202246116 전북대학교 컴퓨터공학부 이재헌

목차

1. 개요

본 과제에서는 Subword Tokenization을 BPE 방식으로 구현하여, 제공된 셰익스피어의 "The Complete Works of William Shakespeare" 를 통해 vocab 및 merge rule을 형성하고, 이를 통해서 새로운 텍스트 파일이 어떻게 tokenization이 진행되는지를 파악하는 것을 목적으로 함.

또한, max_vocab size에 따라 vocab, merge rule 및 학습 결과의 차이에 대해서 분석해 보고, 새로운 텍스트가 얼마나 잘 Tokenization이 되는지에 관해 분석을 진행하여, 궁극적으로 max_vocab size를 어떻게 설정하는 것이 가장 효율적인지를 탐색함.

2. BPE 알고리즘 설명

Byte pair encoding (BPE)는 1994년에 만들어진 데이터 압축 알고리즘임. BPE의 핵심 아이디어는 자주 등장하는 문자 pair를 계속해서 merge 하여 vocab를 구축시키는 것임.

BPE 알고리즘은 다음과 같이 동작함.

1. Corpus 구성
   모든 단어를 문자 단위로 분해하여 공백으로 구분된 형태로 변환
   hello → h e l l o 처럼 분리
2. Vocab 구성

단어가 얼마나 등장했는지에 따라서 빈도를 저장

3. 자주 등장하는 pair를 merge

   자주 등장하는 (최다 빈도) pair를 계산하여 이를 하나의 새로운 subword를 만든 뒤 vocab에 업데이트시킴. 예를 들어 h u g s가 존재하는데 ug가 연속해서 제일 많이 나왔다면 이를 h ug s로 merge시킴

4. Iteration

   설정한 max_vocab size까지 계속해서 반복하고 도달하면 반복을 종료

BPE의 장점

- 모르는 단어도 Subword 조합으로 사용이 가능 (unknown 문제 해결)
- max_vocab size를 정해서 효율적으로 조정이 가능

BPE의 단점

- Context에 따른 적용이 어려움
- 특수문자 처리에 어려움이 존재

3. 알고리즘 설계 및 구현

BPE를 통한 Subword Tokenization과정을 진행할 때 크게 학습 & 추론 두 가지의 가능성이 존재하며, main 함수에서 복잡도를 줄이고, 결합도를 낮추기 위해서 학습(BPE Trainer)과 추론(BPE Tokenizer) 클래스로 나누어서 진행하였음.

BPETrainer

BPE 알고리즘의 학습을 수행하는 부분으로 주어진 pg100.txt를 바탕으로 Vocab을 생성하고, merge rules를 학습하는 역할

1) pg100.txt를 읽고 데이터 전처리 진행

   파일을 읽기 위해서 file_path를 매개변수로 받아와서 lower()를 통해 소문자로 변하게 한 다음 `re.sub(r'[^a-z\s]', '', text)` 를 통해서 텍스트를 분리시킴. 또한, 반복을 돌려서 문자 단위로 분리시켜서 반환해 주는 역할을 함

```
def preprocess_text(self, file_path):
  with open(file_path, 'r', encoding='utf-8') as f:
    text = f.read().lower()
  text = re.sub(r'[^a-z\s]', '', text)
  words = text.strip().split()
  processed = []
  for word in words:
    letters = list(word)
    spaced_word = ' '.join(letters)
    processed.append(spaced_word)
  return processed
```

2) basic vocab를 구성

corpus에 있는 문자열에서 단어를 하나씩 꺼내서 vocab에 존재한다면 1을 더하고 없으면 1로 설정하여서 빈도수를 설정하여 basic vocab를 형성함.

```
def build_vocab(self, corpus):
  self.vocab = {}
  for word in corpus:
    if word in self.vocab:
      self.vocab[word] += 1
    else:
      self.vocab[word] = 1
```

