

## 实验题目

## 实验内容

## 遇到和解决的问题

## 实验步骤

### 英文工具

NLTK

Spacy

StanfordCoreNLP

### 中文工具

jieba

StanfordCoreNLP:

SnowNLP

THULAC

NLPIR

## Bert+BiLstm+CRF

### 1.前言

### 2.数据预处理

#### 2.1本地查看数据转换后的结果

### 3.构建数据集

### 4.数据集分割

### 5.模型架构

#### 5.1模型初始化

#### 5.2前向传播过程

#### 6.1训练一个epoch

#### 6.2训练所有epoch

#### 6.3evaluate函数

### 7.整体训练过程

### 8.训练结果 (50 epoch)

## Bert理论部分

### 1.前言

### 2.Bert

#### 2.1主要任务

##### 2.1.1.完形填空

##### 2.1.2.预测下一个句子

#### 2.2.输入表示

#### 2.3.MLM (Mask Language Model)

#### 2.4.NSP

#### 2.5.总体架构

### 3.Fine-tuning

## 总结

英文命名实体识别

中文命名实体识别

Bert+BiLstm-CRF

# 实验题目

- 命名实体识别
- 1.掌握命名实体识别（NER）相关基础知识点。  
2.使用开源工具以及tensorflow等框架实现命名实体识别模型，加深对相关理论的理解。

# 实验内容

- 1.利用 Chinese.txt 和 English.txt 的中英文句子，在实验二的基础上，继续利用以下给定的中英文工具进行命名实体识别。并对不同工具产生的结果进行简要对比分析，将实验过程与结果写成实验报告，实验课结束后提交。
- 2.使用BERT + Bi-LSTM + CRF 实践命名实体识别，详细代码在BERT-LSTM-CRF压缩包，要求：运行代码，理解过程

# 遇到和解决的问题

多数是代码没明白，函数不明白，模型不理解的问题，均在下方叙述了

# 实验步骤

## 英文工具

- NLTK
- 代码

```
def nltk_nlp(data):
    ans = nltk.word_tokenize(data) # 分词
    tagged = nltk.pos_tag(ans) # 词性标注

    entities = nltk.chunk.ne_chunk(tagged) # 命名实体识别
    out = [i for i in entities]
    print(out)
```

- 结果

可以看出的是，命名实体识别函数即完成了词性标注，又完成了命名实体识别，且识别出的命名实体以 `Tree(实体名,[单词, 词性])` 表示

Named Entity Type	Examples
PERSON	President Obama, Franz Beckenbauer
ORGANIZATION	WHO, ISRO, FC Bayern
LOCATION	Germany, India, USA, Mt. Everest
DATE	December, 2016-12-25
TIME	12:30:00 AM, one thirty pm
MONEY	Twenty dollars, Rs. 50, 100 GBP
PERCENT	20%, forty five percent
FACILITY	Stonehenge, Taj Mahal, Washington Monument
GPE	Asia, Europe, Germany, North America

`GPE` 通常表示地理—政治条目，比如城市，州，国家，洲等。`LOCATION` 除了上述内容外，还能表示名山大川等。`FACILITY` 通常表示知名的纪念碑或人工制品等。

- 识别实体列表： `[PERSON, ORGANIZATION, GPE]`
- 优点：识别的 `ORGANIZATION` 效果不错，基本准确，但是可能会将名词组合中的所有词都认为是 `ORGANIZATION`，比如识别了 `General`。识别 `GPE` 效果也不错，识别了 `China`
- 缺点：识别的两个 `PERSON`，一个是 `HAN`，一个是 `Shaanxi`，两个都不是人名。而且有人名并未识别，比如 `Xi Jinping`。感觉时间（月份）也属于命名实体，但是 `nltk` 并未对其进行识别。`GPE` 识别出了 `Law`，可能不太对

```
[('Xi', 'NN'), ('Jinping', 'NNP'), (',', ','), ('male', 'NN'),
(',', ','), Tree('PERSON', [('Han', 'NNP'])), ('ethnicity', 'NN'),
(',', ','), ('was', 'VBD'), ('born', 'VBN'), ('in', 'IN'),
('June', 'NNP'), ('1953', 'CD'), ('and', 'CC'), ('is', 'VBZ'),
('from', 'IN'), ('Fuping', 'VBG'), (',', ','), Tree('PERSON',
[('Shaanxi', 'NNP'), ('Province', 'NNP')]), ('.', '.'), ('He',
'PRP'), ('began', 'VBD'), ('his', 'PRP$'), ('first', 'JJ'),
('job', 'NN'), ('in', 'IN'), ('January', 'NNP'), ('1969', 'CD'),
('and', 'CC'), ('joined', 'VBD'), ('the', 'DT'),
Tree('ORGANIZATION', [('Communist', 'NNP'), ('Party', 'NNP')]),
('of', 'IN'), Tree('GPE', [('China', 'NNP'])), ('(', '('),
Tree('ORGANIZATION', [('CPC', 'NNP'])), (')', ')'), ('in', 'IN'),
('January', 'NNP'), ('1974', 'CD'), ('.', '.'), ('Xi', 'VB'),
('graduated', 'VBN'), ('from', 'IN'), Tree('ORGANIZATION',
[('School', 'NNP')]), ('of', 'IN'), Tree('ORGANIZATION',
[('Humanities', 'NNP')]), ('and', 'CC'), Tree('ORGANIZATION',
[('Social', 'NNP'), ('Sciences', 'NNPS')]), (',', ','),
Tree('ORGANIZATION', [('Tsinghua', 'NNP'), ('University',
'NNP')]), ('where', 'WRB'), ('he', 'PRP'), ('completed', 'VBD'),
('an', 'DT'), ('in-service', 'JJ'), ('graduate', 'NN'),
('program', 'NN'), ('in', 'IN'), Tree('GPE', [('Marxist',
'NNP')]), ('theory', 'NN'), ('and', 'CC'), ('ideological', 'JJ'),
('and', 'CC'), ('political', 'JJ'), ('education', 'NN'), ('.',
'.'), ('He', 'PRP'), ('holds', 'VBZ'), ('a', 'DT'), ('Doctor',
'NNP'), ('of', 'IN'), Tree('GPE', [('Law', 'NNP'])), ('degree',
'NN'), ('.', '.'), ('Xi', 'NN'), ('is', 'VBZ'), ('currently',
'RB'), Tree('ORGANIZATION', [('General', 'NNP')]), ('Secretary',
'NNP'), ('of', 'IN'), ('the', 'DT'), Tree('ORGANIZATION', [('CPC',
'NNP'), ('Central', 'NNP'), ('Committee', 'NNP')]), (',', ','),
('Chairman', 'NNP'), ('of', 'IN'), ('the', 'DT'),
Tree('ORGANIZATION', [('CPC', 'NNP'), ('Central', 'NNP')]),
('Military', 'NNP'), ('Commission', 'NNP'), (',', ','),
('President', 'NNP'), ('of', 'IN'), ('the', 'DT'),
Tree('ORGANIZATION', [('People', 'NNP'])), ('s', 'POS'),
Tree('ORGANIZATION', [('Republic', 'NNP'])), ('of', 'IN'),
Tree('GPE', [('China', 'NNP'])), ('(', '('), Tree('ORGANIZATION',
[('PRC', 'NNP')]), (')', ')'), (',', ','), ('and', 'CC'),
('Chairman', 'NNP'), ('of', 'IN'), ('the', 'DT'),
Tree('ORGANIZATION', [('PRC', 'NNP'), ('Central', 'NNP')]),
('Military', 'NNP'), ('Commission', 'NNP'), ('.', '.')]

```

- **Spacy**

- 代码

```
def spacy_nlp(data):
    nlp = spacy.load("en_core_web_sm")
    doc = nlp(data)
    ans = []
    for ent in doc.ents:
        ans.append({ent.text: ent.label_})
    print(ans)
```

- 结果

Spacy 的实体识别效果是惊人的，识别出了 PERSON, NORP, DATE, GPE, ORDINAL, ORG 这些类别，这是远远多于并且精细于 nltk 的。

TYPE	DESCRIPTION
PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Companies, agencies, institutions, etc.
GPE	Countries, cities, states.
LOC	Non-GPE locations, mountain ranges, bodies of water.
PRODUCT	Objects, vehicles, foods, etc. (Not services.)
EVENT	Named hurricanes, battles, wars, sports events, etc.
WORK_OF_ART	Titles of books, songs, etc.
LAW	Named documents made into laws.
LANGUAGE	Any named language.
DATE	Absolute or relative dates or periods.
TIME	Times smaller than a day.
PERCENT	Percentage, including "%".
MONEY	Monetary values, including unit.
QUANTITY	Measurements, as of weight or distance.
ORDINAL	"first", "second", etc.
CARDINAL	Numerals that do not fall under another type.

