과제1

PEP 8은 파이썬의 코딩 스타일 가이드입니다. 위 가이드를 따라서 다른 개발자 그룹과의 협업이나, 시간이 지난 후의 나의 코드를 읽는 등의 상황에서 일관된, 높은 가독성의 코드를 짜도록 유도해줍니다. 그러나 PEP 8의 모든 가이드를 강제적으로 항상 따르는 것이 아니라 프로젝트의 상황에 맞게 이를 일부 수정, 혹은 개선해서 따르는 것이 가장 현명합니다.

PEP8의 목차는 다음과 같습니다. Code Lay-out, String Quotes, Whitespace in Expressions and Statements, When to Use Trailing Commas, Comments, Naming Conventions, Programming Recommendations. 이중 Code Lay-out, Whitespace in Expressions and Statements, Comments, Naming Conventions, Programming Recommendations 파트의 일부분을 간략히 정리하겠습니다.

우선 Code Lay-out에서는 들여쓰기, 띄어쓰기를 가이드합니다. 들여쓰기는 공백 4칸, 한 줄은 최대 79자까지로 제한합니다. 함수와 클래스 정의는 2줄씩 띄어 쓰며, 클래스의 메소드 정의는 1줄씩 띄어 씁니다.

Whitespace in Expressions and Statements에서는 불필요한 띄어쓰기(e.g. argument, =, default value 사이의 공백이나 조건문의 수식 사이의 불필요한 공백)은 제거하도록 가이드합니다. [ ]와 ( ), 쉼표, 쌍점(:)과 쌍반점(;) 앞의 공백 또한 불필요한 띄어쓰기 입니다. Comments에서는 코드와 일치하는 필요한 주석만을 영어로 달 것을 권장합니다.

Naming Conventions에서는 다음 변수 명 규칙이 있습니다.

_single_leading_underscore: 이런 형태의 변수는 '내부적으로 사용되는 변수'입니다.

single_trailing_underscore_: 이 형태는 파이썬의 기본 키워드와 변수명이 충돌하는 것을 방지하기 위해 사용됩니다.

__double_leading_underscore: 이 형태는 클래스의 속성명을 변경하는데 사용됩니다. 이를 통해 클래스 속성의 이름 충돌을 방지할 수 있습니다.

double_leading_and_trailing_underscore: 이 형태는 파이썬의 '매직 메소드'를 정의할 때 사용됩니다. (e.g. __init__) 이러한 형태의 표기는 파이썬 문서에 설명된 대로만 사용해야 합니다.

Programming Recommendations의 예시는 다음과 같습니다. 1) None을 비교할 때는 is나 is not을 사용합니다. 2) 단어의 앞이나 뒤를 비교할 때는 startswith()와 endwith()을 사용합니다. 3) 객체의 타입을 비교할 때는 isinstance()를 사용합니다. 4) boolean을 조건문으로 사용할 때는 not사용이면 충분합니다. 5) improt는 파일의 맨 위에, 모듈과 패키지 간에 한 줄씩 띄워서 작성합니다. 6) string모듈보다는string메소드를 사용합니다.

---

1. 과제2

각 class를 구현할 때 고려한 부분

A. TextPreprocessor class: input type에 따라서 분기 구분하기, 기호와 숫자 제거하는 전처리 수행하기
B. Tokenizer class: bpe_tokenizer와 word_tokenizer에서 공통으로 사용되는 메서드 넣기, 하위 클래스에서 overwriting 되는 tokenize 메서드는 raise NotImplementedError 작성하기
C. BPETokenizer class: bpe_algorithm에 맞춰서 적절한 모듈화 진행하기, 가독성이 좋은 코드 작성하기
D. WordTokenizer class: bpe_tokenizer와 비슷하면서도 다른 알고리즘 작성하기

2. BPETokenizer class의 각 method의 구조와 작동 원리

A. compute_word_freqs(self) -> None : 각 단어마다의 frequency를 계산하는 함수.
구조는 for loop을 돌면서 self.word_freqs라는 dictionary에 {word : freq} 형태로 저장하는 방식. Word 안에 있는 letter도 for loop을 돌면서 self.alphabet에 저장한다. Self.splits에는 정상적인 word와 각 글자가 한 칸씩 띄워져있는 word(ex: 'w o r d')를 dictionary 형태로 저장한다.
B. get_stats(self) -> None: 각 pair마다 frequency를 계산하는 함수.
구조는 compute_word_freqs에서 구한 self.word_freqs를 이용해 for loop을 돌면서 pair의 frequency를 self.vocab라는 dictionary에 {pair : freq} 형태로 저장한다.
C. merge_vocab(self, pair: List[tuple[str, str]]) -> None: 인수로 들어온 pair에 대해 merge를 하고, merge로 인해 새롭게 생긴 pair를 self.vocab에 추가하는 함수
구조는 self.word_freqs를 for loop으로 돌면서, word의 길이가 1이거나 해당 pair가 word에 없는 경우는 넘긴다. 해당 pair가 word에 있는 경우에만 self.splits에서 merge를 수행한다. Merge를 수행한 다음에는 새롭게 만들어진 pair의 freq를 계산해야 되므로 앞, 뒤를 확인하고 pair의 freq를 계산하여 self.vocab를 업데이트한다.
D. train(self, n_iter: int) -> None: iteration을 돌면서 merge한 pair를 학습하는 함수. self.vocab에서 freq가 제일 높은 key 값을 찾고, 그 pair를 이용해서 merge를 한 다음 self.vocab를 업데이트하는 과정을 반복하는 구조이다.
E. tokenize(self, text: Union[List[str], str], padding: bool = False, max_length: Optional[int] = None) -> List[List[int]]: 새롭게 들어온 text에 대해 학습한 merge를 적용하여 tokenize하는 함수. 원래 train에서도 봤던 word가 있으면 그대로 사용하고, 없으면 merge 규칙과 하나씩 비교해 가면서 merge를 수행하는 구조이다. 이렇게 tokenize가 완료되었으면, token index로 바꾸는 과정을 거친다. 이때 paddding과 max_length에 따라 자르는 과정도 포함되어 있다.

3. 제시된 조건들을 충족시킨 방법

A. 상속 활용: BPETokenizer와 WordTokenizer에서 공통으로 활용 하는 전처리 관련 함

수, add_corpus, tokenize, __call__ 함수를 부모 class인 Tokenizer로 빼서 상속을 활용하였다.

B. 입력 타입에 따른 분기 처리: 맨 처음 들어가는 TextPreprocessor를 통해 type을 확인하고 나눠 똑 같은 list 형태를 반환하도록 만들었다.

C. 의도하지 않은 입력에 대한 에러처리: 맨 처음 들어가는 TextPreprocesso를 통해 list나 str type이 아니면 ValueError 메시지를 출력하도록 만들었다.

D. 알고리즘 변경: 논문 알고리즘에서는 train의 loop를 돌 때 처음부터 다시 모든 pair에 대해서 freq를 계산했지만, 여기서는 그럴 필요 없이 merge 할 때 새로 생긴 pair에 대해서만 freq를 추가하여 불필요한 시간 낭비를 제거하였다.

4. 협업한 방식

A. 박수연: TextPreprocessor, BPETokenizer class 코드 작성

B. 이승준: BPETokenizer class의 merge_vocab, train 함수 수정, WordTokenizer class 코드 작성