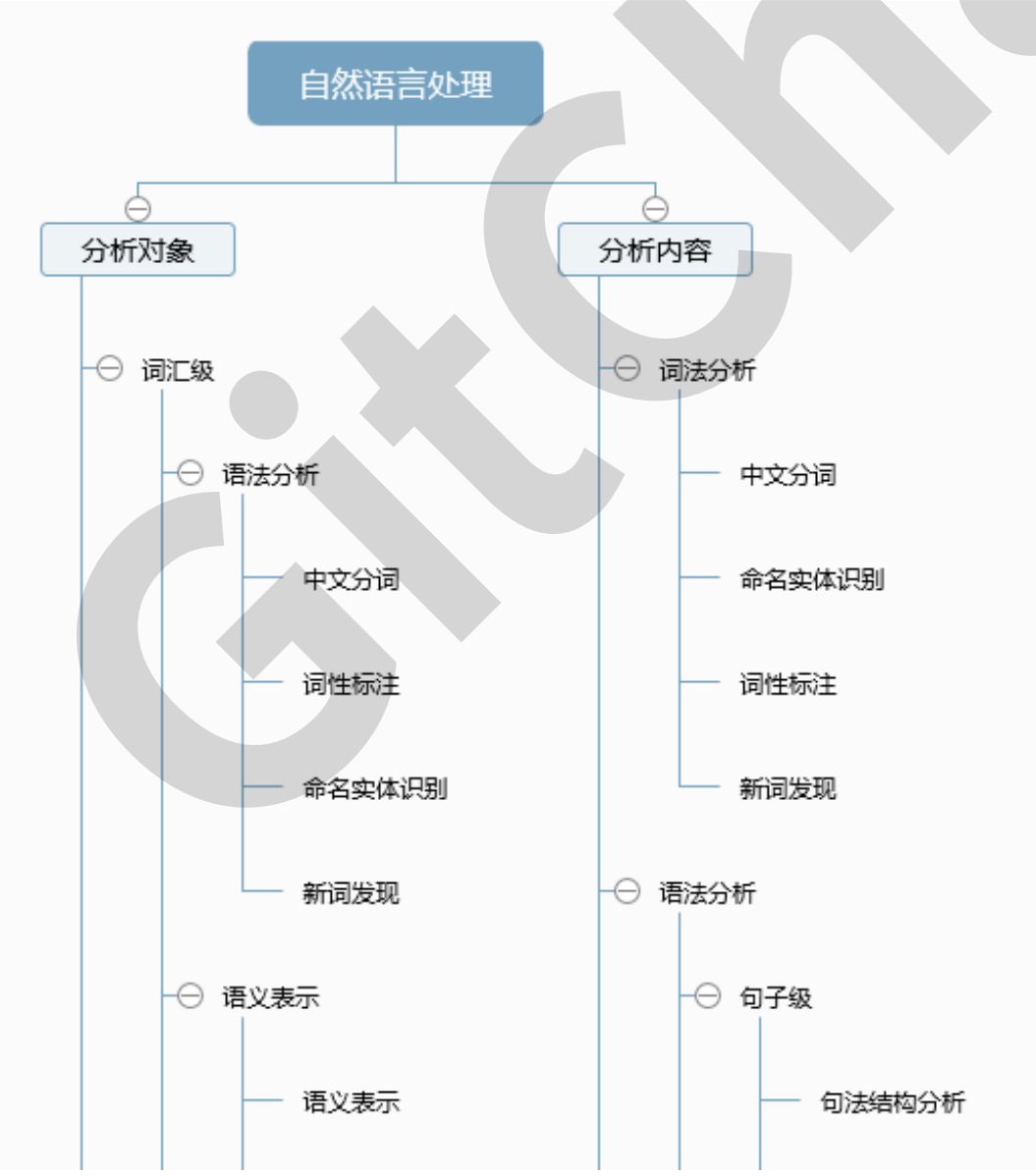
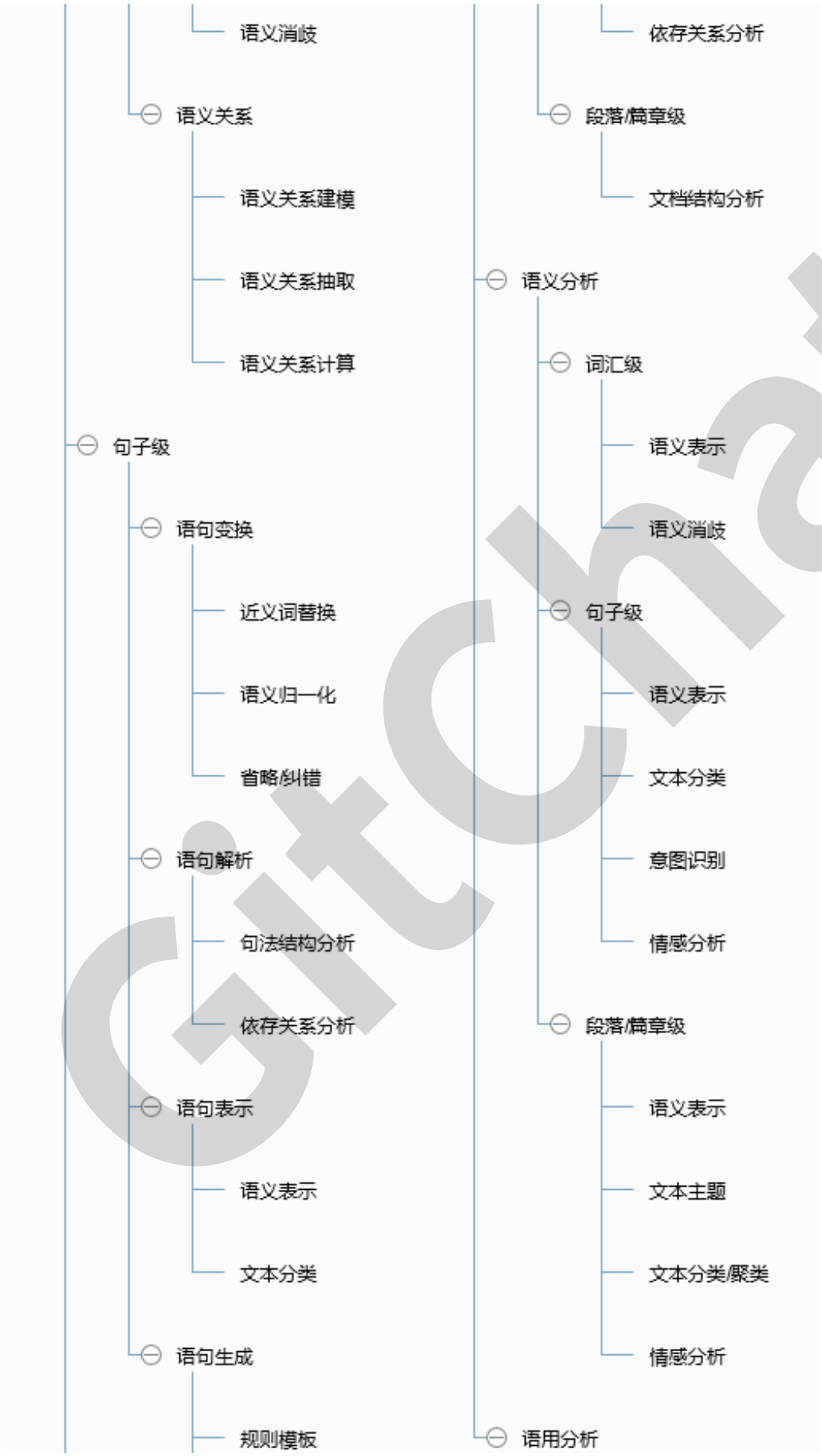
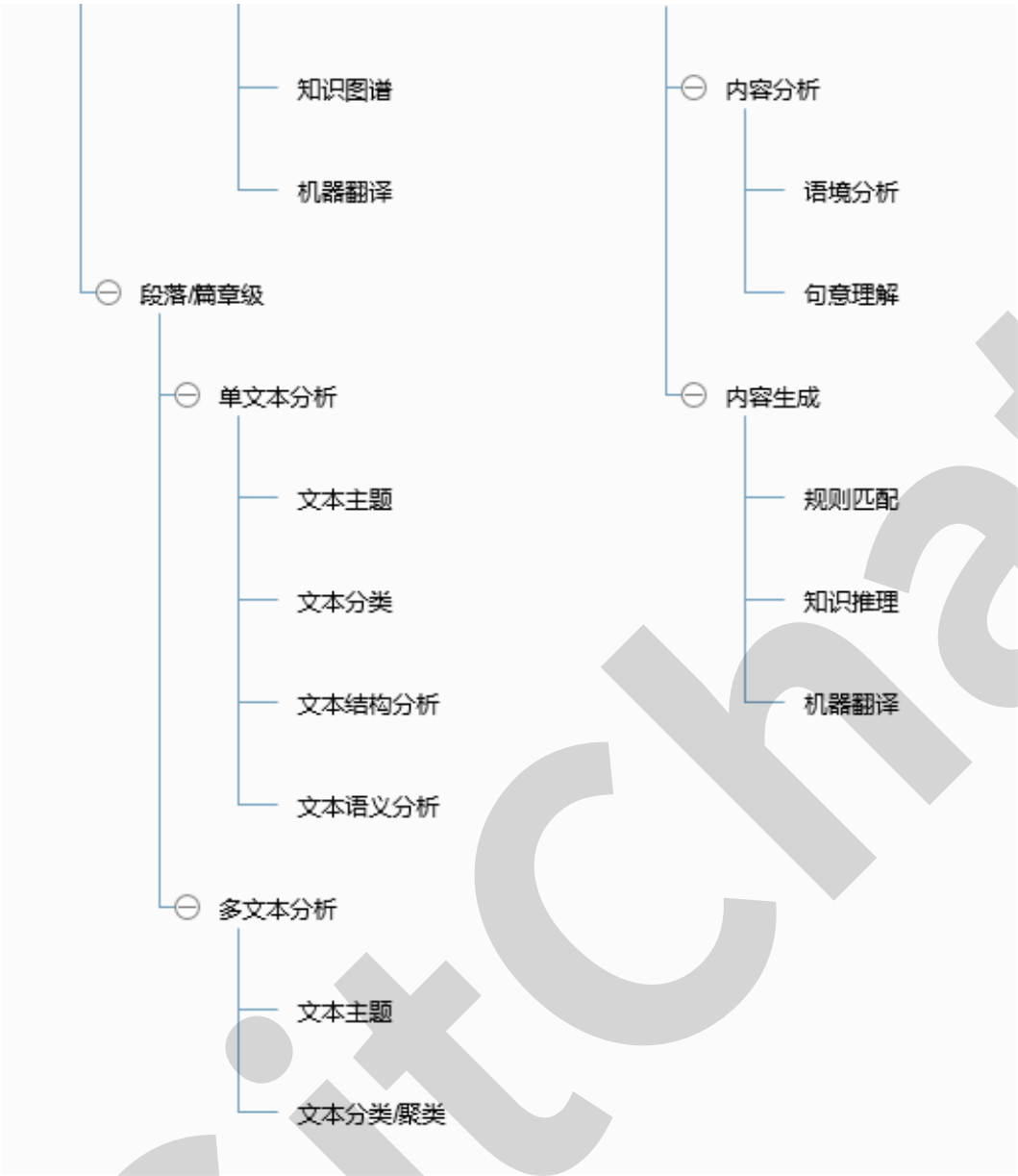
第01课：中文自然语言处理的完整机器处理流程

2016年全球瞩目的围棋大战中，人类以失败告终，更是激起了各种“机器超越、控制人类”的讨论，然而机器真的懂人类吗？机器能感受到人类的情绪吗？机器能理解人类的语言吗？如果能，那它又是如何做到呢？带着这样好奇心，本文将带领大家熟悉和回顾一个完整的自然语言处理过程，后续所有章节所有示例开发都将遵从这个处理过程。

首先我们通过一张图（来源：网络）来了解 NLP 所包含的技术知识点，这张图从分析对象和分析内容两个不同的维度来进行表达，个人觉得内容只能作为参考，对于整个 AI 背景下的自然语言处理来说还不够完整。




语义消歧
语义关系
语义关系建模
语义关系抽取
语义关系计算
句子级
语句变换
近义词替换
语义归一化
省略/纠错
语句解析
句法结构分析
依存关系分析
语句表示
语义表示
文本分类
语句生成
规则模板
依存关系分析
段落/篇章级
文档结构分析
语义分析
词汇级
语义表示
语义消歧
句子级
语义表示
文本分类
意图识别
情感分析
段落/篇章级
语义表示
文本主题
文本分类/聚类
情感分析
语用分析



