به نام خدا

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

درس پردازش زبان طبیعی

پاسخ تمرین شماره یک

نام و نام خانوادگی: امیرحسین شاه قلی

شماره دانشجویی: ۸۱۰۱۹۹۴۴۱

اسفند ماه ۱۴۰۲

Table of Contents

Answer of the question one

Answer of the first part:

```
import re
def custom_tokenizer(text):
    pattern = r'\b\w+\b'
    tokens = re.findall(pattern, text)
    return tokens
```

Above tokenizer is a Word-based tokenizer. The tokenizer uses a regular expression pattern r'\b\w+\b' to find word boundaries (\b) and match one or more word characters (\w+) and another word boundaries at the end of the token. This pattern seeks out sequences of alphanumeric characters (`[a-zA-Z0-9_]`) that are bounded by non-word characters like spaces, punctuation, or the start/end of the string.

Disadvantages: Punctuations like apostrophes in contractions or possessives are not handled properly; "doesn't" would be tokenized into "doesn" and "t,". Also the (\w) character class doesn't match all Unicode word characters, so the tokenizer may not work well with languages that use non-Latin characters; "سلام خوبی" would be tokenized into an empty set. Also the tokenizer doesn't handle case sensitivity issues. While the (\w) class matches both uppercase and lowercase characters; "getPost" would be tokenized into "getPost" which should be tokenized into "get" and "Post".

Answer of the second part:

Output Tokens: ['Just', 'received', 'my', 'M', 'Sc', 'diploma', 'today', 'on', '2024', '02', '10', 'Excited', 'to', 'embark', 'on', 'this', 'new', 'journey', 'of', 'knowledge', 'and', 'discovery', 'MScGraduate', 'EducationMatters']

As we can see (M.sc.) isn't tokenized properly and it should be considered as a one token. Also (2024/02/10!) which is a date isn't tokenized properly and it should be considered as a date in one token. (#) in tokens are ignored and it's sometimes better to consider them in tokens and they help us to classify sentences better.

Answer of the third part:

New pattern: r'\b[\w\.]+(?:/\d{2}){0,2}\b'

The new pattern is designed to match words, abbreviations with periods, hashtags, and dates in a specific format (yyyy/mm/dd).

Output Tokens: ['Just', 'received', 'my', 'M.Sc', 'diploma', 'today', 'on', '2024/02/10', 'Excited', 'to', 'embark', 'on', 'this', 'new', 'journey', 'of', 'knowledge', 'and', 'discovery', 'MScGraduate', 'EducationMatters']

Answer of the question two

Answer of the first part:

Both BERT tokenizer and GPT use subword-based tokenization. BERT uses WordPiece and GPT uses byte pair encoding (BPE) tokenizer.

LLM mostly uses subword-based tokenization for several reasons. subword-based tokenization decreases the chances of encountering words not seen during training and this type of tokenization helps models to do better on new data and handling out of vocabulary (OOV) words also languages with rich morphology can produce a large number of word forms, which would be difficult to cover with a word-level vocabulary. Subword-based tokenization helps to manage this complexity by breaking words into morphemes or other subword units and this helps tokenization to reduce the required vocabulary size to cover the language. Subword-based tokenization can achieve good encoding efficiency while capturing enough semantic information in each token and for generative models like GPT, subword tokenization allows the continuation of text to be handled at a granular level. This gives the model more flexibility and precision in text generation tasks.

Answer of the second part:

The key difference between **BPE** and **WordPiece** is in the way the symbol pairs are chosen for adding to vocabulary. **WordPiece** does not choose the most frequent symbol pair in but it chooses a symbol which maximizes the likelihood of the training data. Maximizing the likelihood of the training data is equivalent to finding the symbol pair, whose probability divided by the probabilities of its first symbol followed by its second symbol is the greatest among all symbol pairs while **BPE** focuses on merging the most frequent adjacent symbols to form a new symbol in the vocabulary.

