Report of Deep Learning for Natural Langauge Processing

Zhijin Cao
21231024@buaa.edu.cn

Abstract

该实验利用一元、二元、三元N-gram模型，在训练集语料库中估算条件概率，以此为依据在测试集上估算文本信息熵的上界。求得中文汉字在一元、二元、三元模型下的信息熵分别为10.022235、7.679201、4.381159，中文词语在一元、二元、三元模型下的信息熵分别为14.758152、7.309117、1.123431，英文字母一元、二元、三元模型下的信息熵分别为3.697409、3.192029、2.942581，英文单词一元、二元、三元模型下的信息熵分别为12.215167、6.750498、0.895930。

Introduction

信息熵

信息熵的概念最早由香农（1916-2001）于1948年借鉴热力学中的“热熵”的概念提出，旨在表示信息的不确定性。熵值越大，则信息的不确定程度越大。其数学公式可以表示为

$$H(x)=\sum_{x\in X}P(x)\log(\frac{1}{p(x)})=-\sum_{x\in X}P(x)\log(P(x))$$

论文中假设 $X=\{...X_{-2},X_{-1},X_0,X_1,X_2,...\}$ 是基于有限字母表的平稳随机过程， P 为 X 的概率分布， E_p 为 P 的数学期望，则 X 的信息熵定义为

$$H(x)=H(P)=-E_p\log P(X_0\mid X_{-1},X_{-2},...)$$

当对数底数为2时，信息熵的单位为bit，相关理论说明此时信息熵可以表示为

$$H(P)=\lim_{n\to\infty}-E_p\log P(X_0\mid X_{-1},X_{-2},...,X_{-n})$$

我们无法精确获取 X 的概率分布，因此无法获取精确的 P ，但我们可以通过数据量足够大的训练集语料库来估计 P ，通过建立 P 的随机平稳过程模型 M 来估算 $H(P)$ 的上界，与上述推理过程相同，我们可以得到以下公式

$$H(P,M)=\lim_{n\to\infty}-E_p\log M(X_0\mid X_{-1},X_{-2},...,X_{-n})$$

$H(P,M)$ 有较大的参考价值，是因为其是 $H(P)$ 的一个上界，即 $H(P)\le H(P,M)$

N-gram模型

N-gram模型用于预测文本序列在语料库中可能出现的概率，给定一个文本序列：$S = W_1, W_2, ..., W_k$，它的概率可以表示为

$$P(S) = P(W_1, W_2, ..., W_k) = P(W_1)P(W_2 \mid W_1)...P(W_k \mid W_1, W_2, ..., W_{k-1}) = \prod_i P(W_i \mid W_1, ..., W_{i-1})$$

然而，这样计算存在参数空间过大的问题。出于简化问题的需要，可以引入N-gram模型，该模型在马尔可夫假设下，下一个序列元素出现的概率只与前面的有限个序列元素有关。因此，前面得到的条件概率的计算可以简化如下

$$P(W_i \mid W_1, ...W_{i-1}) \approx P(W_i \mid W_{i-k}, ...W_{i-1})$$

当$k = 0$时对应的模型为一元模型（unigram），每个W_i相互独立，此时

$$P(W_1, W_2, ..., W_k) = \prod_i P(W_i)$$

当$k = 1$时对应的模型为二元模型（bigram），每个W_i只与前一个相关，此时

$$P(W_1, W_2, ..., W_k) = \prod_i P(W_i \mid W_{i\text{-}1})$$

当$k = 2$时对应的模型为三元模型（trigram），每个W_i只与前两个相关，此时

$$P(W_1, W_2, ..., W_k) = \prod_i P(W_i \mid W_{i\text{-}2}, W_{i\text{-}1})$$

Methodology

实验思路

该实验利用$k = 0,1,2$时的一元、二元、三元模型，在训练集语料库中估算条件概率，以此为依据在测试集上估算信息熵的上界。在实验中，对中文、英文两种语言进行信息熵估算，中文分按汉字和词语两种划分情况，英文分按字母和单词两种划分情况。

