

NLP - Final Project

17341111 Xuehai Liu

2019 年 11 月 30 日

目录

1	数据库的建立——网页新闻抓取	3
1.1	新闻网页地址的提取与过滤	3
1.2	新闻网页的下载与存储	4
1.3	爬虫实现细节	7
2	数据预处理——分词与去噪	8
2.1	结巴分词	8
2.2	去除停用词与添加 token	8
2.3	构造词典	8
3	模型训练与预测	10
3.1	N-gram 模型	10
3.1.1	训练语料	10
3.1.2	最大似然估计	10
3.1.3	数据平滑	12
3.1.4	模型预测	13
3.2	LSTM 模型	14

3.2.1	训练语料	14
3.2.2	神经网络单元	15
3.2.3	模型搭建	15
3.2.4	模型预测	16
4	实验结果分析	18
4.1	N-gram 模型	18
4.2	LSTM 模型	19
4.3	模型综合比较分析	20
5	回顾与展望	23

1 数据库的建立——网页新闻抓取

第一部分的任务是从国内主流新闻网站（如腾讯、新浪、网易等）的科技频道抓取新闻内容，并要求抓取的内容对应的发布时间在 2019 年 1 月 1 日之后。利用这些数据，我们将在后面建立模型训练的数据库。网页的抓取由下面三个方面进行阐述：新闻地址的提取、新闻网页的下载与存储和爬虫的实现细节。

1.1 新闻网页地址的提取与过滤

传统的爬虫方法从一个网页中提取其他网页的链接，像蜘蛛在网上的爬行一样逐步链接到更多的链接。因此第一步我们需要确定要爬取的网页地址，作为爬虫的起点。此外，需要注意爬虫抓取的网页的限制：他们需要是新闻网页的科技频道，并且是在 2019 年 1 月 1 日之后的。



图 1: 新浪科技主页

通过对腾讯科技、新浪科技等频道的观察，我发现新浪科技的新闻是最具有普遍规律的：他们的网页有着下列格式：

`https://tech.sina.com.cn/***/publicDate/***`

因此我自然而然想到了可以使用正则表达式来限制抓取的网页，这样我使用的爬虫模式就仍然可以使用传统的爬虫方法，不需要作过多改动。



图 2: 新浪科技的文章范例

1.2 新闻网页的下载与存储

我在程序中封装了两个类：worker 和 crawler，分别用于创建线程、抓取网页。抓取网页使用了 BeautifulSoup 库，并且抓取了网页中具有 article 标签的文本。Worker 分多线程并维护一个队列，存储抓取到的新闻网页。存储时使用 sqlite 数据库，在主函数中单线程地插入数据库，防止多线程的 RC 问题。

```
class worker(Thread):
    def __init__(self, url, queue):
        super(worker, self).__init__()
        self.url = url
        self.q = queue

    def run(self):
        self.crawlPage()

    def request(self, url):
        print(url)
        send_headers = {
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36",
            "Connection": "keep-alive",
            "Accept-Language": "zh-CN,zh;q=0.8"}
        try:
```

```

        html=requests.get(url, timeout=2, stream=False,headers = send_headers)
    except Exception:
        #print('Could not open %s'%url)
        pass
    else:
        return html

def crawlPage(self):
    with sem:
        respond=self.request(self.url)
        if respond==None:
            return None

        '''
        if 'content-type' in respond.headers.keys():
            if not respond.headers['content-type'].startswith('text'):
                return None
        '''

        respond.encoding='utf-8'
        soup=BS(respond.text,'lxml')
        [x.extract() for x in soup.find_all('script')]
        [x.extract() for x in soup.find_all('style')]
        #排除script和样式标签

        title=soup.find('title')
        if title==None:
            title=''
        else:
            title=title.string
        text=soup.get_text()

    ignoreFiles =('.ppt','.pptx','.bmp','.jpg','.png','.tif','.gif','.pcx','.tga'
                  ', '.exif', '.fpx', '.svg', '.psd', '
                  .cdr', '.pcd', '.dxf', '.ufo', '.eps'
                  ', '.ai', '.raw', '.WMF', '.webp', '.
                  doc', '.docx', '.txt', '.pdf', '.
                  xlsx', '.xls', '.csv', '.rar', '.zip'
                  ', '.arj', '.gz', '.z', '.wav', '.aif'
                  ', '.au', '.mp3', '.ram', '.wma', '.
                  mmf', '.amr', '.aac', '.flac', '.avi'
                  ', '.mpg', '.mov', '.swf', '.int', '.

