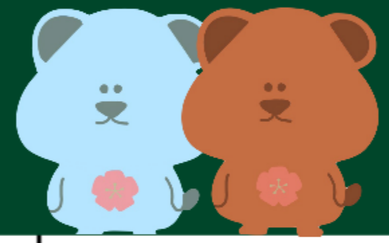
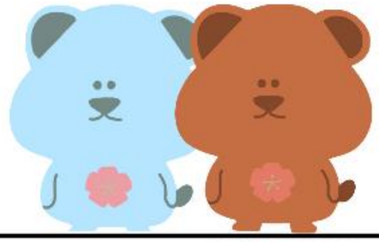


15주차 발표

ML팀 박지운 오연재 이서영

EWHA EURON

목차



EWHA EURON

01. 자연어 처리란

EWHA EURON

1.1 자연어 처리란

자연어 처리: 일상생활에서 사용하는 언어 의미를 분석하여 컴퓨터가 처리할 수 있도록 하는 것.

언어별로 띄어쓰기로 인하여 단어 단위의 임베딩이 상이하다.

맞춤법 검사, 단어 검색은 자연어 처리가 잘 되지만 질의응답, 대화는 자연어 처리의 발전이 더 필요한 부분.

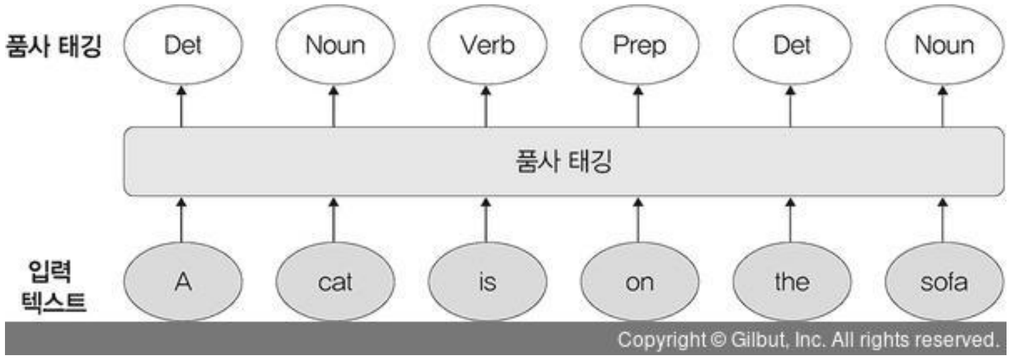
말뭉치(corpus): 자연어 처리 모델 학습을 위한 데이터(특정한 목적에서 표본을 추출)

토큰(token): 단어/문자 단위로 자르는 것(문장 토큰화, 단어 토큰화)

불용어(stop words): 문장 내에서 많이 등장하는 단어, 빈도가 높아 성능에 영향을 미쳐 사전에 제거하기(a, the, she, he..)

어간 추출(stemming): 단어를 기본 형태로 만드는 작업(run, ran, running→ run으로 통일)

품사 태깅(part-of-speech tagging): 품사를 식별하기 위해 붙여주는 태그(식별 정보)



1.2 라이브러리 설치

(1) 문장 토큰화(단어로 쪼개기): nltk.word_tokenize()

```
text=nltk.word_tokenize("Is it possible distinguishing cats and dogs")
text
```

```
['Is', 'it', 'possible', 'distinguishing', 'cats', 'and', 'dogs']
```

(2) 품사 태깅: nltk.pos_tan()

주어진 문장에서 품사를 식별하기 위해 단어+품사 의 형태로 나타난다.

```
nltk.pos_tag(text)
```

```
[('Is', 'VBZ'),
 ('it', 'PRP'),
 ('possible', 'JJ'),
 ('distinguishing', 'VBG'),
 ('cats', 'NNS'),
 ('and', 'CC'),
 ('dogs', 'NNS')]
```

KoNLPy

꼬꼬마, 코모란, 한나눔, 트위터(open korean text), 메카브 등 한글 분석기를 모두 사용할 수 있다.

