KOREATECH 한국기술교육대학교

5주차 과제

수 강 과 목 자연어처리

담 당 교 수 임상훈 교수님

학 과 컴퓨터공학부

학 번 2020136087

이 름 윤아현

제 출 일 2023.04.07

Ⅰ. 서론

이번 5주차 과제를 통해 영화 리뷰 데이터셋을 활용하여 Skip-Gram을 사용한 Word2Vec를 생성한다.

이 Word2Vec을 활용하여 MLP, CNN과 RNN을 사용한 모델을 통해 학습을 진행한다.

Ⅱ. 본론

1. IMDB 데이터 처리 (20 점)

* Stanford 대학에서 제공하는 IMDB 영화 리뷰 데이터(https://ai.stanford.edu/~amaas/data/sentiment/)를 다운 받아 학습, 테스트 데이터를 구성하시오

    * 데이터는 영어 텍스트 데이터로 긍정/부정의 Binary classification 데이터셋임
    * 데이터셋의 압축을 해제했을 때의 각 디렉토리의 용도는 다음과 같음
        * train/pos : 긍정 label 의 학습 데이터
        * train/neg : 부정 label 의 학습 데이터
        * test/pos : 긍정 label 의 테스트 데이터
        * test/neg : 부정 label 의 테스트 데이터
    * 지금껏 배운 다양한 기법을 적용해 tokenizing, nomalizing 등을 진행한 후 vocab 을 구축하여야함

GRADING
* 데이터셋 전처리를 통해 vocab 구축 (+20)

1번. 데이터 불러오기

제시된 링크를 통해 데이터셋을 tar.gz 형식으로 다운로드 받았다. 이 파일을 압축해제 한 뒤, train.txt, test.txt 파일로 저장해주었다.

이 파일의 내용을 로드하여 train_data와 test_data 변수로 지정해주었다.

Code

```
!tar -xzvf aclImdb_v1.tar.gz
```

```python
from pathlib import Path

train_datas = []
test_datas = []

train_data_dir = Path("/content/aclImdb/train")
test_data_dir = Path("/content/aclImdb/test")

for sentiment in ["pos", "neg"]:
    samples = list(train_data_dir.glob(f"{sentiment}/*.txt"))
    train_datas.extend(samples[:len(samples)])

for sentiment in ["pos", "neg"]:
    samples = list(test_data_dir.glob(f"{sentiment}/*.txt"))
    test_datas.extend(samples[:len(samples)])

train_file = open("train.txt", "w")
test_file = open("test.txt", "w")

# dataset 그룹화 진행
for file, datas in [(train_file, train_datas), (test_file, test_datas)]:
    file.write("id\ttext\tlabel\n")
    for data in datas:
        lines = [line.strip() for line in data.open().readlines()]
        text = " ".join(lines)
        id = data.name[:-4]
        label = 1 if "pos" in data.parts else 0
        file.write(f"{id}\t{text}\t{label}\n")

train_file.close()
test_file.close()
```

```
# data 읽어오기
from requests import get
from os.path import exists

# 파일이 있으면 파일을 읽어온다.
def download(url, filename):
    if exists(filename): # file
        print(f"{filename} already exists")
    else:
        with open(filename, "wb") as file:
            response = get(url) # url
            file.write(response.content)

# 가져온 dataset 읽기
with open("train.txt", "r") as file:
    for i in range(5):
        print(file.readline())

# with open("test.txt", "r") as file:
#     for i in range(5):
#         print(file.readline())

with open("train.txt", "r", encoding="utf-8") as file:
    contents = file.read()
    lines = contents.split("\n")[1:]
    train_data = [line.split("\t") for line in lines if len(line) > 0]

with open("test.txt", "r", encoding="utf-8") as file:
    contents = file.read()
    lines = contents.split("\n")[1:]
    test_data = [line.split("\t") for line in lines if len(line) > 0]
```

Output

없음

2번. 데이터 정제하기

데이터 전처리를 수행하여 train와 test 데이터를 각각 token화된 데이터셋으로 저장해주었다.

