KOREATECH 한국기술교육대학교

3주차 과제

수 강 과 목 자연어처리

담 당 교 수 임상훈 교수님

학 과 컴퓨터공학부

학 번 2020136087

이 름 윤아현

제 출 일 2023.03.25

Ⅰ. 서론

이번 3주차 과제를 통해 스스로 문서들을 토큰화하여 Vocabulary를 생성해본다.

Text 정제 과정을 통하여 최적의 성능을 가지는 Vocabulary를 생성하여 Model을 훈련시켜본다.

여러 가지 정제 과정의 성능을 직접 평가해보면서 어떠한 점이 부족하거나 만족스러웠는지 확인해본다.

Ⅱ. 본론

1. 영어로된 영화 리뷰 데이터 전처리 (40 점)

* Kaggle 에서 제공하는 Movie Review Dataset(https://www.kaggle.com/datasets/vipulgandhi/movie-review-dataset?resource=download)를 전처리하여 Vocab 을 구성하시오.

*제공되는 코드는 해당 데이터를 실습에 사용한 NSMC 데이터의 형식으로 바꾸어주는 코드이다.
*단, 실습과 달리 train, val, test 의 세 부분으로 나누어짐
*Vocab 구성시 Whitespace tokenizer 를 사용하여야 하며, Data cleaning, Text cleaning 과정을 거치지 않은 기본 Vocab 을 생성하시오

1번. 데이터 불러오기

Kaggle에 있는 데이터셋을 다운로드 받아 Train, Validation, Test Data로 나누어 File을 읽어왔다.

저장한 Data 파일을 읽어, 각각 train_data, val_data, test_data로 저장해주었다.

Code

```
## 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("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("val.txt", "r", encoding="utf-8") as file:
    contents = file.read()
    lines = contents.split("\n")[1:]
    val_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]
```

2번. Whitespace Tokenizer를 통한 Vocab 생성

Text를 공백을 기준으로 토큰화를 수행한다.

Train, Validation, Test Dataset에 대해 모두 수행하며, Train Dataset에 있는 토큰들을 Vocabulary로 생성한다.

Code

```python
# Whitespace tokenizer를 통해 기본 Vocab 만들기
from collections import Counter
from tqdm import tqdm

# white space vocab 만들기
whitespace_vocab = {"[PAD]":0, "[UNK]":1}
whitespace_vocab_idx = 2

# 공백을 기준으로 token 생성
tokenizer = lambda x: x.split()

tokenized_train_dataset = []
tokenized_val_dataset = []
tokenized_test_dataset = []

# train/test data에 대한 tokenizer을 수행한다.
for data in train_data:
    tokens = tokenizer(data[1])
    labels = data[2]
    tokenized_train_dataset.append((tokens, labels))

for data in val_data:
    tokens = tokenizer(data[1])
    labels = data[2]
    tokenized_val_dataset.append((tokens, labels))

for data in test_data:
    tokens = tokenizer(data[1])
    labels = data[2]
    tokenized_test_dataset.append((tokens, labels))

for tokens, _ in tokenized_train_dataset:
    for token in tokens:
        if token not in whitespace_vocab:
            whitespace_vocab[token] = whitespace_vocab_idx
            whitespace_vocab_idx += 1
```

```python
def check_vocab_properties(vocab):
    print(f"Vocab size: {len(vocab)}") #vocab size 길이 반환
    print(f"Vocab items: {list(vocab.items())[:5]}") # vocab에 어떤 data가 있는지
```

Output

```
Vocab size: 42670
Vocab items: [('[PAD]', 0), ('[UNK]', 1), ('moviemaking', 2), ('is', 3), ('a', 4)]
```

→ Vocab에는 42,670개의 Token이 저장되었으며, check_vocab_properties 함수를 통해 0~5번째 인덱스에 저장된 단어들을 확인할 수 있다.

2. 영어로된 영화 리뷰 데이터 분류 모델 학습 (30 점)

*실습에 사용한 모델 및 Dataset 객체를 통해 긍정 부정 분류 모델을 학습하시오.
*실습코드와 동일한 구조의 모델을 사용해야함
*실습코드와 동일한 Trainer 및 하이퍼파라미터를 유지해야함
***Train, Validation, Test 데이터를 모두 학습에 알맞게 사용할 것**

1번. Train Models 정의하기

3주차 실습 수업시간에 정의한 Model을 그대로 가져와 정의해주었다.

