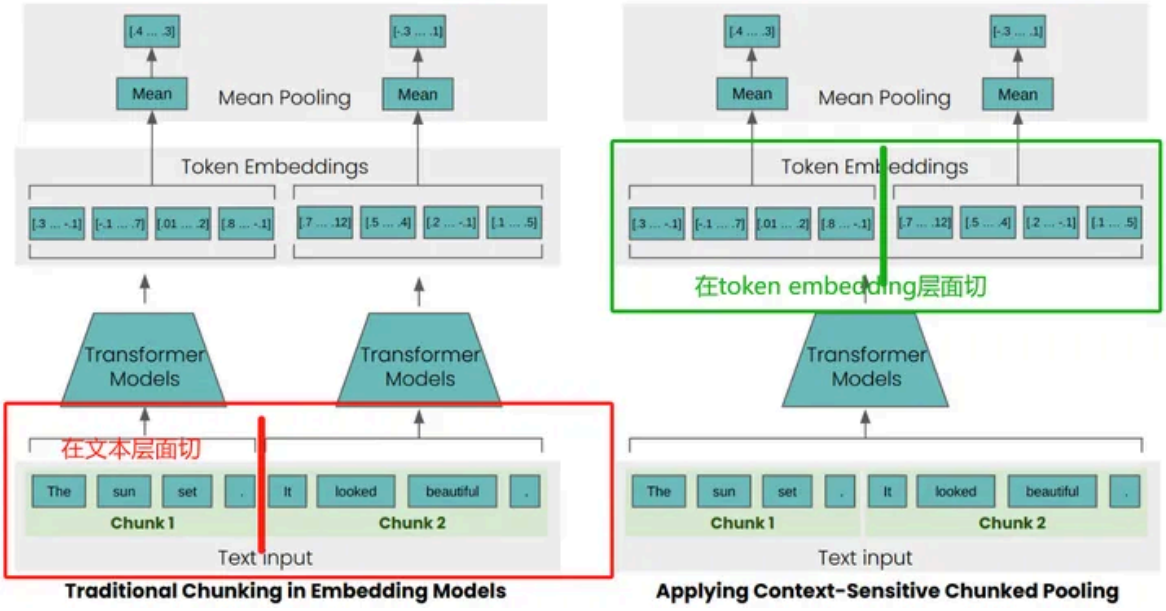
【RAG】文本切块新方法Late Chunking，原理及中文实践

原创 战士金 炼钢AI 2024年09月17日 14:08 北京

① 原理

感觉很久没有让人眼前一亮的RAG相关技术出现了。Jina AI提出的Late Chunking原理简单，但能大大缓解RAG文本切块过程中的语义割裂问题。在常规的RAG文本切块过程中，为了能够让某个文本块保留一些其相邻文本块的信息，最常规的做法就是在切块过程中让相邻文本块之间保留一些重叠的部分。更激进一些的，还有父文本检索策略，如果召回了一小段文本，那么就把这小段文本所属的那一大段文本都塞进LLM的上下文中。但是这些都更倾向于“规则”层面的优化方法，很少有在“算法”层面的技术。Late Chunking就是一种在算法层面增强文本块上下文感知的技术。

常规的RAG文本切块如左下图所示，先进行文本切分，分别过Transformer处理，通过meaning pooling得到每个chunk的句向量。假设有如下这段话：“战士金是个程序员。他最近在玩黑神话悟空。”如果按照句号切分这段话，并分别过transformer处理，模型是并不能感知到第二个chunk里边的“他”是指的谁的。但如果先将这一整段话经过transformer处理，生成token embeddings，再在句号位置处切分，Transformer的self attention机制是能够学习到第二个chunk的“他”指的是第一个chunk里边的“战士金”。这种先将文本变成embedding，再进行切分的策略就是Late Chunking。听起来很简单吧！