전처리 방식은 특수문자 제거, 모든 단어 소문자 변환, nltk tokenizer 적용과 불용어 제거를 진행하였다.

이 때, text에 html "
" 태그의 빈도 수가 많기 때문에 불용어 사전에 "br" 단어를 추가하여 처리해주었다.

또한, 학습을 수행할 때 몇 개의 index에서 label data type 오류가 계속 발생하여 이를 해결하기 위해 token화를 수행할 때 label data를 정수형으로 변환해주었으며 오류가 발생하는 데이터는 무시하는 작업을 수행하였다.

token화를 수행한 후, train data에 대한 dictionary를 생성하여 vocab 변수에 저장해주었다.

Code

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

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer

english_stops = set(stopwords.words('english'))
english_stops.add("br")  # "br"을 불용어 목록에 추가
stemmer = PorterStemmer()
```

```
tokenized_train_dataset = []
tokenized_test_dataset = []

import re

for data in train_data:
    try :
      text = re.sub(r'[,.!?;:()\"\'/<>\d-]', ' ', data[1]) #특수문자 제거
      text = text.lower()  # 소문자 변환
      tokens = word_tokenize(text) #nltk tokenizer 적용
      # stem_tokens = [stemmer.stem(token) for token in tokens] #PorterStemmer 적용
      stop_tokens = [token for token in tokens if token not in english_stops] # 불용어 제거
      labels = int(data[2])
      tokenized_train_dataset.append((stop_tokens, labels))
    # Dataset을 불러올 때 valueerror가 계속 나타나서 label을 데이터 정제할 때 int형으로 변환함
    except ValueError:
      pass

for data in test_data:
    try :
      text = re.sub(r'[,.!?;:()\"\'/<>\d-]', ' ', data[1])
      text = text.lower()
      tokens = word_tokenize(text)
      # stem_tokens = [stemmer.stem(token) for token in tokens]
      stop_tokens = [token for token in tokens if token not in english_stops]
      labels = int(data[2])
      tokenized_test_dataset.append((stop_tokens, labels))
    # Dataset을 불러올 때 valueerror가 계속 나타나서 label을 데이터 정제할 때 int형으로 변환함
    except ValueError:
      pass
```

```
from collections import Counter

token_counter = Counter()

for tokens, _ in tokenized_train_dataset:
    token_counter.update(tokens)

# remove tokens that appear only twice or less
min_count = 2
cleaned_vocab = {"[PAD]":0, "[UNK]":1}
cleaned_vocab_idx = 2

for token, count in token_counter.items():
    if count > min_count:
        cleaned_vocab[token] = cleaned_vocab_idx
        cleaned_vocab_idx += 1
```

Output

없음

2. 데이터셋 통계 분석 (30 점)
* 1 에서 처리한 vocab 을 통해 tokenizing 된 데이터셋의 여러 통계를 계산하시오
    * 통계의 예시
        * 학습/테스트 문서의 수
        * 학습/테스트 데이터의 평균 token 수
        * 데이터의 token histogram
        * 학습/테스트에서의 unk token 의 수
        * 각 token 의 빈도 그래프
        * 긍정/부정의 token 빈도 차이
        * 긍정/부정의 frequent/rare token

* 이전 실습까지 사용한 코드 및 검색을 활용하여 최소 1 개의 그래프를 그려야 함

GRADING
* 분석한 통계의 수 (+5)

1번. 학습/테스트 문서의 수

Train data와 Test data에 저장되어 있는 문서의 수는 동일하게 25,000개이다.

Code

```
# 1번. 학습/테스트 문서의 수
print("Train Dataset의 문서의 수 : ", len(train_data))
print("Test Dataset의 문서의 수 : ", len(test_data))
```

Output

```
Train Dataset의 문서의 수 :  25000
Test Dataset의 문서의 수 :  25000
```

2번. 학습/테스트 데이터의 평균 token 수

Token화를 진행한 후 train data와 test data의 평균 토큰 수를 계산해보았을 때 각각 120단어, 117단어가 나오는 것을 확인할 수 있다.

Code