학습을 수행시키는 check_performace 함수에서 (train_dataset, train_dataloader), (val_dataset, val_dataloader), (test_dataset, test_dataloader)에 dataset을 알맞게 정의해준다.

Code

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

wandb.login()

# 학습 수행
def check_performance(vocab,train_data, val_data, test_data, wandb_log_name):
  # grouping을 통해서 현재 실험들을 wandb에서 한 눈에 볼 수 있도록 한다.
    wandb_logger = WandbLogger(project="NLP_Week3", name=wandb_log_name, group="Lec02_Assignment")

    # 감정 분류 모델
    model = SentimentClassifier(len(vocab))
    # PytorchLighting을 사용하기 위해 model을 감싸준다.
    pl_model = SentimentClassifierPL(model)

    train_dataset = SentimentDataset(train_data, vocab)
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, num_workers=4)
    val_dataset = SentimentDataset(val_data, vocab)
    val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False, 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=1,
                    accelerator="cpu", #gpu 할당량이 끝나버렸어요 ..
                    logger=wandb_logger
                    )

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

    trainer.test(dataloaders=test_loader)

    wandb.finish()
```

SentimentClassifier, SentimentClassidierPL, SentimentDataset은 모두 실습시간과 동일한 code를 사용하였기 때문에 보고서에서는 생략한다.

2번. Whitespace Vocab을 사용한 Modeling 수행

앞에서 정의한 check_performace 함수를 사용하여 학습을 수행한다.

Input

whitespace_vocab(사용할 vocabulary), tokenized_train_dataset, tokenized_val_dataset,

tokenized_test_dataset(학습, 검증, 테스트에 필요한 dataset), "whitespace"(wandb에서 보여줄 때 사용할 이름)를 Input으로 준다.

Code

```
check_performance(whitespace_vocab, tokenized_train_dataset, tokenized_val_dataset, tokenized_test_dataset, "whitespace")
```

Output

Test Accuracy: 51.67%

3. Vocab 개선 (30 점)

* 상기 구축한 Vocab 을 최적화 하여 모델의 성능을 개선하시오
* 남이 구축한 Vocab 을 가져오는 것을 제외, 모든 것을 허용
* 참고하면 좋은 것
* 영어의 불용어(Stopword) : https://gist.github.com/sebleier/554280
* 영어에 적합한 Tokenizer : NLTK 라이브러리 참고 (https://www.nltk.org/api/nltk.tokenize.html)
* 영어의 lemmatizing 및 stemming : NLTK 라이브러리를 참고 (https://www.nltk.org/api/nltk.stem.html)
* Regular Expression 사용시 유용한 사이트 : https://regexr.com/

1번. Text 분석을 통한 개선 방법 설정

Code

```
# 등장빈도 알아보기
from collections import Counter

token_counter = Counter()

for tokens, _ in tokenized_train_dataset: # token vocab
    token_counter.update(tokens)
```

```
 # method를 사용, count가 많은 것들 10개 출력
token_counter.most_common(10)
```

```
token_counter.most_common()[-10:]
```

```
import matplotlib.pyplot as plt

def plot_frequency_distribution(counter):
    plt.plot(counter.values())
    plt.ylabel('Token Frequency')
    plt.show()
```