知乎 @hahakity

- 优点：识别效果很好，能够以短语的形式进行整体识别，而不是将短语拆分成单个词再进行实体识别。而且，识别结果比较精细，总体识别效果非常不错，没有看出明显的错误和不足
- 缺点：个人对 {'Marxist': NORP} 保持怀疑，但是感觉其实是比较合理的，其实实体命名的种类不够精细，都是大类，比如没有学科等实体。但是从实体识别的效果上看，没有明显的缺点。

```
[{'Xi Jinping': 'PERSON'}, {'Han': 'NORP'}, {'June 1953': 'DATE'}, {'Fuping': 'GPE'}, {'Shaanxi Province': 'GPE'}, {'first': 'ORDINAL'}, {'January 1969': 'DATE'}, {'the Communist Party of China': 'ORG'}, {'CPC': 'ORG'}, {'January 1974': 'DATE'}, {'School of Humanities and Social Sciences': 'ORG'}, {'Tsinghua University': 'ORG'}, {'Marxist': 'NORP'}, {'a Doctor of Law degree': 'LAW'}, {'the CPC Central Committee': 'ORG'}, {'the CPC Central Military Commission': 'ORG'}, {"the People's Republic of China": 'GPE'}, {'PRC': 'GPE'}, {'the PRC Central Military Commission': 'ORG'}]
```

## • StanfordCoreNLP

- 代码

```
def stanford_nlp(data):  
    # -*-coding:utf-8 -*-  
    with StanfordCoreNLP(r'D:\stanford-corenlp-full-2018-02-27')  
    as nlp:  
        print(nlp.ner(data))
```

- 结果

首先可以看出的是，该方法对每个词进行划分，然后识别，而不是可以像 Spacy 一样进行短语的实体命名。识别的格式为 (单词, 实体名)，如果不是实体的，实体名为 `0`

总体识别列表为 [PERSON, DATE, STATE\_OR\_PROVINCE, LOCATION, ORDINAL, ORGANIZATION, IDEOLOGY, TITLE, MISC, COUNTRY]

- 优点：具有一定的识别准确度，分类比较精细，出现了特有的实体名称 IDEOLOGY, MISC, TITLE。
- 缺点：不能以短语的形式进行实体命名。有很多遗漏识别的，比如 Han, Fuping 等。有一些识别错误的，比如不该将 Province 识别为 LOCATION。相同的词在不同的语境中识别成了不同的实体，比如将 PRC 识别成了 COUNTRY, ORGANIZATION。而且运行速度比其他方法慢很多。

```
[('Xi', 'PERSON'), ('Jinping', 'PERSON'), ('', 'O'), ('male', 'O'), ('', 'O'), ('', 'O'), ('Han', 'O'), ('ethnicity', 'O'), ('', 'O'), ('was', 'O'), ('born', 'O'), ('in', 'O'), ('June', 'DATE'), ('1953', 'DATE'), ('and', 'O'), ('is', 'O'), ('from', 'O'), ('Fuping', 'O'), ('', 'O'), ('Shaanxi', 'STATE_OR_PROVINCE'), ('Province', 'LOCATION'), ('.', 'O'), ('He', 'O'), ('began', 'O'), ('his', 'O'), ('first', 'ORDINAL'), ('job', 'O'), ('in', 'O'), ('January', 'DATE'), ('1969', 'DATE'), ('and', 'O'), ('joined', 'O'), ('the', 'O'), ('Communist', 'ORGANIZATION'), ('Party', 'ORGANIZATION'), ('of', 'ORGANIZATION'), ('China', 'ORGANIZATION'), ('(', 'O'), ('CPC', 'ORGANIZATION'), (')', 'O'), ('in', 'O'), ('January', 'DATE'), ('1974', 'DATE'), ('.', 'O'), ('Xi', 'O'), ('graduated', 'O'), ('from', 'O'), ('School', 'ORGANIZATION'), ('of', 'ORGANIZATION'), ('Humanities', 'ORGANIZATION'), ('and', 'ORGANIZATION'), ('Social', 'ORGANIZATION'), ('Sciences', 'ORGANIZATION'), ('', 'O'), ('Tsinghua', 'ORGANIZATION'), ('University', 'ORGANIZATION'), ('where', 'O'), ('he', 'O'), ('completed', 'O'), ('an', 'O'), ('in-service', 'O'), ('graduate', 'O'), ('program', 'O'), ('in', 'O'), ('Marxist', 'IDEOLOGY'), ('theory', 'O'), ('and', 'O'), ('ideological', 'O'), ('and', 'O'), ('political', 'O'), ('education', 'O'), ('.', 'O'), ('He', 'O'), ('holds', 'O'), ('a', 'O'), ('Doctor', 'TITLE'), ('of', 'MISC'), ('Law', 'MISC'), ('degree', 'O'), ('.', 'O'), ('Xi', 'O'), ('is', 'O'), ('currently', 'DATE'), ('General', 'TITLE'), ('Secretary', 'O'), ('of', 'O'), ('the', 'O'), ('CPC', 'ORGANIZATION'), ('Central', 'ORGANIZATION'), ('Committee', 'ORGANIZATION'), ('', 'O'), ('Chairman', 'TITLE'), ('of', 'O'), ('the', 'O'), ('CPC', 'ORGANIZATION'), ('Central', 'ORGANIZATION'), ('Military', 'ORGANIZATION'), ('Commission', 'ORGANIZATION'), ('', 'O'), ('President', 'TITLE'), ('of', 'O'), ('the', 'O'), ('People', 'LOCATION'), ('s', 'LOCATION'), ('Republic', 'COUNTRY'), ('of', 'COUNTRY'), ('China', 'COUNTRY'), ('(', 'O'), ('PRC', 'COUNTRY'), (')', 'O'), ('', 'O'), ('and', 'O'), ('Chairman', 'TITLE'), ('of', 'O'), ('the', 'O'), ('PRC', 'ORGANIZATION'), ('Central', 'ORGANIZATION'), ('Military', 'ORGANIZATION'), ('Commission', 'ORGANIZATION'), ('.', 'O')]
```

## 中文工具

### • jieba

根据词性进行提取主要有两种方式

- `jieba.analyse.extract_tags()` : `extract_tags` 使用的是 `tf-idf` 指标
- `jieba.analyse.textrank()` : `textrank` 主要参考了 `PageRank` 的思想

- 对每个句子进行分词和词性标注处理
- 过滤掉除指定词性外的其他单词，过滤掉出现在停用词表的单词，过滤掉长度小于2的单词
- 将剩下的单词中循环选择一个单词，将其与其后面4个单词分别组合成4条边。

我们选择 `extract_tags` 进行测试

- 代码

```
def jieba_test(data):
    kw = jieba.analyse.extract_tags(data, withWeight=True,
    allowPOS=('v'))
    print(kw)
```

- 结果

因为是按照词性标注的结果进行提取的，所以也没法说命名实体识别的效果怎么样，因为这个效果严重依赖于分词结果。从结果来看，这些词确实都是动词，且是以短语的形式呈现的，识别效果还不错

```
[('扳回', 1.4378477974), ('远射', 1.4077025403), ('输给',
1.22169290836625), ('迎战', 1.10010140301875), ('未能',
0.80480783902625), ('进攻', 0.76395465629), ('结束',
0.66719444108375), ('进行', 0.4658905993775)]
```

- **StanfordCoreNLP:**

- 代码

```
def stanford_nlp(data):
    # __coding:utf-8__
    with StanfordCoreNLP(r'D:\stanford-corenlp-full-2018-02-27',
    lang='zh') as nlp:
        print(nlp.ner(data))
```

- 结果

- 优点：能实现一些基本的实体命名识别，比如 `DATE`，`COUNTRY`，`ORGANIZATION`，`ORDINAL`，`NUMBER` 等，对数字的识别比较精准
- 缺点：识别准确度不够，且识别的结果不稳定。在不同上下文中，相同的词会被识别成不同的结果。比如，将泰国识别为 `COUNTRY`，`MISC`。且对 `ORGANIZATION` 的识别也不准确，没有将 `国足`，`青年报` 等识别为 `ORGANIZATION`。识别效果依赖分词结果。