有机器学习相关经验的人都知道，中文自然语言处理的过程和机器学习过程大体一致，但又存在很多细节上的不同点，下面我们就来看看中文自然语言处理的基本过程有哪些呢？

获取语料

语料，即语言材料。语料是语言学研究的内容。语料是构成语料库的基本单元。所以，人们简单地用文本作为替代，并把文本中的上下文关系作为现实世界中语言的上下文关系的替代品。我们把一个文本集合称为语料库（Corpus），当有几个这样的文本集合的时候，我们称之为语料库集合(Corpora)。（定义来源：百度百科）按语料来源，我们将语料分为以下两种：

1.已有语料

很多业务部门、公司等组织随着业务发展都会积累有大量的纸质或者电子文本资料。那么，对于这些资料，在允许的条件下我们稍加整合，把纸质的文本全部电子化就可以作为我们的语料库。

2.网上下载、抓取语料

如果现在个人手里没有数据怎么办呢？这个时候，我们可以选择获取国内外标准开放数据集，比如国内的**中文汉语有搜狗语料**、**人民日报语料**。国外的因为大都是英文或者外文，这里暂时用不到。也可以选择通过爬虫自己去抓取一些数据，然后来进行后续内容。

语料预处理

这里重点介绍一下语料的预处理，在一个完整的中文自然语言处理工程应用中，语料预处理大概会占到整个50%-70%的工作量，所以开发人员大部分时间就在进行语料预处理。下面通过数据洗清、分词、词性标注、去停用词四个大的方面来完成语料的预处理工作。

1.语料清洗

数据清洗，顾名思义就是在语料中找到我们感兴趣的东西，把不感兴趣的、视为噪音的内容清洗删除，包括对于原始文本提取标题、摘要、正文等信息，对于爬取的网页内容，去除广告、标签、HTML、JS 等代码和注释等。常见的数据清洗方式有：人工去重、对齐、删除和标注等，或者规则提取内容、正则表达式匹配、根据词性和命名实体提取、编写脚本或者代码批处理等。

2.分词

中文语料数据为一批短文本或者长文本，比如：句子，文章摘要，段落或者整篇文章组成的一个集合。一般句子、段落之间的字、词语是连续的，有一定含义。而进行文本挖掘分析时，我们希望文本处理的最小单位粒度是词或者词语，所以这个时候就需要分词来将文本全部进行分词。

常见的分词算法有：基于字符串匹配的分词方法、基于理解的分词方法、基于统计的分词方法和基于规则的分词方法，每种方法下面对应许多具体的算法。

当前中文分词算法的主要难点有歧义识别和新词识别，比如：“羽毛球拍卖完了”，这个可以

切分成“羽毛 球拍 卖 完 了”，也可切分成“羽毛球 拍卖 完 了”，如果不依赖上下文其他的句子，恐怕很难知道如何去理解。

3.词性标注

词性标注，就是给每个词或者词语打词类标签，如形容词、动词、名词等。这样做可以让文本在后面的处理中融入更多有用的语言信息。词性标注是一个经典的序列标注问题，不过对于有些中文自然语言处理来说，词性标注不是非必需的。比如，常见的文本分类就不用关心词性问题，但是类似情感分析、知识推理却是需要的，下图是常见的中文词性整理。

词性编码	词性名称	注 解
Ag	形语素	形容词性语素。形容词代码为 a，语素代码 g 前面置以A。
a	形容词	取英语形容词 adjective 的第1个字母。
ad	副形词	直接作状语的形容词。形容词代码 a和副词代码d并在一起。
an	名形词	具有名词功能的形容词。形容词代码 a和名词代码n并在一起。
b	区别词	取汉字“别”的声母。
c	连词	取英语连词 conjunction 的第1个字母。
dg	副语素	副词性语素。副词代码为 d，语素代码 g 前面置以D。
d	副词	取 adverb 的第2个字母，因其第1个字母已用于形容词。
e	叹词	取英语叹词 exclamation 的第1个字母。
f	方位词	取汉字“方”
g	语素	绝大多数语素都能作为合成词的“词根”，取汉字“根”的声母。
h	前接成分	取英语 head 的第1个字母。
i	成语	取英语成语 idiom 的第1个字母。
j	简称略语	取汉字“简”的声母。
k	后接成分	
l	习用语	习用语尚未成为成语，有点“临时性”，取“临”的声母。
m	数词	取英语 numeral 的第3个字母，n，u已有他用。
Ng	名语素	名词性语素。名词代码为 n，语素代码 g 前面置以N。
n	名词	取英语名词 noun 的第1个字母。
nr	人名	名词代码 n和“人(ren)”的声母并在一起。
ns	地名	名词代码 n和处所词代码s并在一起。
nt	机构团体	“团”的声母为 t，名词代码n和t并在一起。
nz	其他专名	“专”的声母的第 1个字母为z，名词代码n和z并在一起。
o	拟声词	取英语拟声词 onomatopoeia 的第1个字母。
p	介词	取英语介词 prepositional 的第1个字母。
q	量词	取英语 quantity 的第1个字母。
r	代词	取英语代词 pronoun 的第2个字母,因p已用于介词。
s	处所词	取英语 space 的第1个字母。
tg	时语素	时间词性语素。时间词代码为 t,在语素的代码g前面置以T。
t	时间词	取英语 time 的第1个字母。
u	助词	取英语助词 auxiliary
vg	动语素	动词性语素。动词代码为 v。在语素的代码g前面置以V。
v	动词	取英语动词 verb 的第一个字母。
vd	副动词	直接作状语的动词。动词和副词的代码并在一起。
vn	名动词	指具有名词功能的动词。动词和名词的代码并在一起。
w	标点符号	
x	非语素字	非语素字只是一个符号，字母 x通常用于代表未知数、符号。
y	语气词	取汉字“语”的声母。
z	状态词	取汉字“状”的声母的前一个字母。
un	未知词	不可识别词及用户自定义词组。取英文Unkonwn首两个字母。(非北大标准，CSW分词中定义)

常见的词性标注方法可以分为基于规则和基于统计的方法。其中基于统计的方法，如基于最大熵的词性标注、基于统计最大概率输出词性和基于 HMM 的词性标注。

4.去停用词

停用词一般指对文本特征没有任何贡献作用的字词，比如标点符号、语气、人称等一些词。所以在一般性的文本处理中，分词之后，接下来一步就是去停用词。但是对于中文来说，去停用词操作不是一成不变的，停用词词典是根据具体场景来决定的，比如在情感分析中，语气词、感叹号是应该保留的，因为他们对表示语气程度、感情色彩有一定的贡献和意义。

特征工程

做完语料预处理之后，接下来需要考虑如何把分词之后的字和词语表示成计算机能够计算的类型。显然，如果要计算我们至少需要把中文分词的字符串转换成数字，确切的说应该是数学中的向量。有两种常用的表示模型分别是词袋模型和词向量。

词袋模型（Bag of Word, BOW)，即不考虑词语原本在句子中的顺序，直接将每一个词语或者符号统一放置在一个集合（如 list），然后按照计数的方式对出现的次数进行统计。统计词频这只是最基本的方式，TF-IDF 是词袋模型的一个经典用法。

词向量是将字、词语转换成向量矩阵的计算模型。目前为止最常用的词表示方法是 One-hot，这种方法把每个词表示为一个很长的向量。这个向量的维度是词表大小，其中绝大多数元素为 0，只有一个维度的值为 1，这个维度就代表了当前的词。还有 Google 团队的 Word2Vec，其主要包含两个模型：跳字模型（Skip-Gram）和连续词袋模型（Continuous Bag of Words，简称 CBOW），以及两种高效训练的方法：负采样（Negative Sampling）和层序 Softmax（Hierarchical Softmax）。值得一提的是，Word2Vec 词向量可以较好地表达不同词之间的相似和类比关系。除此之外，还有一些词向量的表示方式，如 Doc2Vec、WordRank 和 FastText 等。

特征选择

同数据挖掘一样，在文本挖掘相关问题中，特征工程也是必不可少的。在一个实际问题中，构造好的特征向量，是要选择合适的、表达能力强的特征。文本特征一般都是词语，具有语义信息，使用特征选择能够找出一个特征子集，其仍然可以保留语义信息；但通过特征提取找到的特征子空间，将会丢失部分语义信息。所以特征选择是一个很有挑战的过程，更多的依赖于经验和专业知识，并且有很多现成的算法来进行特征的选择。目前，常见的特征选择方法主要有 DF、 MI、 IG、 CHI、WLLR、WFO 六种。

模型训练

在特征向量选择好之后，接下来要做的事情当然就是训练模型，对于不同的应用需求，我们使用不同的模型，传统的有监督和无监督等机器学习模型， 如 KNN、SVM、Naive Bayes、决策树、GBDT、K-means 等模型；深度学习模型比如 CNN、RNN、LSTM、 Seq2Seq、FastText、TextCNN 等。这些模型在后续的分类、聚类、神经序列、情感分析等示例中都会用到，这里不再赘述。下面是在模型训练时需要注意的几个点。

1.注意过拟合、欠拟合问题，不断提高模型的泛化能力。

过拟合：模型学习能力太强，以至于把噪声数据的特征也学习到了，导致模型泛化能力下降，在训练集上表现很好，但是在测试集上表现很差。

常见的解决方法有：

- 增大数据的训练量；
- 增加正则化项，如 L1 正则和 L2 正则；
- 特征选取不合理，人工筛选特征和使用特征选择算法；
- 采用 Dropout 方法等。

欠拟合：就是模型不能够很好地拟合数据，表现在模型过于简单。

常见的解决方法有：

- 添加其他特征项；
- 增加模型复杂度，比如神经网络加更多的层、线性模型通过添加多项式使模型泛化能力更强；
- 减少正则化参数，正则化的目的是用来防止过拟合的，但是现在模型出现了欠拟合，则需要减少正则化参数。

2.对于神经网络，注意梯度消失和梯度爆炸问题。

评价指标

训练好的模型，上线之前要对模型进行必要的评估，目的让模型对语料具备较好的泛化能力。具体有以下这些指标可以参考。

1.错误率、精度、准确率、精确度、召回率、F1 衡量。

错误率：是分类错误的样本数占样本总数的比例。对样例集 D，分类错误率计算公式如下：

$$E(f;D)=\frac{1}{m}\sum_{i=1}^{m}\mathbb{I}(f(\boldsymbol{x}_i)\neq y_i)$$

精度：是分类正确的样本数占样本总数的比例。这里的分类正确的样本数指的不仅是正例分类正确的个数还有反例分类正确的个数。对样例集 D，精度计算公式如下：

$$\begin{aligned}\mathrm{acc}(f;D) &= \frac{1}{m}\sum_{i=1}^{m}\mathbb{I}(f(\boldsymbol{x}_i)=y_i)\\ &= 1-E(f;D)\ .\end{aligned}$$

对于二分类问题，可将样例根据其真实类别与学习器预测类别的组合划分为真正例（True Positive）、假正例（False Positive）、真反例（True Negative)、假反例（False Negative）四种情形，令 TP、FP、TN、FN 分别表示其对应的样例数，则显然有 TP+FP++TN+FN=样例总数。分类结果的“混淆矩阵”（Confusion Matrix）如下：

真实情况	预测结果	
	正例	反例
正例	TP(真正例)	FN(假反例)
反例	FP(假正例)	TN(真反例)

准确率，缩写表示用 P。准确率是针对我们预测结果而言的，它表示的是预测为正的样例中有多少是真正的正样例。定义公式如下：

$$\mathrm{P}=\frac{TP}{TP+FP}$$

精确度，缩写表示用 A。精确度则是分类正确的样本数占样本总数的比例。Accuracy 反应了分类器对整个样本的判定能力（即能将正的判定为正的，负的判定为负的）。定义公式如下：

$$A = \frac{TP+TN}{TP+FN+FP+TN}$$

召回率，缩写表示用 R。召回率是针对我们原来的样本而言的，它表示的是样本中的正例有多少被预测正确。定义公式如下：

$$R = \frac{TP}{TP+FN}$$

F1 衡量，表达出对查准率/查全率的不同偏好。定义公式如下：

$$F1 = \frac{2 \times P \times R}{P+R} = \frac{2 \times TP}{样例总数 + TP - TN}$$

2.ROC 曲线、AUC 曲线。

ROC 全称是“受试者工作特征”（Receiver Operating Characteristic）曲线。我们根据模型的预测结果，把阈值从0变到最大，即刚开始是把每个样本作为正例进行预测，随着阈值的增大，学习器预测正样例数越来越少，直到最后没有一个样本是正样例。在这一过程中，每次计算出两个重要量的值，分别以它们为横、纵坐标作图，就得到了 ROC 曲线。