3) Frequent Pair를 찾아서 병합

가장 빈번히 등장한 pair를 찾아야 함. 이전의 build_vocab에서 진행했던 word와 freq(빈도)를 찾아서 이를 character 단위의 리스트로 구성한 다음 인접한 두 문자를 하나의 tuple로 묶어서 pair를 만든 뒤 이걸 반복이 끝날 때까지 빈도수를 구해서 이 빈도 리스트를 반환하여 가장 빈번히 등장한 pair를 찾을 수 있도록 함.

```
def get_pair_frequency(self):
  pair_freq = {}
  for word, freq in self.vocab.items():
    symbols = word.split()

    for i in range(len(symbols) - 1):
      pair = (symbols[i], symbols[i + 1])
      if pair in pair_freq:
        pair_freq[pair] += freq
      else:
        pair_freq[pair] = freq
```

```
    return pair_freq
```

4) 기존 vocab 갱신

매개변수로 전달된 best_pair를 list형태로 만든 뒤 vocab에 연속된 문자가 존재하는 경우 이를 하나의 token으로 merge 한 다음 기존의 vocab와 merge rules를 갱신함.

```
  def merge_most_frequent_pair(self, best_pair):
    pattern = re.escape(' '.join(best_pair))
    replacement = ''.join(best_pair)
    target = list(best_pair)
    new_vocab = {}

    for word, freq in self.vocab.items():
      symbols = word.split()
      new_symbols = []
      i = 0

      while i < len(symbols):
        if i < len(symbols) - 1 and symbols[i] == target[0] and
symbols[i + 1] == target[1]:
          new_symbols.append(replacement)
          i += 2
        else:
          new_symbols.append(symbols[i])
          i += 1

      new_word = ' '.join(new_symbols)
      new_vocab[new_word] = freq

    self.vocab = new_vocab
    self.merge_rules.append(best_pair)
```

5) 위의 과정을 max_vocab size에 도달할 때까지 반복

다음의 함수는 main함수에서 호출하는 함수로서 하나만 호출하면 main 함수에서 간편하게 사용할 수 있음. 학습의 전체 흐름을 볼 수 있는 로직임.

초기에 preprocess_text()함수를 통해 전처리를 해서 Corpus 생성, 이를 build_vocab를 해서 basic vocab를 생성하고, merge rule의 수가 max_vocab size보다 작을 때까지 반복.

반복 과정에서 get_pair_frequency()를 통해 인접 pair의 빈도를 확인하고 최다 빈도를 구한 뒤 merge_most_frequent_pair()를 통해서 vocab에 존재하는 모든 단어에서 frequent pair를 찾아서 merge함.

```
  def train(self, corpus_path):
    corpus = self.preprocess_text(corpus_path)
    self.build_vocab(corpus)
    prev_vocab_size = len(self.vocab)

    initial_char_count = len(
      set(symbol for word in self.vocab for symbol in
word.split()))

    while len(self.merge_rules) + initial_char_count <
self.max_vocab_size:
      pair_freq = self.get_pair_frequency()
      if not pair_freq:
        break
      best_pair = max(pair_freq, key=pair_freq.get)
      self.merge_most_frequent_pair(best_pair)
```

6) vocab.txt에 최종 vocab와 merge rules를 저장

초기에는 merge rules와 vocab를 따로 저장하려고 하였으나, 요구사항에서 하나의 vocab.txt를 언급하여 vocab는 "==== Vocabulary ===="에 저장하였고, merge rules는 "==== Merge Ruls ===="에 저장하였음.

```
  def save_vocab_and_rules(self, vocab_path):
    with open(vocab_path, 'w', encoding='utf-8') as f:
      f.write("==== Vocabulary ====\n")
      for word, freq in self.vocab.items():
        f.write(f"{word} {freq}\n")

      f.write("\n==== Merge Rules ====\n")
      for pair in self.merge_rules:
        f.write(f"{pair[0]} {pair[1]}\n")
```