```
plot_frequency_distribution(token_counter)
```

Output

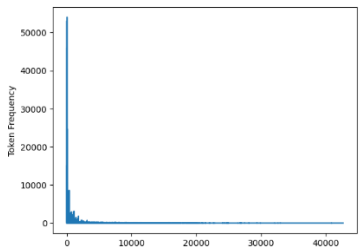
가장 많이 나온 단어 10개:

[(',', 54009), ('the', 53023), ('.', 46018), ('a', 26539), ('and', 24616), ('of', 23604), ('to', 22107), ('is', 17408), ('in', 15054), ('"', 12449)]

가장 많이 적게 단어 10개:

[('allegorically', 1), ('combing', 1), ("capote's", 1), ('true-crime', 1), ("change-i'll", 1), ('yourself-but', 1), ("movie-he's", 1), ('chandler/ross', 1), ('genre-last', 1), ('example-', 1)]

인덱스별 빈도수:



※ Text 분석으로 알 수 있는 점

1번. 토큰이 한국어보다는 변형이 덜하니, token의 수가 한국어보다 더 줄어든 것을 확인할 수 있다.

2번. Most common word를 보면, 조사와 특수문자가 흔히 나온다는 것을 확인할 수 있다.

3번. 빈도수가 적은 word를 보면, '-' 혹은 '/'로 단어들이 결합되어 있는 것을 확인할 수 있다.

4번. Word가 나오는 빈도수를 보면, 극단적인 'L' 형태를 가지고 있는 것을 확인할 수 있다.

※ Text 분석을 통해 수행할 vocab 개선 방법

1번. 등장빈도가 1만 나오는 경우 다 삭제 (4번 개선)

2반. 정규표현식을 통한 필요 없는 문자 제거 (2, 3번)

3번. Stopword 제거 (2번)

4번. NLTK 적용 → Lemmatizer, Porter Stemmer, Pos Tagging 사용 (2번)

이 4가지 방법을 통해 가장 좋았던 Vocab 생성 방식을 적용한다.

2번. Vocab 개선 수행

2-1번. 등장빈도가 1만 나오는 경우 다 삭제

Code

```
# 1번. 빈도 수가 1인 경우, 제거 진행
min_count = 1
min1removed_vocab = {"[PAD]":0, "[UNK]":1}
min1removed_vocab_idx = 2

for token, count in token_counter.items():
    if count > min_count: # 빈도가 1보다 큰 값들만 vocab에 저장하겠다.
        min1removed_vocab[token] = min1removed_vocab_idx
        min1removed_vocab_idx += 1
```

```
# 단어의 수가 절반정도 줄어든 것을 볼 수 있다.
check_vocab_properties(min1removed_vocab)
```

```
check_performance(min1removed_vocab, tokenized_train_dataset, tokenized_val_dataset, tokenized_test_dataset, "whitespace_min_count_1")
```

Output

```
Vocab size: 23763
Vocab items: [('[PAD]', 0), ('[UNK]', 1), ('moviemaking', 2), ('is', 3), ('a', 4)]
```

Test Accuracy: 49.33%

Vocabulary의 길이가 절반 정도 줄어든 것을 보아 data의 부족으로 whitespace vocab model보다 성능이 떨어진다고 짐작된다.

2-2번. 정규표현식을 통한 필요 없는 문자 제거

Code

```
# 2번. 정규표현식 제거
import re

tokenizer = lambda x: x.split()

tokenized_train_dataset = []
tokenized_val_dataset = []
tokenized_test_dataset = []

for data in train_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1]) #특수문자 제거
    text = text.lower()  # 소문자 변환
    tokens = tokenizer(text)
    labels = data[2]
    tokenized_train_dataset.append((tokens, labels))

for data in val_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = tokenizer(text)
    labels = data[2]
    tokenized_val_dataset.append((tokens, labels))

for data in test_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = tokenizer(text)
    labels = data[2]
    tokenized_test_dataset.append((tokens, labels))

token_counter = Counter()

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

정규표현식을 통해 특수문자를 제거하고 대문자를 소문자로 변경해주었다.

```
cleaned_vocab = {"[PAD]":0, "[UNK]":1}
cleaned_vocab_idx = 2

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

```
check_performance(cleaned_vocab, tokenized_train_dataset, tokenized_val_dataset, tokenized_test_dataset, "whitespace_cleaned")
```

Output

Test Accuracy: 54.67%

Text의 정제를 수행하니, 그 전보다 성능이 개선된 것을 볼 수 있다.

대문자로 인해 공통 단어가 2개로 vocab에 저장되는 상황을 막을 수 있어 정확도가 높아졌다고 생각한다.

2-3번. stopword 제거

불용어 제거는 "영어의 불용어 사전 참조" 방식과 "NLTK 라이브러리의 stopwords" 방식 2개를 수행하였다.

1번. 영어의 불용어 사전 참조 (https://gist.github.com/sebleier/554280)

Code

```
# 1번. 불용어 사전 참조
stopwords = []

# stopwords.txt 파일 열기
with open('stopwords.txt', 'r') as file:
    for line in file:
        stopwords.append(line.strip())

stopwords[:10]
```

불용어 파일을 txt로 다운받아 stopwords 리스트에 저장한다.

