自动文本摘要

19351024 林威

1. 背景介绍

文本摘要旨在将文本或文本集合转换为包含关键信息的简短文本。近年来，互联网中的信息呈井喷式喷发，如何从中快速有效地获取所需信息显得极为重要。自动文本摘要技术的出现可以有效缓解该问题，利用计算机自动地从长文本或文本集合中提炼一段能准确反映原文中心内容的简洁连贯的短文。

自动文摘的形式化定义：设$D=\{w_1,w_2,\ldots,w_n\}$为包含n个单词的原始文档，自动文摘的目标是得到一个由单词y_i组成的包含原文中心内容的摘要$Y=\{y_1,y_2,\ldots,y_m\}$且需要$n>>m$.

2. 数据集

本次作业的训练集包含1789508个数据，测试集包含3000个数据。每个数据包含title和content两块，下面展示了一条数据：

```
{
 "title": "通俗小说之王--大仲马",
 "content": "他的小说大多以真实的历史作背景，情节曲折生动，往往出人意料，被后人誉为“通俗小说之王”。他的代表作《三个火枪手》、《基督山伯爵》至今影响深远。但是他曾说，自己最得意的作品是儿子小仲马。1802的今天，小说之王大仲马出生。缅怀！[蜡烛] "
}
```

3. 评价指标

本次作业采用了4个评价指标，分别是Rouge-1，Rouge-2，Rouge-L以及BLEU。

其中Rouge将模型产生的系统摘要和参考摘要进行对比，通过计算它们之间的重叠的基本单元数目来评价系统摘要的质量。Rouge有很多类别，其后面的数字分别代表基于1元词，2元词和最长公共子序列（longest common subsequence, LCS）

基于$n-gram$的Rouge可以用下式来计算：

$$Rouge_n=\frac{\sum_{sen\in S}\sum_{gram_n\in sen}Count_{match}(gram_n)}{\sum_{sen\in S}\sum_{gram_n\in sen}Count(gram_n)}$$

其中S是参考摘要，sen是句子。这个式子中，分母是n-gram的个数，分子是参考摘要和自动摘要共有的n-gram的个数。

Rouge-L可以用下式来计算：

$$R_{LCS}=\frac{LCS(C,S)}{len(S)},\ P_{LCS}=\frac{LCS(C,S)}{len(C)},\ F_{LCS}=\frac{(1+\beta^2)R_{LCS}P_{LCS}}{R_{LCS}+\beta^2P_{LCS}}$$

其中C表示系统摘要，S表示参考摘要，R_{LCS}表示召回率，P_{LCS}表示精确率，F_{LCS}表示Rouge-L。一般β会被设置为很大的数，因此F_{LCS}几乎只考虑了召回率。

BLEU是一种基于精确度的相似性度量方法，用于分析候选摘要和参考摘要中n元组共同出现的程度。其表达式如下：

$$BLEU = BP \times exp(\sum_{n=1}^{N} W_n \times logP_n)$$

$$BP == \begin{cases} 1, & lc > lr \\ exp(1 - lr/lc), & lc \leq lr \end{cases}$$

$$P_n = \frac{\sum_{i \in n\_gram} min(h_i(C), max_{j \in m} h_i(S_j))}{\sum_{i \in n\_gram} h_i(C)}$$

其中，BP是惩罚因子，lc是候选摘要的长度，lr是最短的参考摘要的长度，W_n是n_gram 的权重，一般设置为均匀权重，即对于任意n都有$W_n = \frac{1}{n}$，P_n候选摘要中n元组的精确度，$h_i(C)$表示n_gram第i个词组在候选摘要中出现的次数，$h_i(S_j)$表示n_gram第i个词组在第j个参考摘要中出现的次数。