```python
# 2번. 학습/테스트 데이터의 평균 token 수
train_token_counts = [len(tokens) for tokens, label in tokenized_train_dataset]
train_avg_token_count = sum(train_token_counts) / len(tokenized_train_dataset)
print(f"Train Dataset의 평균 토큰 수: {train_avg_token_count}")

test_token_counts = [len(tokens) for tokens, label in tokenized_test_dataset]
test_avg_token_count = sum(test_token_counts) / len(tokenized_train_dataset)
print(f"Test Dataset의 평균 토큰 수: {test_avg_token_count}")
```

Output

```
Train Dataset의 평균 토큰 수: 120.02281094925564
Test Dataset의 평균 토큰 수: 117.20918040659517
```

Output

3번. 학습/테스트에서 unk token의 수

Train data로 vocabulary를 생성하였기 때문에 Train data에는 [UNK] 토큰의 수는 없다. 그러므로 Test data에 [UNK] 토큰이 몇 개 있는지 확인을 해보았을 때 89,166개가 있는 것을 파악하였다.

이 숫자는 전체 토큰에서 약 3.04%를 차지한다.

Code

```python
# 3번. 학습/테스트에서의 unk token의 수
total_token_count = 0
unk_count = 0

for tokens, _ in tokenized_test_dataset:
    total_token_count += len(tokens)
    for token in tokens:
        if token not in cleaned_vocab:
            unk_count += 1

print("Total Tokens In Test:", total_token_count)
print("Number of [UNK] tokens In Test:", unk_count)
```

Output

```
Total Tokens In Test: 2928823
Number of [UNK] tokens In Test: 89166
```

4번. 각 토큰의 빈도 수

Train data와 Test data에서 각 인덱스별 토큰 빈도수를 출력해보았다. 사전에서 앞의 인덱스를 가질수록 많은 토큰들이 출현한 것을 볼 수 있었다.

앞의 10개의 토큰과 빈도수를 출력해보았을 때 아래와 같은 결과가 나왔다.

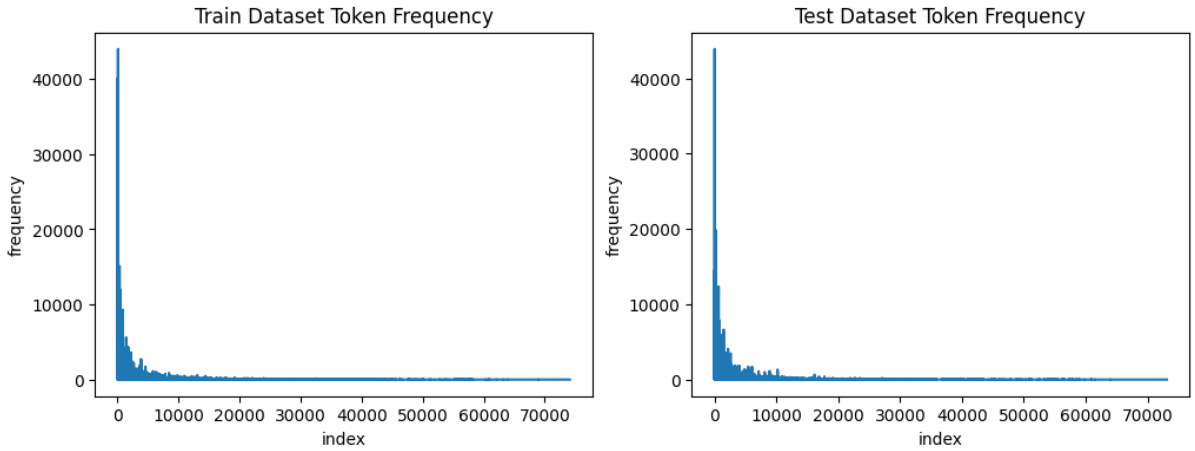
※ test_counter을 사용하여 test_vocab을 추가로 생성하였다.

Code