[('3月', 'DATE'), ('23日', 'DATE'), ('下午', 'TIME'), ('', ' ', 'O'), ('“', 'O'), ('青年报', 'O'), ('杯赛', 'O'), ('”', 'O'), ('U19', 'O'), ('邀请赛', 'O'), ('在', 'O'), ('越南', 'COUNTRY'), ('芽', 'O'), ('庄', 'O'), ('进行', 'O'), ('', ' ', 'O'), ('前', 'O'), ('国脚', 'O'), ('曲波', 'PERSON'), ('挂帅', 'O'), ('的', 'O'), ('中国', 'COUNTRY'), ('U19', 'O'), ('B', 'O'), ('队', 'O'), ('迎战', 'O'), ('泰国', 'COUNTRY'), ('U19', 'O'), ('', ' ', 'O'), ('上半场', 'ORGANIZATION'), ('国青队', 'ORGANIZATION'), ('的', 'O'), ('门户', 'O'), ('大开', 'O'), ('', ' ', 'O'), ('泰国', 'COUNTRY'), ('在', 'O'), ('第11', 'ORDINAL'), ('分钟', 'MISC'), ('和', 'O'), ('第17', 'ORDINAL'), ('分钟', 'MISC'), ('连', 'O'), ('进', 'O'), ('2', 'NUMBER'), ('球', 'O'), ('', ' ', 'O'), ('半场', 'MISC'), ('国青', 'MISC'), ('0', 'NUMBER'), ('射门', 'O'), ('0', 'NUMBER'), ('角球', 'O'), ('', ' ', 'O'), ('几乎', 'O'), ('被', 'O'), ('完全', 'O'), ('压制', 'O'), ('。', 'O'), ('下半场', 'O'), ('', ' ', 'O'), ('国青', 'O'), ('的', 'O'), ('进攻', 'O'), ('一度', 'O'), ('有所', 'O'), ('起色', 'O'), ('', ' ', 'O'), ('并', 'O'), ('由', 'O'), ('马辅渔', 'PERSON'), ('利用', 'O'), ('远射', 'O'), ('扳回', 'O'), ('一', 'NUMBER'), ('球', 'O'), ('', ' ', 'O'), ('但', 'O'), ('最终', 'O'), ('未', 'O'), ('能', 'O'), ('扳平', 'O'), ('比分', 'O'), ('。', 'O'), ('全', 'O'), ('场', 'O'), ('比赛', 'O'), ('结束', 'O'), ('', ' ', 'O'), ('国青', 'O'), ('1-2', 'NUMBER'), ('输球', 'O'), ('', ' ', 'O'), ('继', 'O'), ('中国', 'COUNTRY'), ('杯', 'O'), ('国足', 'O'), ('0-1', 'MISC'), ('输给', 'MISC'), ('泰国', 'MISC'), ('之后', 'MISC'), ('', ' ', 'O'), ('3', 'NUMBER'), ('天', 'MISC'), ('内', 'MISC'), ('遭遇', 'O'), ('泰国', 'COUNTRY'), ('足球', 'O'), ('双杀', 'O'), ('。', 'O')]

## • SnowNLP

### • 代码

因为没有提供命名实体识别功能，我们根据词性标注的结果自行实现，按 词性 分类

```
def snow_nlp(data):
    s = SnowNLP(data)
    ans = []
    ent = ['n'] # 需要提取的实体种类
    for i in s.tags:
        if i[1] in ent:
            ans.append(i)
    print(ans)
```

### • 结果（识别常见出现的常见名词）

- 优点：部分识别效果比较准确，像一般的 青年报，邀请赛 等识别结果都正确。

- 缺点：有很多识别错误的和遗漏识别的。上次实验我们也提到过，SnowNLP 因为分词效果不好，导致词性标注的效果也不是很理想，从而又引起了命名实体识别结果不理想。**单个字太多，甚至无法识别出正常的短语，长一些的名词**。更不用说，将这些词的词性，实体识别准确。而且遗漏了 比赛 等常见的名词，在 SnowNLP 中是将其当做动词来标注的。

```
[('3', 'ns'), ('月', 'n'), ('青年报', 'n'), ('邀请赛', 'n'), ('越南', 'ns'), ('芽', 'n'), ('庄', 'nr'), ('国', 'n'), ('脚曲', 'nr'), ('波', 'nr'), ('中国', 'ns'), ('队', 'n'), ('泰国', 'ns'), ('上半场', 'n'), ('国', 'n'), ('门户', 'n'), ('泰国', 'ns'), ('连', 'n'), ('球', 'n'), ('国', 'n'), ('青', 'nr'), ('0', 'nr'), ('角球', 'n'), ('下半场', 'n'), ('国', 'n'), ('起色', 'n'), ('马', 'nr'), ('辅', 'nr'), ('球', 'n'), ('比分', 'n'), ('全场', 'n'), ('国', 'n'), ('青', 'nr'), ('1-2', 'nr'), ('球', 'n'), ('中国', 'ns'), ('国', 'n'), ('泰国', 'ns'), ('泰国', 'ns'), ('足球', 'n')]
```

## • THULAC

因为 THULAC 并未提供自带的命名实体识别方法，我们根据其词性标注结果手动实现命名实体识别。

- 代码

```
def thulac_nlp(data):
    thu1 = thulac.thulac() # 默认模式
    text = thu1.cut(data, text=True) # 进行一句话分词
    text = text.split(' ')
    ent = ['np'] # 调整命名实体种类
    ans = [] # 存储结果
    for i in text:
        i = i.split('_')
        if i[1] in ent:
            ans.append(i)
    print(ans)
```

- 结果

- 优点：可以区分人名，机构名，量词等。总体效果十分不错。
- 缺点：分词效果太细致，导致词性标注太细致，容易将原本是一个词性的词分开标注。部分词性标注不准确，将 0 标注为 v。容易分不清副词和形容词。部分专有名词被识别成了普通名词，如 国青赛。
- np
  - 优点：识别出来的结果是准确的，曲波，马辅渔 都是不太容易识别的，没有发现遗漏识别情况

```
[['曲波', 'np'], ['马辅渔', 'np']]
```

- ni

- 优点：识别出的结果是正确的，没有发现遗漏识别的情况

```
[['国青队', 'ni']]
```

- ns

- 优点：识别结果基本准确，能够识别出 芽庄 这种不太容易识别的地点
- 缺点：出现了 杯国 这种明显不正确的实体，应该是由不正确的分词导致的

```
[['越南', 'ns'], ['芽庄', 'ns'], ['中国', 'ns'], ['泰国', 'ns'],  
['泰国', 'ns'], ['中国', 'ns'], ['杯国', 'ns'], ['泰国', 'ns'], ['泰  
国', 'ns']]
```

## • NLPIR

因为 NLPIR 并未提供自带的命名实体识别方法，我们根据其词性标注结果手动实现命名实体识别。

- 代码

```
def pynlpir_nlp(data):  
    pynlpir.open()  
    tagged = pynlpir.segment(data)  
    ent = ['noun']  
    ans = []  
    for i in tagged:  
        if i[1] in ent:  
            ans.append(i)  
    print(ans)
```

- 结果

- 优点：可以看出，准确度是比较不错的，识别为名词的基本都是名词。而且标注方式为正常的英文，而不是自定义的词性表，看起来非常易懂。
- 缺点：由于分词效果比较差，从而导致了词性标注和命名实体识别的效果也比较差。出现了 马,杯 这种单个名词，明显错误。

```
[('杯', 'noun'), ('U19', 'noun'), ('邀请赛', 'noun'), ('越南', 'noun'), ('芽', 'noun'), ('庄', 'noun'), ('国脚', 'noun'), ('曲波', 'noun'), ('中国', 'noun'), ('U19B', 'noun'), ('队', 'noun'), ('泰国', 'noun'), ('U19', 'noun'), ('上半场', 'noun'), ('国青', 'noun'), ('队', 'noun'), ('门户', 'noun'), ('泰国', 'noun'), ('球', 'noun'), ('国青', 'noun'), ('角球', 'noun'), ('国青', 'noun'), ('起色', 'noun'), ('马', 'noun'), ('球', 'noun'), ('比分', 'noun'), ('全场', 'noun'), ('国青', 'noun'), ('球', 'noun'), ('中国', 'noun'), ('杯', 'noun'), ('国', 'noun'), ('泰国', 'noun'), ('泰国', 'noun'), ('足球', 'noun')]
```

# Bert+BiLstm+CRF

## 1.前言

- 下面主要分析本次实验的代码，讲解主要流程和代码含义，而关注参数的选择和模型的选择
- 若分析过程出现错误，请及时指正，谢谢

## 2.数据预处理

- 原始标注：对句子中的每个字标注上一个标签，可以简单地看成是直接对每个字分类（需要融合上下文信息），因此可以使用一个多分类器，分类器输出类别就是该字的标签
- 联合标注：对一串连续的字标注相同的标签。在NER任务中，实体由一个或多个字组成，所以它属于联合标注任务。

但是在联合标注中，相邻词语标签之间可能会存在依赖关系。**这一问题可以通过标签转化的方式，把联合标注转化成原始标注解决。**

我们这里使用的是 BIOS标注

标签	含义
B-X	该字是词片段 X 的起始字
I-X	该字是词片段 X 起始字之后的字
S-X	该字单独标记为 X 标签
O	该字不属于事先定义的任何词片段类型