以上参考指标都可以在python中调用rouge库和nltk.translate.bleu_score库来计算。

4. 方法

自动文摘方法可以分为抽取式和生成式两类。抽取式从原始文档中提取关键文本单元来组成摘要。其二者各自的优缺点如下：

	优点	缺点
抽取式	会保留源文章的显著信息，有着正确的语法	产生的摘要往往不够简洁；容易产生大量的冗余信息；连贯性上无法得到很好的保证 对于短文本摘要不太友好。
生成式	模型试图去理解文本的内容，可以生成原文中没有的单词； 更加接近摘要的本质，具有生成高质量摘要的潜力。	需要利用大量的训练数据训练模型，训练数据的质量决定了模型性能的峰值 ； 训练过程普遍耗时较长 部分重要的模型参数需要人工设置、优化

在本次作业中，我采用了两种方法来解决自动文摘的问题。第一个是基于TF-IDF的文摘方法，它属于抽取式方法。第二个是基于BART的方法，它属于生成式方法。

（一）基于TF-IDF的文摘方法

该方法实际上是通过关键词信息量、句子位置和句子相似度三个权重指数来计算句子的权重，然后抽取句子权重最高的句子作为摘要。

先来介绍一下TF-IDF值，它是用来衡量一个词的重要性的。首先TF表示词频，可以由下式来计算：

$$TF = \frac{\text{某个词在文章中出现的次数}}{\text{文章的总词数}}$$

IDF则表示逆文档频率，可以由下式来计算：

$$IDF = log(\frac{\text{语料库的文档总数}}{\text{包含该词的文档数} + 1})$$

一个词的TF-IDF值则可以由$TF - IDF = TF \times IDF$来计算。

接着我们就可以来计算句子的关键词信息量了，它实际上就是该句子中的词的TF-IDF值得总和。

位置权重在这里我设置的是$1 - \frac{\text{句子的位置}}{\text{文章中句子总数}}$,即认为句子出现的位置越前，句子越重要。

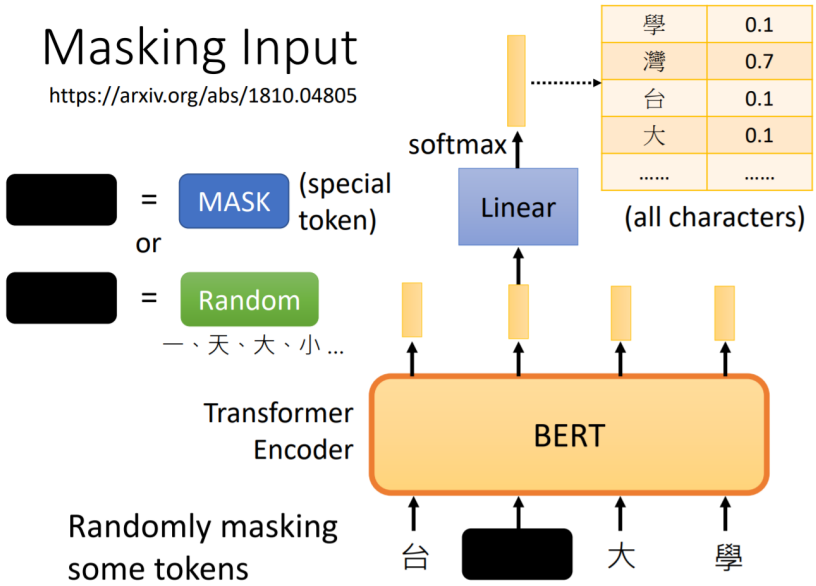
两个句子的相似度可以由余弦相似度来计算。某个句子的相似度是该句子与其他句子的相似度的总和。如果相似度越高，则句子越重要。