BPETokenizer

BPETrainer를 사용하여 학습한 merge rules를 바탕으로 새로운 텍스트 (infer.txt)를 Subword Tokenization하는 작업을 수행

1) vocab.txt에서 merge rules 불러오기

모든 줄을 리스트로 읽어온 다음 merge rules를 vocab.txt에서 찾은 뒤 각 줄을 순회하면서 parsing을 진행. 즉, merge rules를 불러들여서 class의 merge_rules에 저장한 것.

```
  def load_merge_rules(self, merge_path: str):
    merge_rules = []
    with open(merge_path, 'r', encoding='utf-8') as f:
      lines = f.readlines()

    in_merge_section = False
    for line in lines:
      line = line.strip()
      if line == "==== Merge Rules ====":
        in_merge_section = True
        continue

      if in_merge_section and line:
        parts = line.split()
        if len(parts) == 2:
          merge_rules.append((parts[0], parts[1]))
    return merge_rules
```

2) infer.txt를 읽고 전처리

3) merge rules를 하나씩 적용하여 순차적으로 merge

미리 학습된 merge rule이 있기 때문에 인접한 문자쌍이 merge rule과 같은 경우 merge를 진행. 또한, 첫 토큰을 제외한 이후의 토큰부터는 ##를 붙여서 반환함.

```
  def apply_BPE(self, word: str):
    tokens = list(word)

    for merge_rule in self.merge_rules:
      i = 0
      while i < len(tokens) - 1:
        if tokens[i] == merge_rule[0] and tokens[i + 1] ==
merge_rule[1]:
          tokens = tokens[:i] + [''.join(merge_rule)] + tokens[i +
2:]
          i = max(i - 1, 0)
        else:
          i += 1

    if not tokens:
      return []
    return [tokens[0]] + [f"##{t}" for t in tokens[1:]]
```

4) merge해서 만들어진 subword를 파일에 저

파일을 읽어서 전처리 과정을 거친 뒤 apply_BPE함수를 호출해서 이를 텍스트 파일을 Subword Tokenization을 적용한 결과를 저장하는 가장 메인이 되는 로직임

```
def tokenize_file(self, input_path, output_path):
  with (open(input_path, 'r', encoding='utf-8') as infile,
        open(output_path, 'w', encoding='utf-8') as outfile):
    for line in infile:
      words = line.strip().lower().split()
      tokenized_line = []

      for word in words:
        tokens = self.apply_BPE(word)
        tokenized_line.extend(tokens)

      outfile.write(' '.join(tokenized_line) + '\n')
```

4. 실험 및 결과

infer.txt(어린 왕자의 한 구절)가 다음과 같을 때 max_vocab 크기 변경에 따른 결과를 비교하였음.

```
Grown-ups never understand anything by themselves,

and it is tiresome for children to be always

and forever explaining things to them
```

예상되는 분할 Token은 다음과 같다.

```
Grown ##- ##ups ##never under ##stand any ##thing by them ##selves ##,

and it is ##tire ##some for children to be always

and for ##ever ##ex ##plain ##ing things to them
```

max_vocab == 100

```
학습 완료 소요 시간: 3.97초
```

```
전체 토큰 수: 67
전체 원래 단어 수: 21
평균 분할 수 (토큰 수 / 단어 수): 3.19
```

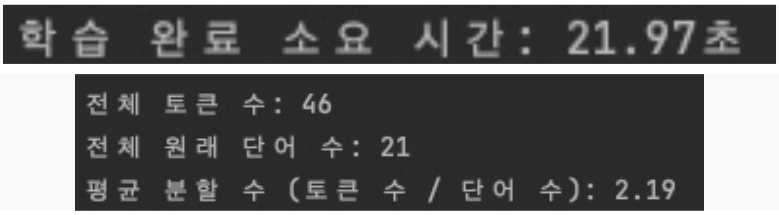
max_vocab를 가장 잘게 쪼갰을 때는 다음과 같이 의미를 알 수 없는 수준의 문자 수준의 token이 빈번히 나타남을 확인할 수 있음. 또한, 희귀 단어가 나타날 경우 유연하다는 특징을 지님.