需要注意的是，要使以上范式合理，构建模型M的训练集语料库应与测试用的文本信息完全独立。若没有这一限制，将语料库既用作训练集也用作测试集，则条件概率将比实际值偏大，造成最终信息熵比实际信息熵上限偏小。

由于选取的训练集的不完全性，可能出现这样一种情况，即测试集中的词元组合在训练集中未出现过或出现的频次极低（由此求得的条件概率低于浮点数的最小精度），此时应该如何对条件概率进行估计呢？论文中采用了如下的平滑处理方法，其基本思想是：由于N-gram比N+1-gram出现的可能性大得多，所以使用N-gram估计N+1-gram的概率，下面

以三元模型的二阶条件概率举例

$$P(W_i \mid W_{i-2}W_{i-1}) = \lambda_3 P(W_i \mid W_{i-2}W_{i-1}) + \lambda_2 P(W_i \mid W_{i-1}) + \lambda_1 P(W_i) + \lambda_0 P_0$$

其中 $\sum_{j=1}^{3} \lambda_j = 1$， $P(W_i)$ 为每个词的概率， $P_0 = 1/R, R$ 为统计语料中出现的总词/字数。关于参数 λ_i 的确定：训练数据分为两部分，一部分用于估计条件概率，另一部分用于计算参数 λ_i，求使得语言模型perplexity最小的 λ_i

本文中采取了更为简化的方法处理这一问题，即若频次为0，直接给频次加1使得条件概率可以被计算，该方法出于以下假设：假设频次为0的情况相对而言是极少数，此时这一处理对最终信息熵结果的影响可忽略不计。

实验数据

该实验中文数据集为维基百科语料库的部分内容，其中训练集为wiki_zh/AA中的全部内容，测试集为wiki_zh/AM中wiki_00.txt到wiki_10.txt。该实验英文数据库为nltk中gutenberg语料库的全部内容，其中训练集为gutenberg语料库中除了austen-emma.txt的全部内容，测试集为austen-emma.txt。

代码解析

该实验的主函数主要分为加载数据，获取一元、二元、三元字典，计算信息熵三个部分，主函数代码如下

```
if name == "main":

    modetype = ['char', 'token']
    languagetype = ['ch', 'en']

    for mode in modetype:
        for language in languagetype:

            tfdictrain = {}
            bigramtfdictrain = {}
            trigramtfdictrain = {}
            tfdictest = {}
            bigramtfdictest = {}
            trigramtfdictest = {}

            if language == 'ch':

                datatrain = loaddatachinese(traindatachinesepath, mode)
                datatest = loaddatachinese(testdatachinesepath, mode)
                print(f'中文信息熵计算结果({mode}): ')

            if language == 'en':

                datatrain = loaddataenglishtrain(mode)
                datatest = loaddataenglishtest(mode)
                print(f'英文信息熵计算结果({mode}): ')
```

```
            gettf(tfdictrain, datatrain)
            getbigramtf(bigramtfdictrain, datatrain)
            gettrigramtf(trigramtfdictrain, datatrain)
            gettf(tfdictest, datatest)
            getbigramtf(bigramtfdictest, datatest)
            gettrigramtf(trigramtfdictest, datatest)

            calculateunigramentropy(tfdictrain, tfdictest)
            calculatebigramentropy(tfdictrain, bigramtfdictrain, bigramtfdictest)
            calculatetrigramentropy(bigramtfdictrain, trigramtfdictrain, trigramtfdictest)
```