```

```

sys', '.dll', '.adt', '.map', '.bak'
, '.bat', '.cmd')

newpages=set()
links=soup('a')
if(soup.find("div", class_="article")):
    text = soup.find("div", class_="article").get_text()
for link in links:
    if ('href' in link.attrs):
        url=parse.urljoin(self.url,link['href'])
        if url.endswith(ignoreFiles) or not re.match(r'https://tech.sina.com
                                                    .cn/.*/2019-\d{2}-\d{2}
                                                    /*',url):

            continue
        if url.find('"')!=-1:continue

```

```

def crawl(self, pages,depth=5):
    for i in range(depth):
        q=Queue()
        newpages=set()
        thread_list=[]
        for page in pages:
            p=worker(page,q)
            p.start()
            thread_list.append(p)

        #分多线程开始爬取，并将爬到的东西都加入到队列中。

        for thread in thread_list:
            thread.join()
        #print('read finish')

        while not q.empty():
            (title,url,text,url_list)=q.get()
            #print('inserting '+url)
            #单线程地插入数据库。防止出现rc问题。
            self.addtoindex(url,title,text)
            for newurl in url_list:
                #print(k)
                if not self.isindexed(newurl):
                    newpages.add(newurl)

```

```
self.dbcommit()

#print('commit finish')

pages=newpages
```

1.3 爬虫实现细节

一些具体的实现细节说明：

- 抓取网页时需要使用 try,except 语句，防止出现意外导致整个程序停止。此外，爬虫可以设置 timeout，在网页响应过慢时跳过，防止在一个网页上浪费太多时间。
- 部分网页具有反爬虫机制，因此需要伪装爬虫头部为浏览器。
- 使用 sqlite 数据库存储，并提供了 class printer，帮助输出数据库内容以进行查看。

爬虫结果展示：



图 3: 爬虫结果

2 数据预处理——分词与去噪

第二部分对爬虫抓取的数据进行分句、分词、去除停用词和构造词典的操作。

2.1 结巴分词

分句使用简单的根据标点符号进行划分即可，我使用了正则表达式进行划分。

而核心分词代码如下：

```
sentences_seg = [list(jieba.cut(sentence)) for sentence in sentences]
```

结巴分词的 cut 方法返回一个迭代器，使用 python 的 list 类型进行转化后得到每个句子的词语构成列表。

2.2 去除停用词与添加 token

中文分词后，需要对分词结果中的停用词进行删除。使用的词表为百度停用词表。核心代码如下：

```
def movestopwordslist(list):
    stopwords = stopwordslist('语料/bd_stop_words.txt')
    outlist = []
    for word in list:
        #print (word)
        if word not in stopwords:
            outlist.append(word)
    return outlist
```

此外，需要在每个句子的句首和末尾添加 token，以便后面构造数据集时对句子进行区分。

```
sentences_seg = [["<BOS>"] + sentence + ["<EOS>"] for sentence in
                  sentences_seg ] #添加 token
```

2.3 构造词典

最后，需要统计每个词出现的次数并构造词典。最后，将词典写入到文件中，以便后续构造数据库使用。

```
#计数器，此处将所有标点符号和长度小于等于1的无意义字符删去。
c=Counter()
for x in all_words:
    if len(x)>1 and x != '\r\n':
        c[x] += 1
```



```

#获取词典
vocab = c.most_common(vocabulary_size)
index_to_word = [x[0] for x in vocab]
word_to_index = dict([(w,i) for i,w in enumerate(index_to_word)]) #['hello',100]{word,'
                                index'}

#保存到文件
with open("word_dict_n-gram.txt",'w',encoding = "utf-8") as text:
    for i in range(len(vocab)):
        string = str(i) + " " + vocab[i][0] + " " + str(vocab[i][1]) + "\n"
        text.write(string)