1.3 KoNLPy

```
# 사용하고자 하는 객체 설정
hannanum = Hannanum()
kkma = Kkma()
komoran = Komoran()
okt = Okt()
```

(1) .pos() : 형태소 분석 함수. 형태소를 분석해 각 형태소별 품사까지 태그해주는 함수.

```
print(okt.pos('형태소 분석하면 어떻게?'))
```

```
[('형태소', 'Noun'), ('분석', 'Noun'), ('하면', 'Verb'), ('어떻게', 'Adjective'), ('?', 'Punctuation')]
```

(2) .morphs() : 형태소 분석 함수. 형태소를 분석해서 나눈 뒤 리스트로 만들어 준다. 품사를 태깅하진 않는다.

```
print(okt.morphs('형태소 분석하면 어떻게?'))
```

```
['형태소', '분석', '하면', '어떻게', '?']
```

(3) 객체 별로 품사표현 방식은 다르다.

```
# hannanum 인 경우.
wordType=hannanum.tagset
print(wordType)
```

```
{'E': '어미', 'EC': '연결 어미', 'EF': '종결 어미', 'EP': '선어말어미', 'ET': '전성 어미', 'F': '외국어', 'I': '독립언', 'II': '감탄
사', 'J': '관계언', 'JC': '격조사', 'JP': '서술격 조사', 'JX': '보조사', 'M': '수식언', 'MA': '부사', 'MM': '관형사', 'N': '체언', 'N
B': '의존명사', 'NC': '보통명사', 'NN': '수사', 'NP': '대명사', 'NQ': '고유명사', 'P': '용언', 'PA': '형용사', 'PV': '동사', 'PX':
'보조 용언', 'S': '기호', 'X': '접사', 'XP': '접두사', 'XS': '접미사'}
```

참조: https://sosoeasy.tistory.com/328
https://wonhwa.tistory.com/24

1.4 Genism – 토픽 모델링

토픽모델링: 텍스트 본문의 숨겨진 의미 구조를 발견하기 위해 사용되는 텍스트 마이닝 기법이다.

(1) LDA(latent dirichlet allocation)

가정: 문서들은 토픽의 혼합으로 구성 + 토픽들은 확률 분포에 기반하여 단어를 생성. → 문서가 생성된 과정을 역추적

1) 사용자는 알고리즘에게 토픽의 개수 k를 알려준다. (k: 하이퍼 파라미터)

2) 모든 단어를 k개 중 하나의 토픽에 할당한다.

3) 문서의 각 단어 w는 자신은 잘못된 토픽에 할당되어져 있지만, 다른 단어들은 전부 올바른 토픽에 할당되어져 있는 상태라고 가정한다. 이에 따라 단어 w는 아래의 두 가지 기준에 따라서 토픽이 재할당된다. *수렴까지 반복

- p(topic t | document d) : 문서 d의 단어들 중 토픽 t에 해당하는 단어들의 비율
- p(word w | topic t) : 각 토픽들 t에서 해당 단어 w의 분포

doc1

word	apple	banana	apple	dog	dog
topic	B	B	???	A	A

doc2

word	cute	book	king	apple	apple
topic	B	B	B	B	B

참조: https://heytech.tistory.com/328

1.5 사이킷런 라이브러리

(2) 사이킷런 라이브러리

Countvectorizer: 단어의 등장 횟수를 기준으로 특성을 추출한다.

Tfidvectorizer: TF-IDF 값을 사용하여 텍스트 특성을 추출한다. (카운팅 방식의 단점을 보완)

- **TF**: 특정 단어가 하나의 데이터 안에서 등장하는 횟수

Ex) A라는 단어가 doc1에서 10번, doc3에서 5번 사용 => doc1-단어A의 TF는 10, doc3-단어A의 TF는 5 이다.

- **DF** : 특정 단어가 여러 데이터에 자주 등장하는지를 알려주는 지표.

Ex) A라는 단어가 doc1, doc3에 등장했다면 DF는 2 (DF가 클수록 범용적인 단어)

- **IDF**: 전체 단어수를 해당 단어의 DF로 나눈 뒤 로그를 취한 값
- **TF-IDF** : TF와 IDF를 곱한 값. 즉 TF가 높고, DF가 낮을수록 값이 커지는 것을 이용하는 것이다.

	Doc1	Doc2	Doc3
Term1	5	0	0
Term2	1	0	0
Term3	5	5	5
Term4	3	3	3
Term5	3	0	1

Doc1	TF	DF	IDF	TF-IDF
Term1	5	1	Log3	5log3
Term2	1	1	Log3	1log3
Term3	5	3	Log1	0
Term4	3	3	Log1	0
Term5	3	2	Log(3/2)	3log(3/2)

=> TF-IDF클수록 특이한 단어이다.