```
# 4번. 각 토큰의 빈도 수
import matplotlib.pyplot as plt

plt.figure(figsize = (12,4))

plt.subplot(1, 2, 1)
plt.plot(token_counter.values())
plt.title('Train Dataset Token Frequency')
plt.xlabel("index")
plt.ylabel("frequency")
print()

plt.subplot(1, 2, 2)
plt.plot(test_counter.values())
plt.title('Test Dataset Token Frequency')
plt.xlabel("index")
plt.ylabel("frequency")
plt.show()
```

```
# Train data에서 10개 토큰과 그 빈도 수 출력
print("Train Dataset의 처음 10개 토큰과 빈도 수:")
for token, freq in list(token_counter.items())[:10]:
    print(f"{token}: {freq}")

print("\nTest Dataset의 처음 10개 토큰과 빈도 수:")
# Test data에서 처음 10개 토큰과 그 빈도 수 출력
for token, freq in list(test_counter.items())[:10]:
    print(f"{token}: {freq}")
```

Output



```
Train Dataset의 처음 10개 토큰과 빈도 수:
*: 7047
spoilers: 580
herein: 26
really: 11731
scares: 189
killer: 1456
sharks: 42
maybe: 2339
ghosts: 181
trying: 2472

Test Dataset의 처음 10개 토큰과 빈도 수:
okay: 804
ghoulies: 101
kind: 2763
bad: 9160
really: 11345
even: 12192
acting: 6367
storyline: 759
stupid: 1829
forget: 738
```

5번. Train Data에서 긍정/부정 toke 빈도 차이

Train data와 Test data에서 긍정과 부정 문서의 개수는 비슷하며, token의 수 또한 비슷하게 나타나는 것을 볼 수 있다.

Code

```python
# 5번. Train/Test Data에서 긍정/부정의 token 빈도 차이

## train
positive_train = 0
negative_train = 0
positive_tokens_train = []
negative_tokens_train = []

for tokens, label in tokenized_train_dataset:
    # 긍정
    if label == 1:
        positive_tokens_train.extend(tokens)
        positive_train += 1
    # 부정
    else:
        negative_tokens_train.extend(tokens)
        negative_train += 1

print("긍정 문서 개수 (Train) : ", positive_train)
print("부정 문서 개수 (Train)  : ", negative_train)

positive_tokens_set_train = set(positive_tokens_train)
negative_tokens_set_train  = set(negative_tokens_train)

print("긍정 Token의 개수 (Train) : ", len(positive_tokens_set_train))
print("부정 Token의 개수 (Train) : ", len(negative_tokens_set_train))
```

```
## test
positive_test = 0
negative_test = 0
positive_tokens_test = []
negative_tokens_test = []

for tokens, label in tokenized_test_dataset:
    # 긍정
    if label == 1:
        positive_tokens_test.extend(tokens)
        positive_test += 1
    # 부정
    else:
        negative_tokens_test.extend(tokens)
        negative_test += 1

print("긍정 문서 개수 (Test) : ", positive_test)
print("부정 문서 개수 (Test)  : ", negative_test)

positive_tokens_set_test = set(positive_tokens_test)
negative_tokens_set_test  = set(negative_tokens_test)

print("긍정 Token의 개수 (Test) : ", len(positive_tokens_set_test))
print("부정 Token의 개수 (Test) : ", len(negative_tokens_set_test))
```

Output