```

构造结果展示：

```

<BOS> 一个月前，毕福康刚刚 贾跃亭 手中接过 CEO 职位，告诉记者，精力花两件事情上，一个找融资，公司项目管理 <EOS>
<BOS> 毕福康，FF 财务计划 <EOS>
<BOS> 希望明年第一季度新一轮融资，12 - 15 月寻求 IPO 机会 <EOS>
<BOS> IPO，FF 还 8.5 亿美元资金 <EOS>
<BOS> 毕福康，贾跃亭 申请 破产 重组 一个利好，数年间，贾跃亭 债务 阻碍 FF 发展 因素 <EOS>
<BOS> 贾跃亭 标签 方式 弱化 掉，还得知，接下来毕福康 考验，FF 赢得 投资人 信赖 <EOS>
<BOS> 汽车 老将 毕福康 接棒 CEO 毕福康 汽车 圈 老将，FF，毕福康 曾 担任 中国 电动汽车 创业 公司 拜腾 董事长 兼 首席执行官，再往
<BOS> 毕福康 告诉 21 世纪 经济 报道 记者，选择 FF，关注 完整 出行 方式 生态系统 打造 <EOS>
<BOS> 未来 汽车 将会 出行 方式 一个 组成部分，汽车 关注 重点 不再 驾驶者，乘客 使用者 <EOS>
<BOS> 汽车 产业 不再 出售 汽车 商业模式，汽车 看成 一个 智能 设备，出售 智能 设备 搭载 生态系统 内容 盈利 <EOS>
<BOS> 毕福康 称，未来 看法，贾跃亭 想法 不谋而合 <EOS>
<BOS> 理念，再加上 宝马 造车 经验 拜腾 管理 经验，都 毕福康 接替 贾跃亭 不二 人选 <EOS>
<BOS> 毕福康 记者，FF 后 发现，公司拥有 先进 产品 技术，却 缺乏 执行力，世人 证明 FF 能力 交付 第一款 车 能力 <EOS>
<BOS> 角度，执行力 公司 最 目标 分配 团队，明年 9 月，第一款 车 交付 <EOS>

```

图 4: 数据预处理

3 模型训练与预测

模型部分，我构造了 n-gram 模型和 LSTM 模型，其中 n-gram 使用传统的统计方法进行计算，而 LSTM 模型使用神经网络进行搭建。下面进行详细说明。

3.1 N-gram 模型

n-gram 部分我主要实现了二元的 n-gram 语法。

3.1.1 训练语料

首先构造数据库。我将第二步骤中构建的分好词的数据集读入，去除长度 ≤ 1 的无用信息构造了 n-gram 的数据集。数据集中每个句子保留开始的 <BOS> 和 <EOS> 标记。
在本次训练中，我只使用了之前提交的 1000 条新闻文本作为数据集。

```
def load_data(filepath):
    text = []
    file_list = os.listdir(filepath)
    for file_path in file_list:
        full_name = filepath + "/" + file_path
        # print("当前处理的文件是:", full_name)
        with open(full_name, "r", encoding="utf-8") as f:
            for line in f.readlines():
                linelist = line.split()
                newlist = []
                for word in linelist:
                    if (len(word) > 1):
                        newlist.append(word)
                text += newlist
    return text
```

3.1.2 最大似然估计

首先我们明确一下 n-gram 的原理。一个句子出现的概率可以用全概率公式描述。

$$\begin{aligned} p(s) &= p(w_1)p(w_2 | w_1)p(w_3 | w_1 w_2) \cdots p(w_t | w_1 \cdots w_{t-1}) \\ &= \prod_{i=1}^t p(w_i | w_1 \cdots w_{i-1}) \end{aligned} \quad (5-1)$$

<http://blog.csdn.net/firpar>

运用马尔科夫链规则，可以估计此概率为条件概率的累乘（二元规则）

$$p(s) = \prod_{i=1}^l p(w_i | w_1 \cdots w_{i-1}) \approx \prod_{i=1}^l p(w_i | w_{i-1})$$

既然如此，要计算一个句子中某个位置的单词的可能性，我们就计算这个单词附近的 n-gram 内条件概率的累乘，取最大值作为预测值即可。那么如何计算条件概率？下面我们就使用最大似然估计的方法进行求解，公式如下：

$$p(w_i | w_{i-n+1}^{i-1}) = f(w_i | w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^i)}{\sum_{w_i} c(w_{i-n+1}^i)}$$