(문서를 가려내는 단어이다.)

참조: https://ratsgo.github.io/from%20frequency%20to%20semantics/2017/03/28/tfidf/

2. 전처리

EWHA EURON

2.1 전처리 과정 및 결측치 확인

머신러닝/딥러닝에서는 텍스트 자체를 이해하지 못하기 때문에 텍스트 데이터 전처리가 필요함

[전반적인 전처리 과정]

▲ 그림 9-15 전처리 과정

[결측치 확인]
1) is.null() 을 이용해서 결측치 확인
2) 결측치 처리
- 모든 행에 결측치 존재 시 행 삭제 drop.na(how= 'all')
- 하나라도 존재시 행 삭제 drop.na()
- 결측치를 다른 값으로 채우기 fillna(?) 0, 평균, 최빈값 등

2.2 토큰화

텍스트를 단어 or 문자 단위로 자르기

1) 단어 토큰화
- 띄어쓰기 단위로 구분
- From nltk import word_tokenize

▲ 그림 9-17 단어 토큰화

- '(어퍼스트로피)를 구분하려면 from nltk.tokenize import WordPunctTokenizer 사용

```
['it', ''', 's', 'nothing', 'that', 'you', 'don', ''', 't', 'already', 'know', 'excep
t', 'most', 'people', 'aren', ''', 't', 'aware', 'of', 'how', 'their', 'inner', 'worl
d', 'works', '.']
```

2.2 토큰화

2) 문장 토큰화
- 문장 마지막 기호에 따라 구분
- from nltk import sent_tokenize

```
from nltk import sent_tokenize
text_sample = 'Natural Language Processing, or NLP, is the process of extracting the me
aning, or intent, behind human language. In the field of Conversational artificial inte
lligence (AI), NLP allows machines and applications to understand the intent of human l
anguage inputs, and then generate appropriate responses, resulting in a natural convers
ation flow.'
tokenized_sentences = sent_tokenize(text_sample)
print(tokenized_sentences))
```

2.2 토큰화

한국어 토큰화 -> KoNLPy

- from konlpy.tag import Okt/Komoran/Kkma
- 오픈소스 한글 형태소 분석기

```
twitter = Okt()

result = []
for line in rdw:
    malist = twitter.pos( line[1], norm=True, stem=True)
    r = []
    for word in malist:
        if not word[1] in ["Josa","Eomi","Punctuation"]:
            r.append(word[0])
    rl = (" ".join(r)).strip()
    result.append(rl)
    print(rl)
```

```
정치 현실 적 모습 담다 영화 열정 채우다 스티븐 냉소 적 정치 겪다 후 변하다 모습 소름
다 진짜 서스펜스 스릴러 뛰어나다 배우 들 연기 보다 재미 있다 남자 여자 관계 조심하다
정치 이면
인사 이드 읏 보고 생각나다 찾아오다 ㅋㅋㅋ 어리다 때 보다 그때 느끼다 감동 아직도 살
봇 들 감정 이렇게 표현 해내다 있다
90년 대 SF 혁명 가다 준 추억 명작 정말 재미있다 즐기다 보다 90년 대 향수 느끼다
쥬드 멋지다 미소 반하다 것 같다 표정 연기 디테일 살다 카메론 너무 예쁘다
버리다
지루하다 느슨하다
말 필요없다 쓰레기 정말 정말 기대하다 더욱 실망 감 크다
```

```
>>> from konlpy.tag import Kkma
>>> from konlpy.utils import pprint
>>> pprint(kkma.nouns(u'질문이나 건의사항은 깃헙 이슈 트래커에 남겨주세요.'))
[질문,
 건의,
 건의사항,
 사항,
 깃헙,
 이슈,
 트래커]
>>> pprint(kkma.pos(u'오류보고는 실행환경, 에러메세지와함께 설명을 최대한상세히!^^
[(오류, NNG),
 (보고, NNG),
 (는, JX),
 (실행, NNG),
 (환경, NNG),
 (,, SP),
 (에러, NNG),
```