ROC 曲线的纵轴是“真正例率”（True Positive Rate, 简称 TPR)，横轴是“假正例率”（False Positive Rate,简称FPR），两者分别定义为：

$$TPR = \frac{TP}{TP+FN}$$

$$FPR = \frac{FP}{TN+FP}$$

ROC 曲线的意义有以下几点：

1. ROC 曲线能很容易的查出任意阈值对模型的泛化性能影响；
2. 有助于选择最佳的阈值；
3. 可以对不同的模型比较性能，在同一坐标中，靠近左上角的 ROC 曲所代表的学习器准确性最高。

如果两条 ROC 曲线没有相交，我们可以根据哪条曲线最靠近左上角哪条曲线代表的学习器性能就最好。但是实际任务中，情况很复杂，若两个模型的 ROC 曲线发生交叉，则难以一般性的断言两者孰优孰劣。此时如果一定要进行比较，则比较合理的判断依据是比较 ROC 曲线下的面积，即AUC（Area Under ROC Curve）。

AUC 就是 ROC 曲线下的面积，衡量学习器优劣的一种性能指标。AUC 是衡量二分类模型优劣的一种评价指标，表示预测的正例排在负例前面的概率。

前面我们所讲的都是针对二分类问题，那么如果实际需要在多分类问题中用 ROC 曲线的话，一般性的转化为多个“一对多”的问题。即把其中一个当作正例，其余当作负例来看待，画出多个 ROC 曲线。

模型上线应用

模型线上应用，目前主流的应用方式就是提供服务或者将模型持久化。

第一就是线下训练模型，然后将模型做线上部署，发布成接口服务以供业务系统使用。

第二种就是在线训练，在线训练完成之后把模型 pickle 持久化，然后在线服务接口模板通过读取 pickle 而改变接口服务。

模型重构（非必须）

随着时间和变化，可能需要对模型做一定的重构，包括根据业务不同侧重点对上面提到的一至七步骤也进行调整，重新训练模型进行上线。

参考文献

1. 周志华《机器学习》
2. 李航《统计学习方法》

3. 伊恩·古德费洛《深度学习》

第02课：简单好用的中文分词利器 jieba 和 HanLP

前言

从本文开始，我们就要真正进入实战部分。首先，我们按照中文自然语言处理流程的第一步获取语料，然后重点进行中文分词的学习。中文分词有很多种，常见的比如有中科院计算所 NLPIR、哈工大 LTP、清华大学 THULAC 、斯坦福分词器、Hanlp 分词器、jieba 分词、IKAnalyzer 等。这里针对 jieba 和 HanLP 分别介绍不同场景下的中文分词应用。

jieba 分词

jieba 安装

（1）Python 2.x 下 jieba 的三种安装方式，如下：

- **全自动安装**：执行命令 `easy_install jieba` 或者 `pip install jieba` / `pip3 install jieba`，可实现全自动安装。
- **半自动安装**：先下载 jieba，解压后运行 `python setup.py install`。
- **手动安装**：将 jieba 目录放置于当前目录或者 site-packages 目录。

安装完通过 `import jieba` 验证安装成功与否。

（2）Python 3.x 下的安装方式。

Github 上 jieba 的 Python3.x 版本的路径是：https://github.com/fxsjy/jieba/tree/jieba3k。

通过 `git clone https://github.com/fxsjy/jieba.git` 命令下载到本地，然后解压，再通过命令行进入解压目录，执行 `python setup.py install` 命令，即可安装成功。

jieba 的分词算法

主要有以下三种：

1. 基于统计词典，构造前缀词典，基于前缀词典对句子进行切分，得到所有切分可能，根据切分位置，构造一个有向无环图（DAG）；
2. 基于DAG图，采用动态规划计算最大概率路径（最有可能的分词结果），根据最大概率路径分词；
3. 对于新词(词库中没有的词），采用有汉字成词能力的 HMM 模型进行切分。

jieba 分词

下面我们进行 jieba 分词练习，第一步首先引入 jieba 和语料:

```
    import jieba
    content = "现如今，机器学习和深度学习带动人工智能飞速的发展，并在图片处理、语音识别领域取得巨大成功。"
```

（1）精确分词

精确分词：精确模式试图将句子最精确地切开，精确分词也是默认分词。

```
segs_1 = jieba.cut(content, cut_all=False)
print("/".join(segs_1))
```

其结果为：

> 现如今/，/机器/学习/和/深度/学习/带动/人工智能/飞速/的/发展/，/并/在/图片/处理/、/语音/识别/领域/取得/巨大成功/。

（2）全模式

全模式分词：把句子中所有的可能是词语的都扫描出来，速度非常快，但不能解决歧义。

```
    segs_3 = jieba.cut(content, cut_all=True)
    print("/".join(segs_3))
```

结果为：

> 现如今/如今///机器/学习/和/深度/学习/带动/动人/人工/人工智能/智能/飞速/的/发展///并/在/图片/处理///语音/识别/领域/取得/巨大/巨大成功/大成/成功//

（3）搜索引擎模式

搜索引擎模式：在精确模式的基础上，对长词再次切分，提高召回率，适合用于搜索引擎分词。

```
segs_4 = jieba.cut_for_search(content)
print("/".join(segs_4))
```

结果为：

> 如今/现如今/，/机器/学习/和/深度/学习/带动/人工/智能/人工智能/飞速/的/发展/，/并/在/图片/处理/、/语音/识别/领域/取得/巨大/大成/成功/巨大成功/。

（4）用 lcut 生成 list

jieba.cut 以及 `jieba.cut_for_search` 返回的结构都是一个可迭代的 Generator，可以使用 for 循环来获得分词后得到的每一个词语（Unicode）。jieba.lcut 对 cut 的结果做了封装，l 代表 list，即返回的结果是一个 list 集合。同样的，用 `jieba.lcut_for_search` 也直接返回 list 集合。

```
segs_5 = jieba.lcut(content)
print(segs_5)
```

结果为：

> ['现如今', ' ，', '机器', '学习', '和', '深度', '学习', '带动', '人工智能', '飞速', '的', '发展', ' ，', '并', '在', '图片', '处理', '、', '语音', '识别', '领域', '取得', '巨大成功', '。']

（5）获取词性

jieba 可以很方便地获取中文词性，通过 jieba.posseg 模块实现词性标注。

```
    import jieba.posseg as psg
    print([(x.word,x.flag) for x in psg.lcut(content)])
```

结果为：

> [('现如今', 't'), (' ，', 'x'), ('机器', 'n'), ('学习', 'v'), ('和', 'c'), ('深度', 'ns'), ('学习', 'v'), ('带动', 'v'), ('人工智能', 'n'), ('飞速', 'n'), ('的', 'uj'), ('发展', 'vn'), (' ，', 'x'), ('并', 'c'), ('在', 'p'), ('图片', 'n'), ('处理', 'v'), ('、', 'x'), ('语音', 'n'), ('识别', 'v'), ('领域', 'n'), ('取得', 'v'), ('巨大成功', 'nr'), ('。', 'x')]

（6）并行分词

并行分词原理为文本按行分隔后，分配到多个 Python 进程并行分词，最后归并结果。

用法：

```
jieba.enable_parallel(4) # 开启并行分词模式，参数为并行进程数 。
jieba.disable_parallel() # 关闭并行分词模式 。
```

注意： 并行分词仅支持默认分词器 jieba.dt 和 jieba.posseg.dt。目前暂不支持 Windows。

（7）获取分词结果中词列表的 top n

```
    from collections import Counter
    top5= Counter(segs_5).most_common(5)
    print(top5)
```

结果为：

> [(' ，', 2), ('学习', 2), ('现如今', 1), ('机器', 1), ('和', 1)]

（8）自定义添加词和字典

默认情况下，使用默认分词，是识别不出这句话中的“铁甲网”这个新词，这里使用用户字典提高分词准确性。

```
txt = "铁甲网是中国最大的工程机械交易平台。"
print(jieba.lcut(txt))
```

结果为：

> ['铁甲网', '是', '中国', '最大', '的', '工程机械', '交易平台', '。']

如果添加一个词到字典，看结果就不一样了。

```
jieba.add_word("铁甲网")
print(jieba.lcut(txt))
```

结果为：

> ['铁甲网', '是', '中国', '最大', '的', '工程机械', '交易平台', '。']

但是，如果要添加很多个词，一个个添加效率就不够高了，这时候可以定义一个文件，然后通过 `load_userdict()` 函数，加载自定义词典，如下：

```
jieba.load_userdict('user_dict.txt')
print(jieba.lcut(txt))
```

结果为：

> ['铁甲网', '是', '中国', '最大', '的', '工程机械', '交易平台', '。']

注意事项：

jieba.cut 方法接受三个输入参数: 需要分词的字符串；`cut_all` 参数用来控制是否采用全模式；HMM 参数用来控制是否使用 HMM 模型。

`jieba.cut_for_search` 方法接受两个参数：需要分词的字符串；是否使用 HMM 模型。该方法适合用于搜索引擎构建倒排索引的分词，粒度比较细。

HanLP 分词

pyhanlp 安装

其为 HanLP 的 Python 接口，支持自动下载与升级 HanLP，兼容 Python2、Python3。

安装命令为 `pip install pyhanlp`，使用命令 hanlp 来验证安装。

pyhanlp 目前使用 jpype1 这个 Python 包来调用 HanLP，如果遇到：

> building '_jpype' extensionerror: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft VisualC++ Build Tools": http://landinghub.visualstudio.com/visual-cpp-build-tools

则推荐利用轻量级的 Miniconda 来下载编译好的 jpype1。

```
conda install -c conda-forge jpype1
pip install pyhanlp
```

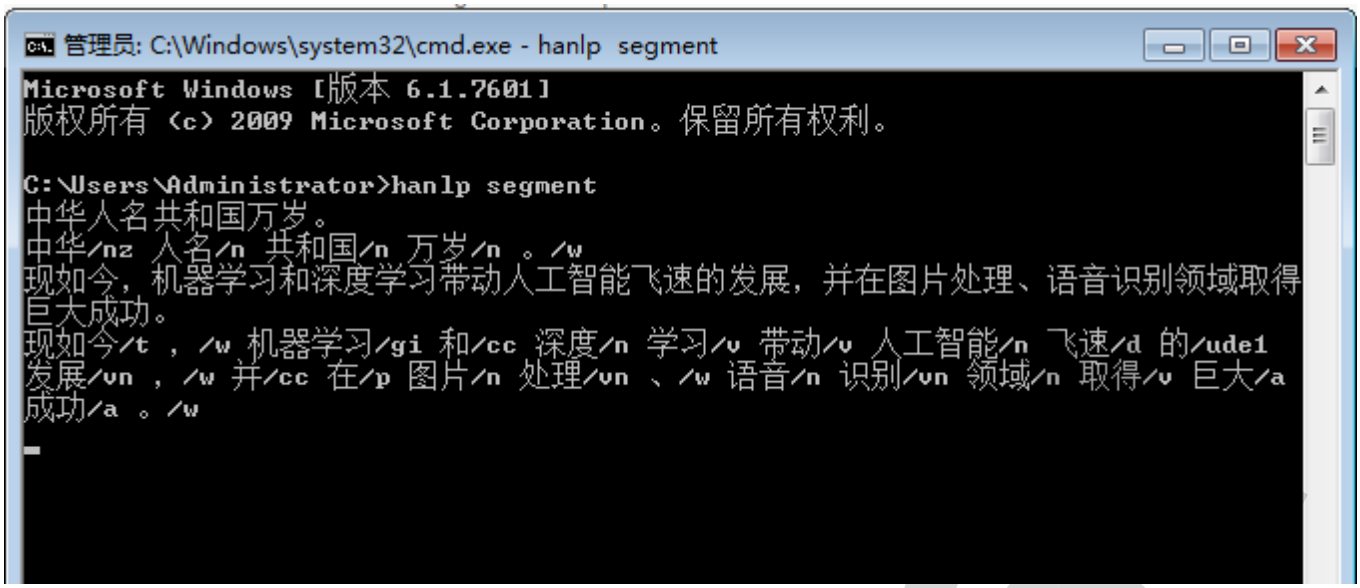
未安装 Java 时会报错：

> jpype._jvmfinder.JVMNotFoundException: No JVM shared library file (jvm.dll) found. Try setting up the JAVA_HOME environment variable properly.