其中，f 表示历史串出现的情况下，wi 出现的概率。分母是历史串在语料库中出现的总次数。分子表示语料库中历史串和 wi 同时出现的总次数。

上述过程用代码表示如下：我使用了三个函数实现计算一个马尔可夫链的概率。

```
#比较两个数列，二元语法
def compareList(ori_list, test_list):
    #申请空间
    count_list = [0] * (len(test_list))
    #遍历测试的字符串
    for i in range(0, len(test_list) - 1):
        #遍历语料字符串，且因为是二元语法，不用比较语料字符串的最后一个字符
        for j in range(0, len(ori_list) - 2):
            #如果测试的第一个词和语料的第一个词相等则比较第二个词
            if test_list[i] == ori_list[j]:
                if test_list[i + 1] == ori_list[j + 1]:
                    count_list[i] += 1
    return count_list

def count_tuple(word1, word2, ori_list):
    count = 0
    for i, word in enumerate(ori_list):
        if (word == word1):
```

```

        if(ori_list[i+1] ==word2):
            count +=1
    return count

#计算概率
def probability(test_list,count_list,ori_dict):
    flag=0
    #概率值为p
    p=1
    for key in test_list:
        #数据平滑处理：加1法
        if(key not in ori_dict):
            continue
        p*=(float(count_list[flag]+1)/float(ori_dict[key]+1))
        flag+=1
    return p

```

3.1.3 数据平滑

正如上述代码陈述的，我使用了加一法进行数据平滑，这样做的目的是防止数据稀疏的情况下出现计数分子为 0，导致整个概率计算为 0 的情况。

对于2-gram 有：

$$\begin{aligned}
 p(w_i | w_{i-1}) &= \frac{1 + c(w_{i-1}w_i)}{\sum_{w_i} [1 + c(w_{i-1}w_i)]} \\
 &= \frac{1 + c(w_{i-1}w_i)}{|V| + \sum_{w_i} c(w_{i-1}w_i)}
 \end{aligned}$$

其中， V 为被考虑语料的词汇量(全部可能的基元数)。

核心代码：

```

p*=(float(count_list[flag]+1)/float(ori_dict[key]+1))

```

3.1.4 模型预测

n-gram 模型进行预测是线上的过程，即对于每个测试样例，都直接在数据集上进行计算其可能概率。因此，在预测时可以直接计算其 n-gram 对应的 prob，取预测值最大的一项作为输出即可。需要注意的是，由于数据集中许多出现次数太少的词语严重影响模型的训练效率，我在训练时规定词表中出现次数小于 10 的词语不参与训练。因此我的预测词典大小为 5000. 核心代码如下：

```
index = templist.index("MM")
templist[index] = key
#test_list = test_list.replace("MM",key)
count_list = compareList(ori_list, templist)

pred = probability(templist,count_list,ori_dict)

predlist.append((pred,key))
#pred = predict(sentence_ori,testdata,ori_dict)
if(pred>maxprob):
    maxprob = pred
    maxword = key
```

实验结果请阅报告的 4.1 节。

3.2 LSTM 模型

LSTM 模型的搭建基于 pytorch, 因此实现上和 n-gram 的传统方法有相当大的不同。下面介绍详细细节。

3.2.1 训练语料

训练语料方面, 我自定义了一个数据库类 MyDataset, 它继承自 torch.nn, 读取数据集中的所有数据, 将其载入到 text 列表中, 作为一个整个数据库长的词语列表。然后根据该列表构造 xdata 和 ydata。取 seq-size (这里取 32) 作为一个单位长度划分词语列表, 由 lstm 的性质, ydata 是 xdata 序列集的每个序列对应的往右移一位对应的序列集。故, xdata 和 ydata 的维度都是 32* 词表长度。

```
class MyDataset(data.Dataset):
    def __init__(self,filepath, batch_size,seq_size):
        text = []
        file_list = os.listdir(filepath)
        for file_path in file_list:
            full_name = filepath + "/" + file_path
            #print("当前处理的文件是:",full_name)
            with open(full_name, "r", encoding="utf-8") as f:
                for line in f.readlines():
                    linelist = line.split()
                    newlist = []
                    for word in linelist:
                        if(len(word)>1):
                            newlist.append(word)
                    text += newlist

        int_data =[word_to_int[word] for word in text]
        num_bacthes = int(len(int_data) / (seq_size * batch_size) )
        x_data = int_data[:num_bacthes*batch_size *seq_size]
        y_data = np.zeros_like(x_data)
        y_data[:-1] = x_data[1:]
        y_data[-1] = x_data[0]

        self.x_data = np.reshape(x_data,(-1,seq_size))
        self.y_data = np.reshape(y_data,(-1,seq_size))

    def __len__(self):
        return len(self.x_data)

    def __getitem__(self, id):
```

```
x = self.x_data[id]
y = self.y_data[id]
return x,y
```

3.2.2 神经网络单元

神经网络单元使用了一个比较基本的 lstm 单元。需要注意的是 lstm 的输入和输出是递归状态.

```
class LSTMModule(nn.Module):https://www.overleaf.com/project/5de13d4a039669000148be2796
```