2.3 불용어 제거

- 문장에서 빈번하게 등장하나 의미를 부여하기엔 어려운 단어인 불용어를 미리 제거
- 불용어를 제거하지 않는다면 효율성 감소 및 처리 시간 증가하는 문제
- 영어 불용어 리스트는 nltk에서 제공하나 한국어는 없음

```
from nltk.corpus import stopwords

print(stopwords.words('english'))
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're",
"you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he',
'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's",
'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which',
'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are',
'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do',
'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because',
'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against',
'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to',
'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again',
'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all',
'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no',
'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can',
'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o',
're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't",
'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn',
"isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't",
'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't",
'won', "won't", 'wouldn', "wouldn't"]
```

- 보통 직접 불용어 리스트를 정의해서 for문을 통해 제거
- Ex) 을, 를, 에, 의, 가, 으로, 에게, 이다, 되다, 하다, 네, 등, 겨우, 단지, 다만, 훨씬, 약간 등

EWHA EURON

2.4 어간 추출

단어의 원형 찾기 -> writes wrote writing 형태는 다르나 모두 같은 의미니까 -> write

- 품사가 달라도 가능
- 사전에 없는 단어도 추출 가능, 단 품사 보존하지 않기에 어간 추출 결과가 사전에 존재하지 않는 경우가 다수

1) 포터

- 원형 보존을 잘하는 편
- 규칙 기반 접근 ex) ALIZE > AL/ICAL > IC

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()

print(stemmer.stem('obesses'),stemmer.stem('obssesed'))
print(stemmer.stem('standardizes'),stemmer.stem('standardization'))
print(stemmer.stem('national'), stemmer.stem('nation'))
print(stemmer.stem('absentness'), stemmer.stem('absently'))
print(stemmer.stem('tribalical'), stemmer.stem('tribalicalized'))
```

```
obess obsses
standard standard
nation nation
absent absent
tribal tribalic
```

2) 랭커스터

- 원형 축소 -> 정확도 상대적으로 낮음
- 데이터셋 축소시킬 때 유용

```
from nltk.stem import LancasterStemmer
stemmer = LancasterStemmer()

print(stemmer.stem('obsesses'),stemmer.stem('obsessed'))
print(stemmer.stem('standardizes'),stemmer.stem('standardization'))
print(stemmer.stem('national'), stemmer.stem('nation'))
print(stemmer.stem('absentness'), stemmer.stem('absently'))
print(stemmer.stem('tribalical'), stemmer.stem('tribalicalized'))
```

```
obsess obsess
standard standard
nat nat
abs abs
trib trib
```

2.4 표제어 추출

- 기본 사전형 단어 ex) am are is → be
- 어간과 접사를 분리하는 개념
- 단어가 문장 속에서 어떤 품사로 쓰였는지를 고려 -> 품사 같아야 함
- 사전에 없는 단어는 추출 불가
- 문법과 문장 내 단어의 의미를 고려하기 때문에 어간 추출보다 성능 더 좋으나 시간은 더 오래 걸림

```
import nltk
nltk.download('wordnet')

from nltk.stem import WordNetLemmatizer
lemma = WordNetLemmatizer()

print(stemmer.stem('obsesses'),stemmer.stem('obsessed'))
print(lemma.lemmatize('standardizes'),lemma.lemmatize('standardization'))
print(lemma.lemmatize('national'), lemma.lemmatize('nation'))
print(lemma.lemmatize('absentness'), lemma.lemmatize('absently'))
print(lemma.lemmatize('tribalical'), lemma.lemmatize('tribalicalized'))
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
obsess obsess
standardizes standardization
national nation
absentness absently
tribalical tribalicalized
```

2.4 표제어 추출

단어의 품사 정보를 추가해서 성능을 높일 수 있음

```
print(lemma.lemmatize('obsesses', 'v'),lemma.lemmatize('obsessed','a'))
print(lemma.lemmatize('standardizes','v'),lemma.lemmatize('standardization','n'))
print(lemma.lemmatize('national','a'), lemma.lemmatize('nation','n'))
print(lemma.lemmatize('absentness','n'), lemma.lemmatize('absently','r'))
print(lemma.lemmatize('tribalical','a'), lemma.lemmatize('tribalicalized','v'))
```