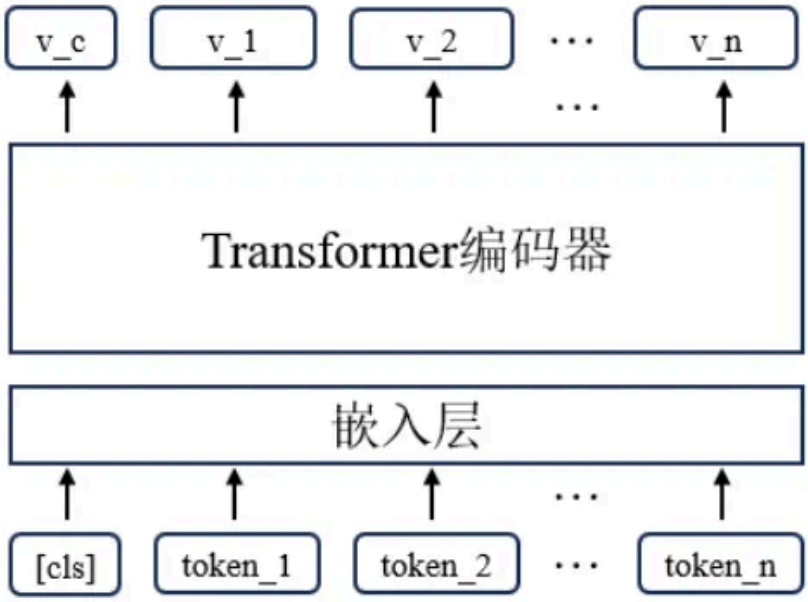


现在我们来思考一个问题，为啥明明原理这么简单的技术之前就没啥人提呢？或者说有人提（只是笔者不太清楚）但使用也不广泛呢？

（1）句向量表征方式的问题

transformer最终输出的是每个token的embedding，如下所示。如果想用一个embedding对整个句子进行表征，主要有两大流派（Transformer encoder架构），一派是对所有token的embedding取平均；另一派是用[CLS]位置的token embedding表示句子，因为大部分文本向量化模型是双向注意力的，[CLS]位置的embedding能感知到全局文本信息。某一个模型到底是用CLS embedding表示句子，还是用所有token embedding的平均值表示句子是由训练时候决定，并不能随意切换。如果一个模型在训练时用CLS embedding

表示句子，那么推理时用所有token embedding平均表示句子效果会非常差。目前使用CLS embedding表示句向量的模型占大多数，比如被大家广泛使用的BGE、BCE模型等。只有少部分模型，比如GTE系列的个别模型、JINA的模型等才用mean pooling token embedding的方式表征句子。而late chunking技术只有那些用mean pooling token embeding表征句子的模型才能用.



（2）文本向量化模型支持长度的问题

输入的序列越长，在做句子表征时候信息损失的越多。并且大家还都会对文本进行提前chunk，因此文本向量模型通常只支持几百长度的文本（只要显存足够，你当然可以输入几万长度的文本，但是效果就很差了）。支持长文本向量化的模型不多（训练时就需要用长文本），笔者只知道Jina的模型开源模型支持8192长度，openai的闭源模型支持8192的长度。Late Chunking技术需要先对文本进行embedding，再进行分段，因此比常规的文本向量化需要模型有更强的长文本处理能力。

总结一些，首先，Late chunking需要用mean pooling的方式表征句子，其次，还需要模型支持长文本的输入，这两个条件Jina AI的模型都满足。Jina AI中文模型链接如下：

```
https://hf-mirror.com/jinaai/jina-embeddings-v2-base-zh
```

② 实践

Late Chunking有官方的代码库，链接如下：

```
https://github.com/jina-ai/late-chunking/tree/main
```

这篇文章涉及的中文late chunking完整代码在如下仓库：

```
https://github.com/zhanshijinwat/Late-Chunking-Chinese
```