```
긍정 문서 개수 (Train) :  12492
부정 문서 개수 (Train)  :  12496
긍정 Token의 개수 (Train) :  55200
부정 Token의 개수 (Train) :  53510
긍정 문서 개수 (Test) :  12493
부정 문서 개수 (Test)  :  12492
긍정 Token의 개수 (Test) :  54054
부정 Token의 개수 (Test) :  52797
```

3. Classification 모델 구축 및 학습 (50 점)

* 이론 및 실습 수업을 통해 배운 MLP, CNN, RNN 을 사용하여 각자의 모델을 구축하시오
    * 모델의 크기는 ModelSummary 기준 500MB 의 메모리를 초과하면 안됨
    * 모델은 최대 10 epoch 학습 할 수 있음 (적게 학습하는 것은 ok)

* 최대한 높은 성능을 기록하는 모델을 구축하여야 함
    * 학습엔 주어진 학습 데이터만을 사용하여야 함
    * 테스트 데이터를 학습에 사용하면 0 점
* 모델 구성에 있어 왜 자신이 그런 모델 구조를 설계 하였는지 설명을 하여야함

GRADING
* 모델 구축 및 학습 (+20)
* 모델에 대한 설명 (+10)
* 모델 성능에 따른 성적
    * 상위 0~30% : +20

```
    * 상위 30~50% :  +15
    * 상위 50~70% :  +10
    * 상위 70~100% : +5
```

* 상위 50~70% : +10
* 상위 70~100% : +5

1번. Word2Vec 및 Skip-Gram 설정, 필요한 함수 설정

여기서 두 가지의 튜닝 과정을 거쳤다.

첫 번째, Skip-Gram에서의 Vector size와 window 수를 조정해주었다. Vector size를 300으로 설정한 이유는 더 많은 단어의 정보를 포착하기 위해 차원을 300까지 늘렸으며, 150 ~ 300까지 size를 늘려보면서 여러 model 학습을 수행하였을 때 300에서 가장 좋은 성능을 보였다.

두 번째, 토큰의 길이를 120까지 늘려서 dataset을 만들었다. 그 이유는 앞에서 통계적으로 분석을 수행하였을 때 train dataset에서의 평균 token 수가 120개였으므로 이 만큼의 단어 수를 허용하도록 하였다.

추가로 epoch는 3, 5, 7, 10번 반복으로 수행해보았는데 7번과 10번 반복 학습의 성능 차이가 없어 7번으로 설정하였다.

Code

```
import os
import torch
import random
import numpy as np

np.random.seed(0)
random.seed(0)
torch.manual_seed(0)
```

```
# make word2vec train data
word2vec_train_datas = []
for train_text, _ in tokenized_train_dataset:
    word2vec_train_datas.append([word for word in train_text])
```

```
from gensim.models import Word2Vec

# call CBOW or SkipGram
SkipGram_W2V = Word2Vec(sentences = word2vec_train_datas, vector_size = 300,
                        window = 10, min_count = 1, workers = 4, sg = 1)
```

```python
import numpy as np

# make embedding lookup matrix
embedding_list = []

for token, idx in cleaned_vocab.items():
    if token in SkipGram_W2V.wv:
        embedding_list.append(SkipGram_W2V.wv[token])
    elif token == "[PAD]":
        embedding_list.append(np.zeros(SkipGram_W2V.wv.vectors.shape[1]))
    elif token == "[UNK]":
        embedding_list.append(np.random.uniform(-1, 1, SkipGram_W2V.wv.vectors.shape[1]))
    else:
        embedding_list.append(np.random.uniform(-1, 1, SkipGram_W2V.wv.vectors.shape[1]))

embedding_loopup_matrix = np.vstack(embedding_list)

print(embedding_loopup_matrix.shape)
print(len(cleaned_vocab))
```

```python
from torch.utils.data import Dataset, DataLoader

# Tokeninizing된 Data를 Input으로 받는다.
class SentimentDataset(Dataset):
    def __init__(self, data, vocab):
        self.data = data
        self.vocab = vocab

    def __len__(self):
        return len(self.data)

    # TOkeninizing된 Data
    # 수행 과정: 하나의 text data에 대해서 vocabulary에 있는 인덱스를 mapping한다.
    # mapping 할 때, 시퀀스 길이를 초과하면 truncating 수행
    # 반대면 padding 작업 수행
    def __getitem__(self, index):
        label = self.data[index][1] #현재 인덱스에 해당하는 데이터 샘플 정답
        tokens = self.data[index][0] #token화된 text data