HanLP 主项目采用 Java 开发，所以需要 Java 运行环境，请安装 JDK。

命令行交互式分词模式

在命令行界面，使用命令 hanlp segment 进入交互分词模式，输入一个句子并回车，HanLP 会输出分词结果：



可见，pyhanlp 分词结果是带有词性的。

服务器模式

通过 hanlp serve 来启动内置的 HTTP 服务器，默认本地访问地址为：http://localhost:8765 。

也可以访问官网演示页面：http://hanlp.hankcs.com/。

通过工具类 HanLP 调用常用接口

通过工具类 HanLP 调用常用接口，这种方式应该是我们在项目中最常用的方式。

（1）分词

```
1.      from pyhanlp import *
2.      content = "现如今，机器学习和深度学习带动人工智能飞速的发展，并在图片处理、语音识别领域取得巨大成功。"
3.      print(HanLP.segment(content))
```

结果为：

> [现如今/t, ，/w, 机器学习/gi, 和/cc, 深度/n, 学习/v, 带动/v, 人工智能/n, 飞速/d, 的/ude1, 发展/vn, ，/w, 并/cc, 在/p, 图片/n, 处理/vn, 、/w, 语音/n, 识别/vn, 领域/n, 取得/v, 巨大/a, 成功/a, 。/w]

（2）自定义词典分词

在没有使用自定义字典时的分词。

```
txt = "铁甲网是中国最大的工程机械交易平台。"
print(HanLP.segment(txt))
```

结果为：

> [铁甲/n, 网/n, 是/vshi, 中国/ns, 最大/gm, 的/ude1, 工程/n, 机械/n, 交易/vn, 平台/n, 。/w]

添加自定义新词：

```
CustomDictionary.add("铁甲网")
CustomDictionary.insert("工程机械", "nz 1024")
CustomDictionary.add("交易平台", "nz 1024 n 1")
print(HanLP.segment(txt))
```

结果为：

> [铁甲网/nz, 是/vshi, 中国/ns, 最大/gm, 的/ude1, 工程机械/nz, 交易平台/nz, 。/w]

当然了，jieba 和 pyhanlp 能做的事还有很多，关键词提取、自动摘要、依存句法分析、情感分析等，后面章节我们将会讲到，这里不再赘述。

参考文献：

1. https://github.com/fxsjy/jieba
2. https://github.com/hankcs/pyhanlp

第03课：动手实战中文文本中的关键字提取

前言

关键词提取就是从文本里面把跟这篇文章意义最相关的一些词语抽取出来。这个可以追溯到文献检索初期，关键词是为了文献标引工作，从报告、论文中选取出来用以表示全文主题内容信息的单词或术语，在现在的报告和论文中，我们依然可以看到关键词这一项。因此，关键词在文献检索、自动文摘、文本聚类/分类等方面有着重要的应用，它不仅是进行这些工作不可或缺的基础和前提，也是互联网上信息建库的一项重要工作。

关键词抽取从方法来说主要有两种：

- 第一种是关键词分配：就是给定一个已有的关键词库，对于新来的文档从该词库里面匹配几个词语作为这篇文档的关键词。
- 第二种是关键词提取：针对新文档，通过算法分析，提取文档中一些词语作为该文档的关键词。

目前大多数应用领域的关键词抽取算法都是基于后者实现的，从逻辑上说，后者比前者在实际应用中更准确。

下面介绍一些关于关键词抽取的常用和经典的算法实现。

基于 TF-IDF 算法进行关键词提取

在信息检索理论中，TF-IDF 是 Term Frequency - Inverse Document Frequency 的简写。TF-IDF 是一种数值统计，用于反映一个词对于语料中某篇文档的重要性。在信息检索和文本挖掘领域，它经常用于因子加权。TF-IDF 的主要思想就是：如果某个词在一篇文档中出现的频率高，也即 TF 高；并且在语料库中其他文档中很少出现，即 DF 低，也即 IDF 高，则认为这个词具有很好的类别区分能力。

TF 为词频（Term Frequency），表示词 t 在文档 d 中出现的频率，计算公式：

$$\mathrm{tf_{i,j}} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

其中，$n_{i,j}$ 是该词 t_i 在文件 d_j 中的出现次数，而分母则是在文件 d_j 中所有字词的出现次数之和。

IDF 为逆文档频率（Inverse Document Frequency），表示语料库中包含词 t 的文档的数目的倒数，计算公式：

$$\mathrm{idf_i} = \log \frac{|D|}{|\{j : t_i \in d_j\}|}$$

其中，`|D|` 表示语料库中的文件总数，$|\{j : t_i \in d_j\}|$ 包含词 t_i 的文件数目，如果该词语不在语料库中，就会导致被除数为零，因此一般情况下使用 $1 + |\{j : t_i \in d_j\}|$。

TF-IDF 在实际中主要是将二者相乘，也即 TF * IDF， 计算公式：

$$\mathrm{tfidf_{i,j}} = \mathrm{tf_{i,j}} \times \mathrm{idf_i}$$

因此，TF-IDF 倾向于过滤掉常见的词语，保留重要的词语。例如，某一特定文件内的高频率词语，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的 TF-IDF。

好在 jieba 已经实现了基于 TF-IDF 算法的关键词抽取，通过命令 `import jieba.analyse` 引入，函数参数解释如下：

```
1.  jieba.analyse.extract_tags(sentence, topK=20, withWeight=False,
    allowPOS=())
```

- sentence：待提取的文本语料；
- topK：返回 TF/IDF 权重最大的关键词个数，默认值为 20；
- withWeight：是否需要返回关键词权重值，默认值为 False；
- allowPOS：仅包括指定词性的词，默认值为空，即不筛选。

接下来看例子，我采用的语料来自于百度百科对人工智能的定义，获取 Top20 关键字，用空格隔开打印：

```
    import jieba.analyse
    sentence  = "人工智能（Artificial Intelligence），英文缩写为AI。它是研究、开发用于模拟、延伸和扩展人的智能的理论、方法、技术及应用系统的一门新的技术科学。人工智能是计算机科学的一个分支，它企图了解智能的实质，并生产出一种新的能以人类智能相似的方式做出反应的智能机器，该领域的研究包括机器人、语言识别、图像识别、自然语言处理和专家系统等。人工智能从诞生以来，理论和技术日益成熟，应用领域也不断扩大，可以设想，未来人工智能带来的科技产品，将会是人类智慧的“容器”。人工智能可以对人的意识、思维的信息过程的模拟。人工智能不是人的智能，但能像人那样思考、也可能超过人的智能。人工智能是一门极富挑战性的科学，从事这项工作的人必须懂得计算机知识，心理学和哲学。人工智能是包括十分广泛的科学，它由不同的领域组成，如机器学习，计算机视觉等等，总的说来，人工智能研究的一个主要目标是使机器能够胜任一些通常需要人类智能才能完成的复杂工作。但不同的时代、不同的人对这种“复杂工作”的理解是不同的。2017年12月，人工智能入选“2017年度中国媒体十大流行语”。"
    keywords = "  ".join(jieba.analyse.extract_tags(sentence , topK=20, withWeight=False, allowPOS=()))
    print(keywords)
```

执行结果：

> 人工智能 智能 2017 机器 不同 人类 科学 模拟 一门 技术 计算机 研究 工作 Artificial Intelligence AI 图像识别 12 复杂 流行语

下面只获取 Top10 的关键字，并修改一下词性，只选择名词和动词，看看结果有何不同？

```
    keywords =(jieba.analyse.extract_tags(sentence , topK=10, withWeight=True, allowPOS=(['n','v'])))
    print(keywords)
```

执行结果：

> [('人工智能', 0.9750542675762887), ('智能', 0.5167124540885567), ('机器', 0.20540911929525774), ('人类', 0.17414426566082475), ('科学', 0.17250169374402063), ('模拟', 0.15723537382948452), ('技术', 0.14596259315164947), ('计算机', 0.14030483362639176), ('图像识别', 0.12324502580309278), ('流行语', 0.11242211730309279)]

基于 TextRank 算法进行关键词提取

TextRank 是由 PageRank 改进而来，核心思想将文本中的词看作图中的节点，通过边相互连接，不同的节点会有不同的权重，权重高的节点可以作为关键词。这里给出 TextRank 的公式：

$$WS(V_i) = (1-d) + d * \sum_{V_j \in In(V_i)} \frac{w_{ji}}{\sum_{V_k \in Out(V_j)} w_{jk}} WS(V_j)$$

节点 i 的权重取决于节点 i 的邻居节点中 i-j 这条边的权重 / j 的所有出度的边的权重 * 节点 j 的权重，将这些邻居节点计算的权重相加，再乘上一定的阻尼系数，就是节点 i 的权重，阻尼系数 d 一般取 0.85。

TextRank 用于关键词提取的算法如下：

（1）把给定的文本 T 按照完整句子进行分割，即:

$$T=[S_1, S_2, \cdots, S_m]$$

（2）对于每个句子，进行分词和词性标注处理，并过滤掉停用词，只保留指定词性的单词，如名词、动词、形容词，其中 $t_{i,j}$ 是保留后的候选关键词。

$$S_i=[t_{i,1}, t_{i,2}, \cdots, t_{i,n}]$$

（3）构建候选关键词图 G = (V,E)，其中 V 为节点集，由（2）生成的候选关键词组成，然后采用共现关系（Co-Occurrence）构造任两点之间的边，两个节点之间存在边仅当它们对应的词汇在长度为 K 的窗口中共现，K 表示窗口大小，即最多共现 K 个单词。

（4）根据 TextRank 的公式，迭代传播各节点的权重，直至收敛。

（5）对节点权重进行倒序排序，从而得到最重要的 T 个单词，作为候选关键词。

（6）由（5）得到最重要的 T 个单词，在原始文本中进行标记，若形成相邻词组，则组合成多词关键词。

同样 jieba 已经实现了基于 TextRank 算法的关键词抽取，通过命令 `import jieba.analyse`

引用，函数参数解释如下：

```
jieba.analyse.textrank(sentence, topK=20, withWeight=False, allowPOS=('ns', 'n', 'vn', 'v'))
```

直接使用，接口参数同 TF-IDF 相同，注意默认过滤词性。

接下来，我们继续看例子，语料继续使用上例中的句子。

```
    result = "  ".join(jieba.analyse.textrank(sentence, topK=20, withWeight=False, allowPOS=('ns', 'n', 'vn', 'v')))
    print(result)
```

执行结果：

> 智能 人工智能 机器 人类 研究 技术 模拟 包括 科学 工作 领域 理论 计算机 年度 需要 语言 相似 方式 做出 心理学

如果修改一下词性，只需要名词和动词，看看结果有何不同？

```
    result = "  ".join(jieba.analyse.textrank(sentence, topK=20, withWeight=False, allowPOS=('n','v')))
    print(result)
```

执行结果：

> 智能 人工智能 机器 人类 技术 模拟 包括 科学 理论 计算机 领域 年度 需要 心理学 信息 语言 识别 带来 过程 延伸

基于 LDA 主题模型进行关键词提取

其实，使用 LDA 获取文本关键词在我的第一次 Chat《NLP 中文短文本分类项目实践（上）》已经讲过了，为了保持内容的完整性，在这里我继续写一下。

语料是一个关于汽车的短文本，下面通过 Gensim 库完成基于 LDA 的关键字提取。整个过程

的步骤为：文件加载 -> jieba 分词 -> 去停用词 -> 构建词袋模型 -> LDA 模型训练 -> 结果可视化。

```
    #引入库文件
    import jieba.analyse as analyse
    import jieba
    import pandas as pd
    from gensim import corpora, models, similarities
    import gensim
    import numpy as np
    import matplotlib.pyplot as plt
    %matplotlib inline
    #设置文件路径
    dir = "D://ProgramData//PythonWorkSpace//study//"
    file_desc = "".join([dir,'car.csv'])
    stop_words = "".join([dir,'stopwords.txt'])
    #定义停用词
    stopwords=pd.read_csv(stop_words,index_col=False,quoting=3,sep="\t",names=['stopword'], encoding='utf-8')
    stopwords=stopwords['stopword'].values
    #加载语料
    df = pd.read_csv(file_desc, encoding='gbk')
    #删除nan行
    df.dropna(inplace=True)
    lines=df.content.values.tolist()
    #开始分词
    sentences=[]
    for line in lines:
        try:
            segs=jieba.lcut(line)
            segs = [v for v in segs if not str(v).isdigit()]#去数字
            segs = list(filter(lambda x:x.strip(), segs))    #去左右空格
            segs = list(filter(lambda x:x not in stopwords, segs)) #去掉停用词
            sentences.append(segs)
        except Exception:
            print(line)
            continue
    #构建词袋模型
    dictionary = corpora.Dictionary(sentences)
    corpus = [dictionary.doc2bow(sentence) for sentence in sentences]
    #lda模型，num_topics是主题的个数，这里定义了5个
    lda = gensim.models.ldamodel.LdaModel(corpus=corpus, id2word=dictionary, num_topics=10)
```