3.2.3 模型搭建

模型的训练和常规的深度学习的步骤基本相同。

- 每轮训练中，先将梯度置零，防止前面轮次造成的梯度累积
- 从 dataloader 中获取数据并转接到 cuda 上
- 神经网络前向传播，计算损失 loss
- 反向传播，优化神经网络参数

重复上述训练过程，训练至达到最大 epoch 或模型收敛即可。下面是 train 部分的核心代码:

```
for i,(x,y) in enumerate(train_loader):
    iter += 1
    network.train()      #use train mode
    optimizer.zero_grad() # 梯度清零
    x = x.long()
    y = y.long()
    x = torch.LongTensor(x).to(device)
    y = torch.LongTensor(y).to(device)      #转化模型输入为longtensor
    logits, (state_h,state_c) = network(x,(state_h,state_c))
    #print(logits.size(),y.size())

    loss = criterion(logits.transpose(1,2), y)

    loss_value = loss.item()

    state_h = state_h.detach()
    state_c = state_c.detach()

    loss.backward()
```

```

        _ = torch.nn.utils.clip_grad_norm_(network.parameters(), param.gradients_norm
                                           )

    optimizer.step()
    losses.append(loss_value)

    if iter % 1000 == 0:
        torch.save(network.state_dict(), '{} / model-{}.pth'.format(param.
                                                                    checkpoint_path, iter))

    if loss_value < minloss:
        minloss = loss_value
        torch.save(network.state_dict(), '{} / model-final.pth'.format(param.
                                                                    checkpoint_path))

```

3.2.4 模型预测

最后的步骤就是模型预测。与 n-gram 的模型预测不同，LSTM 的预测过程是离线的。这意味着只要训练好模型，预测时只需要将测试数据放入模型即可得到对应的输出，而不是像 n-gram 一样需要遍历整个模型获取最优解。这令 LSTM 在测试上的效率比 n-gram 要高了许多。

核心代码如下。

```

def predict(device, network, quest, numWord, topk = 5):
    network.eval()
    q_index = quest.index("MM")
    pre_q , post_q = quest[:q_index], quest[q_index+len("MM"):]
    state_h, state_c = network.zero_state(1)
    state_h, state_c = state_h.to(device), state_c.to(device)

    for word in pre_q:
        index = word_to_int.get(word)
        if(index == None):
            continue
        ix = torch.tensor([[index]]).to(device)
        #label chaoguoyuzhi
        out, (state_h, state_c) = network(ix, (state_h, state_c))
    _, topix = torch.topk(out[0], k = topk)
    choices = topix.tolist()
    return [int_to_word[x] for x in choices[0]]

```

几点实现细节说明：

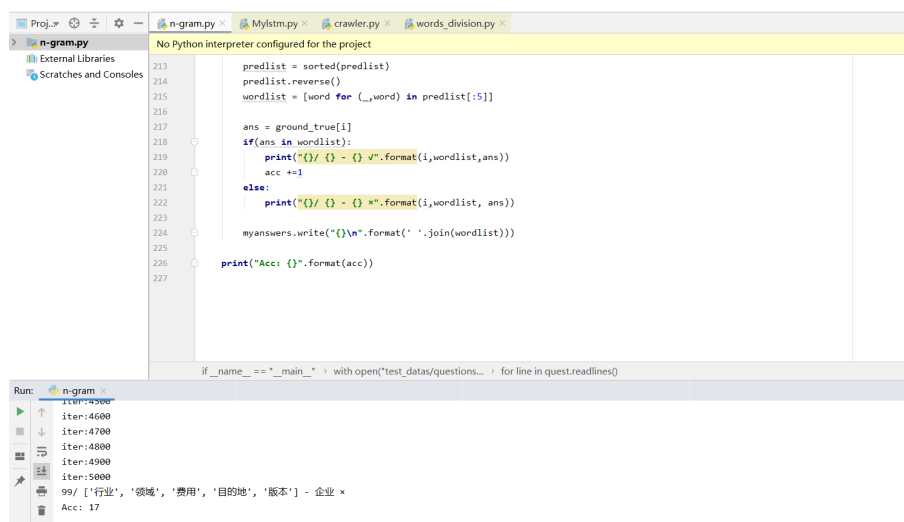
- 原 quests 数据集中的”[MASK]”已被我用”MM”代替，按 MM 的位置分割出预测词的前置序列。
 - 使用待预测单词的前置序列进行预测：获取前置序列的每个单词的序号，递归向后地放入神经网络运行。如果前置序列中的单词在词典中不存在则跳过。
 - 最后，使用 pytorch 提供的 topk method，返回最有可能的 k 个值。k 默认为 5
- 实验结果请阅 4.2 节。