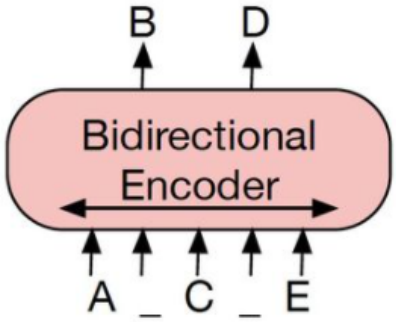
计算完这三个权重指数之后，把它们加权相加，然后作为该句子的重要性。取句子重要性最高的前n个句子作为文章的摘要。其中n是个可调节参数。

（二）Bart Fine-tune

在文本理解任务，预训练模型已经取得了质的飞跃，语言模型预训练+下游任务fine-tune基本上已经成为标配。很多人曾经尝试将BERT等预训练语言模型应用于文本生成任务，然而结果并不理想。究其原因，是在于预训练阶段和下游任务阶段的差异。BART这篇文章提出的是一种符合生成任务的预训练方法，BART的全称是**B**idirectional and **A**uto-**R**egressive **T**ransformers，顾名思义，就是兼具上下文语境信息和自回归特性的Transformer。BART 是一个适用于序列到序列模型的去噪自编码器，可应用于大量终端任务。预训练包括两个阶段：1）使用任意噪声函数破坏文本；2）学得序列到序列模型来重建原始文本。BART 使用基于 Transformer 的标准神经机器翻译架构，可泛化 BERT、GPT 等近期提出的预训练模型。

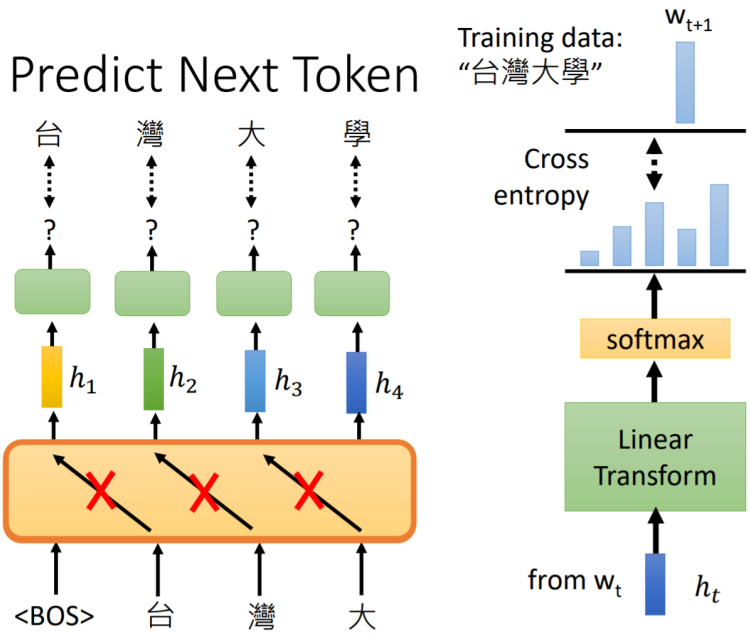
Bert是一种Auto-Encoding的语言模型。它也可以看作是Transformer model的Encoder部分。Bert属于自监督模型，即意味着它的训练集当中并没有label。下图（左）显示了Bert的预训练过程：在输入端输入一段文本，并且随机地给文本制造一些噪声，如用一种特殊的Mask token来随机地替换文本中的字。然后我们将该Mask对应地输出扔进一个MLP当中，最后再过一个Softmax层，我们以本来被mask替换掉地那个字作为label，对softmax层出来地预测结果进行学习。Bert的优点是能获取双向上下文信息。但缺点也很明显，它引入了fine-tune阶段不会出现的mask，而且它也对序列的联合概率做了独立性假设。

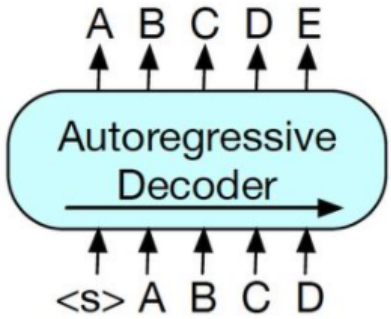




(a) BERT: Random tokens are replaced with masks, and the document is encoded bidirectionally. Missing tokens are predicted independently, so BERT cannot easily be used for generation.