```
import re

tokenizer = lambda x: x.split()

tokenized_train_dataset = []
tokenized_val_dataset = []
tokenized_test_dataset = []

for data in train_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1]) #특수문자 제거
    text = text.lower()  # 소문자 변환
    tokens = tokenizer(text)
    stop_tokens = [token for token in tokens if token not in stopwords]
    labels = data[2]
    tokenized_train_dataset.append((stop_tokens, labels))

for data in val_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = tokenizer(text)
    stop_tokens = [token for token in tokens if token not in stopwords]
    labels = data[2]
    tokenized_val_dataset.append((stop_tokens, labels))

for data in test_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = tokenizer(text)
    stop_tokens = [token for token in tokens if token not in stopwords]
    labels = data[2]
    tokenized_test_dataset.append((stop_tokens, labels))

token_counter = Counter()

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

위에서 성능이 좋게 나왔던 특수문자 제거 및 소문자 변환을 수행한 뒤, 토큰이 stopwords에 있으면 제거하는 방식으로 수행하였다.

```
stop_vocab = {"[PAD]":0, "[UNK]":1}
stop_vocab_idx = 2

for token, count in token_counter.items():
    # if token not in stopwords:
        stop_vocab[token] = stop_vocab_idx
        stop_vocab_idx += 1
```

```
check_performance(stop_vocab, tokenized_train_dataset, tokenized_val_dataset, tokenized_test_dataset, "whitespace_stopwords")
```

Output

Test Accuracy: 45.67%

2번. NLTK 라이브러리에서 제공하는 stopwords 사용

Code

```
# 2번. nltk stopwords 사용한 경우

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

```
from nltk.corpus import stopwords

english_stops = set(stopwords.words('english'))
```

NLTK 라이브러리를 import하여 stopwords를 다운받아 변수로 지정한다.

```
import re

tokenizer = lambda x: x.split()

tokenized_train_dataset = []
tokenized_val_dataset = []
tokenized_test_dataset = []

for data in train_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1]) #특수문자 제거
    text = text.lower()  # 소문자 변환
    tokens = tokenizer(text)
    stop_tokens = [token for token in tokens if token not in english_stops]
    labels = data[2]
    tokenized_train_dataset.append((stop_tokens, labels))

for data in val_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = tokenizer(text)
    stop_tokens = [token for token in tokens if token not in english_stops]
    labels = data[2]
    tokenized_val_dataset.append((stop_tokens, labels))

for data in test_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = tokenizer(text)
    stop_tokens = [token for token in tokens if token not in english_stops]
    labels = data[2]
    tokenized_test_dataset.append((stop_tokens, labels))

token_counter = Counter()

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

```
stop_eng_vocab = {"[PAD]":0, "[UNK]":1}
stop_eng_vocab_idx = 2

for token, count in token_counter.items():
      stop_eng_vocab[token] = stop_eng_vocab_idx
      stop_eng_vocab_idx += 1
```

```
check_performance(stop_vocab, tokenized_train_dataset, tokenized_val_dataset, tokenized_test_dataset, "whitespace_nltk_stopwords")
```

Output

Test Accuracy: 52.00%

1번의 방식보다 NLTK 라이브러리의 stopwords를 통한 text 정제가 더 성능이 좋은 것을 확인할 수 있다.