我想在中文上边做一些late-chunking实践尝试，需要在官方代码的基础上做一些改进，首先找到了代码里的如下函数实现，主要功能是将一段话按照"."切分，然后返回切分后的文本列表chunks，以及在token粒度下的每个chunk的索引范围（span_annotations）。

```
def chunk_by_sentences(input_text: str, tokenizer: callable):
    """
    Split the input text into sentences using the tokenizer
    :param input_text: The text snippet to split into sentences
    :param tokenizer: The tokenizer to use
    :return: A tuple containing the list of text chunks and their correspondi
    """
    inputs = tokenizer(input_text, return_tensors='pt', return_offsets_mappir
    punctuation_mark_id = tokenizer.convert_tokens_to_ids('.')
    sep_id = tokenizer.convert_tokens_to_ids('[SEP]')
    token_offsets = inputs['offset_mapping'][0]
    token_ids = inputs['input_ids'][0]
    chunk_positions = [
        (i, int(start + 1))
        for i, (token_id, (start, end)) in enumerate(zip(token_ids, token_off
        if token_id == punctuation_mark_id
        and (
            token_offsets[i + 1][0] - token_offsets[i][1] > 0
            or token_ids[i + 1] == sep_id
        )
    ]
    chunks = [
        input_text[x[1] : y[1]]
        for x, y in zip([(1, 0)] + chunk_positions[:-1], chunk_positions)
    ]
    span_annotations = [
        (x[0], y[0]) for (x, y) in zip([(1, 0)] + chunk_positions[:-1], chunk
    ]
    return chunks, span_annotations
```

第一反应，嘿那我直接把“punctuation_mark_id = tokenizer.convert_tokens_to_ids('.')”这个里边的分割符从"."改成"。"不就能分割中文了么，简单的很。改完之后发现chunks是一个空列表，失败。这代码看着乱七八糟，还不如自己重新花几分钟写个实现。基本逻辑如下代码所示，在文本粒度按照“。”切分后的文本片段分别转换成token粒度，每一块的token数量求和不就是完整文本的token数量了么？

```
def chunk_by_sentences_chinese(sentence, tokenizer, split_token = "。"):
    split_id = tokenizer.convert_tokens_to_ids(split_token)
    text_chunks = sentence.split(split_token)
    print(text_chunks)
    for i, chunk in enumerate(text_chunks):
```

```
        if i == len(text_chunks)-1:
            if sentence[-1]=="。":
                text_chunks[i] += "。"
        else:
            text_chunks[i] = text_chunks[i]+"。"
    # print(text_chunks)
    text_chunks = [x for x in text_chunks if x!="。"]
    id_chunks = tokenizer(text_chunks)
    start = 0
    end = -1
    span_indexs = []
    for id_chunk in id_chunks.input_ids:
        end = start + len(id_chunk)
        span_indexs.append((start, end))
        start = end
    return text_chunks, span_indexs
```

事情并没有这么简单。我用“**战士金的新书已经出版了。他的新书名字是大模型RAG实战。这本书由机械工业出版社出版。可以在京东上购买。**”这句话当作测试用例跑了一下代码。这段话按照“。”切块后分别转换成token，将　每块的token数量求和，得到token数是27，但是直接对完整的这段话转换成token的数量是29。这是为啥呢？tokenizer将哪些文本映射到哪个token id是根据训练数据统计出来的，尽量将那些出现频率高的长文本子串映射为一个token id。举个例子，假设tokenizer学出来“出版了。他”对应一个token id，但是如果将这句话按照“。”切分成["出版了。", "他"]，那么至少需要2个token id表示。

只能去看看他的代码逻辑了。官方代码里在调用tokenizer时候（inputs = tokenizer(input_text, return_tensors='pt', return_offsets_mapping=True)）用了一个不太常见的参数，return_offsets_mapping，如果设置为True，那么就会返回每个token对应的的文本索引范围，有了这个信息就能准确的找到一段token对应的文本了。但是现在还是没解决为啥我直接把官方代码的分割符从"."换成"。"之后没法输出分割结果的问题。。。其实问题出现在了分割文本时需要满足如下这个判断条件：当我们找到分割符的token位置后，需要他下一个token对应的文本起始位置大于分割符对应的文本结束位置。

```
token_offsets[i + 1][0] - token_offsets[i][1] > 0
```