g ##ro ##w ##n ##- ##u ##p ##s n ##e ##v ##er un ##d ##er ##st ##and an ##y ##th ##ing b ##y the ##m ##se ##l ##v ##es ##,

and it is t ##ir ##es ##o ##me for ch ##i ##ld ##r ##en to be al ##w ##ay ##s

and for ##e ##v ##er e ##x ##p ##la ##in ##ing th ##ing ##s to the ##m

max_vocab == 500



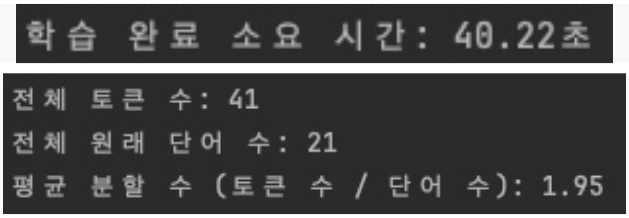
의미 있는 단어가 하나씩 생기기 나타남. 하지만, 아직 의미 있는 단어의 파악은 어려움. max_vocab을 100개로 한 것보다 토큰의 개수가 줄어들었으며, 일부가 병합되었음을 확인해 볼 수 있음.

g ##rown ##- ##up ##s never un ##der ##stand any ##thing by them ##sel ##ves ##,

and it is t ##ir ##es ##o ##me for ch ##i ##ld ##r ##en to be al ##way ##s

and fore ##ver ex ##pla ##in ##ing thing ##s to them

max_vocab == 1000



under 병합, stand 병합 등 의미 있는 수준에서 병합이 이루어짐을 확인할 수 있음. 비교적으로 적절한 수준의 단어 분할이 이루어졌음.

g ##rown ##- ##up ##s never under ##stand any ##thing by them ##selves ##,

and it is t ##ir ##es ##o ##me for child ##r ##en to be al ##way ##s

and fore ##ver ex ##pla ##in ##ing things to them

max_vocab == 2500

```
학습 완료 소요 시간: 87.91초
```

```
전체 토큰 수: 33
전체 원래 단어 수: 21
평균 분할 수 (토큰 수 / 단어 수): 1.57
```

대부분의 단어가 보존되었음. ex(understand, anything등) 시퀀스의 길이가 짧아졌음을 확인할 수 있음.

g ##rown ##- ##up ##s never understand anything by themselves ##,

and it is t ##ires ##o ##me for children to be al ##ways

and fore ##ver ex ##plain ##ing things to them

max_vocab == 5000

```
학습 완료 소요 시간: 164.41초
```

```
전체 토큰 수: 30
전체 원래 단어 수: 21
평균 분할 수 (토큰 수 / 단어 수): 1.43
```

거의 대부분의 단어가 단일 토큰으로 구성되어있으며, 문맥의 이해가 매우 쉬워졌음. 하지만, 처음 보는 단어가 나온다면 corpus에 없으면 분리가 어려워짐.

grown ##- ##up ##s never understand anything by themselves ##,

and it is t ##ires ##ome for children to be always

and fore ##ver ex ##plain ##ing things to them

max_vocab == 10000

```
학습 완료 소요 시간: 292.06초
```

```
전체 토큰 수: 29
전체 원래 단어 수: 21
평균 분할 수 (토큰 수 / 단어 수): 1.38
```

대부분의 단어가 의미가 있는 수준의 단일 토큰으로 구성되었음을 확인해 볼 수 있으며, 많은 메모리가 사용됨. 또한, 새로운 단어가 등장할 때 처리가 매우 어려워진다는 문제가 발생함.

```
grown ##- ##up ##s never understand anything by themselves ##,
and it is tires ##ome for children to be always
and fore ##ver ex ##plain ##ing things to them
```