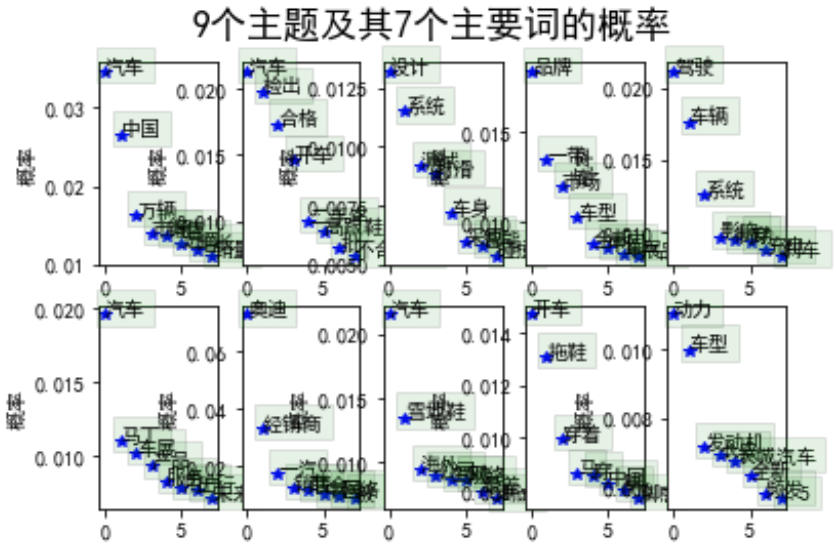
```
        #我们查一下第1号分类，其中最常出现的5个词是：
        print(lda.print_topic(1, topn=5))
        #我们打印所有5个主题，每个主题显示8个词
        for topic in lda.print_topics(num_topics=10, num_words=8):
            print(topic[1])
```

执行结果如下图所示：

```
0.021*"汽车" + 0.020*"检出" + 0.017*"合格" + 0.015*"开车" + 0.010*"二手车"
0.035*"汽车" + 0.027*"中国" + 0.016*"万辆" + 0.014*"市场" + 0.014*"销售" + 0.013*"进口" + 0.012*"增长" + 0.011*"销量"
0.021*"汽车" + 0.020*"检出" + 0.017*"合格" + 0.015*"开车" + 0.010*"二手车" + 0.009*"高跟鞋" + 0.008*"批" + 0.007*"不合格率"
0.013*"设计" + 0.011*"系统" + 0.009*"测试" + 0.009*"打滑" + 0.007*"车身" + 0.006*"采用" + 0.006*"智能" + 0.005*"碰撞"
0.018*"品牌" + 0.013*"一带" + 0.012*"市场" + 0.010*"车型" + 0.009*"全新" + 0.009*"中国" + 0.008*"车展" + 0.008*"产品"
0.021*"驾驶" + 0.018*"车辆" + 0.013*"系统" + 0.010*"影响" + 0.009*"制动" + 0.009*"穿" + 0.009*"充电" + 0.008*"刹车"
0.019*"汽车" + 0.011*"马丁" + 0.010*"车展" + 0.009*"产品" + 0.008*"服务" + 0.008*"用户" + 0.008*"出行" + 0.007*"未来"
0.073*"奥迪" + 0.033*"经销商" + 0.017*"一汽" + 0.012*"销售" + 0.012*"联会" + 0.010*"中国" + 0.009*"网络" + 0.008*"被"
0.021*"汽车" + 0.013*"雪地鞋" + 0.009*"海外" + 0.009*"公司" + 0.009*"驾驶" + 0.009*"网络" + 0.008*"最差" + 0.007*"测试"
0.015*"开车" + 0.013*"拖鞋" + 0.010*"穿着" + 0.009*"马" + 0.009*"穿" + 0.008*"中国" + 0.008*"脚" + 0.008*"脚感"
0.011*"动力" + 0.010*"车型" + 0.007*"发动机" + 0.007*"长安" + 0.007*"长城汽车" + 0.006*"全新" + 0.006*"频发" + 0.006*"1.5"
```

```
        #显示中文matplotlib
        plt.rcParams['font.sans-serif'] = [u'SimHei']
        plt.rcParams['axes.unicode_minus'] = False
        # 在可视化部分，我们首先画出了九个主题的7个词的概率分布图
        num_show_term = 8 # 每个主题下显示几个词
        num_topics  = 10
        for i, k in enumerate(range(num_topics)):
            ax = plt.subplot(2, 5, i+1)
            item_dis_all = lda.get_topic_terms(topicid=k)
            item_dis = np.array(item_dis_all[:num_show_term])
            ax.plot(range(num_show_term), item_dis[:, 1], 'b*')
            item_word_id = item_dis[:, 0].astype(np.int)
            word = [dictionary.id2token[i] for i in item_word_id]
            ax.set_ylabel(u"概率")
            for j in range(num_show_term):
                ax.text(j, item_dis[j, 1], word[j], bbox=dict(facecolor='gr
    een',alpha=0.1))
        plt.suptitle(u'9个主题及其7个主要词的概率', fontsize=18)
        plt.show()
```

执行结果如下图所示：



基于 pyhanlp 进行关键词提取

除了 jieba，也可以选择使用 HanLP 来完成关键字提取，内部采用 TextRankKeyword 实现，语料继续使用上例中的句子。

```
from pyhanlp import *
result = HanLP.extractKeyword(sentence, 20)
print(result)
```

执行结果：

> [人工智能, 智能, 领域, 人类, 研究, 不同, 工作, 包括, 模拟, 新的, 机器, 计算机, 门, 科学, 应用, 系统, 理论, 技术, 入选, 复杂]

总结

本节内容的重点就是掌握关键字提取的基本方法，常规的关键词提取方法如上所述，当然还有其他算法及其改进，有深入研究需求的，可以下载关键字提取方面的论文阅读。

参考文献：

1. Mihalcea R, Tarau P. TextRank: Bringing order into texts[C]//Proceedings of EMNLP. 2004, 4(4): 275.

2. Witten I H, Paynter G W, Frank E, et al. KEA: Practical automatic keyphrase extraction[C]//Proceedings of the fourth ACM conference on Digital libraries. ACM, 1999: 254-255.
3. Chien L F. PAT-tree-based keyword extraction for Chinese information retrieval[C]//ACM SIGIR Forum. ACM, 1997, 31(SI): 50-58.

第04课：了解数据必备的文本可视化技巧

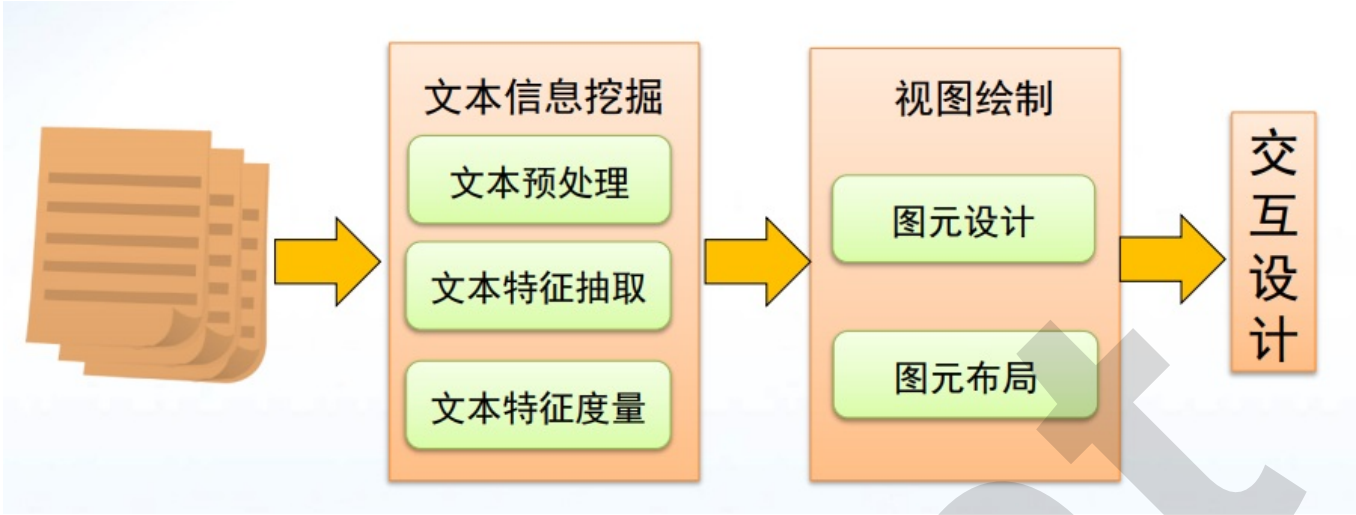
为什么要文本数据可视化

文字是传递信息最常用的载体，随着海量文本的涌现，信息超载和数据过剩等问题日益凸显，当大段大段的文字摆在面前，已经很少有人耐心、认真把它读完，人们急需一种更高效的信息接收方式，从视觉的角度出发，文本可视化正是解药良方。所谓一图胜千言，其实就是文本可视化的一种表现。

因此，文本可视化技术将文本中复杂的或者难以通过文字表达的内容和规律以视觉符号的形式表达出来，使人们能够利用与生俱来的视觉感知的并行化处理能力，快速获取文本中所蕴含的关键信息。

文本可视化的流程

文本可视化依赖于自然语言处理，因此词袋模型、命名实体识别、关键词抽取、主题分析、情感分析等是较常用的文本分析技术。文本分析的过程主要包括特征提取，通过分词、抽取、归一化等操作提取出文本词汇级的内容，利用特征构建向量空间模型并进行降维，以便将其呈现在低维空间，或者利用主题模型处理特征，最终以灵活有效的形式表示这些处理过的数据，以便进行可视化呈现。下图（来源：网络）是一个文本可视化的基本流程图：



文本可视化类型，除了包含常规的图表类，如柱状图、饼图、折线图等的表现形式，在文本领域用的比较多的可视化类型有：

（1）基于文本内容的可视化。

基于文本内容的可视化研究包括基于词频的可视化和基于词汇分布的可视化，常用的有词云、分布图和 Document Cards 等。

（2）基于文本关系的可视化。

基于文本关系的可视化研究文本内外关系，帮助人们理解文本内容和发现规律。常用的可视化形式有树状图、节点连接的网络图、力导向图、叠式图和 Word Tree 等。

（3）基于多层面信息的可视化

基于多层面信息的可视化主要研究如何结合信息的多个方面帮助用户从更深层次理解文本数据，发现其内在规律。其中，包含时间信息和地理坐标的文本可视化近年来受到越来越多的关注。常用的有地理热力图、ThemeRiver、SparkClouds、TextFlow 和基于矩阵视图的情感分析可视化等。

动手实战文本可视化

词云

在 Chat《NLP 中文短文本分类项目实践（上）》中已经讲过如何绘制 Wordcloud，这里只给出关键代码。具体过程是分词、去停用词和统计词频，然后绘制 Wordcloud 词云，这里提

供下面两种方式。

```
        #**第一种是默认的样式**
          wordcloud=WordCloud(font_path=simhei,background_color="white",max_font_size=80)
          word_frequence = {x[0]:x[1] for x in words_stat.head(1000).values}
          wordcloud=wordcloud.fit_words(word_frequence)
          #**第二种是自定义图片**
          text = " ".join(words_stat['segment'].head(100).astype(str))
          abel_mask = imread(r"china.jpg")  #这里设置了一张中国地图
          wordcloud2 = WordCloud(background_color='white',  # 设置背景颜色
              mask = abel_mask,  # 设置背景图片
              max_words = 3000,  # 设置最大现实的字数
              font_path = simhei,  # 设置字体格式
              width=2048,
              height=1024,
              scale=4.0,
              max_font_size= 300,  # 字体最大值
                                  random_state=42).generate(text)
          # 根据图片生成词云颜色
          image_colors = ImageColorGenerator(abel_mask)
          wordcloud2.recolor(color_func=image_colors)
          # 以下代码显示图片
          plt.imshow(wordcloud2)
          plt.axis("off")
          plt.show()
          wordcloud2.to_file(r'wordcloud_2.jpg') #保存结果
```