在 `process.py` 中，我们将 `.json` 文件中的语句和标签，按照 `BIOS` 方式，处理转换成了 `.npz` 文件。主要代码如下。分析过程写在注释中，依据样例 `.json`。

```
text = json_line['text']
words = list(text) # 自动将句子按字符分开
# 如果没有label，则返回None
label_entities = json_line.get('label', None) # 参照下面的例子，该项对应 label 之后的内容
labels = ['0'] * len(words) # [len(words) 个 '0'] 都初始化为 `0`

if label_entities is not None:
    for key, value in label_entities.items(): # key 对应 name 和 company, value 对应后面存储内容
        for sub_name, sub_index in value.items(): # sub_name 对应 叶老桂等, sub_value 对应后面的索引
            for start_index, end_index in sub_index: # 对应列表中的两个数,是标签开始和结束的位置
                assert ''.join(words[start_index:end_index + 1]) == sub_name

                if start_index == end_index: # 单个字作为索引
                    labels[start_index] = 'S-' + key
                else:
                    labels[start_index] = 'B-' + key # 开头
                    labels[start_index + 1:end_index + 1] = ['I-' + key] * (len(sub_name) - 1) # 中间的字
```

- 字符串转 `list` 验证

- 这里很重要的一点是，输入的字符串都转成单字符了，下面使用 `tokenize` 的时候会看到为什么

```
a = "你好,我是nsy,哈哈"
print(list(a))
>>['你', '好', ',', '我', '是', 'n', 's', 'y', ',', '哈', '哈', '哈']
```

`.json` 文件中，数据存储结构如下所示

```
{
  "text": "浙商银行企业信贷部叶老桂博士则从另一个角度对五道门槛进行了解读。叶老桂认为，对目前国内商业银行而言，",
  "label": {
    "name": {
      "叶老桂": [
        [9, 11],
        [32, 34]
      ]
    },
  },
}
```

```

        "company": {
            "浙商银行": [
                [0, 3]
            ]
        }
    }
}

```

## • 2.1本地查看数据转换后的结果

- code

```

import numpy as np
a = np.load(r'D:\2022 spring\nlp\exp4\code\BERT-LSTM-CRF\data\clue\test.npz', allow_pickle=True)
index = 0
words = a['words']
labels = a['labels']
print(words[0])
print(labels[0])

```

- 结果

```

['彭', '小', '军', '认', '为', ' ', ' ', '国', '内', '银', '行', '现',
'在', '走', '的', '是', '台', '湾', '的', '发', '卡', '模', '式',
', ', '先', '通', '过', '跑', '马', '圈', '地', '再', '在', '圈',
'的', '地', '里', '面', '选', '择', '客', '户', ' ', ' ']
['B-name', 'I-name', 'I-name', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'B-address', 'I-address', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']

```

## 3.构建数据集

我们构建自己的数据集 `Dataset` 类。该类主要属性为

```

self.tokenizer = BertTokenizer.from_pretrained(config.bert_model,
do_lower_case=True)
self.label2id = config.label2id
self.id2label = {_id: _label for _label, _id in
list(config.label2id.items())}
self.dataset = self.preprocess(words, labels)
self.word_pad_idx = word_pad_idx # 起初始化作用的
self.label_pad_idx = label_pad_idx # 起初始化作用的
self.device = config.device

```

- 因为我们加载的数据是 `.npz` 文件中的，数据（不是 `label`）是存在列表中的单个字符，我们不进行分词工作了。所以这里的 `tokenizer` 属性主要是将大写字母转化为小写字母
- 一个比较难理解的属性是 `self.dataset`，我们来看看里面到底是什么内容。`preprocess` 函数如下。函数主要功能为
  - 在每句话前面加一个开头 `CLS`
  - 将原始字符/字都转换成 `id`，并存储有 `label` 的字的开始位置的索引
  - 将 `label` 转成 `id`
  - 注意：代码中 `token` 的长度都是 `1`，这是由 `.npz` 中的数据作为输入决定的

```

def preprocess(self, origin_sentences, origin_labels): # 输入
    的是 .npz 里面的数据
    """
    Maps tokens and tags to their indices and stores them in
    the dict data.
    examples:
        word: ['[CLS]', '浙', '商', '银', '行', '企', '业', '信',
'贷', '部']
        sentence: ([101, 3851, 1555, 7213, 6121, 821, 689, 928,
6587, 6956],
                    array([ 1,  2,  3,  4,  5,  6,  7,  8,  9,
10]))
        label: [3, 13, 13, 13, 0, 0, 0, 0, 0]
    """
    data = []
    sentences = []
    labels = []
    for line in origin_sentences: # 处理每一句话,类型为 list
        words = []
        word_lens = []
        for token in line: # 一句话中的每个词
            words.append(self.tokenizer.tokenize(token)) #
tokennize结果:'浙'->['浙']
            word_lens.append(len(token)) # len(token) 全是 1
        # 开头加上[CLS]

```

```

        words = ['[CLS]'] + [item for token in words for item
in token] # token 是字符列表, item 是 token 中的项, 也就是单个字
        token_start_idx = 1 + np.cumsum([0] + word_lens[:-1])
        # 除了 `[CLS]` 之外的索引, 写成一个列表

    sentences.append((self.tokenizer.convert_tokens_to_ids(words),
token_start_idx)) # 将 token 的 id 和 index 一起加入 sentences

    for tag in origin_labels: # tag 是每一行的
origin_sentences 中的字对应的 label
        label_id = [self.label2id.get(t) for t in tag] # 每个
字的 label -> id
        labels.append(label_id)
    for sentence, label in zip(sentences, labels):
        data.append((sentence, label))
    return data # 作为 self.dataset

```

#### ○ 比较难理解的部分

```

for token in line:
    words.append(self.tokenizer.tokenize(token))
    word_lens.append(len(token))
words = ['[CLS]'] + [item for token in words for item in
token]
token_start_idx = 1 + np.cumsum([0] + word_lens[:-1])

```

- 对上面的例子来说, `tokenize` 效果就是 浙 -> ['浙']
  - `tokenize` 其实有分词的作用, 比如

```

import torch
import numpy as np
from transformers import BertTokenizer

tokenizer =
BertTokenizer.from_pretrained('pretrained_bert_models/b
ert-base-chinese/', do_lower_case=True)
print(tokenizer.tokenize("unwanted"))
print(tokenizer.tokenize("===+"))
>>['u', '##n', '##wan', '##ted']
>>['=', '=', '=', '+']

```

- 如果有上述功能, `len(token)` 和 真实索引开始位置就对不上了, 比如 `unwanted` 进行 `tokenize` 之后提供四个部分, 但是索引却要 + 8
- 这里的字符全是单个的 (上面解释过), 因此只有大写 -> 小写的 作用



- for 循环之后, 得到 `words = [['浙'], ['商'], ['银'], ['行'], ['企'], ['业'], ['信'], ['贷'], ['部']]`, 大写变小写在这里没有体现。 `word_lens=[1,1,1,1,1,1,1,1,1]`
- 下一步 `words->['[CLS]', '浙', '商', '银', '行', '企', '业', '信', '贷', '部']`
- `word_lens` 去掉最后一个, 前面添加一个 0, 然后前向求和 + 1, 得到 `[ 1, 2, 3, 4, 5, 6, 7, 8, 9]`。我认为样例的数字错了 (代码没问题, 本人已经测试过)。我感觉直接对 `word_lens` 前向求和就行
- 该部分还有一个主要函数是 `collate_fn(self, batch)`。主要功能为:
  - 将每个 `batch` 的 `data` 扩充到同一长度 ( `batch` 中最长的 `data` 的长度)
    - 先找到最大的长度
    - 初始化一个矩阵 (句子个数, 最大句子长度), 初始化为 0
    - 将相应的值放到对应的索引上
  - 将每个 `batch` 的 `label` 扩充到统一长度 ( `batch` 中最长的 `label` 的长度)
    - 先找到最大的长度
    - 初始化一个矩阵 (句子个数, 最大 `label` 长度)
    - 将相应的值放到对应的索引上
  - 将 `batch_data`, `batch_label_starts`, `batch_labels` 转换为 `tensor` 并移动到 GPU 上, 然后返回

## 4.数据集分割

我们按照 9:1 的比例, 将训练数据分割成训练集和验证集, 代码在 `run.py` 中。

```
# 分离出验证集
word_train, word_dev, label_train, label_dev = load_dev('train')
```