5. 분석

실험 대상

텍스트(infer.txt): 어린왕자의 구절

Tokenizer: BPE를 이용한 Subword Tokenization

조작 변인: max_vocab size (100, 500, 1000, 2500, 5000, 10000)

평가 기준: 정량적 분석, 정성적 분석, 성능 분석

정량적 분석

max_vocab	총 토큰	평균 분할 (토큰 / 단어)
100	67	3.19
500	46	2.19
1000	41	1.95
2500	33	1.57
5000	30	1.43
10000	29	1.38

(평균 분할이 의미하는 것은 하나의 단어가 몇 개의 토큰으로 평균적으로 나눠졌는지를 나타내는 지표)

위의 결과를 통해서 max_vocab가 커질수록 전체 토큰의 수가 줄어들며, 평균 분할이 줄어들어 큰 단위로 토큰화를 했다는 것을 의미함.

max_vocab가 클 때

max_vocab가 커질수록 문장이 짧고 효율적으로 처리가 가능함. 또한, 의미 단위가 크기 때문에 문맥 파악이 쉬워진다는 장점을 가짐.

하지만, 새로운 단어가 등장하는 경우에는 처리가 어려울 수 있으며, vocab 사이즈가 충분히 커야 함. 따라서 학습하는 과정에서 데이터가 많이 필요하며, 따라서 메모리의 부담이 증가하고 학습하는 과정에서 시간이 오래 걸린다는 단점이 존재함.

max_vocab가 작을 때

max_vocab가 작을수록 작은 수의 subword로 대부분의 단어를 표현할 수 있으며, 새로운 단어가 있어도 조각을 통해서 의미를 유추해 낼 수 있음.

하지만, 하나의 문장이 너무 여러 개의 토큰으로 구성되며, 의미 단위가 너무 작아져서 문장의 전체 의미를 파악하기가 어려워진다는 단점이 존재함.

Word Coverage

이는 Subword Tokenizer가 주어진 단어를 얼마나 그대로 재현하였는지를 나타내는 비율임. 이를 통해서 단어들이 얼마나 단일 토큰으로 보존되었는지를 확인할 수 있음. word coverage가 높을수록 의미 단위로 잘 분리한 것이며, 문맥 파악이 쉬워짐. 하지만 너무 높아지면, 처음 확인한 단어를 마주하 처리하지 못한다는 단점이 존재한다.

max_vocab	Word Coverage
100	38.10%
500	52.38%
1000	57.14%
2500	71.43%
5000	76.19%
10000	76.19%

위의 표를 보면, 100과 100 사이에서는 약 60% 아래이지만, 2500을 넘어가면서부터 약 70%를 넘기는 것을 확인해 볼 수 있음. 또한, 5000을 넘어가면서 vocab의 size를 늘려도 더 이상 새로운 단어를 단일 토큰으로 보존할 수 없다. 즉, 5000이상에서는 메모리는 낭비되지만 성능은 동일한 결과를 초래함.

즉, Word Coverage가 커질수록 의미 단위가 더욱 보존됨을 확인할 수 있으며, 문맥 파악에 용이하다는 것을 확인할 수 있다. 하지만, Word Coverage가 줄어들수록 새로운 단어에 대응하는 것이 더욱 용이함

정성적 분석

분석의 기준은 다음과 같음

- 어색한 결합이 존재하는가
- 의미 단위로 잘 쪼개졌는가

max_vocab가 100, 500일 때는 완전히 의미를 이해하기 힘들 정도의 토큰임. 대부분의 경우에 어색한 결합이 존재하였음.
(ex_an ##y ##th ##ing b ##y the ##m ##se ##l ##v ##es)

max_vocab == 1000,2500: 의미 단위가 적절히 유지되며, 언어적으로 유의미한 단위로 나눠짐을 확인해 볼 수 있음. 또한, 적절한 결합이 존재하였으나, 여전히 어색한 결합이 존재함을 확인해볼 수 있음.
(ex_any ##thing by them ##selves)

max_vocab == 5000, 10000: 의미가 명확하며, 하나의 토큰이 하나의 의미를 지니는 경우가 많음. 또한, 대부분의 경우에 어색하지 않고 자연스러운 결합으로 구성되어있음.
(ex_anything by themselves)

성능 분석

시간 복잡도

다음은 각 max_vocab에 따른 실제 걸리는 시간을 코드를 실행해서 기록한 것임.