在英文里"."**后边会接一个空格，但是在中文里"。"后边是不接空格的。**分割符后边是否有空格会导致offsets_mapping返回的文本范围有所区别。举个如下：

```
test = ".hi"
test_id = tokenizer(test, return_tensors='pt', return_offsets_mapping=True)
print(test_id)
# output:
{'input_ids': tensor([[    0,    44, 23233,     2]]), 'attention_mask': tensc
          [0, 1],
          [1, 3],
```

```
          [0, 0]]])}
#================================================================
test = ". hi"
test_id = tokenizer(test, return_tensors='pt', return_offsets_mapping=True)
print(test_id)
# output:
{'input_ids': tensor([[    0,    44, 23233,     2]]), 'attention_mask': tensc
          [0, 1],
          [2, 4],
          [0, 0]]])}
```

英文和中文语言格式上的区别需要我们简单修改一下代码，将"token_offsets[i + 1][0] - token_offsets[i][1] > 0"改为"token_offsets[i + 1][0] - token_offsets[i][1] >= 0"。至此，我们就可以对中文使用late-chunking了。后面我们先对完整的文本段落转为embedding序列，然后调用chunked_pooling找到每个chunk的token索引范围，分别做mean pooling得到每个文本chunk的向量表示。

```
def chunked_pooling(
    model_output, span_annotation: list, max_length=None
):
    token_embeddings = model_output[0]
    outputs = []
    for embeddings, annotations in zip(token_embeddings, span_annotation):
        if (
            max_length is not None
        ):  # remove annotations which go bejond the max-length of the model
            annotations = [
                (start, min(end, max_length - 1))
                for (start, end) in annotations
                if start < (max_length - 1)
            ]
        pooled_embeddings = [
            embeddings[start:end].sum(dim=0) / (end - start)
            for start, end in annotations
            if (end - start) >= 1
        ]
        pooled_embeddings = [
            embedding.detach().cpu().numpy() for embedding in pooled_embeddir
        ]
        outputs.append(pooled_embeddings)

    return outputs
```

最后我们来做个测试，case还是：“战士金的新书已经出版了。他的新书名字是大模型RAG实战。这本书由机械工业出版社出版。可以在京东上购买。”，用”。“当作分割符，query为”战士金的新书叫什么“，使用传统chunk方法，和late chunking方法的得分分别如下。正确答案包含在第2个chunk中，但是这个chunk并不包含关键字”战士金“，因此使用原始chunk方法时和query的相似度只有0.526。但是如果使用late chunking，是能够感知到”他“指的是第一个chunk中的战士金的，因此相似度score能增长到0.728。

```
native chunk score("战士金的新书叫什么", "战士金的新书已经出版了。"): 0.838993
late chunking score("战士金的新书叫什么", "战士金的新书已经出版了。"): 0.9393679
===
native chunk score("战士金的新书叫什么", "他的新书名字是大模型RAG实战。"): 0.72893
late chunking score("战士金的新书叫什么", "他的新书名字是大模型RAG实战。"): 0.5264
===
native chunk score("战士金的新书叫什么", "这本书由机械工业出版社出版。"): 0.713182
late chunking score("战士金的新书叫什么", "这本书由机械工业出版社出版。"): 0.30559
===
native chunk score("战士金的新书叫什么", "可以在京东上购买。"): 0.6868561
late chunking score("战士金的新书叫什么", "可以在京东上购买。"): 0.12445604
===
```

最后放个小彩蛋，我们是真的有新书出版了，名字也真的是《大模型RAG实战》，按照出版社的节奏还没到宣发阶段，先偷跑放个链接。祝大家中秋快乐~