2-4번. NLTK 적용 → Lemmatizer, Porter Stemmer, Pos Tagging 사용

1번. Lemmatizer

Code

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

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

```
# 1번. 표제어 추출 - Lemmatization 사용
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
```

```
tokenized_train_dataset = []
tokenized_val_dataset = []
tokenized_test_dataset = []

for data in train_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1]) #특수문자 제거
    text = text.lower()  # 소문자 변환
    tokens = word_tokenize(text) #nltk tokenizer 적용
    lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens] #lemmatizer 적용
    stop_tokens = [token for token in lemmatized_tokens if token not in english_stops] # 불용어 제거
    labels = data[2]
    tokenized_train_dataset.append((stop_tokens, labels))

for data in val_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = word_tokenize(text)
    lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens]
    stop_tokens = [token for token in lemmatized_tokens if token not in english_stops]
    labels = data[2]
    tokenized_val_dataset.append((stop_tokens, labels))

for data in test_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = word_tokenize(text)
    lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens]
    stop_tokens = [token for token in lemmatized_tokens if token not in english_stops]
    labels = data[2]
    tokenized_test_dataset.append((stop_tokens, labels))

token_counter = Counter()

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

이 때, 앞서 성능이 좋았던 특수문자 제거 및 소문자 변환을 수행한 뒤 Lemmatizer을 통해 표제어를 추출하였고 추가적으로 Stopwords 제거도 진행하였다.

```
lem_vocab = {"[PAD]":0, "[UNK]":1}
lem_vocab_idx = 2

for token, count in token_counter.items():
      lem_vocab[token] = lem_vocab_idx
      lem_vocab_idx += 1
```

```
check_performance(lem_vocab, tokenized_train_dataset, tokenized_val_dataset, tokenized_test_dataset, "lemmatizer_nltk")
```

Output

Test Accuracy: 50.67%

2번. Porter Stemmer

Code

```
# 2번. Porter Stemmer 적용
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()
```

```
tokenized_train_dataset = []
tokenized_val_dataset = []
tokenized_test_dataset = []

for data in train_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', 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 stem_tokens if token not in english_stops] # 불용어 제거
    labels = data[2]
    tokenized_train_dataset.append((stop_tokens, labels))

for data in val_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = word_tokenize(text)
    stem_tokens = [stemmer.stem(token) for token in tokens]
    stop_tokens = [token for token in stem_tokens if token not in english_stops]
    labels = data[2]
    tokenized_val_dataset.append((stop_tokens, labels))

for data in test_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = word_tokenize(text)
    stem_tokens = [stemmer.stem(token) for token in tokens]
    stop_tokens = [token for token in stem_tokens if token not in english_stops]
    labels = data[2]
    tokenized_test_dataset.append((stop_tokens, labels))

token_counter = Counter()

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

1번 방식과 동일하게 진행하였다.

```
stem_vocab = {"[PAD]":0, "[UNK]":1}
stem_vocab_idx = 2

for token, count in token_counter.items():
      stem_vocab[token] = stem_vocab_idx
      stem_vocab_idx += 1
```

```
check_performance(stem_vocab, tokenized_train_dataset, tokenized_val_dataset, tokenized_test_dataset, "stemmer_nltk")
```

Output

Test Accuracy: 54.67%

3번. Pos Tagging

Code

```
# 3번. pos tagging 사용하기
import nltk
nltk.download('averaged_perceptron_tagger')
```

```
from nltk.tokenize import word_tokenize

my_tag_set = ['NN', 'VB', 'JJ'] #명사, 동사, 형용사만 가져온다.
```

명사, 동사, 형용사만 추출하여 이를 활용하여 vocabulary를 만든다.

```
import re

tokenized_train_dataset = []
tokenized_val_dataset = []
tokenized_test_dataset = []

for data in train_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1]) #특수문자 제거
    text = text.lower()  # 소문자 변환
    tokens = word_tokenize(text) #nltk tokenizer 적용
    pos_tokens = [word for word, tag in nltk.pos_tag(tokens) if tag in my_tag_set] #원하는 품사만 가져옴
    stop_tokens = [token for token in pos_tokens if token not in english_stops] # 불용어 제거
    labels = data[2]
    tokenized_train_dataset.append((stop_tokens, labels))

for data in val_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = word_tokenize(text)
    pos_tokens = [word for word, tag in nltk.pos_tag(tokens) if tag in my_tag_set]
    stop_tokens = [token for token in pos_tokens if token not in english_stops]
    labels = data[2]
    tokenized_val_dataset.append((stop_tokens, labels))

for data in test_data:
    text = re.sub(r'[,.!?;:()\"\'/-]', ' ', data[1])
    text = text.lower()
    tokens = word_tokenize(text)
    pos_tokens = [word for word, tag in nltk.pos_tag(tokens) if tag in my_tag_set]
    stop_tokens = [token for token in pos_tokens if token not in english_stops]
    labels = data[2]
    tokenized_test_dataset.append((stop_tokens, labels))

token_counter = Counter()

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

1번, 2번 방식과 동일하게 진행한다.