4 实验结果分析

我这次训练结果基于的数据集是 1000 大小，与之前上交的数据集一致。下面展示模型测试结果：

4.1 N-gram 模型

N-gram 方面，我主要训练了 2-gram 的模型，并且在这个数据集上取得了 0.17 的 top-5 准确率：



```
Proj: n-gram.py x MyIstm.py x crawler.py x words_division.py x
> n-gram.py
No Python interpreter configured for the project
213 predlist = sorted(predlist)
214 predlist.reverse()
215 wordlist = [word for (_,word) in predlist[:5]]
216
217 ans = ground_true[i]
218 if ans in wordlist:
219     print("{} / {} - {} {}".format(i,wordlist,ans))
220     acc +=1
221 else:
222     print("{} / {} - {} {}".format(i,wordlist, ans))
223     myanswers.write("{}\n".format(' '.join(wordlist)))
224
225 print("Acc: {}".format(acc))
226
227
if __name__ == '__main__':
    with open("test_data/questions...") as f:
        for line in f.readlines():
```

Run: n-gram x
iter:4500
iter:4600
iter:4700
iter:4800
iter:4900
iter:5000
99 / ['行业', '领域', '费用', '目的地', '版本'] - 企业 x
Acc: 17

图 5: 2-gram acc



prediction_n_gram.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
模式 青少年 屏幕 遭受 电子产品
到来 中国 技术 产物 一个
中国 网络 美国 一个 世界各地
设计 iPhone 深色 型号 芯片
做生意 机会 黄牛 音乐 软件
一个 一种 月球 超过 联邦快递
业务 发展 用户 有限 提供
清晰 担忧 光明 行业 绝不
讨论 网易 项目 这项 网络
地区 时间 手机 电子 最终
成长 胆固醇 神经网络 研究 生理学
机器人 鼠标 鼓励 默认 黑色
规模 用户 表现 空间 收入
发展 下滑 推进 创新 稳定
手机 提供 正式版 更新 两个
灭绝 裁员 部署 物种 投资
QQ 覆盖 发现 用户 位置
美国 市场 丹麦 鼠标 鼓励
新浪 厂商 业务 市场 销量
球鞋 鼠标 鼓励 默认 黑色
补贴 优惠 折扣 选择 程度
发生 几年 还会 研究工作
霸王龙 材料 代表 鼠标 鼓励
外罚 制裁 虐待 多年 事情

图 6: top-5 prediction in txt

4.2 LSTM 模型

LSTM 的准确率稍微复杂一些，其准确率很大程度上取决于数据集的大小，并受模型超参数的影响。我使用下列超参数，在此数据集上获得了 0.12 的 top-5 准确率。

```
param = Namespace(  
    train_file_path="语料/train_seg/train",  
    checkpoint_path='checkpoint2',  
    seq_size=32,  
    batch_size=64,  
    embedding_size=128, # embedding dimension  
    lstm_size=128, # hidden dimension  
    gradients_norm=5, # gradient clipping  
    top_k=5,  
    num_epochs=40,  
    learning_rate=0.001  
)
```

此外，LSTM 的训练时间也受到超参数的影响。（尤其是 learning-rate 和 epoch）。使用上述超参数，我的训练时间在 15 分钟左右。

下面展示 LSTM 的预测准确率：

```
for i in range(len(quests)):
    pred = predict(device, network, quests[i], len(word_to_int), 5)
    ans = ground_true[i]
    if (ans in pred):
        acc += 1
        print("{} - pred: {} ans: {} ✓".format(i, pred, ans))
    else:
        print("{} - pred: {} ans: {} ✗".format(i, pred, ans))
acc = acc / len(quests)
print("Acc: {}".format(acc))

if __name__ == '__main__':
    test_lstm
```