```
obsess obsessed
standardize standardization
national nation
absentness absently
tribalical tribalicalized
```

2.5 정규화

정규화: 칼럼의 모든 데이터가 동일한 정도의 범위(스케일 혹은 중요도)를 갖도록 하는 것

Monthly Income	Age	PercentSalary Hike	Relationship Satisfaction	TrainingTimes LastYear	YearsInCurrent Role
5993	23	11	1	0	4
5130	55	23	4	3	7
2090	45	15	2	3	0
2909	60	11	3	3	7
3468	47	12	4	3	2
3068	51	13	3	2	7
2670	19	20	1	3	0
2693	33	22	2	2	0
9526	37	21	2	2	7
5237	59	13	2	3	7

정규화를 하지 않고 데이터를 분석하면 MonthlyIncome 값이 더 크기 때문에 상대적으로 더 많은 영향을 미치게 됨.
=> 값이 크다고 해서 분석에 더 중요한 요소라고 간주할 수 없기 때문에 정규화가 필요한 것!

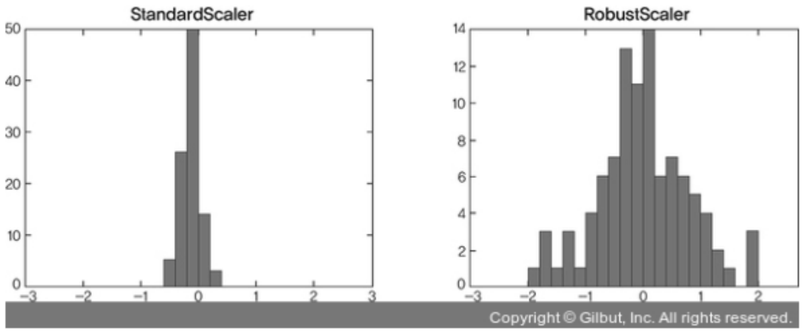
2.5 정규화

$$MinMaxScaler() = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

- 모든 칼럼이 0과 1 사이에 위치하도록 값의 범위를 조정
- 이상치에 매우 민감할 수 있기 때문에 주의

$$StandardScaler() = \frac{x - \mu}{\sigma}$$

- 각 특성의 평균을 0, 분산을 1로 변경하여 칼럼 값의 범위를 조정



▲ 그림 9-18 StandardScaler와 RobustScaler 비교

RobustScaler(): 평균과 분산 대신 중간 값과 사분위수 범위를 사용

MaxAbsScaler(): 절댓값이 0~1 사이가 되도록 조정. 즉, 모든 데이터가 -1~1의 사이가 되도록 조정하기 때문에 양의 수로만 구성된 데이터는 MinMaxScaler()와 유사하게 동작. 또한, 큰 이상치에 민감하다는 단점이 있음

2.5 정규화

```
class customdataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
        self.len = len(self.X)
    def __getitem__(self, index):
        return self.X[index], self.y[index]
    def __len__(self):
        return self.len
```

미니 배치나 데이터를 셔플 하는 등의 용도로 사용 가능

```
train_data = customdataset(torch.FloatTensor(X_train), torch.FloatTensor(y_train))
test_data = customdataset(torch.FloatTensor(X_test), torch.FloatTensor(y_test))

train_loader = DataLoader(dataset=train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_data, batch_size=64, shuffle=False)
```

<- 데이터로더에 데이터 담기

- 방대한 양의 데이터를 배치 단위로 쪼개어 처리 및 셔플 가능
- 데이터 병렬 학습 가능

=> 데이터양이 많을 때 주로 사용

EWHA EURON

2.5 정규화

네트워크 생성(배치 정규화가 포함된 선형 계층)

```
class binaryClassification(nn.Module):
    def __init__(self):
        super(binaryClassification, self).__init__()
        self.layer_1 = nn.Linear(8, 64, bias=True) #칼럼이 여덟 개이므로 입력 크기는 8을 사용
        self.layer_2 = nn.Linear(64, 64, bias=True)
        self.layer_out = nn.Linear(64, 1, bias=True) #출력으로는 당뇨인지 아닌지를 나타내는 0과 1의 값만 가
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=0.1)
        self.batchnorm1 = nn.BatchNorm1d(64)
        self.batchnorm2 = nn.BatchNorm1d(64)

    def forward(self, inputs):
        x = self.relu(self.layer_1(inputs))
        x = self.batchnorm1(x)
        x = self.relu(self.layer_2(x))
        x = self.batchnorm2(x)
        x = self.dropout(x)
        x = self.layer_out(x)
        return x
```

손실 함수와 옵티마이저 지정