        # vocab에 token이 있으면 ID를 반환하고 아니면 1을 반환한다.
        # 1은 대부분 [UNK] 특수 토큰의 ID를 가리킨다.
        token_ids = [self.vocab[token] if token in self.vocab else 1 for token in tokens]

        # Padding 수행 (token 시퀀스 길이 맞추기)
        # text 평균 길이 확인해보기 :
        if len(token_ids) > 120:
            token_ids = token_ids[:120]
        # sequence 길이가 100이 안되면 0으로 padding 추가
        else:
            token_ids = token_ids[:120] + [0] * (120 - len(token_ids))

        return torch.tensor(token_ids), torch.tensor(label)
```

```
import lightning as pl

class SentimentClassifierPL(pl.LightningModule):
    def __init__(self, sentiment_classifier):
        super(SentimentClassifierPL, self).__init__()
        self.model = sentiment_classifier
        self.loss = nn.CrossEntropyLoss()

        self.validation_step_outputs = []
        self.test_step_outputs = []
        self.save_hyperparameters()

    def training_step(self, batch, batch_idx):
        inputs, labels = batch
        outputs = self.model(inputs)
        loss = self.loss(outputs, labels)
        self.log("train_loss", loss)
        return loss

    def test_step(self, batch, batch_idx):
        inputs, labels = batch
        outputs = self.model(inputs)
        loss = self.loss(outputs, labels)
        self.log("test_loss", loss)
        self.test_step_outputs.append((loss, outputs, labels))
        return loss, outputs, labels

    def on_test_epoch_end(self):
        outputs = self.test_step_outputs
        avg_loss = torch.stack([x[0] for x in outputs]).mean()
        self.log("avg_test_loss", avg_loss)

        all_outputs = torch.cat([x[1] for x in outputs])
        all_labels = torch.cat([x[2] for x in outputs])
        all_preds = all_outputs.argmax(dim=1)
        accuracy = (all_preds == all_labels).float().mean()
        self.log("test_accuracy", accuracy)
        self.test_step_outputs.clear()
```

```
 def configure_optimizers(self):
     optimizer = torch.optim.Adam(self.model.parameters(), lr=1e-3)
     return optimizer
```

```python
import wandb
from lightning.pytorch.loggers import WandbLogger
from lightning.pytorch.callbacks import ModelSummary

wandb.login()

def check_vocab_properties(vocab):
    print(f"Vocab size: {len(vocab)}")
    print(f"Vocab items: {list(vocab.items())[:5]}")


def check_performance(model, vocab,train_data, test_data, wandb_log_name):
    wandb_logger = WandbLogger(project="NLP_Assignment02", name=wandb_log_name, group="word2vec")

    pl_model = SentimentClassifierPL(model)

    train_dataset = SentimentDataset(train_data, vocab)
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, num_workers=4)
    test_dataset = SentimentDataset(test_data, vocab)
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False, num_workers=4)

    trainer = pl.Trainer(
        max_epochs= 7, # 3, 5, 7, 10번 실행
        accelerator="gpu",
        logger=wandb_logger,
        callbacks=[ModelSummary(max_depth=2)]
    )

    trainer.fit(
        model=pl_model,
        train_dataloaders=train_loader,
        # val_dataloaders=val_loader
    )

    trainer.test(dataloaders=test_loader)
```

Output

없음

2번. MLP 모델

차원의 수만 token의 길이(120개)와 word2vec의 vector size(300개)에 맞게 수정해주었다.

더 자세하게 단어의 특성들을 학습하고 싶어 은닉층의 개수를 1개 더 늘려보았지만, 성능은 훨씬 더 낮아졌다.

→ 은닉층이 많아질수록 기울기 소실 문제를 발생시켰을 것이라고 예상한다.

Code

```python
import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self, vocab_size):
        super(MLP, self).__init__()
        # lookup matrix의 가중치 학습 x
        self.embedding = nn.Embedding.from_pretrained(torch.FloatTensor(embedding_loopup_matrix),
                                                      freeze=False)
        self.fc1 = nn.Linear(300 * 120, 120)
        self.fc3 = nn.Linear(120, 2)