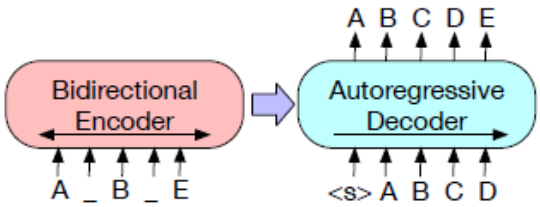
GPT是一种自回归的语言模型。它也可以看作是Transformer model的Decoder部分。由下图可见，GPT的预训练模式实际上就是输入一段文本，然后预测下一个字。因此GPT具有生成文本的能力。GPT的优点是它对序列的联合概率并没有做任何假设。它的缺点是它是从左到右单向的一个建模方式，并没有获取双向上下文信息。





(b) GPT: Tokens are predicted auto-regressively, meaning GPT can be used for generation. However words can only condition on leftward context, so it cannot learn bidirectional interactions.

Bart吸收了BERT的bidirectional encoder和GPT的left-to-right decoder各自的特点，建立在标准的seq2seq Transformer model的基础之上，这使得它比BERT更适合文本生成的场景；相比GPT，也多了双向上下文语境信息。在生成任务上获得进步的同时，它也可以在一些文本理解类任务上取得SOTA。



(c) BART: Inputs to the encoder need not be aligned with decoder outputs, allowing arbitary noise transformations. Here, a document has been corrupted by replacing spans of text with mask symbols. The corrupted document (left) is encoded with a bidirectional model, and then the likelihood of the original document (right) is calculated with an autoregressive decoder. For fine-tuning, an uncorrupted document is input to both the encoder and decoder, and we use representations from the final hidden state of the decoder.

Bart的Loss Function就是reconstruction loss, 也就是decoder的输出和原文ground truth之间的cross entropy。如上图所示，Bart的结构实际上就是一个bert+GPT，但有一点不同之处在于，相对于bert中单一的mask token的训练模式，bart在encoder端尝试了多种"毁坏"原本文本的方式。具体有以下几种模式：

- Token Masking：tokens被随机选取并替换为mask。

- Token Deletion Random： 输入中的tokens被随机地删除。
- Text Infilling: 随机将一段连续的token替换成一个[MASK]，span的长度服从 $\lambda = 3$ 的泊松分布。
- Sentence Permutation: 将一段文本的字的顺序打乱。
- Document Rotation： 从文本序列中随机选择一个token，然后使得该token作为文本的开头。

自动摘要可以被认为是bart的一个下游任务。我们只需要加载预训练的bart模型，然后在此基础上根据我们的训练集微调一下，即可获得可以生成自动摘要的模型。

5. 具体实现

（一）基于TF-IDF的文摘方法

先计算TF-IDF值的矩阵：

```
def get_tfidf_matrix(sentence_set,stop_word):
    corpus = []
    for sent in sentence_set:
        sent_cut = jieba.cut(sent)
        sent_list = [word for word in sent_cut if word not in stop_word]
        sent_str = ' '.join(sent_list)
        corpus.append(sent_str)

    vectorizer=CountVectorizer()
    transformer=TfidfTransformer()
    tfidf=transformer.fit_transform(vectorizer.fit_transform(corpus))
    tfidf_matrix=tfidf.toarray()
    return np.array(tfidf_matrix)
```

再计算句子的关键词信息量：

```
def get_sentence_with_words_weight(tfidf_matrix):
    sentence_with_words_weight = {}
    for i in range(len(tfidf_matrix)):
        sentence_with_words_weight[i] = np.sum(tfidf_matrix[i])

    max_weight = max(sentence_with_words_weight.values()) #归一化
    min_weight = min(sentence_with_words_weight.values())
    for key in sentence_with_words_weight.keys():
        x = sentence_with_words_weight[key]
        sentence_with_words_weight[key] = (x-min_weight)/(max_weight-
min_weight)

    return sentence_with_words_weight
```