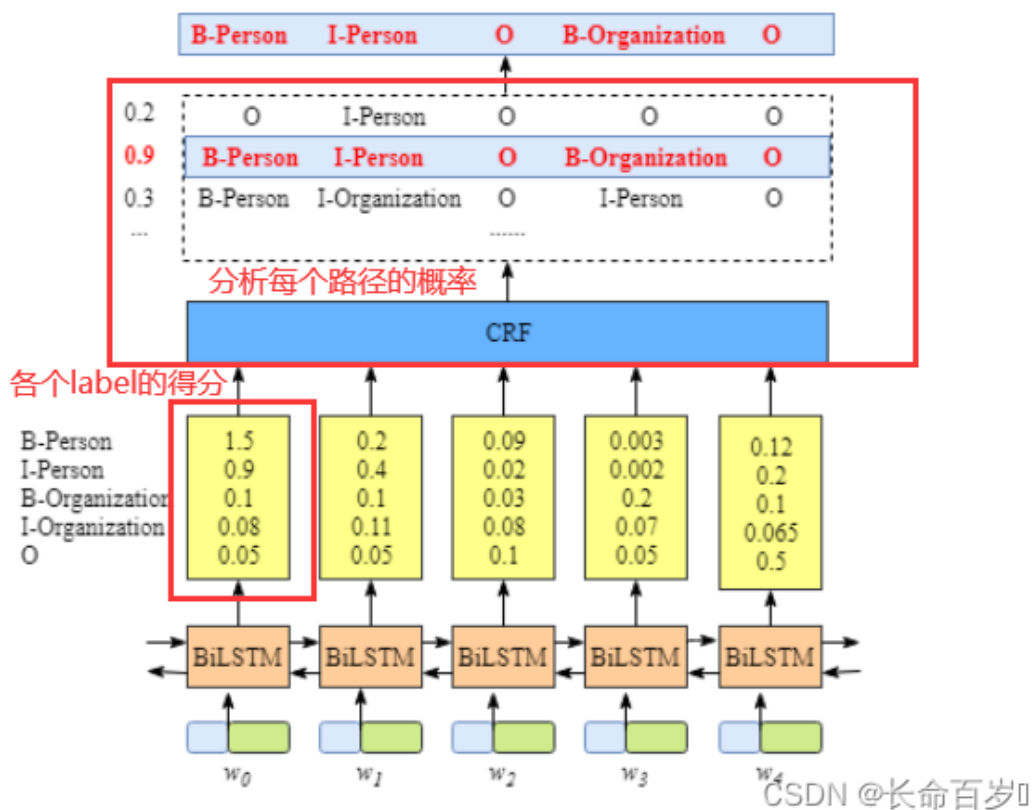
函数 `load_dev()` 代码如下

```
def dev_split(dataset_dir): # 分出训练集和验证集 参数: BERT-LSTM-CRF/data/clue/train.npz
    """split dev set"""
    data = np.load(dataset_dir, allow_pickle=True)
    words = data["words"]
    labels = data["labels"]
    x_train, x_dev, y_train, y_dev = train_test_split(words, labels,
        test_size=config.dev_split_size, random_state=0) # 测试集大小为 0.1
    return x_train, x_dev, y_train, y_dev
```

## 5.模型架构

### • 5.1模型初始化

- 我们的模型继承了一个预训练模型 BertPreTrainedModel
- 主要属性：
  - 一个 bert 模型 (Transformer 的堆叠, bert 作为 Encoding 来使用, 对输入数据进行编码)
  - dropout 层
  - 一个两层的 bilstm(双向lstm) : 输出
  - 一个线性分类器
  - 一个 crf 模型
  - bilstm-CRF 模型结构如下所示, 代码下面有各层的作用



```
class BertNER(BertPreTrainedModel):
    def __init__(self, config):
        super(BertNER, self).__init__(config)
        self.num_labels = config.num_labels # label 的数目




        self.bert = BertModel(config) # 定义 bert 模型
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.bilstm = nn.LSTM(
```

```

        input_size=config.lstm_embedding_size, # 1024
        hidden_size=config.hidden_size // 2, # 1024
        batch_first=True,
        num_layers=2,
        dropout=config.lstm_dropout_prob, # 0.5
        bidirectional=True
    )
    self.classifier = nn.Linear(config.hidden_size,
config.num_labels)
    self.crf = CRF(config.num_labels, batch_first=True)

    self.init_weights()

```

组件名称	说明	维度说明例子
输入	假设输入 batch 是四句话，每句话 6 个字	输入的整数数组 shape=[4,6]
Embedding 层	把每个字符转化成一个向量  用的是BERT	如果 embedding size=512, 输出 shape=[4,6,512]
BiLSTM 层	由于结合“上”“下”文可以得到更准确的标注，所以使用了双向 LSTM	假设 LSTM 中神经元为 100 个，正向 LSTM 输出 shape=[4,6,100]，反向 LSTM 输出 shape=[4,6,100]，二者合并，总的输出 shape=[4,6,200]
Encode 层	可以用一个普通线性层，把高维向量映射成符合标签的维度。  得到的是发射概率	假设标签维度为 12，此处的层为一个[200,12]的向量，输出的 shape=[4,6,12]
CRF 层	CRF 层的本质是一个方形矩阵，它表示标签之间的状态转移信息，也就是 CRF 的特征函数集合  得到的是转移概率	假设标签维度为 12，此处的层为一个[12,12]的矩阵，通过此层得到 CRF loss

## • 5.2前向传播过程

- 先利用 bert 处理输入数据
  - 输入是每个 token 对应的表征
  - 输出是对输入 token 的编码

```

input_ids, input_token_starts = input_data # 训练数据，已经扩充到最大维度的
outputs = self.bert(input_ids, # 用 bert 处理
                    attention_mask=attention_mask,
                    token_type_ids=token_type_ids,
                    position_ids=position_ids,
                    head_mask=head_mask,
                    inputs_embeds=inputs_embeds)
sequence_output = outputs[0]

```

- 将原来有 label 的位置对应的输出提取出来

```
# 去除[CLS]标签等位置，获得与label对齐的pre_label表示
origin_sequence_output = [layer[starts.nonzero().squeeze(1)]
                           for layer, starts in
                           zip(sequence_output, input_token_starts)]
```

- 将 `origin_sequence_output` 填充到最大长度

```
# 将sequence_output的pred_label维度padding到最大长度
padded_sequence_output = pad_sequence(origin_sequence_output,
                                       batch_first=True)
```

- 将 `padded_sequence_output` 输入 `bilstm`

```
# dropout pred_label的一部分feature
padded_sequence_output = self.dropout(padded_sequence_output) #
遮住一部分
lstm_output, _ = self.bilstm(padded_sequence_output)
```

- 进行结果的判别，返回结果

```
# 得到判别值
logits = self.classifier(lstm_output)
outputs = (logits,)
if labels is not None:
    loss_mask = labels.gt(-1) # 我们在对labels长度填充的时候,初始化
    值为 -1, 这里是遮住填充的位置
    loss = self.crf(logits, labels, loss_mask) * (-1)
    outputs = (loss,) + outputs
# contain: (loss), scores
return outputs
```

## 6.模型训练

### • 6.1训练一个epoch

- 首先开启训练模式，本次实验中其实就是开启 `dropout`。关于这样做的理由，请参考 [Pytorch model.train\(\)\\_长命百岁的博客-CSDN博客](#)

```
# set model to training mode
model.train() # 开启训练模式，为了避开测试模式的影响
```

- 利用 `Dataloader` 类的实例 `train_loader` 进行分批训练（一次训练一个 `batch`），`train_epoch` 代码如下：

```

for idx, batch_samples in enumerate(tqdm(train_loader)): # tqdm
    是加了一个进度条
    batch_data, batch_token_starts, batch_labels = batch_samples
    batch_masks = batch_data.gt(0) # token 是用 0 初始化的,
    # 前向传播, 计算结果并产生 loss
    loss = model((batch_data, batch_token_starts),
                  token_type_ids=None, attention_mask=batch_masks,
                  labels=batch_labels)[0]
    train_losses += loss.item()
    # 梯度归0, 反向传播
    model.zero_grad()
    loss.backward()
    # 梯度裁剪, 梯度爆炸的裁剪掉
    nn.utils.clip_grad_norm_(parameters=model.parameters(),
                             max_norm=config.clip_grad)
    # 更新
    optimizer.step()
    scheduler.step()

```

- 这里的 `mask` 是因为我们对一句话进行了 `padding`, `self-attention` 会关注所有位置, 但是我们不想关注 `padding` 的位置。因此我们就提取出来这些位置 (为 0), 然后进行 `mask`。
- 返回结果

```

train_loss = float(train_losses) / len(train_loader)
logging.info("Epoch: {}, train loss: {}".format(epoch,
train_loss))

```

## • 6.2 训练所有 epoch

- 遍历 `epoch`, 调用 `train_epoch` 进行参数更新和 `loss` 计算

```

for epoch in range(1, config.epoch_num + 1): # 遍历 epoch
    train_epoch(train_loader, model, optimizer, scheduler, epoch)
    val_metrics = evaluate(dev_loader, model, mode='dev') #
    evaluate是自定义函数
    val_f1 = val_metrics['f1']

```

- 根据 `f1_score` 的变化考虑是否保存当前模型, 并设置停止训练的条件, 若满足条件, 则停止训练。

## • 6.3 evaluate 函数

在这里, `mode = 'dev'`。利用当前 `epoch` 的模型对验证集进行预测, 计算出 `metrics['loss'] = float(dev_losses) / len(dev_loader)`。并利用预测 `label` 与真实 `label` 计算出 `f1_score = metrics['f1']`。

- 要注意的是, 我们调用 `model` 函数前向传播时, 有的输入了 `label`, 然后接收 `output[0]`, 是 `loss`
- 有的没输入 `label`, 返回的结果是每个位置对所有 `label` 的得分

## 7. 整体训练过程

- 数据预处理

```
# set the logger
utils.set_logger(config.log_dir)
logging.info("device: {}".format(config.device))
# 处理数据, 分离文本和标签
processor = Processor(config)
processor.process()
logging.info("-----Process Done!-----")
```