91 - pred: ['中国', '发展', '5G', '繁荣', '全球化'] ans: 速度 ✗
92 - pred: ['产业', '发展', '金融', '企业', '科技'] ans: 人工智能 ✗
93 - pred: ['全球', '中国', '手机', 'OPPO', '2019'] ans: 智能机 ✗
94 - pred: ['垂直', '社交', '用户', '上演', '一个'] ans: 产品 ✗
95 - pred: ['AI', '意义', '协议', '合伙人', '马斯克'] ans: 无线电 ✗
96 - pred: ['垄断', '约会', '收购', '排名', '人民币'] ans: 营收 ✗
97 - pred: ['一家独大', 'CEO', '收入', '一个', '网易'] ans: 安全 ✗
98 - pred: ['金属', '出手', '2016', '哔哩', '提高'] ans: 价格 ✗
99 - pred: ['新兴', '一个', '互联网', '用户', '推进'] ans: 企业 ✗
Acc:0.12

图 7: LSTM acc

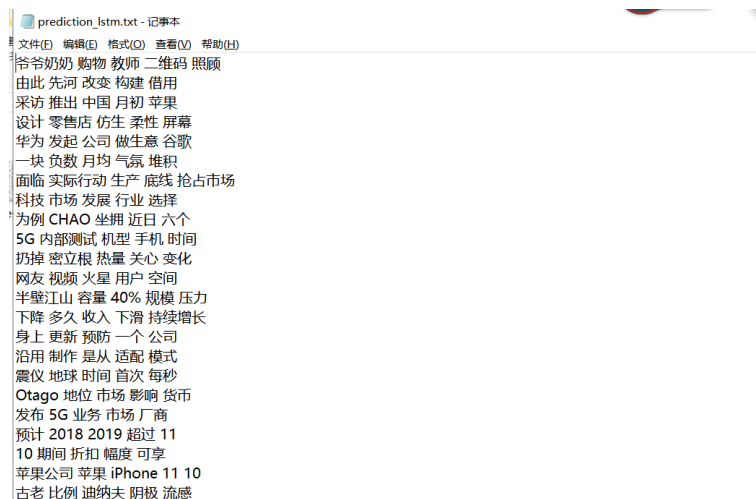


图 8: top-5 prediction in txt

4.3 模型综合比较分析

下面, 我将根据两个模型的训练时间、准确率和在一些数据上的表现对两个模型作综合的评价分析。

- 训练时间

首先, 训练时间方面, 我在将 2-gram 方法大幅简化的情况下, (限制预测词表大小、简化概率计算) 仍然训练了一个晚上才训练完。这是因为 n-gram 方法预测时要遍历整个词表查找最优的预测值, 这导致其预测的时间复杂度较高。相对的, LSTM 模型在 15 分钟左右训练完毕, 预测时的耗时则几乎忽略不计。可见 LSTM 模型在训练和预测上的效率要比传统方法的 n-gram 要高不少。