得到的词云如下图所示：

关系图

关系图法，是指用连线图来表示事物相互关系的一种方法。最常见的关系图是数据库里的 E-R 图，表示实体、关系、属性三者之间的关系。在文本可视化里面，关系图也经常被用来表示有相互关系、原因与结果和目的与手段等复杂关系，下面我们来看看如何用 Python 实现关系图制作。

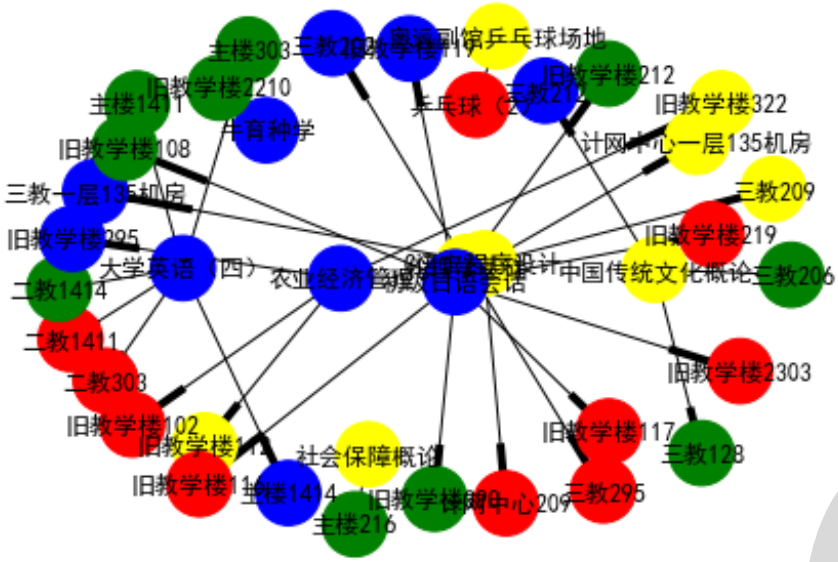
基本步骤：

- 安装 Matplotlib、NetworkX；
- 解决 Matplotlib 无法写中文问题。

我们需要知道 NetworkX 绘制关系图的数据组织结构，节点和边都是 list 格式，边的 list 里面是成对的节点。下面我们看一个真实的例子，学生课程和上课地点的关系图。

```
    classes= df['class'].values.tolist()
    classrooms=df['classroom'].values.tolist()
    nodes = list(set(classes + classrooms))
    weights = [(df.loc[index,'class'],df.loc[index,'classroom'])for index in df.index]
    weights =  list(set(weights))
    # 设置matplotlib正常显示中文
    plt.rcParams['font.sans-serif']=['SimHei']    # 用黑体显示中文
    plt.rcParams['axes.unicode_minus']=False
    colors = ['red', 'green', 'blue', 'yellow']
    #有向图
    DG = nx.DiGraph()
    #一次性添加多节点，输入的格式为列表
    DG.add_nodes_from(nodes)
    #添加边，数据格式为列表
    DG.add_edges_from(weights)
    #作图，设置节点名显示，节点大小，节点颜色
    nx.draw(DG,with_labels=True, node_size=1000, node_color = colors)
    plt.show()
```

得到的关系图如下图所示：



地理热力图

地理热力图，是以特殊高亮的形式显示用户的地理位置，借助热力图，可以直观地观察到用户的总体情况和偏好。

基本步骤：

- 安装 Folium；
- 将地理名词通过百度转换成经纬度。

在通过分词得到城市名称后，将地理名词通过百度转换成经纬度。首先注册密钥，使用百度 Web 服务 API 下的 Geocoding API 接口来获取你所需要地址的经纬度坐标，并转化为 JSON 结构的数据（个人接口，百度每天限制调用6000次），接下来定义经纬度获取函数：

```
    #经纬度转换
    def getlnglat(address):
        url = 'http://api.map.baidu.com/geocoder/v2/'
        output = 'json'
        ak = 'sqGDDvCDEZPSz24bt4b0BpKLnMk1dv6d'
        add = quote(address) #由于本文城市变量为中文，为防止乱码，先用quote进行编码
        uri = url + '?' + 'address=' + add  + '&output=' + output + '&ak=' + ak
        req = urlopen(uri)
        res = req.read().decode() #将其他编码的字符串解码成unicode
        temp = json.loads(res)  #对json数据进行解析
```

```
            return temp
```

转换后数据格式：

```
北京,116.39564503787867,39.92998577808024,840
成都,104.06792346330406,30.679942845419564,291
重庆,106.53063501341296,29.54460610888615,261
昆明,102.71460113878045,25.049153100453157,238
潍坊,119.14263382297052,36.71611487305138,214
济南,117.02496706629023,36.68278472716141,212
```

然后，使用 Folium 库进行热力图绘制地图：

```
    lat = np.array(cities["lat"][0:num])    # 获取维度之维度值
    lon = np.array(cities["lng"][0:num])    # 获取经度值
    pop = np.array(cities["count"][0:num],dtype=float)     # 获取人口数，转化为numpy浮点型
    data1 = [[lat[i],lon[i],pop[i]] for i in range(num)]     #将数据制作成[lats,lons,weights]的形式
    map_osm = folium.Map(location=[35,110],zoom_start=5)     #绘制Map，开始缩放程度是5倍
    HeatMap(data1).add_to(map_osm)  # 将热力图添加到前面建立的map里
    file_path = dir + "heatmap.html"
    map_osm.save(file_path)
```

得到的地图热力图如下图所示：

上面列举了三种典型的文本可视化方式，当然还有更多漂亮的方式。个人在开发过程中，觉得采用前端技术实现的可视化效果最好，但是这次课程仅限于 Pyhton，所以就不讲前端知识了。

最后，我推荐三个前端可视化学习网站，第一个是百度的 Echarts，基于 Canvas，适合刚入门的新手，遵循了数据可视化的一些经典范式，只要把数据组织好，就可以轻松得到很漂亮的图表；第二个推荐 D3.js，基于 SVG 方便自己定制，D3 V4 支持 Canvas+SVG，D3.js 比 Echarts 稍微难点，适合有一定开发经验的人；第三个 three.js，是一个基于 WebGL 的 3D 图形的框架，可以让用户通过 JavaScript 搭建 WebGL 项目。

第05课：面向非结构化数据转换的词袋和词向量模型

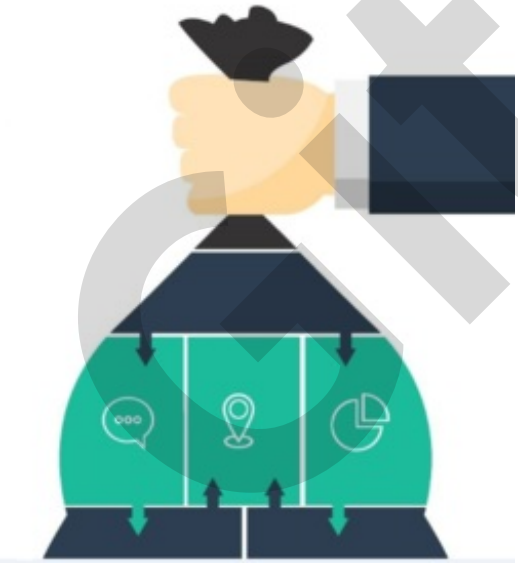
通过前面几个小节的学习，我们现在已经学会了如何获取文本预料，然后分词，在分词之后的结果上，我们可以提取文本的关键词查看文本核心思想，进而可以通过可视化技术把文档从视觉的角度表达出来。

下面，我们来看看，文本数据如何转换成计算机能够计算的数据。这里介绍两种常用的模型：词袋和词向量模型。

词袋模型（Bag of Words Model）

词袋模型的概念

先来看张图，从视觉上感受一下词袋模型的样子。



词袋模型看起来好像一个口袋把所有词都装进去，但却不完全如此。在自然语言处理和信息检索中作为一种简单假设，词袋模型把文本（段落或者文档）被看作是无序的词汇集合，忽略语法甚至是单词的顺序，把每一个单词都进行统计，同时计算每个单词出现的次数，常常被用在文本分类中，如贝叶斯算法、LDA 和 LSA 等。

动手实战词袋模型

（1）词袋模型

本例中，我们自己动手写代码看看词袋模型是如何操作的。

首先，引入 jieba 分词器、语料和停用词（标点符号集合，自己可以手动添加或者用一个文本字典代替）。

```
import jieba
#定义停用词、标点符号
punctuation = [",", "。", ":", ";", "?"]
#定义语料
content = ["机器学习带动人工智能飞速的发展。",
           "深度学习带动人工智能飞速的发展。",
           "机器学习和深度学习带动人工智能飞速的发展。"
          ]
```

接下来，我们先对语料进行分词操作，这里用到 lcut() 方法：

```
#分词
segs_1 = [jieba.lcut(con) for con in content]
print(segs_1)
```

得到分词后的结果如下：

> [['机器', '学习', '带动', '人工智能', '飞速', '的', '发展', '。'], ['深度', '学习', '带动', '人工智能', '飞速', '的', '发展', '。'], ['机器', '学习', '和', '深度', '学习', '带动', '人工智能', '飞速', '的', '发展', '。']]

因为中文语料带有停用词和标点符号，所以需要去停用词和标点符号，这里语料很小，我们直接去标点符号：

```
tokenized = []
for sentence in segs_1:
    words = []
    for word in sentence:
        if word not in punctuation:
            words.append(word)
```

```
            tokenized.append(words)
        print(tokenized)
```

去标点符号后，我们得到结果如下：

> [['机器', '学习', '带动', '人工智能', '飞速', '的', '发展'], ['深度', '学习', '带动', '人工智能', '飞速', '的', '发展'], ['机器', '学习', '和', '深度', '学习', '带动', '人工智能', '飞速', '的', '发展']]

下面操作就是把所有的分词结果放到一个袋子（List）里面，也就是取并集，再去重，获取对应的特征词。

```
    #求并集
    bag_of_words = [ x for item in segs_1 for x in item if x not in punctuation]
    #去重
    bag_of_words = list(set(bag_of_words))
    print(bag_of_words)
```

得到的特征词结果如下：

> ['飞速', '的', '深度', '人工智能', '发展', '和', '机器', '学习', '带动']

我们以上面特征词的顺序，完成词袋化：

```
    bag_of_word2vec = []
    for sentence in tokenized:
        tokens = [1 if token in sentence else 0 for token in bag_of_words ]
        bag_of_word2vec.append(tokens)
```

最后得到词袋向量：

> [[1, 1, 0, 1, 1, 0, 1, 1, 1], [1, 1, 1, 1, 1, 0, 0, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1]]
>
> 词袋的长度

上面的例子在编码时，对于 for 循环多次直接用到列表推导式。在 Python 中，列表推导式的

效率比 for 快很多，尤其在数据量大的时候效果更明显，建议多使用列表推导式。

（2）Gensim 构建词袋模型

下面我们介绍 Gensim 库的使用，继续沿用上面的例子：

```
    from gensim import corpora
    import gensim
    #tokenized是去标点之后的
    dictionary = corpora.Dictionary(tokenized)
    #保存词典
    dictionary.save('deerwester.dict')
    print(dictionary)
```

这时我们得到的结果不全，但通过提示信息可知道共9个独立的词：

> Dictionary(9 unique tokens: ['人工智能', '发展', '学习', '带动', '机器']...)