- 划分训练集和验证集, 并使用上面构建的 `Dataset` 类, 构建数据集 (可用于 `Dataloader`)

```
# 分离出验证集
word_train, word_dev, label_train, label_dev = load_dev('train')
# build dataset
train_dataset = NERDataset(word_train, label_train, config) # 训练数据
dev_dataset = NERDataset(word_dev, label_dev, config) # 验证数据
logging.info("-----Dataset Build!-----")
# get dataset size
train_size = len(train_dataset)
```

- 将 `Dataset` 类放入 `DataLoader` 中, 以进行后续的分 `batch` 训练

```
# build data_loader
train_loader = DataLoader(train_dataset,
batch_size=config.batch_size, # 训练集的 DataLoader
shuffle=True, collate_fn=train_dataset.collate_fn)
dev_loader = DataLoader(dev_dataset, batch_size=config.batch_size,
# 验证集的 DataLoader
shuffle=True, collate_fn=dev_dataset.collate_fn)
logging.info("-----Get Dataloader!-----")
```

- 准备模型

```
device = config.device # 选择设备,这里选的 GPU
model = BertNER.from_pretrained(config.bert_model,
num_labels=len(config.label2id)) # 读取预训练模型
model.to(device) # 将模型移动到 GPU 上
```

- 下面就是模型的参数选择，优化器的选择，调优策略的配置
- 模型训练，保存最优模型
- 模型测试
- 上面三个内容可以参见[用BERT做NER? 教你用PyTorch轻松入门Roberta! - 知乎 \(zhihu.com\)](https://zhuanlan.zhihu.com/p/101111111)。本文只对代码内容进行讲解，不研究训练的参数选择。

## 8.训练结果 (50 epoch)

```
Epoch: 40, train loss: 0.874740464852588  
Epoch: 40, dev loss: 817.9405167523553, f1 score: 0.7881613622542064  
100%|██████████  
Epoch: 41, train loss: 1.1204660843701253  
Epoch: 41, dev loss: 814.3854747099035, f1 score: 0.7880633373934226  
100%|██████████  
Epoch: 42, train loss: 1.0356536977755355  
Epoch: 42, dev loss: 819.2944461598116, f1 score: 0.7917852785685239  
100%|██████████  
Epoch: 43, train loss: 0.9158407107438191  
Epoch: 43, dev loss: 833.570400462431, f1 score: 0.7860919072793819  
100%|██████████  
Epoch: 44, train loss: 0.6764180054365605  
Epoch: 44, dev loss: 841.9643303366269, f1 score: 0.7859463850528025  
Best val f1: 0.7941296371789645  
Training Finished!
```

```
Model config {
  "attention_probs_dropout_prob": 0.1,
  "directionality": "bidi",
  "finetuning_task": null,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "is_decoder": false,
  "layer_norm_eps": 1e-12,
  "lstm_dropout_prob": 0.5,
  "lstm_embedding_size": 768,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "num_labels": 31,
```

```
"output_ attentions": false,  
"output_hidden_states": false,  
"output_past": true,  
"pooler_fc_size": 768,  
"pooler_num_attention_heads": 12,  
"pooler_num_fc_layers": 3,  
"pooler_size_per_head": 128,  
"pooler_type": "first_token_transform",  
"pruned_heads": {},  
"torchscript": false,  
"type_vocab_size": 2,  
"use_bfloat16": false,  
"vocab_size": 21128  
}
```

-----Bad Cases reserved !-----

test loss: 804.8633270263672, f1 score: 0.779121578612349

f1 score of address: 0.6111833550065019

f1 score of book: 0.7820512820512822

f1 score of company: 0.7854356306892067

f1 score of game: 0.831715210355987

f1 score of government: 0.8122605363984674

f1 score of movie: 0.8160535117056857

f1 score of name: 0.865546218487395

f1 score of organization: 0.7903225806451613

f1 score of position: 0.7895335608646189

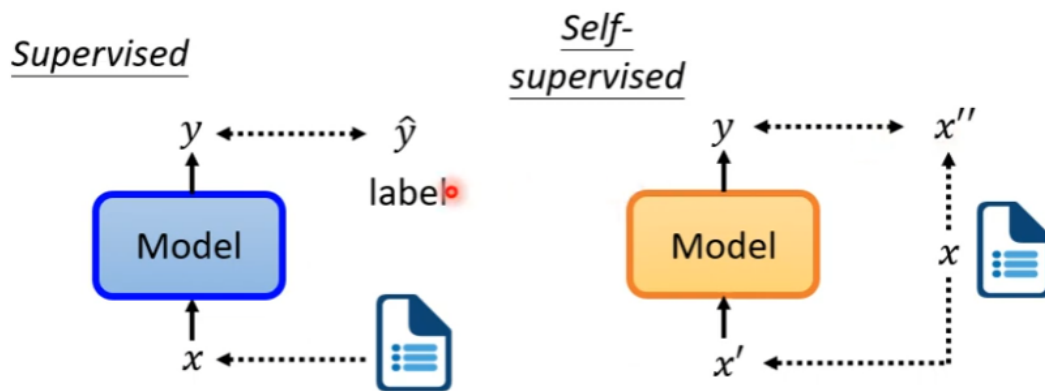
f1 score of scene: 0.6904761904761905

# Bert理论部分

## 1.前言



# Self-supervised Learning



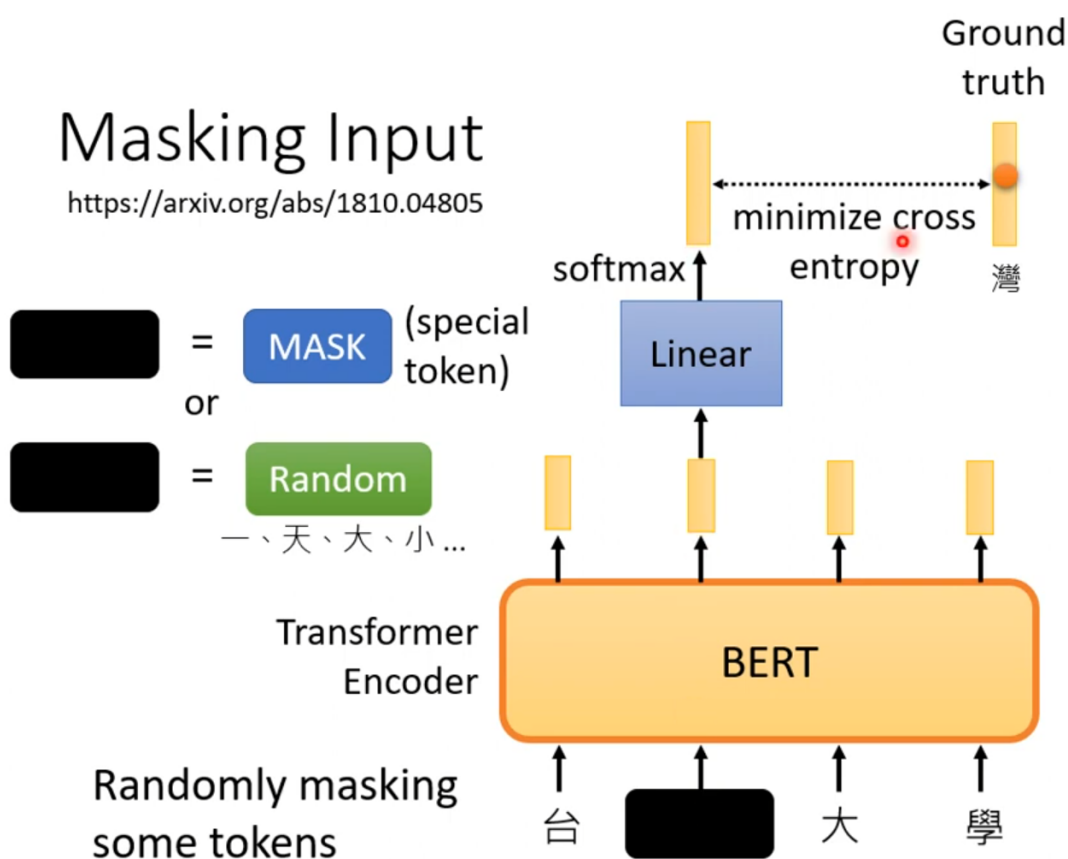
- 监督学习：给定训练数据  $x$  和 标签  $y$ 。使得 Model 对  $x$  的输出越接近  $y$  越好
- 自监督学习：没有标签  $y$ ，我们将数据分成两份，使得 Model 对其中一部分的输出越接近另一部分越好