- max_vocab == 100: 3.97초
- max_vocab == 500: 21.97초
- max_vocab == 1000: 40.22초
- max_vocab == 2500: 87.91초
- max_vocab == 5000: 164.41초
- max_vocab == 10000: 292.02초

전체적으로 학습하는 데 걸리는 시간은 max_vocab size가 증가함에 따라서 학습 시간이 선형적으로 증가함. get_frequency() 함수와 merge_most_frequent_pair() 함수는 vocab 전체를 순회하기 때문에 max_vocab 값이 커질수록 시간이 급격히 증가함을 확인할 수 있다.

또한 merge_rule의 수가 많아질수록 이를 저장하고 tokenization을 진행할 때 (infer) 순차적으로 적용하기 때문에 추론 시간 성능이 저하된다. apply_BPE() 함수에서 볼 수 있듯 merge_rule을 순차적으로 텍스트에 적용하는 구조를 가지고 있기 때문에, tokenize 하는 데 걸리는 시간이 늘어남.

6. 결론 및 향후 나아갈 방향

실험 결론

본 보고서에는 BytePairEncoding(BPE) 알고리즘을 직접 구현하고 max_vocab의 크기를 달리하며 이에 따른 Subword Tokenization결과를 정량적, 정성적으로 평가하였음.

학습 코퍼스로는 셰익스피어 전집을 사용하였으며, 추론 데이터로는 어린 왕자의 한 구절을 활용하였음.

max_vocab가 증가함에 따라 평균 분할 수가 줄어들며, 전체 토큰의 수가 줄어듬을 확인할 수 있었고 이를 통해서 vocab가 커질수록 더 큰 의미단위의 subword로 표현이 가능하다는 것을 의미함. 정성적으로는 의미가 얼마나 유지되었는지를 기준으로 확인하였으며, 1000~2500 사이의 max_vocab에서 가장 적절하게 의미를 지니면서 새로운 text가 나타나도 잘 분석할 수 있을 것으로 판단되었음.

그러나 max_vocab가 너무 큰 경우에는 단어 하나하나가 하나의 개별 토큰으로 처리가 되기 때문에 단어의 의미를 파악하기는 쉬웠으나, 학습 및 추론 과정에서 시간복잡도가 증가하는 것을 확인할 수 있었음. 또한, 새로운 text를 마주했을 때 이를 제대로 subword 하지 못함.

현재 실험한 결과를 보면, max_vocab가 2500인 경우에 word coverage가 71.43% 로 준수한 의미단위로 분리된 것을 확인해볼 수 있으며, 학습 시간이 87초로 준수하다는 것을 확인할 수 있음. 또한, 새로운 단어가 등장하였을 때 적절히 대응할 수 있을 것으로 예상됨.

따라서 max_vocab를 2500으로 설정하는 것이 가장 효율적임.

향후 나아갈 방향

1. BPE 말고 다른 알고리즘을 사용해서 각 Tokenizer별 성능 비교 및 분석
2. 구축한 Subword Tokenization을 적용하여 딥러닝 모델을 직접 구축 또는 서비스에 적용
3. Out-of-Vocabulary(OOV)에 관해서 얼마나 잘 분리가 되는지를 정량적으로 평가

이상으로 BPE기반의 Subword Tokenization 알고리즘의 개요, 동작원리, 구현, 분석, 평가까지 다루었으며, 이를 통해서 max_vocab의 선택이 Tokenization에 있어서 핵심 요소임을 파악할 수 있었음.