```
epochs = 1000+1
print_epoch = 100
LEARNING_RATE = 1e-2

model = binaryClassification()
model.to(device)
print(model)
BCE = nn.BCEWithLogitsLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)
# 훈련 데이터셋에서 무작위로 샘플을 추출하고 그 샘플만 이용해서 기울기를 계산
```

이진 분류 손실 함수로 BCEWithLogitsLoss 함수 외에 ‘이진 교차 엔트로피’ 사용 가능

* BCEWithLogitsLoss 손실 함수: BCELoss 손실 함수에 시그모이드(sigmoid) 함수가 함께 결합된 것

torch.nn.BCEWithLogitsLoss = torch.nn.BCELoss + torch.sigmoid

2.5 정규화

모델 성능 측정 함수 정의

```python
def accuracy(y_pred, y_test):
    y_pred_tag = torch.round(torch.sigmoid(y_pred))
    correct_results_sum = (y_pred_tag == y_test).sum().float()
    # 실제 정답과 모델의 결과가 일치하는 개수를 실수 형태로 변수에 저장
    acc = correct_results_sum/y_test.shape[0]
    acc = torch.round(acc * 100)
    return acc
```

```
Train: epoch: 0 - loss: 0.67931; acc: 56.556
Test: epoch: 0 - loss: 0.67165; acc: 66.000
Train: epoch: 100 - loss: 0.49428; acc: 79.333
Test: epoch: 100 - loss: 0.51525; acc: 72.500
Train: epoch: 200 - loss: 0.59079; acc: 81.333
Test: epoch: 200 - loss: 0.56171; acc: 71.000
Train: epoch: 300 - loss: 0.58855; acc: 79.778
Test: epoch: 300 - loss: 0.52654; acc: 78.750
Train: epoch: 400 - loss: 0.56491; acc: 87.111
Test: epoch: 400 - loss: 0.56405; acc: 69.750
Train: epoch: 500 - loss: 0.52361; acc: 85.778
Test: epoch: 500 - loss: 0.56466; acc: 72.000
Train: epoch: 600 - loss: 0.41998; acc: 78.222
Test: epoch: 600 - loss: 0.54360; acc: 72.000
Train: epoch: 700 - loss: 0.55602; acc: 85.222
Test: epoch: 700 - loss: 0.56167; acc: 72.000
Train: epoch: 800 - loss: 0.55300; acc: 78.889
Test: epoch: 800 - loss: 0.53710; acc: 74.000
Train: epoch: 900 - loss: 0.55601; acc: 88.778
Test: epoch: 900 - loss: 0.54972; acc: 74.000
Train: epoch: 1000 - loss: 0.40115; acc: 82.444
Test: epoch: 1000 - loss: 0.53259; acc: 74.750
```

학습이 진행될수록 훈련 및 테스트 데이터셋에 대한 성능 UP
데이터양이 적어 극적인 효과는 보이지 않음.

모델 학습

```python
for epoch in range(epochs):
    iteration_loss = 0. # 변수를 0으로 초기화
    iteration_accuracy = 0.

    model.train() # 모델 학습
    for i, data in enumerate(train_loader): # 데이터로더에서 훈련 데이터셋을 배치 크기만큼 불러옵니다.
        X, y = data
        y_pred = model(X.float()) # 독립 변수를 모델에 적용하여 훈련
        loss = BCE(y_pred, y.reshape(-1,1).float()) # 모델에 적용하여 훈련시킨 결과와 정답(레이블)을 손실 함수

        iteration_loss += loss # 오차 값을 변수에 누적하여 저장
        iteration_accuracy += accuracy(y_pred, y) # 모델 성능(정확도)을 변수에 누적하여 저장
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    if(epoch % print_epoch == 0):
        print('Train: epoch: {0} - loss: {1:.5f}; acc: {2:.3f}'.format(epoch, iteration_loss/(i+1), iteration_

    iteration_loss = 0.
    iteration_accuracy = 0.
    model.eval() # 모델 검증(테스트)
    for i, data in enumerate(test_loader): # 데이터로더에서 테스트 데이터셋을 배치 크기만큼 불러옵니다.
        X, y = data
        y_pred = model(X.float())
        loss = BCE(y_pred, y.reshape(-1,1).float())
        iteration_loss += loss
        iteration_accuracy += accuracy(y_pred, y)
    if(epoch % print_epoch == 0):
        print('Test: epoch: {0} - loss: {1:.5f}; acc: {2:.3f}'.format(epoch,
              iteration_loss/(i+1), iteration_accuracy/(i+1)))
```