    # 원하는 연산을 수행하기 위해서 차원을 찍어본 뒤에 올바르게 설정해야 한다.
    def forward(self, x):
        x = self.embedding(x)
        x = x.view(-1, 300 * 120)
        x = F.relu(self.fc1(x)) # fully-connected-Layer 설정
        x = self.fc3(x)
        return x
```

```python
mlp_model = MLP(len(cleaned_vocab))
```

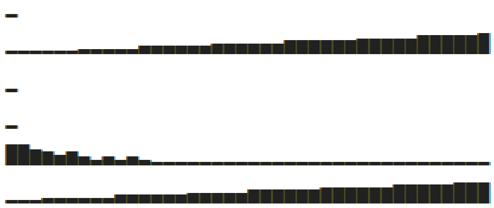
```python
check_performance(mlp_model, cleaned_vocab, tokenized_train_dataset, tokenized_test_dataset, "mlp")
```

Output

Test metric	DataLoader 0
avg_test_loss	0.7466390132904053
test_accuracy	0.8456673622131348
test_loss	0.7463622689247131

Run history:

avg_test_loss	
epoch	
test_accuracy	
test_loss	
train_loss	
trainer/global_step	



Run summary:

avg_test_loss	0.74664
epoch	7
test_accuracy	0.84567
test_loss	0.74636
train_loss	3e-05
trainer/global_step	2737

정확도: 84.5%

3번. TextCNN 모델

Filter를 3x3, 4x4, 5x5 size 3개를 설정하여 학습을 수행하였다.

Word2vec의 vector차원을 300으로 수정해주었다.

Code

```
# 1. Embedding이 2개이다.
# 2. 합성곱층의 Filter가 다 다르다.
# 3. 시간축을 기준으로 max-pooling을 수행한다.
class TextCNN(nn.Module):
    def __init__(self, vocab_size):
        super(TextCNN, self).__init__()
        self.SG_embedding = nn.Embedding.from_pretrained(torch.FloatTensor(embedding_loopup_matrix),
                                                         freeze=True) # skip-gram
        self.RD_embedding = nn.Embedding(vocab_size, 300) # random embedding

        # skip-gram filter를 3개 선언
        # input channel 1개
        # 32개의 filter 사용
        # 3x3, 4x4, 5x5 filter 사용
        self.SG_conv1 = nn.Conv2d(1, 300, (3, 300))
        self.SG_conv2 = nn.Conv2d(1, 300, (4, 300))
        self.SG_conv3 = nn.Conv2d(1, 300, (5, 300))

        # random filter를 3개 선언
        self.RD_conv1 = nn.Conv2d(1, 300, (3, 300))
        self.RD_conv2 = nn.Conv2d(1, 300, (4, 300))
        self.RD_conv3 = nn.Conv2d(1, 300, (5, 300))

        self.fc = nn.Linear(6*300, 2)


    def forward(self, x):
        SG_embedding = self.SG_embedding(x).unsqueeze(1)
        RD_embedding = self.RD_embedding(x).unsqueeze(1)

        SG_conv1_feature = F.relu(self.SG_conv1(SG_embedding).squeeze(3))
        SG_conv2_feature = F.relu(self.SG_conv2(SG_embedding).squeeze(3))
        SG_conv3_feature = F.relu(self.SG_conv3(SG_embedding).squeeze(3))

        RD_conv1_feature = F.relu(self.RD_conv1(RD_embedding).squeeze(3))
        RD_conv2_feature = F.relu(self.RD_conv2(RD_embedding).squeeze(3))
        RD_conv3_feature = F.relu(self.RD_conv3(RD_embedding).squeeze(3))
```

```
 # 시간축 → token의 수
 # 정해진 filter 개수만큼 가져온다.
 SG_max1 = F.max_pool1d(SG_conv1_feature, SG_conv1_feature.size(2)).squeeze(2)
 SG_max2 = F.max_pool1d(SG_conv2_feature, SG_conv2_feature.size(2)).squeeze(2)
 SG_max3 = F.max_pool1d(SG_conv3_feature, SG_conv3_feature.size(2)).squeeze(2)