Answer of the third part:

BPE Result: vocab size: 10000

“original sentence”: This darkness is absolutely killing! If we ever take this trip again, it must be about the time of the sNew Moon!

“tokens”: ['This', ' darkness', ' is', ' absolutely', ' kill', 'ing', '!', ' If', ' we', ' ever', ' take', ' this', ' trip', ' again', ',', ' it', ' must', ' be', ' about', ' the', ' time', ' of', ' the', ' s', 'New', ' Moon', '!']

“original sentence”: This is a tokenization task. Tokenization is the first step in a NLP pipeline. We will be comparing the tokens generated by each tokenization model.

“tokens”: ['This', ' is', ' a', ' to', 'ken', 'ization', ' task', '.', ' T', 'ok', 'en', 'ization', ' is', ' the', ' first', ' step', ' in', ' a', ' N', 'L', 'P', ' pip', 'el', 'ine', '.', ' We', ' will', ' be', ' comp', 'aring', ' the', ' to', 'k', 'ens', ' gener', 'ated', ' by', ' each', ' to', 'ken', 'ization', ' mod', 'el', '.']

WordPiece Result: vocab size: 10000

“original sentence”: This darkness is absolutely killing! If we ever take this trip again, it must be about the time of the sNew Moon!

“tokens”: ['Th', '##i', '##s', 'd', '##a', '##r', '##k', '##n', '##e', '##s', '##s', 'is', 'absolut', '##e', '##ly', 'killing', '!', 'If', 'w', '##e', 'ev', '##e', '##r', 'tak', '##e', 'th', '##i', '##s', 't', '##r', '##i', '##p', 'a', '##g', '##a', '##i', '##n', ',', 'it', 'must', 'b', '##e', 'about', 'th', '##e', 't', '##i', '##m', '##e', 'of', 'th', '##e', 's', '##N', '##e', '##w', 'Moon', '!']

“original sentence”: This is a tokenization task. Tokenization is the first step in a NLP pipeline. We will be comparing the tokens generated by each tokenization model.

“tokens”: ['Th', '##i', '##s', 'is', 'a', 'to', '##k', '##e', '##niz', '##ation', 't', '##a', '##s', '##k', '[UNK]', 'Tok', '##e', '##niz', '##ation', 'is', 'th', '##e', 'f', '##i', '##r', '##s', '##t', 's', '##t', '##e', '##p', 'in', 'a', 'N', '##L', '##P', 'p', '##i', '##p', '##e', '##l', '##i', '##n', '##e', '[UNK]', 'W', '##e', 'w', '##i', '##l', '##l', 'b', '##e', 'comparing', 'th', '##e', 'to', '##k', '##e', '##n', '##s', 'g', '##e', '##n', '##e', '##r', '##a', '##t', '##e', '##d', 'by', 'each', 'to', '##k', '##e', '##niz', '##ation', 'mod', '##e', '##l', '[UNK]']

Conclusion:

BPE splits words into subword units based on the most frequent byte pairs in the corpus. It aims to find a balance between character-level and word-level tokenization, resulting in tokens like 'kill' and 'ing' in the first sentence, and 'to', 'ken', 'ization' in the second sentence. This approach can handle rare or OOV words by breaking them down into known subword units, WordPiece operates similarly but focuses on creating a more efficient vocabulary by iteratively merging the most frequently co-occurring character sequences and iit produces tokens with '##' to indicate continuation subword units, such as '##a', '##r', '##k', in the first sentence, and '##k', '##e', '##niz', '##ation' in the second sentence.