```
pos_vocab = {"[PAD]":0, "[UNK]":1}
pos_vocab_idx = 2

for token, count in token_counter.items():
      pos_vocab[token] = pos_vocab_idx
      pos_vocab_idx += 1
```

```
check_performance(pos_vocab, tokenized_train_dataset, tokenized_val_dataset, tokenized_test_dataset, "pos_nltk")
```

Output

Test Accuracy: 50.00%

결론

표제어 추출을 위한 Post Stemmer 사용하기

[사용한 전처리 프로세스]

nltk word tokenizer 사용

정규표현식을 활용한 특수문자 제거

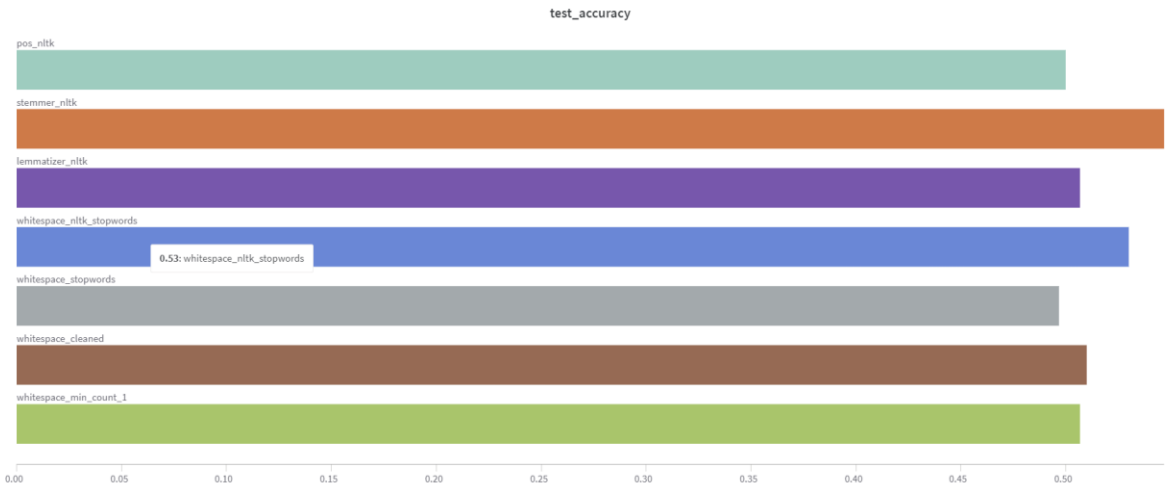
대문자를 소문자로 대체

Post stemmer 사용

불용어(stopwords) 제거

성능(Test 정확도): 54.67%

[3번 문항에 대한 Wandb Test Accuracy 첨부]



4. 왜 Vocab 에 따라 성능이 달라지는가? (Bonus 20 점)

* 실습 및 과제를 통해 Vocab 에 따라 성능이 차이남을 확인할 수 있음
* 왜 성능이 하락하는지에 대한 자신의 생각을 기술하시오

1번. data의 부족 Issue

정제 방식에 따라, data가 점점 더 줄어든다. (stopwords, 소문자 변환 등 수행)

이로 인해 학습할 수 있는 data 수도 비례적으로 줄어들기 때문에 성능이 조금밖에 증가하지 못한다고 생각한다.

영어 같은 경우에, 한글조합보다는 더 적은 조합을 가지고 있으므로 현저히 적은 vocab data set이 만들어지는 것을 볼 수 있다.

Data가 많다면, 지금보다 더 좋은 성능을 낼 수 있을 것이라고 생각한다.

2번. [UNK] Token이 존재한다.

만일, Train vocab에 Token이 없으면, [UNK] Token으로 대체되기 때문에 Validation set이나 Test set에서 실질적이고 중요한 의미를 가지는 단어여도 [UNK] Token이 되어버릴 수 있다.

<ValidationSet과 TestSet에서의 [UNK] 수 확인>