BERT 就是自监督学习，利用非 Mask 的来预测 Mask 的部分

## 2.Bert

### • 2.1主要任务

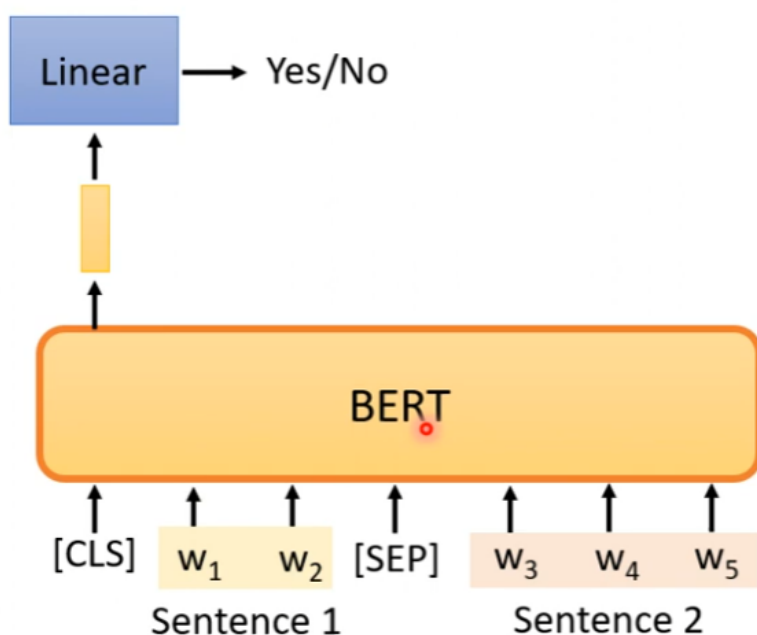
#### - 2.1.1.完形填空



- 随机选择一些输入进行遮挡
  - 被遮住可以是将字换成了 `MASK`：一个特殊的字符，代表被盖住，不在字典里出现
  - 可以是利用字典中的其他字随机替换
- 被盖住的字输出的结果经过一个线性层，然后做 `softmax` 得到结果，这个结果和真实字的 `one-hot` 编码越接近越好

### 2.1.2. 预测下一个句子

## Next Sentence Prediction



- `[CLS]` 放在一个序列的最前面，没有特别具体的语义。我们通过 `[CLS]` 对应的输出，来判断两个句子是不是连接在一起的。

### 2.2. 输入表示

原文说：我们的输入表示能够在 `token` 序列中明确表示单个句子 和 一对句子。一个“句子”可以是连续文本的任意跨度，而不是一个实际的语言句子。

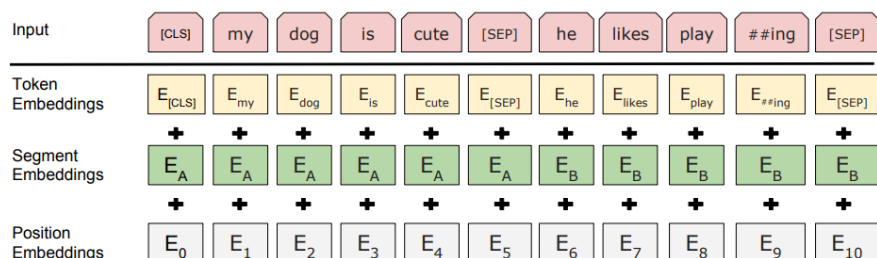


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

上图是论文中的示例。比如输入的一个序列包含两个句子，“my dog is cute”，“he likes playing”。

- 我们在序列的最前面加上一个特殊的 Token [CLS]，其在最后一个隐藏层的隐藏状态被当做整个序列的聚合表示，从而应用于分类任务
- 在一个句子的末尾加上一个 [SEP] 表示一个句子的结束，以区分不同的句子

我们在进行了分词，添加了特殊 Token 之后的序列作为模型的输入。

接着对每个Token进行3个Embedding：词的Embedding，Segment的Embedding和位置的Embedding

- 对词的 Embedding 就是将词表示成向量，使用 WordPiece embedding
- Segment Embedding 就是给词打上标签，代表其属于哪个句子。这里使用的是学习好的 Embedding
- 位置的 Embedding 就是对其位置进行编码

## • 2.3.MLM (Mask Language Model)

BERT 使用的是 MASK 语言模型。Mask语言模型有点类似与完形填空——给定一个句子，把其中某些词遮挡起来，让模型猜测可能的词。

BERT 对输入序列随机进行 15% Mask，然后预测这些被遮住的 Token。通过调整模型的参数使得模型预测正确的概率尽可能大。

但是这有一个问题：在 Pretraining Mask LM 时会出现特殊的Token [MASK]，但是在后面的fine-tuning时却不会出现，这会出现Mismatch的问题。为了减轻这种不匹配，在BERT中，如果某个Token在被选中的15%个Token里，则按照下面的方式随机的执行

- 80%的概率替换成[MASK]，比如my dog is hairy → my dog is [MASK]
- 10%的概率替换成随机的一个词，比如my dog is hairy → my dog is apple
- 10%的概率替换成它本身，比如my dog is hairy → my dog is hairy，就是不变

## • 2.4.NSP

在有些任务中，比如问答，前后两个句子有一定的关联关系，我们希望BERT Pretraining的模型能够学习到这种关系。因此BERT还增加了一个新的任务——预测两个句子是否有关联关系。

这通过为 BERT 添加一个任务来实现：我们预训练了一个二分类的 是否是下一个句子的任务。

当选择句子 A 和 B 作为预训练样本时，B 有 50% 的概率是 A 的真实的下一句，也有 50% 的概率是随机的一句。

比如下面两个句子是相关的

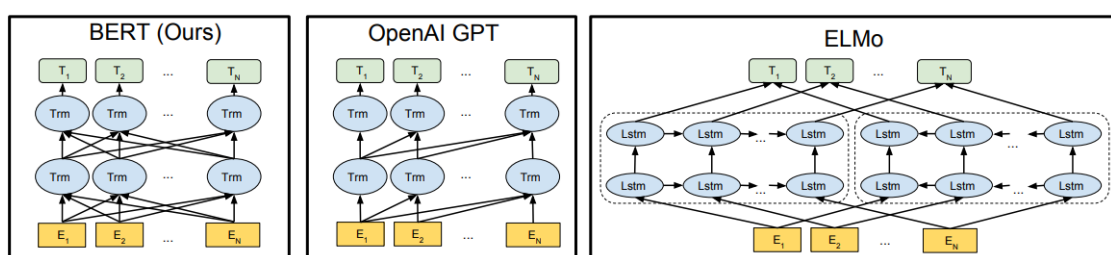
[CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk [SEP]

下面两个句子是不相关的

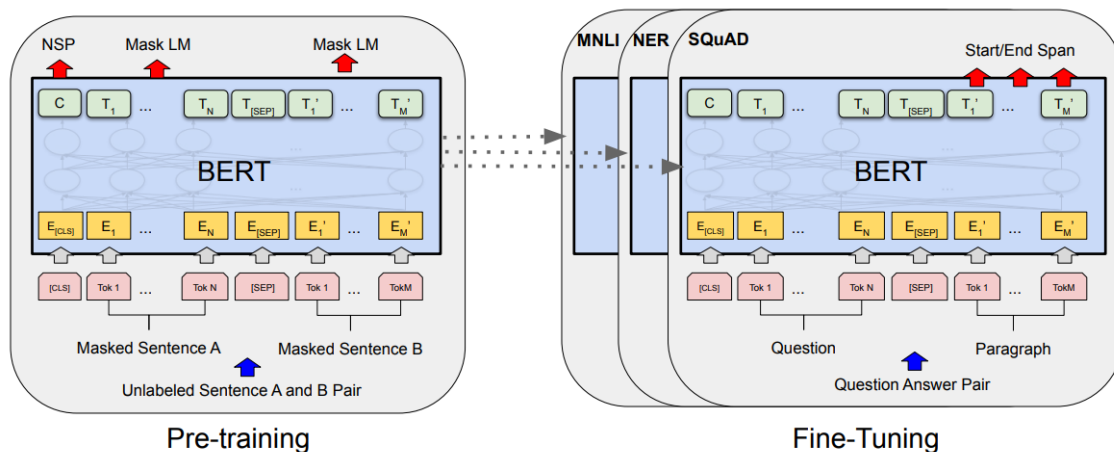
[CLS] the man [MASK] to the store [SEP] penguin [MASK] are flight ##less birds [SEP]

## • 2.5.总体架构

BERT的架构是多层的双向 Transformer encoder。我们定义层数（Transformer blocks）为  $L$ ，hidden size 为  $H$ ，self-attention 的头的个数为  $A$ 。一共提供了两版 BERT



- $BERT_{BASE}(L = 12, H = 768, A = 12)$ ，总参数量为 110M
  - BASE 版本的模型大小和 OpenAI GPT相同，只不过使用了双向的 self-attention
- $BERT_{LARGE}(L = 24, H = 1024, A = 16)$ ，总参数量为 340M



- $E$  代表 对输入进行 Embedding 的结果
- $C$  是  $[CLS]$  在最后一个隐藏层对应的隐藏状态
- $T$  是其它位置对应的隐藏状态