以及位置权重：

```
def get_sentence_with_position_weight(sentence_set):
    sentence_with_position_weight = {}
    total_sent = len(sentence_set)
    for i in range(total_sent):
        sentence_with_position_weight[i] = (total_sent - i) / total_sent
    return sentence_with_position_weight
```

以及句子相似度权重：

```
def similarity(sent1,sent2):
    return np.sum(sent1 * sent2) / 1e-6+(np.sqrt(np.sum(sent1 * sent1)) *\
                                    np.sqrt(np.sum(sent2 * sent2)))

def get_similarity_weight(tfidf_matrix):
    sentence_score = collections.defaultdict(lambda :0.)
    for i in range(len(tfidf_matrix)):
        score_i = 0.
        for j in range(len(tfidf_matrix)):
            score_i += similarity(tfidf_matrix[i],tfidf_matrix[j])
        sentence_score[i] = score_i

    max_score = max(sentence_score.values()) #归一化
    min_score = min(sentence_score.values())
    for key in sentence_score.keys():
        x = sentence_score[key]
        sentence_score[key] = (x-min_score)/(max_score-min_score)

    return sentence_score
```

最后对所有句子计算三个权重指数的加权和，并对所有句子进行排序。取前K个句子（或者前K%的句子）作为摘要：

```
def ranking_base_on_weigth(sentence_with_words_weight,
                           sentence_with_position_weight,
                           sentence_score, feature_weight = [1,1,1]):
    sentence_weight = collections.defaultdict(lambda :0.)
    for sent in sentence_score.keys():
        sentence_weight[sent] =
feature_weight[0]*sentence_with_words_weight[sent] +\

 feature_weight[1]*sentence_with_position_weight[sent] +\
                                feature_weight[2]*sentence_score[sent]

    sort_sent_weight = sorted(sentence_weight.items(),key=lambda d: d[1],
reverse=True)
    return sort_sent_weight

def get_summarization(sentence_with_index,sort_sent_weight,topK_ratio =0.3):
    topK = int(len(sort_sent_weight)*topK_ratio+1)
    summarization_sent = sorted([sent[0] for sent in
sort_sent_weight[:topK]])

    summarization = []
    for i in summarization_sent:
        summarization.append(sentence_with_index[i])

    summary = ''.join(summarization)
    return summary
```

（二）Bart Fine-tune

Bart的模型体量巨大，需要使用到预训练的模型，并在此基础上进行微调。这里我使用的是hugging face上的模型。该模型是用英文来训练的，并且用CNN Daily Mail的数据集进行了微调。

首先，import一些需要用到的package，导入数据:（这里我将原文件已经预先转化为csv文件了，方便pandas读取）

```
import pandas as pd
from fastai.text.all import *
from transformers import *
from blurr.data.all import *
from blurr.modeling.all import *

df = pd.read_csv('test_data.csv', error_bad_lines=False, sep=';')
df = df.dropna().reset_index()
df = df[(df['language']=='english') & (df['type']=='bs')].reset_index()
df = df[['title','text']]
df['text'] = df['text'].apply(lambda x: x.replace('\n',''))
```