(Notice: Both algorithms vocabulary size set to 10000)

BPE source code

WordPiece source code

Answer of the question three

Answer of the first part:

```
import re

import nltk

from nltk import word_tokenize

from nltk.util import ngrams

with open("Tarzan.txt", "r", encoding="utf-8") as file:

   corpus = file.read()

clean_text = re.sub(r"[^\w\s]", "", corpus).replace('\ufeff', '')

token = word_tokenize(clean_text, language='english')
```

Answer of the second part:

Laplace smoothing solves data sparsity by adding a small positive value (typically 1) (in this project we use 0.01) to the count of each n-gram in the training data before the probabilities are calculated. This ensures that no n-gram has a zero probability, allowing the model to generate predictions even for unseen n-grams.

The rest of the steps in the file part_3_ph2.ipynb

Answer of the third part:

original: knowing well the windings of the trail he

Generated sentence (with random choice from top 10): knowing well the windings of the trail he could do so much excited was to be no difficulty

Generated sentence: knowing well the windings of the trail he fear defiantly Behind let safety decided goat Shes prone ascended

---

original: For half a day he lolled on the huge back and

Generated sentence (with random choice from top 10): for half a day he lolled on the huge back and so they saw his sword was to them that

Generated sentence: for half a day he lolled on the huge back and it was gone battle space hat jest board stout patch

Answer of the fourth part:

3-gram:

original: knowing well the windings of the trail he

Generated sentence: knowing well the windings of the trail he took himself quite seriously and would be victorious and the

5-gram:

original: knowing well the windings of the trail he

Generated sentence: knowing well the windings of the trail he took short cuts swinging through the branches of the trees

3-gram:

original: For half a day he lolled on the huge back and

Generated sentence: For half a day he lolled on the huge back and join me but that of Gobred these were champions whose

5-gram:

original: For half a day he lolled on the huge back and

Generated sentence: For half a day he lolled on the huge back and overtake you without much loss of time If you catch

(**notice:** for 3-gram we used add-1 for smoothing and also for 5-gram, in 5-gram we consider 4-gram and 3-gram for next word if 5-gram choices be empty)

Conclusion:

As we can see in the results, sentences of 3-gram and 5-gram make more sense to be valid sentences and it's not like random words together like bigrams sentences but these sentences are still meaningless.

Answer of the fifth part:

Technically we can increase N but increasing N not always leads to better sentences, also increasing N have some disadvantages like computational complexity and also as the value of N increases, the number of possible n-grams becomes exponentially larger and this leads to data sparsity, also using high n-grams can lead to overfitting and its consider a larger context, which can make it difficult to find local dependencies between words so with these problems we **can not** increase N without limit (this will be very costful).

Answer of the question four

Answer of the first part:

Function Implementation:

```
def test_ngram(data, positive_freq, negative_freq, n):

   pred_labels = []

   vocab_size = len(set(positive_freq.keys()).union(set(negative_freq.keys())))

   log_pos_denominator = math.log(sum(positive_freq.values()) + vocab_size)

   log_neg_denominator = math.log(sum(negative_freq.values()) + vocab_size)

   for _, row in data.iterrows():

       grams = get_ngrams(row['review'], n)

       log_positive_prob = 0

       log_negative_prob = 0

       review_counter = Counter(grams)

       for gram, _ in review_counter.items():

           if gram in positive_freq or gram in negative_freq:

               log_positive_prob += math.log(positive_freq.get(gram, 0) + 1) -
log_pos_denominator

               log_negative_prob += math.log(negative_freq.get(gram, 0) + 1) -
log_neg_denominator
```

```
        if log_positive_prob >= log_negative_prob:
            pred_labels.append(1)
        else:
            pred_labels.append(0)
    return pred_labels
```

The function calculates log probabilities for each review's n-grams by checking their occurrence in the positive and negative frequency dictionaries, applying Laplace smoothing to avoid zero probabilities. It sums these log probabilities to get a total log probability score for each positive and negative class for every review and then compares the scores for the positive and negative classes and assigns a label based on which score is higher.

Answer of the second part:

Accuracy: 0.8547486033519553

Precision: 0.8461538461538461

Recall: 0.6226415094339622

F1 Score: 0.717391304347826