加载数据部分，其内容包括将文本数据按路径加载，按模式对数据进行拆分，将拆分后的数据进行筛选并去除停词，代码如下

```
def loaddatachinese(datapath, mode):
    """
        加载中文数据
        :return:
    """
    data = ""

    #加载路径下所有内容，去除停用词后返回至data
    for root, dirs, files in os.walk(datapath):
        for file in files:
            filepath = os.path.join(root, file)
            with open(filepath, 'r', encoding='utf-8') as f:
                filedata = f.read()
            data += filedata

    #按照模式进行分词
    splitwords = []
    if mode == 'token':
        splitwords = list(jieba.cut(data))
    elif mode == 'char':
        splitwords = [ch for ch in data]

    # 保留所有的中文字符并去除分词中的停用词
    with open(stopwordschinesepath, 'r', encoding='utf-8') as f:
        stopwords = [line.strip() for line in f.readlines()]
    splitwords = [word for word in splitwords if re.match(r'^[u4e00-u9fff]+$', word) and word not in

    return splitwords

def loaddataenglishtrain(mode):
    """
        # 加载英文训练集数据
        # :return:
    """
    nltk.download('gutenberg')
    fileidtoskip = 'austen-emma.txt'
    data = ""

    # 加载训练集中所有内容，此处将2以后的内容划为训练集
    for fileid in gutenberg.fileids():
        if fileid == fileidtoskip:
            continue  # 跳过这个文件ID
        data += gutenberg.raw(fileid)

    # 按照模式进行分词
    splitwords = []
    if mode == 'token':
        splitwords = wordtokenize(data.lower())
    elif mode == 'char':
        splitwords = list(data)

    #去除分词中的停用词
```

```
    nltk.download('punkttab')
    nltk.download('stopwords')
    splitwords = [word for word in splitwords if word not in string.punctuation]
    stopwords = set(stopwords.words('english'))
    splitwords = [word for word in splitwords if word not in stopwords]

    return splitwords


def loaddataenglishtest(mode):
    """
        加载英文测试集数据
        :return:
    """
    nltk.download('gutenberg')
    fileid = 'austen-emma.txt'

    # 加载测试集中所有内容，此处将1的内容划为测试集
    data = gutenberg.raw(fileid)

    # 按照模式进行分词
    splitwords = []
    if mode == 'token':
        splitwords = wordtokenize(data.lower())
    elif mode == 'char':
        splitwords = list(data)

    # 去除分词中的停用词
    nltk.download('punkttab')
    nltk.download('stopwords')
    splitwords = [word for word in splitwords if word not in string.punctuation]
    stopwords = set(stopwords.words('english'))
    splitwords = [word for word in splitwords if word not in stopwords]

    return splitwords
```

获取字典部分，其内容包括根据加载好且预处理后的数据生成一元、二元、三元词字典，代码如下

```
def gettf(tfdic, words):
    """
    获取一元词词频
    :return:一元词词频dic
    """
    for i in range(len(words)):
        tfdic[words[i]] = tfdic.get(words[i], 0) + 1


def getbigramtf(tfdic, words):
    """
    获取二元词词频
    :return:二元词词频dic
    """
    for i in range(len(words)-1):
        tfdic[(words[i], words[i+1])] = tfdic.get((words[i], words[i+1]), 0) + 1


def gettrigramtf(tfdic, words):
    """
    获取三元词词频
    :return:三元词词频dic
    """
    for i in range(len(words)-2):
        tfdic[((words[i], words[i+1]), words[i+2])] = tfdic.get(((words[i], words[i+1]), words[i+2])
```