那我们如何查看所有词呢？通过下面方法，可以查看到所有词和对应的下标：

```
    #查看词典和下标 id 的映射
    print(dictionary.token2id)
```

最后结果如下：

> {'人工智能': 0, '发展': 1, '学习': 2, '带动': 3, '机器': 4, '的': 5, '飞速': 6, '深度': 7, '和': 8}

根据得到的结果，我们同样可以得到词袋模型的特征向量。这里顺带提一下函数 doc2bow()，作用只是计算每个不同单词的出现次数，将单词转换为其整数单词 id 并将结果作为稀疏向量返回。

不是词袋的长度（小）

```
corpus = [dictionary.doc2bow(sentence) for sentence in segs_1]
    print(corpus )
```

得到的稀疏向量结果如下：

> [[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1)],
> [(0, 1), (1, 1), (2, 1), (3, 1), (5, 1), (6, 1), (7, 1)],
> [(0, 1), (1, 1), (2, 2), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1)]]

词向量（Word Embedding）

深度学习带给自然语言处理最令人兴奋的突破是词向量（Word Embedding）技术。词向量技术是将词语转化成为稠密向量。在自然语言处理应用中，词向量作为机器学习、深度学习模型的特征进行输入。因此，最终模型的效果很大程度上取决于词向量的效果。

词向量的概念

在 Word2Vec 出现之前，自然语言处理经常把字词进行独热编码，也就是 One-Hot Encoder。

> 大数据 [0,0,0,0,0,0,0,1,0,……，0,0,0,0,0,0,0]
>
> 云计算[0,0,0,0,1,0,0,0,0,……，0,0,0,0,0,0,0]
>
> 机器学习[0,0,0,1,0,0,0,0,0,……，0,0,0,0,0,0,0]
>
> 人工智能[0,0,0,0,0,0,0,0,0,……，1,0,0,0,0,0,0]

比如上面的例子中，大数据 、云计算、机器学习和人工智能各对应一个向量，向量中只有一个值为1，其余都为0。所以使用 One-Hot Encoder有以下问题：

- 第一，词语编码是随机的，向量之间相互独立，看不出词语之间可能存在的关联关系。
- 第二，向量维度的大小取决于语料库中词语的多少，如果语料包含的所有词语对应的向量合为一个矩阵的话，那这个矩阵过于稀疏，并且会造成维度灾难。

而解决这个问题的手段，就是使用向量表示（Vector Representations）。比如 Word2Vec 可以将 One-Hot Encoder 转化为低维度的连续值，也就是稠密向量，并且其中意思相近的词也将被映射到向量空间中相近的位置。经过降维，在二维空间中，相似的单词在空间中的距离也很接近。

这里简单给词向量一个定义，词向量就是要用某个固定维度的向量去表示单词。也就是说要把单词变成固定维度的向量，作为机器学习（Machine Learning）或深度学习模型的特征向量输入。

动手实战词向量

（1）Word2Vec

Word2Vec 是 Google 团队2013年推出的，自提出后被广泛应用在自然语言处理任务中，并且受到它的启发，后续出现了更多形式的词向量模型。Word2Vec 主要包含两种模型：Skip-Gram 和 CBOW，值得一提的是，Word2Vec 词向量可以较好地表达不同词之间的相似和类比关系。

下面我们通过代码实战来体验一下 Word2Vec。通过 `pip install gensim` 安装好库后，即可导入使用。

先导入 Gensim 中的 Word2Vec 和 jieba 分词器，再引入从百度百科抓取的黄河和长江的语料：

```
from gensim.models import Word2Vec
```

```
    import jieba
    #定义停用词、标点符号
    punctuation = [",","。", ":", ";", ".", "'", '"', "’", "?", "/", "-", "+", "&", "(", ")"]
    sentences = [
    "长江是中国第一大河，干流全长6397公里（以沱沱河为源），一般称6300公里。流域总面积一百八十余万平方公里，年平均入海水量约九千六百余亿立方米。以干流长度和入海水量论，长江均居世界第三位。",
    "黄河，中国古代也称河，发源于中华人民共和国青海省巴颜喀拉山脉，流经青海、四川、甘肃、宁夏、内蒙古、陕西、山西、河南、山东9个省区，最后于山东省东营垦利县注入渤海。干流河道全长5464千米，仅次于长江，为中国第二长河。黄河还是世界第五长河。",
    "黄河,是中华民族的母亲河。作为中华文明的发祥地,维系炎黄子孙的血脉.是中华民族民族精神与民族情感的象征。",
    "黄河被称为中华文明的母亲河。公元前2000多年华夏族在黄河领域的中原地区形成、繁衍。",
    "在兰州的"黄河第一桥"内蒙古托克托县河口镇以上的黄河河段为黄河上游。",
    "黄河上游根据河道特性的不同，又可分为河源段、峡谷段和冲积平原三部分。 ",
    "黄河,是中华民族的母亲河。"
    ]
```

上面定义好语料，接下来进行分词，去标点符号操作 ：

```
    sentences = [jieba.lcut(sen) for sen in sentences]
    tokenized = []
    for sentence in sentences:
        words = []
        for word in sentence:
            if word not in punctuation:
                words.append(word)
        tokenized.append(words)
```

这样我们获取的语料在分词之后，去掉了标点符号，如果做得更严谨，大家可以去停用词，然后进行模型训练：

```
    model = Word2Vec(tokenized, sg=1, size=100,  window=5,  min_count=2,  negative=1, sample=0.001, hs=1, workers=4)
```

参数解释如下：

- sg=1 是 `skip-gram` 算法，对低频词敏感；默认 sg=0 为 CBOW 算法。
- size 是输出词向量的维数，值太小会导致词映射因为冲突而影响结果，值太大则会耗内存

并使算法计算变慢，一般值取为100到200之间。

- window 是句子中当前词与目标词之间的最大距离，3表示在目标词前看3-b 个词，后面看 b 个词（ b 在0-3之间随机）。
- `min_count` 是对词进行过滤，频率小于 `min-count` 的单词则会被忽视，默认值为5。
- negative 和 sample 可根据训练结果进行微调，sample 表示更高频率的词被随机下采样到所设置的阈值，默认值为 1e-3。
- hs=1 表示层级 softmax 将会被使用，默认 hs=0 且 negative 不为0，则负采样将会被选择使用。
- 详细参数说明可查看 Word2Vec 源代码。

训练后的模型可以保存与加载，如下代码所示：

```
1.      model.save(model)   #保存模型
2.      model = Word2Vec.load(model)    #加载模型
```

模型训练好之后，接下来就可以使用模型，可以用来计算句子或者词的相似性、最大匹配程度等。

例如，我们判断一下黄河和黄河自己的相似度：

```
1.      print(model.similarity('黄河', '黄河'))
```

结果输出为：

```
1.0000000000000002
```

例如，当输入黄河和长江来计算相似度的时候，结果就比较小，因为我们的语料实在太小了。

```
1.      print(model.similarity('黄河', '长江'))
```

结果输出为：

```
-0.036808977457324699
```

下面我们预测最接近的词，预测与黄河和母亲河最接近，而与长江不接近的词：

```
print(model.most_similar(positive=['黄河', '母亲河'], negative=['长江']))
```

得到结果如下，可以根据相似度大小找到与黄河和母亲河最接近的词（实际处理建议增大数据量和去停用词）。

```
[('是', 0.14632007479667664),
('以', 0.14630728960037231),
('长河', 0.13878652453422546),
('河道', 0.13716217875480652),
('在', 0.11577725410461426),
('全长', 0.10969121754169464),
('内蒙古', 0.07590540498495102),
('入海', 0.06970417499542236),
('民族', 0.06064444035291672),
('中华文明', 0.057667165994644165)]
```

上面通过小数据量的语料实战，加强了对 Word2Vec 的理解，总之 Word2Vec 是一种将词变成词向量的工具。通俗点说，只有这样文本预料才转化为计算机能够计算的矩阵向量。

（2）Doc2Vec

Doc2Vec 是 Mikolov 在 Word2Vec 基础上提出的另一个用于计算长文本向量的工具。在 Gensim 库中，Doc2Vec 与 Word2Vec 都极为相似。但两者在对输入数据的预处理上稍有不同，Doc2vec 接收一个由 ~~LabeledSentence~~ TaggedDocument 对象组成的迭代器作为其构造函数的输入参数。其中，LabeledSentence 是 Gensim 内建的一个类，它接收两个 List 作为其初始化的参数：word list 和 label list。

Doc2Vec 也包括两种实现方式：DBOW（Distributed Bag of Words）和 DM（Distributed Memory）。DBOW 和 DM 的实现，二者在 gensim 库中的实现用的是同一个方法，该方法中参数 dm = 0 或者 dm=1 决定调用 DBOW 还是 DM。Doc2Vec 将文档语料通过一个固定长度的向量表达。

下面是 Gensim 中 Doc2Vec 模型的实战，我们把上述语料每一句话当做一个文本，添加上对应的标签。接下来，定义数据预处理类，作用是给每个文章添加对应的标签：

```
    #定义数据预处理类，作用是给每个文章添加对应的标签                    TaggedDocument
    from gensim.models.doc2vec import Doc2Vec,LabeledSentence
    doc_labels = ["长江","黄河","黄河","黄河","黄河","黄河","黄河"]
    class LabeledLineSentence(object):
        def __init__(self, doc_list, labels_list):
           self.labels_list = labels_list
           self.doc_list = doc_list
        def __iter__(self):
            for idx, doc in enumerate(self.doc_list):
                yield LabeledSentence(words=doc,tags=[self.labels_list[
idx]])                          TaggedDocument

        model = Doc2Vec(documents,dm=1, size=100, window=8, min_count=5
, workers=4)
        model.save(model )
        model = Doc2Vec.load(model )
```

上面定义好了数据预处理函数，我们将 Word2Vec 中分词去标点后的数据，进行转换：

```
    iter_data = LabeledLineSentence(tokenized, doc_labels)
```

得到一个数据集，我开始定义模型参数，这里 dm=1，采用了 Gensim 中的 DM 实现。

```
    model = Doc2Vec(dm=1, size=100, window=8, min_count=5, workers=4)
    model.build_vocab(iter_data)
```

接下来训练模型， 设置迭代次数1000次， `start_alpha` 为开始学习率， `end_alpha` 与 `start_alpha` 线性递减。

```
    model.train(iter_data,total_examples=model.corpus_count,epochs=1000
,start_alpha=0.01,end_alpha =0.001)
```

最后我们对模型进行一些预测：

```
    #根据标签找最相似的，这里只有黄河和长江，所以结果为长江，并计算出了相似度
    print(model.docvecs.most_similar('黄河'))
```

得到的结果：

```
[('长江', 0.25543850660324097)]
```

然后对黄河和长江标签做相似性计算：

```
print(model.docvecs.similarity('黄河','长江'))
```

得到的结果：

```
0.25543848271351405
```

上面只是在小数据量进行的小练习，而最终影响模型准确率的因素有：文档的数量越多，文档的相似性越好，也就是基于大数据量的模型训练。在工业界，Word2Vec 和 Doc2Vec 常见的应用有：做相似词计算；相关词挖掘，在推荐系统中用在品牌、用户、商品挖掘中；上下文预测句子；机器翻译；作为特征输入其他模型等。

总结，本文只是简单的介绍了词袋和词向量模型的典型应用，对于两者的理论和其他词向量模型，比如 TextRank 、FastText 和 GloVe 等，阅读文末给出参考文献将了解更多。

参考文献：

1. https://radimrehurek.com/gensim/tut1.html
2. https://radimrehurek.com/gensim/models/word2vec.html
3. https://radimrehurek.com/gensim/summarization/summariser.html
4. https://radimrehurek.com/gensim/models/fasttext.html
5. https://nlp.stanford.edu/projects/glove/

第06课：动手实战基于 ML 的中文短文本分类

文本分类，属于有监督学习中的一部分，在很多场景下都有应用，下面通过小数据的实例，一步步完成中文短文本的分类实现，整个过程尽量做到少理论重实战。

开发环境，我们选择：

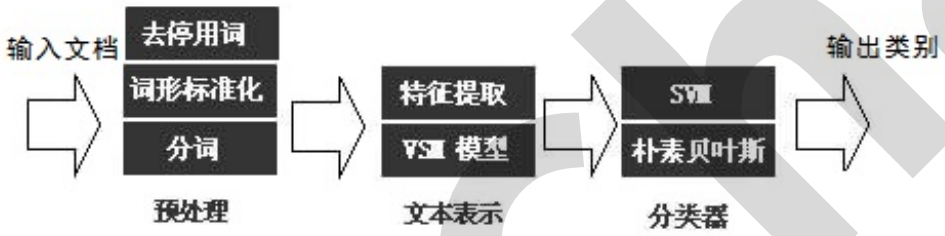
1. Windows 系统
2. Python 3.6
3. Jupyter Notebook

本文使用的数据是我曾经做过的一份司法数据，需求是对每一条输入数据，判断事情的主体是谁，比如报警人被老公打，报警人被老婆打，报警人被儿子打，报警人被女儿打等来进行文本有监督的分类操作。

整个过程分为以下几个步骤：

- 语料加载
- 分词
- 去停用词
- 抽取词向量特征
- 分别进行算法建模和模型训练
- 评估、计算 AUC 值
- 模型对比

基本流程如下图所示：



下面开始项目实战。

1. 首先进行语料加载，在这之前，引入所需要的 Python 依赖包，并将全部语料和停用词字典读入内存中。

第一步，引入依赖库，有随机数库、jieba 分词、pandas 库等：

```
1.      import random
2.      import jieba
3.      import pandas as pd
```

