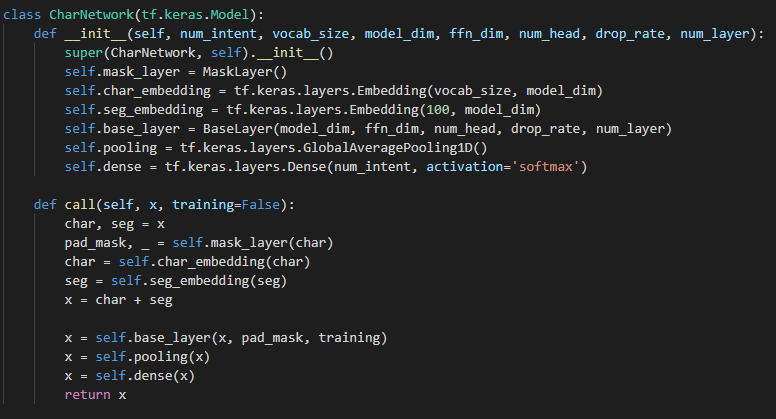
## **6.1. Network**

네트워크의 구조는 일반적인 Transformer기반 분류기와 다르지 않다. Embedding layer와 Transformer layer를 거친 후, Pooling layer로 문장의 표상 벡터(representation vector)을 추출한 뒤 최종적으로 Softmax layer로 intent별 확률 값을 출력한다.

제안 네트워크는 입력 시퀀스의 단위가 토큰이 아닌 글자로 구성된다. 따라서 Embedding layer 또한 글자별 벡터를 출력한다. 하지만 같은 토큰으로 묶인 글자들의 경우, 유사한 의미를 지니고 있기 때문에 이러한 정보를 처리하기 위해 segment embedding layer를 추가하였다. 같은 토큰의 글자들은 같은 segment index 값을 받아, 같은 segment embedding vector를 갖게 된다. 최종적으로 하나의 글자의 embedding vector는 character level embedding vector와 segment embedding vector의 합으로 산출된다.



Character Classifier Network 구조

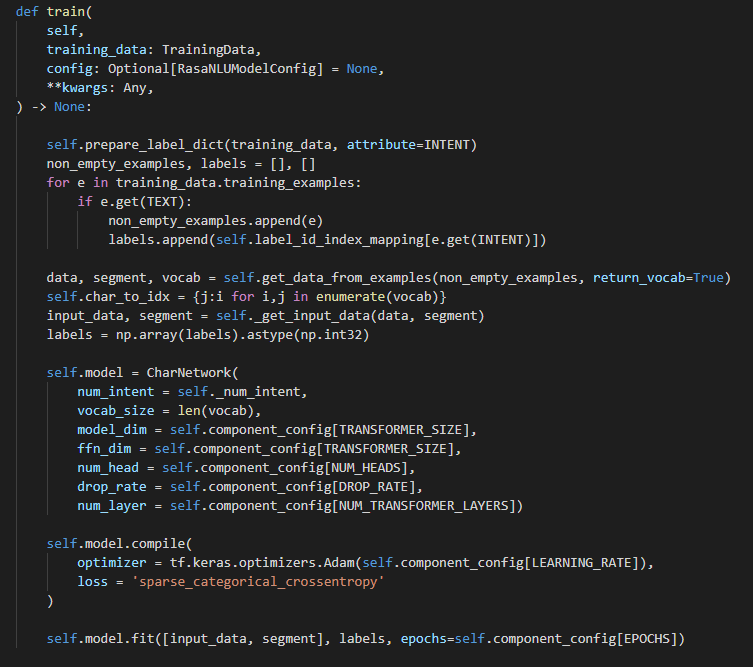


Character Classifier Network 코드

## **6.2. train**

train 메소드에서는 다른 component들과 마찬가지로 모델의 파라미터를 학습시킨다. 이때 코드의 간소화를 위해 RasaModel 객체를 사용하지 않았다. 이러한 방식은 4.1.3. Flair Embedding과 동일하다.

제안 모델은 토큰 단위가 아닌 글자 단위이기 때문에 추가적인 전처리가 필요하다. 전처리는 get\_data\_from\_examples 메소드에서 진행되며, 글자 분리, [BOS], [SEP]과 같은 특수 토큰 추가 등의 과정이 포함되어 있다.



Character Classifier train 메소드

## **6.3. persist / load**

persist와 load 에서는 모델과 label-index dictionary 데이터를 저장하고 불러온다. 구성 방식은 다른 컴포넌트와 동일하다.