 RD_max1 = F.max_pool1d(RD_conv1_feature, RD_conv1_feature.size(2)).squeeze(2)
 RD_max2 = F.max_pool1d(RD_conv2_feature, RD_conv2_feature.size(2)).squeeze(2)
 RD_max3 = F.max_pool1d(RD_conv3_feature, RD_conv3_feature.size(2)).squeeze(2)

 x = torch.cat([SG_max1, SG_max2, SG_max3, RD_max1, RD_max2, RD_max3], dim=1)

 x = self.fc(x)

 return x
```

```
textcnn_model = TextCNN(len(cleaned_vocab))
```

```
check_performance(textcnn_model, cleaned_vocab, tokenized_train_dataset,
                  tokenized_test_dataset, "textcnn")
```

Output

Test metric	DataLoader 0
avg_test_loss	0.43273279070854187
test_accuracy	0.8707624077796936
test_loss	0.4329492151737213

Run history:

Run summary:

avg_test_loss	0.43273
epoch	7
test_accuracy	0.87076
test_loss	0.43295
train_loss	0.00056
trainer/global_step	2737

정확도: 87.07%

4번. BiLSTM 모델

더 완벽한 학습을 수행하기 위해서 모델이 양방향으로 앞의 단어와 뒤의 단어 모두 학습하는 것이 더 좋은 성능을 낼 것이라고 생각하여 양방향 LSTM을 사용하였다.

이 또한, word2vec size를 300으로 수정해주었다.

Code

```
class biLSTM(nn.Module):
    def __init__(self, vocab_size):
        super(biLSTM, self).__init__()
        hidden_size = 300
        self.embedding = nn.Embedding.from_pretrained(torch.FloatTensor(embedding_loopup_matrix),
                                                      freeze=False)
        # 양방향 수행 (이전, 이후 단어를 사용한다.) > Right, Left 둘 다 사용
        # hidden state 2개를 사용한다.
        self.rnn = nn.LSTM(300, 300, batch_first=True, num_layers=2, bidirectional=True)
        self.fc = nn.Sequential(
          nn.Linear(hidden_size * 2, hidden_size),  # 양방향이므로 hidden_size * 2
          nn.ReLU(),
          nn.Linear(hidden_size, 2)  # 최종 출력 크기는 작업에 따라 결정
        )

    def forward(self, x):
        x = self.embedding(x)
        x, _ = self.rnn(x)
        x = x.mean(dim=1)
        x = self.fc(x)
        return x
```

```
bilstm_model = biLSTM(len(cleaned_vocab))
```

```
check_performance(bilstm_model, cleaned_vocab, tokenized_train_dataset,
                  tokenized_test_dataset, "bilstm")
```

Output

Test metric	DataLoader 0
avg_test_loss	1.3727058172225952
test_accuracy	0.8258554935455322
test_loss	1.374728798866272

Run history:

avg_test_loss	
epoch	
test_accuracy	
test_loss	
train_loss	
trainer/global_step	

Run summary:

avg_test_loss	1.37271
epoch	7
test_accuracy	0.82586
test_loss	1.37473
train_loss	4e-05
trainer/global_step	2737

정확도: 82.58%

Wandb를 통한 전체적인 성능 비교

Ⅲ. 결론

3가지의 모델을 사용하였을 때 TextCNN이 가장 좋은 성능을 보였다. 87.01%로 다른 모델에 비해 훨씬 좋은 성능을 보였다. 다양한 커널을 사용하여 텍스트의 지역적 특징들을 더 잘 포착할 수 있어서 좋은 성능을 보인 것이라고 판단하였다.

이전의 과제에서 영어 단어 정제를 한 뒤 모델을 훈련했을 때보다 여러 유용한 딥러닝 모델을 사용하니까 더 좋은 성능이 나왔다고 예측한다.