### 3.Fine-tuning

Fine-tuning是简单的，因为Transformer中的 self-attention 机制提供了 BERT 对许多下游任务进行建模的能力，无论他们包含单个文本还是文本对。我们只需要替换掉输入和输出就好了。

对于每个任务，我们只需将特定于任务的输入和输出插入到BERT中，并对所有参数进行端到端微调。

在输出端，token 表示被输入到输出层用于 token 级任务，如序列标记或问题回答，而  $[CLS]$  表示被输入到输出层用于分类，如逻辑蕴涵和情感分析

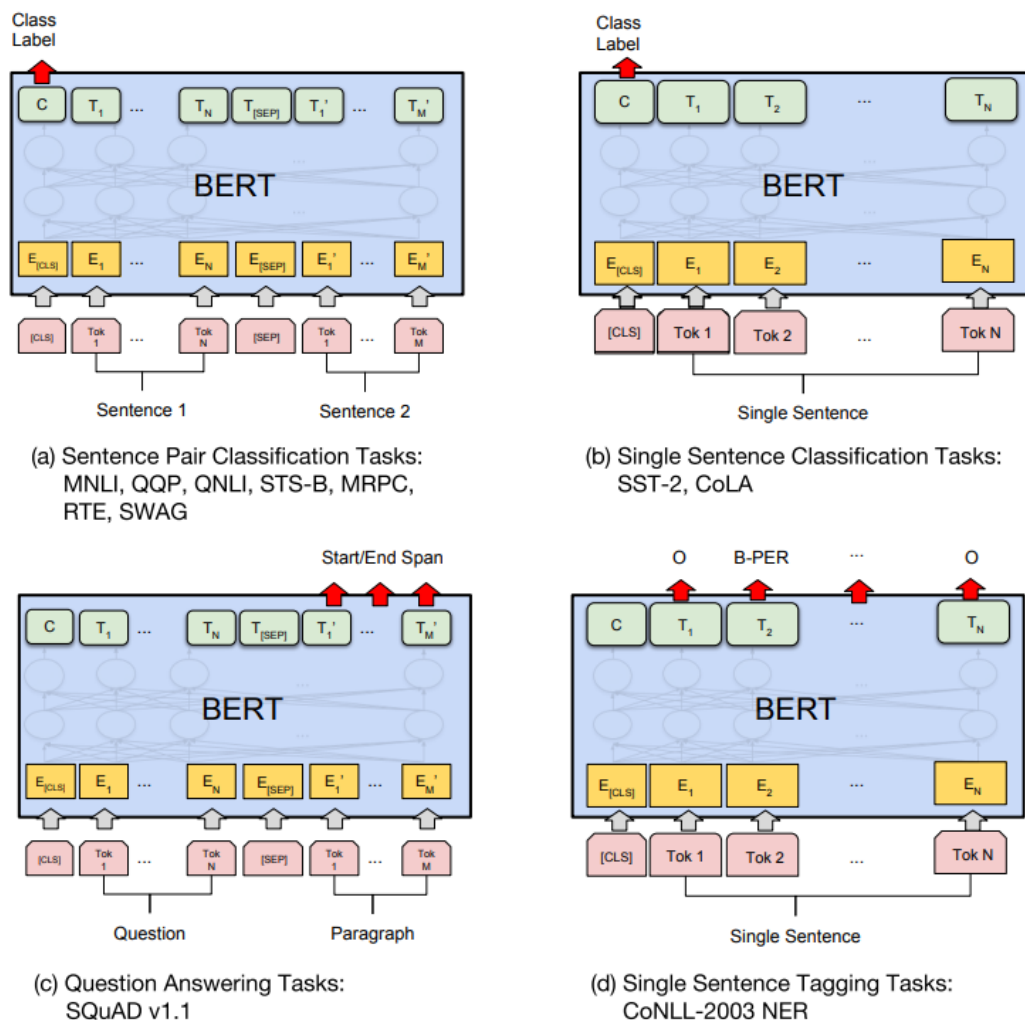


Figure 4: Illustrations of Fine-tuning BERT on Different Tasks.

- 对于相似度计算等任务，输入是两个序列，如任务 (a) 那样。我们用第一个特殊的 Token `[CLS]` 的最后一层输出，接上 `softmax` 进行分类，用分类的数据来进行 Fine-tuning
- 对于普通的分类任务，输入是一个序列。像任务 (b) 那样。我们用第一个特殊的 Token `[CLS]` 的最后一层输出，接上 `softmax` 进行分类，用分类的数据来进行 Fine-tuning
- 第三类是问答类问题（如 (c) 所示），输入是一个问题和一段很长的包含答案的文字，输出在这段文字中找到的答案。我们首先将问题和 Paragraph 表示成一个很长的序列，中间用 `[SEP]` 分开，这里假设答案是 Paragraph 中一段连续的文字 (Span)。BERT 把寻找答案的问题转化成寻找这个 Span 的开始下标和结束下标的问题。

对于 Paragraph 的第  $i$  个 Token，BERT 的最后一层把它编码成  $T_i$ ，然后我们用一个向量  $S$  (这是模型的参数，需要根据训练数据调整) 和它相乘 (内积) 计算它是开始位置的得分，因为 Paragraph 的每一个 Token (当然 WordPiece 的中间，比如 `##ing` 是不可能是开始的) 都有可能是开始可能，我们用 `softmax` 把它变成概率，然后选择概率最大的作为答案的开始

类似的有个向量  $T$ ，用于计算答案结束的位置

- 第四类任务是序列标注，比如命名实体识别，输入是一个句子(Token序列)，除了 [CLS] 和 [SEP] 的每个时刻都会有输出的Tag，比如B-PER表示人名的开始。然后用输出的Tag来进行Fine-Tuning，如任务 (d) 所示

本文此部分来自本人博客[Bert简介 长命百岁的博客-CSDN博客](#)

## 总结

### • 英文命名实体识别

以上三种英文方法都是具有命名实体识别功能的，直接提供了实体识别的函数接口，可以直接得到结果

- 我认为，识别效果最好的是 Spacy，其识别效果已经很难找到明显的错误，识别出了 PERSON, NORP, DATE, GPE, ORDINAL, ORG 这些类别。在类别的细分程度上和准确度上，Spacy 明显领先于 nltk，其类别的细分程度虽然不如 StanfordCoreNLP 多，但是准确度比 StanfordCoreNLP 更好，这是非常关键的。在三种方法中，Spacy 是唯一一个能够以短语的形式进行整体识别，而不是将短语拆分成单个词再进行实体识别，这是非常突出的。
- 效果第二好的是 StanfordCoreNLP，其识别种类细分程度非常高，在本次识别中，共发现 [PERSON, DATE, STATE\_OR\_PROVINCE, LOCATION, ORDINAL, ORGANIZATION, IDEOLOGY, TITLE, MISC, COUNTRY] 这些类。具有一定的识别准确度，分类比较精细，出现了特有的实体名称 IDEOLOGY, MISC, TITLE。但是其不能识别短语，仍有明显的遗漏识别和错误识别，识别结果对上下文有一定的依赖，且运行速度明显慢于其它方法。
- 最后的是 nltk，其识别种类细分程度不够高，上面得到的种类为 [PERSON, ORGANIZATION, GPE]。识别的 ORGANIZATION 效果不错，基本准确，但是可能会将名词组合中的所有词都认为是 ORGANIZATION。存在明显的错误识别和遗漏识别。

### • 中文命名实体识别

中文方法中，只有 StanfordCoreNLP 提供了实体识别的函数接口，其余方法均依靠词性标注结果手动实现。

- StanfordCoreNLP 能实现一些基本的实体命名识别，比如 DATE, COUNTRY, ORGANIZATION, ORDINAL, NUMBER 等，对数字的识别比较精准，但是总体的识别精度不够，像英文识别时提到的那样，识别结果一定程度上依赖于上下文。
- jieba 中，我们测试了动词。效果严重依赖于分词结果。从结果来看，这些词确实都是动词，且是以短语的形式呈现的，识别效果还不错



- SnowNLP 部分识别效果比较准确，像一般的 青年报，邀请赛 等识别结果都正确。有很多识别错误的和遗漏识别的。分词效果不好，导致词性标注的效果也不是很理想，从而又引起了命名实体识别结果不理想。**单个字太多，甚至无法识别出正常的短语，长一些的名词**
- THULAC 可以区分人名，机构名，量词等。总体效果十分不错，算是**这些中文方法中效果最好的了**。分词效果太细致，导致词性标注太细致，容易将原本是一个词性的词 分开标注。部分词性标注不准确。
- NLPPIR 准确度是及格的，识别为名词的基本都是名词。而且标注方式为正常的英文，而不是自定义的词性表，看起来非常易懂。但是，由于分词效果比较差，从而导致了词性标注和命名实体识别的效果也比较差。

- **Bert+ BiLstm-CRF**

该部分原理及实验过程，实验内容均已详细论述，但不足的是，模型在训练过程中发生了过拟合