```
total_token_count_val = 0
unk_val_count = 0

for tokens, _ in tokenized_val_dataset:
    total_token_count_val += len(tokens)
    for token in tokens:
        if token in stem_vocab:
            token_counter[token] += 1
        else :
            unk_val_count += 1

print("Total Tokens In validation:", total_token_count_val)
print("Most Common Word In validation:", token_counter.most_common(20))
print("Number of [UNK] tokens In validation:", unk_val_count)
```

```
Total Tokens In validation: 113936
Most Common Word In validation: [('film', 24575), ('thi', 21337), ('hi', 20693), ('movi', 15778), ('one', 13249), ('wa', 10864),
Number of [UNK] tokens In validation: 2597
```

```
total_token_count_test = 0
unk_test_count = 0

for tokens, _ in tokenized_test_dataset:
    total_token_count_test += len(tokens)
    for token in tokens:
        if token in stem_vocab:
            token_counter[token] += 1
        else :
            unk_test_count += 1

print("Total Tokens In Test: ", total_token_count_test)
print("Most Common Word In test:", token_counter.most_common(20))
print("Number of [UNK] tokens In test:", unk_test_count)
```

```
Total Tokens In Test:  113388
Most Common Word In test: [('film', 27977), ('thi', 24277), ('hi', 23569), ('movi', 18040), ('one', 15129), ('wa', 12366),
Number of [UNK] tokens In test: 2558
```

3번. 의미가 없는 단어들의 빈도 수

Vocab에 있는 단어들을 확인해보았을 때, 'is', 'a'와 같이 의미가 없는 조사들의 빈도 수가 많이 나오는 것을 볼 수 있다. 이러한 단어들이 문맥 간의 유사도 혹은 의미를 구별하는 데 영향을 미쳐 학습 성능이 저하될 수 있다.

이를 해결하기 위해서 위에서 사용한 Stopwords 제거 혹은 TF-IDF와 같은 방식을 사용해야 한다.

3번. 경험 부족

여러 가지 경우의 수를 가지고 vocab을 생성할 수 있는데, 경우의 수는 셀 수 없이 많으며 이로 인해 성능 개선이 될 가능성이 있다.

여러 text 정제 방식들을 조합하여 사용하였을 경우, stopwords를 직접 text를 보고 찾아내어 단어를 추가적으로 삭제하였을 경우, 혹은 pos tagging을 수행하였을 때 적절한 형태소를 선택하였을 때, 등 여러가지 경우의 수가 있기 때문에 성능 개선의 여지가 있다.

추가적인 사항. 모델 성능 개선

현재는 단순한 모델로 학습을 돌렸고 epoch 수가 현저히 적기 때문에, 학습이 제대로 안 이루어졌을 가능성이 높다. LSTM 모델 혹은 사전 학습된 모델을 통해 학습을 수행한다면, 더 좋은 성능을 보일 것이다.

Ⅲ. 결론

이번 과제를 통하여 직접 Vocab을 구성해보고 성능을 평가해보는 시간을 가졌다.

성능을 개선하기 위해서 사용할 수 있는 여러 가지 방법을 수행하였다. 방법이 많다 보니, 여러가지 방법들을 어떻게 효율적이게 사용할 것인지 혹은 이 방식을 내가 왜 사용하려고 하는지에 대해 생각을 많이 해보게 되었다.

Text 분석 과정을 통해서 이 과정을 수행해야할 이유에 대해서 분석해보고 직접 수행도 해보았다.

생각보다 WhiteSpace만을 수행한 Vocab보다 성능이 많이 증가하진 않아서 아쉽지만(약 3% 증가), 나중에는 모델링 과정을 직접 수행해보고 탐색적 분석을 통해 어떠한 방식을 추가적으로 사용할지 생각해보는 시간을 가지고 싶다.