- 准确率

我统计了两个模型的 top-1, top-3 和 top-5 的情况，绘图如下：

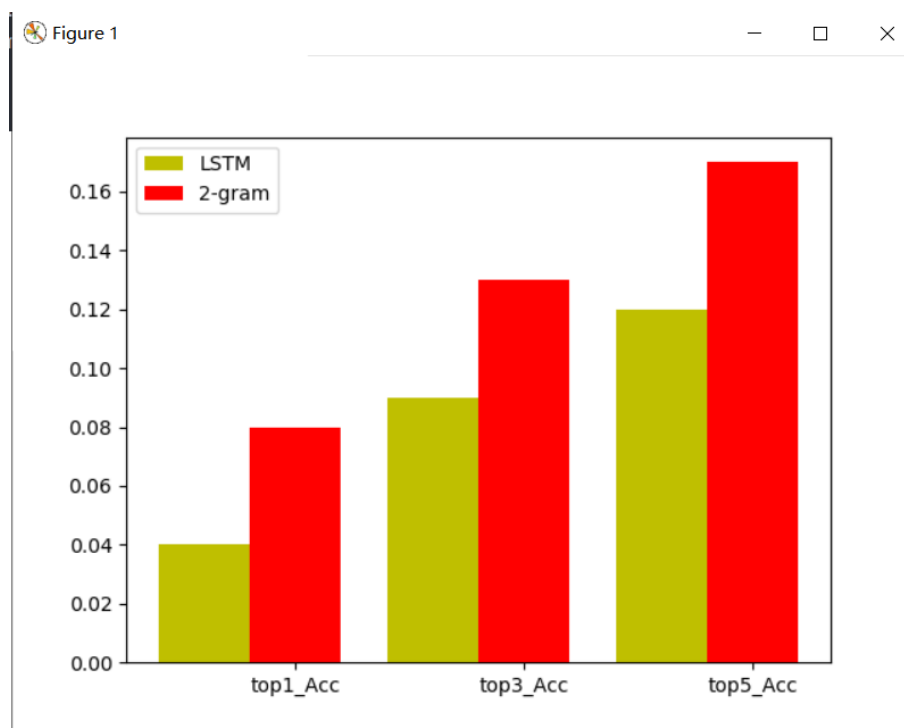


图 9: comparation

可以看出，尽管 2-gram 使用的是较为简单的统计的方法，但它在各个级别的 acc 上都要优于 LSTM 的神经网络方法。这给我带来了不小的疑惑：从两者的原理上来看，2-gram 使用了较为极端的预测方法：仅假设预测词的概率和它前面出现的 1 个词有关。这在现实中很多场景下是不正确的：因为我们现实中使用的句子往往有着逻辑上的联系，这是 n-gram 无法学习的。而 lstm 则是在句子的词语之间学习线性的函数不断拟合，理论上 lstm 应当是更加符合我们的造句习惯的做法。

那么这种情况下，LSTM 方法弱于 n-gram 的一个原因是什么呢？很简单，数据库（词表）过小了。我们知道，神经网络的训练必须基于足够大的数据集，否则它学习到的仍然是一个较弱的预测器。如果数据集过小，神经网络中的神经元参数拟合能力就差，反而表现得还不如传统的 n-gram 方法。

- 具体数据具体分析

然而，LSTM 是不是真的就弱于 n-gram 呢？让我们看一个案例来回答这个问题：

quest = ”然而，若卫视为了挽救业绩，一味向广告商低头，将节目内容更多地倾向于广告商，必将在一定程度上牺牲用户体验，进而陷入 [MASK] 流失、收视下降、广告商离场、业绩持续下滑的恶性循环。”

让我们看看两者的预测：

2-gram: [’资金’, ’困境’, ’1970’, ’认知’, ’第二次’]

lstm: [’朋友圈’, ’用户’, ’一条’, ’页面’, ’注意力’]

此题的正确答案是”用户”。

显然，n-gram 在这种句子前后具有逻辑相关性的时候就显得有些束手无策了。前文提及的牺牲用户体验无法被 n-gram 学习，自然也就无法预测。而相对的，lstm 模型却学习到了句子之间的关系，正确判断了用户体验的牺牲导致用户流失的结果。事实上，这样的案例在 lstm 预测成功而 n-gram 失败的情况中出现了 4 次，占预测成功 12 次的 1/3。

可见，lstm 的泛用性是要强于 n-gram 的——即使它在较小的数据集上表现较差，但是可以预见如果给两个模型足够大的数据集，lstm 所能学习到的表示能力将要强于 n-gram 模型。

5 回顾与展望

本次项目是一个相当全面的学习自然语言处理的过程。

从获取数据到数据清洗，文本分词，语言模型的选择与训练，我对自然语言处理的整个流程熟悉了许多。虽然之前曾经用过神经网络训练图像的零样本分类，但是神经网络模型在自然语言处理中的应用尚是第一次，对我而言充满了新鲜感。回顾整个项目的过程，我还是碰到了不少的问题：如何准确获取训练中需要的数据？如何表示这些数据来让语言模型能够利用他们训练？如何构建语言模型？怎么选择更好的语言模型？这些问题随着这次项目的进展逐一得到了解答，令我受益匪浅。

此外，一个不小的感触是：当今时代，一个模型的成败很大程度上已经不再像以往一样单纯地建立在算法是否足够优秀上，而更多地取决于数据的质量和数量了。如果拥有了足够多的高质量数据，神经网络凭借其足够强大的拟合学习能力将能够学习到比传统算法远远复杂的多的线性与非线性关系。相对的，如果神经网络没有足够多的优质数据，它所学习到的网络将非常受限。新时代，算法和数据要共同进步，我们未来才能够获得更好的服务于人们的技术。

未来的方向：

本学期期末前我计划完成一个人机对话的系统，记录我们日常生活中的时间管理与目标管理。人机对话的系统需要在这次的项目的基础上再进行拓展，模型将不再只是生成简单的单词预测，而将生成更复杂的对话语句，并且需要远远多于本次实验的语料。模型的构建有两种方法：检索法和生成法。检索法简单地根据用户输入从数据库中检索最佳答案，而生成法需要根据用户输入生成一个回应序列，以现在的技术而言挑战性十足。未来如果有机会，我将对上述方法进行尝试实验，并争取做出成品。