计算信息熵部分，其内容包括平滑处理估算条件概率，计算信息熵，输出信息熵，代码如下

```
def calculateunigramentropy(tfdictrain, tfdictest):
    """
    计算一元词的信息熵
    :return:
    """

    begin = time.time()
    wordsnumtrain = sum([item[1] for item in tfdictrain.items()])
    wordsnumtest = sum([item[1] for item in tfdictest.items()])
    entropy = 0
    for item in tfdictest.items():
        jp = item[1]/wordsnumtest
        itemfintrain = tfdictrain.get(item[0], 0)
        # 平滑处理
        if itemfintrain == 0:
            itemfintrain = 1
        cp = itemfintrain / wordsnumtrain
        if cp > 0:
            entropy += -jp  math.log(cp, 2)
    print("一元模型信息熵为：{:.6f} 比特/(词or字）".format(entropy))
    end = time.time()
    print("一元模型运行时间：{:.6f} s".format(end - begin))

    return ['unigram', round(entropy, 6)]


def calculatebigramentropy(tfdictrain, bigramtfdictrain, bigramtfdictest):
    """
    计算二元词的信息熵
    :return:
    """

    begin = time.time()
    biwordsnumtest = sum([item[1] for item in bigramtfdictest.items()])
    entropy = 0
    for biitem in bigramtfdictest.items():
        jp = biitem[1] / biwordsnumtest
        itemfintrain = bigramtfdictrain.get(biitem[0], 0)
        t = tfdictrain.get(biitem[0][0], 0)
        # 平滑处理
        if itemfintrain == 0:
            itemfintrain = 1
        if t == 0:
            t = 1
        cp = itemfintrain / t
        if cp > 0:
            entropy += -jp  math.log(cp, 2)
    print("二元模型信息熵为：{:.6f} 比特/(词or字）".format(entropy))
    end = time.time()
    print("二元模型运行时间：{:.6f} s".format(end - begin))

    return ['bigram', round(entropy, 6)]


def calculatetrigramentropy(bigramtfdictrain, trigramtfdictrain, trigramtfdictest):
    """
    计算三元词的信息熵
    :return:
    """

    begin = time.time()
    triwordsnumtest = sum([item[1] for item in trigramtfdictest.items()])
```

```
entropy = 0
for triitem in trigramtfdictest.items():
    jp = triitem[1] / triwordsnumtest
    itemfintrain = trigramtfdictrain.get(triitem[0], 0)
    t = bigramtfdictrain.get(triitem[0][0], 0)
    # 平滑处理
    if itemfintrain == 0:
        itemfintrain = 1
    if t == 0:
        t = 1
    cp = itemfintrain / t
    if cp > 0:
        entropy += -jp  math.log(cp, 2)
print("三元模型信息熵为: {:.6f} 比特/(词or字）".format(entropy))
end = time.time()
print("三元模型运行时间: {:.6f} s".format(end - begin))

return ['trigram', round(entropy, 6)]
```

Experimental Studies

实验结果如下表所示

Table 1: 不同模型和语言、输入类型下信息熵结果

信息熵（bit/（字/词）	中文汉字	中文词语	英文字母	英文单词
Unigram	10.022235	14.758152	3.697409	12.215167
Bigram	7.679201	7.309117	3.192029	6.750498
Trigram	4.381159	1.123431	2.942581	0.895930

Conclusions

结果分析

1. 对中文和英文两种语言进行对比，可以看到，无论是在字还是词情况下，使用相同的模型，中文的信息熵总是高于英文的信息熵，这表明中文的混乱程度更高，更难以预测，蕴含更多的信息量。这一结果的原因可能是，中文汉字的数量远远多于英文字母的数量，这为不同字的组合带来了更多可能，字与字之间的关系也相对更为复杂。

2. 对两种语言的字和词进行对比，可以看到在一元情况下，词的信息熵均高于字的信息熵，这是因为词语是由字组成的，词的可能情况比字的可能情况多得多。而随着N的增长，到三元模型时，字的信息熵均高于词的信息熵。这是因为在文本中，组成句子的相邻词语之间多有关联，而相邻字（尤其是3个字，不同容易组成常见词）的关联相对较小，因此此时字组的混乱程度高于词组的。

3. 对一元、二元、三元模型进行对比，可以看到随着N的增大，信息熵逐渐减小，这是因为N越大，预测下一个元素所用到的已知元素就越多，预测结果越精确，因而信息熵减小。

不足

1. 由于采用了比较粗糙的平滑方法，随着N的增大，N元组的种类越来越多，越发有可能出现在测试集中出现的词元组合在训练集中没有出现或出现频率极低的情况，这使得最终结果比实际结果偏小，对比论文中结果：对于英语单词token部分三元模型下的信息熵为1.61bit，而本文结果为0.90bit，可知平滑方法的不同对结果造成的影响还是比较大的。改进方法是按照论文中的思路，进行插值删除平滑。

2. 本文只计算了token部分的信息熵，而在论文中实际信息熵的上限由四部分组成，分别是token部分、spelling部分、case部分、spacing部分，更精确的结果需要计算其余三部分，得到整体信息熵上限。

References

[1] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, Jennifer C. Lai, and Robert L. Mercer. 1992. An Estimate of an Upper Bound for the Entropy of English.