接下来，导入预训练的模型；创建mini-batch用于训练；创建数据集。

```
pretrained_model_name = "facebook/bart-large-cnn"
hf_arch, hf_config, hf_tokenizer, hf_model = BLURR.get_hf_objects(pretrained_model_name,
                                    model_cls=BartForConditionalGeneration)

hf_batch_tfm = HF_Seq2SeqBeforeBatchTransform(hf_arch, hf_config, hf_tokenizer, hf_model,
    task='summarization',
    text_gen_kwargs=
 {'max_length': 248,'min_length': 56,'do_sample': False, 'early_stopping': True, 'num_beams': 4, 'temperature': 1.0,
  'top_k': 50, 'top_p': 1.0, 'repetition_penalty': 1.0, 'bad_words_ids': None, 'bos_token_id': 0, 'pad_token_id': 1,
 'eos_token_id': 2, 'length_penalty': 2.0, 'no_repeat_ngram_size': 3, 'encoder_no_repeat_ngram_size': 0,
 'num_return_sequences': 1, 'decoder_start_token_id': 2, 'use_cache': True, 'num_beam_groups': 1,
 'diversity_penalty': 0.0, 'output_attentions': False, 'output_hidden_states': False, 'output_scores': False,
 'return_dict_in_generate': False, 'forced_bos_token_id': 0, 'forced_eos_token_id': 2, 'remove_invalid_values': False})
blocks = (HF_Seq2SeqBlock(before_batch_tfm=hf_batch_tfm), noop)
dblock = DataBlock(blocks=blocks, get_x=ColReader('text'), get_y=ColReader('title'), splitter=RandomSplitter())
dls = dblock.dataloaders(articles, batch_size = 2)
```

接着，定义loss function，训练模型。

```
seq2seq_metrics = {
        'rouge': {
            'compute_kwargs': { 'rouge_types': ["rouge1", "rouge2", "rougeL"], 'use_stemmer': True },
            'returns': ["rouge1", "rouge2", "rougeL"]
        },
```

```
        'bertscore': {
            'compute_kwargs': { 'lang': 'fr' },
            'returns': ["precision", "recall", "f1"]}}

model = HF_BaseModelWrapper(hf_model)
learn_cbs = [HF_BaseModelCallback]
fit_cbs = [HF_Seq2SeqMetricsCallback(custom_metrics=seq2seq_metrics)]

learn = Learner(dls, model,
                opt_func=ranger,loss_func=CrossEntropyLossFlat(),
                cbs=learn_cbs,splitter=partial(seq2seq_splitter,
arch=hf_arch)).to_fp16()
learn.create_opt()
learn.freeze()
learn.fit_one_cycle(3, lr_max=3e-5, cbs=fit_cbs)
```

最后，获取摘要结果。

```
outputs = learn.blurr_generate(text_to_generate, early_stopping=False,
num_return_sequences=1)

for idx, o in enumerate(outputs):
    print(f'=== Prediction {idx+1} ===\n{o}\n')
```

6. 实验结果

在测试集上用该方法的实验结果如下：

	Rouge-1	Rouge-2	Rouge-L	BLEU
TF-IDF based	0.3273	0.2103	0.2958	0.1210
Bart Fine-tune	0.2872	0.1687	0.2597	0.0933

令人意外的是，基于深度学习的方法竟然结果不如传统机器学习的方法。究其原因，我觉得可能还是因为在这里我没有发挥出bart的真正实力。一是因为这里的bart是基于英文数据集预训练的，在中文数据集上的微调效果可能不是很好。二是由于算力原因，我只训练了一个epoch，远没有到收敛的程度，因此会影响结果质量。

下面展示一个例子，原文是：

> title: 关灯别玩手机！易致青光眼
>
> content: 很多人习惯睡前关灯拿出手机看会小说、玩玩游戏，武警总医院眼科主任吴志鸿提醒，在昏暗光线下用眼，会造成瞳孔长时间散大，堵塞眼内液体循环流通，很容易导致青光眼的发生。而青光眼是我国第一位不可逆致盲性眼病。现在你关灯了吗？快别玩手机了！via@央视新闻

基于TF-IDF的结果是：

> 很多人习惯睡前关灯拿出手机看会小说、玩玩游戏，而青光眼是我国第一位不可逆致盲性眼病。

基于Bart的结果是：

> 中国武警警告,不要关灯玩手机。