第二步，加载停用词字典，停用词词典为 stopwords.txt 文件，可以根据场景自己在该文本里面添加要去除的词（比如冠词、人称、数字等特定词）：

```
1.      #加载停用词
2.      stopwords=pd.read_csv('stopwords.txt',index_col=False,quoting=3,sep
    ="\t",names=['stopword'], encoding='utf-8')
3.      stopwords=stopwords['stopword'].values
```

第三步，加载语料，语料是4个已经分好类的 csv 文件，直接用 pandas 加载即可，加载之后可以首先删除 nan 行，并提取要分词的 content 列转换为 list 列表：

```
    #加载语料
    laogong_df = pd.read_csv('beilaogongda.csv', encoding='utf-8', sep=',')
    laopo_df = pd.read_csv('beilaogongda.csv', encoding='utf-8', sep=',')
    erzi_df = pd.read_csv('beierzida.csv', encoding='utf-8', sep=',')
    nver_df = pd.read_csv('beinverda.csv', encoding='utf-8', sep=',')
    #删除语料的nan行
    laogong_df.dropna(inplace=True)
    laopo_df.dropna(inplace=True)
    erzi_df.dropna(inplace=True)
    nver_df.dropna(inplace=True)
    #转换
    laogong = laogong_df.segment.values.tolist()
    laopo = laopo_df.segment.values.tolist()
    erzi = erzi_df.segment.values.tolist()
    nver = nver_df.segment.values.tolist()
```

2. 分词和去停用词。

第一步，定义分词、去停用词和批量打标签的函数，函数包含3个参数：`content_lines` 参数为语料列表；sentences 参数为预先定义的 list，用来存储分词并打标签后的结果；category 参数为标签 ：

```
    #定义分词和打标签函数preprocess_text
    #参数content_lines即为上面转换的list
    #参数sentences是定义的空list，用来储存打标签之后的数据
    #参数category 是类型标签
    def preprocess_text(content_lines, sentences, category):
        for line in content_lines:
            try:
                segs=jieba.lcut(line)
                segs = [v for v in segs if not str(v).isdigit()]#去数字
                segs = list(filter(lambda x:x.strip(), segs))    #去左右空格
                segs = list(filter(lambda x:len(x)>1, segs)) #长度为1的字符
```

```
                segs = list(filter(lambda x:x not in stopwords, segs))
#去掉停用词
                sentences.append((" ".join(segs), category))# 打标签
            except Exception:
                print(line)
                continue
```

第二步，调用函数、生成训练数据，根据我提供的司法语料数据，分为报警人被老公打，报警人被老婆打，报警人被儿子打，报警人被女儿打，标签分别为0、1、2、3，具体如下：

```
    sentences = []
    preprocess_text(laogong, sentences,0)
    preprocess_text(laopo, sentences, 1)
    preprocess_text(erzi, sentences, 2)
    preprocess_text(nver, sentences, 3)
```

第三步，将得到的数据集打散，生成更可靠的训练集分布，避免同类数据分布不均匀：

```
    random.shuffle(sentences)
```

第四步，我们在控制台输出前10条数据，观察一下：

```
    for sentence in sentences[:10]:
        print(sentence[0], sentence[1])  #下标0是词列表，1是标签
```

得到的结果如图所示：

```
报警 老公 持械 老公 项链 上址 民警 到场 民警 携带 防护 装备 0
报警 老公 拿头 民警 到场 0
报警 人称 老公 持械 无人 0
老公 人伤 无需 民警 1
报警 人称 老公 民警 到场 0
丈夫 拖鞋 无人 民警 到场 1
报警 丈夫 民警 到场 1
报警 儿子 无需 救护 民警 到场 2
家暴 老公 持械 人伤 无需 救护 民警 到场 1
报警 人称 儿子 民警 到场 2
```

3. 抽取词向量特征。

第一步，抽取特征，我们定义文本抽取词袋模型特征：

```
    from sklearn.feature_extraction.text import CountVectorizer
    vec = CountVectorizer(
        analyzer='word', # tokenise by character ngrams
        max_features=4000,  # keep the most common 1000 ngrams
    )
```

第二步，把语料数据切分，用 `sk-learn` 对数据切分，分成训练集和测试集：

```
    from sklearn.model_selection import train_test_split
    x, y = zip(*sentences)
    x_train, x_test, y_train, y_test = train_test_split(x, y, random_st
ate=1256)
```

第三步，把训练数据转换为词袋模型：

```
    vec.fit(x_train)
```

4. 分别进行算法建模和模型训练。

定义朴素贝叶斯模型，然后对训练集进行模型训练，直接使用 sklearn 中的 MultinomialNB：

```
    from sklearn.naive_bayes import MultinomialNB
    classifier = MultinomialNB()
    classifier.fit(vec.transform(x_train), y_train)
```

5. 评估、计算 AUC 值。

第一步，上面步骤1-4完成了从语料到模型的训练，训练之后，我们要用测试集来计算 AUC 值：

```
    print(classifier.score(vec.transform(x_test), y_test))
```

得到的结果评分为：0.647331786543。

第二步，进行测试集的预测：

```
    pre = classifier.predict(vec.transform(x_test))
```

6. 模型对比。

整个模型从语料到训练评估步骤1-5就完成了，接下来我们来看看，改变特征向量模型和训练模型对结果有什么变化。

（1）改变特征向量模型

下面可以把特征做得更强一点，尝试加入抽取 `2-gram` 和 `3-gram` 的统计特征，把词库的量放大一点。

```
    from sklearn.feature_extraction.text import CountVectorizer
    vec = CountVectorizer(
        analyzer='word', # tokenise by character ngrams
        ngram_range=(1,4),  # use ngrams of size 1 and 2
        max_features=20000,  # keep the most common 1000 ngrams
    )
    vec.fit(x_train)
    #用朴素贝叶斯算法进行模型训练
    classifier = MultinomialNB()
    classifier.fit(vec.transform(x_train), y_train)
    #对结果进行评分
    print(classifier.score(vec.transform(x_test), y_test))
```

得到的结果评分为：0.649651972158，确实有一点提高，但是不太明显。

（2）改变训练模型

使用 SVM 训练：

```
    from sklearn.svm import SVC
    svm = SVC(kernel='linear')
    svm.fit(vec.transform(x_train), y_train)
    print(svm.score(vec.transform(x_test), y_test))
```

使用决策树、随机森林、XGBoost、神经网络等等：

```
    import xgboost as xgb
    from sklearn.model_selection import StratifiedKFold
    import numpy as np
    # xgb矩阵赋值
    xgb_train = xgb.DMatrix(vec.transform(x_train), label=y_train)
    xgb_test = xgb.DMatrix(vec.transform(x_test))
```

在 XGBoost 中，下面主要是调参指标，可以根据参数进行调参：

```
    params = {
            'booster': 'gbtree',      #使用gbtree
            'objective': 'multi:softmax',  # 多分类的问题、
            # 'objective': 'multi:softprob',   # 多分类概率
            #'objective': 'binary:logistic',  #二分类
            'eval_metric': 'merror',   #logloss
            'num_class': 4,  # 类别数，与 multisoftmax 并用
            'gamma': 0.1,  # 用于控制是否后剪枝的参数,越大越保守，一般0.1、0.2这样子。
            'max_depth': 8,  # 构建树的深度，越大越容易过拟合
            'alpha': 0,   # L1正则化系数
            'lambda': 10,  # 控制模型复杂度的权重值的L2正则化项参数，参数越大，模型越不容易过拟合。
            'subsample': 0.7,  # 随机采样训练样本
            'colsample_bytree': 0.5,  # 生成树时进行的列采样
            'min_child_weight': 3,
            # 这个参数默认是 1，是每个叶子里面 h 的和至少是多少，对正负样本不均衡时的 0-1 分类而言
            # 假设 h 在 0.01 附近，min_child_weight 为 1 叶子节点中最少需要包含 100 个样本。
            'silent': 0,  # 设置成1则没有运行信息输出，最好是设置为0.
            'eta': 0.03,  # 如同学习率
            'seed': 1000,
            'nthread': -1,  # cpu 线程数
            'missing': 1
        }
```

总结

上面通过真实司法数据，一步步实现中文短文本分类的方法，整个示例代码可以当做模板来用，从优化和提高模型准确率来说，主要有两方面可以尝试：

1. 特征向量的构建，除了词袋模型，可以考虑使用 word2vec 和 doc2vec 等；

2. 模型上可以选择有监督的分类算法、集成学习以及神经网络等。

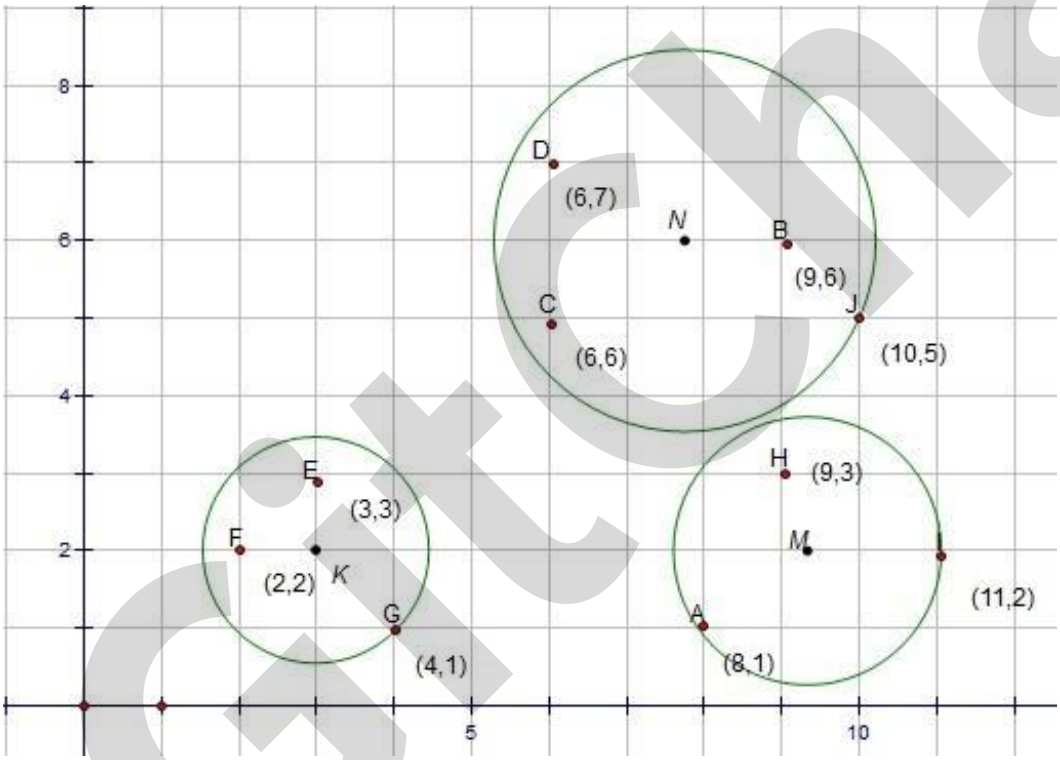
最后如果想了解更多，推荐我的两篇 Chat 文章：

- NLP 中文短文本分类项目实践（上）
- NLP 中文短文本分类项目实践（下）

第07课：动手实战基于 ML 的中文短文本聚类

关于文本聚类，我曾在 Chat《NLP 中文文本聚类之无监督学习》中介绍过，文本聚类是将一个个文档由原有的自然语言文字信息转化成数学信息，以高维空间点的形式展现出来，通过计算哪些点距离比较近，从而将那些点聚成一个簇，簇的中心叫做簇心。一个好的聚类要保证簇内点的距离尽量的近，但簇与簇之间的点要尽量的远。

如下图，以 K、M、N 三个点分别为聚类的簇心，将结果聚为三类，使得簇内点的距离尽量的近，但簇与簇之间的点尽量的远。



开发环境，我们选择：：

1. Windows 系统
2. Python 3.6
3. Jupyter Notebook

本文继续沿用上篇文本分类中的语料来进行文本无监督聚类操作。

整个过程分为以下几个步骤：

- 语料加载
- 分词
- 去停用词
- 抽取词向量特征
- 实战 TF-IDF 的中文文本 K-means 聚类
- 实战 word2Vec 的中文文本 K-means 聚类

下面开始项目实战。

1. 首先进行语料加载，在这之前，引入所需要的 Python 依赖包，并将全部语料和停用词字典读入内存中。

第一步，引入依赖库，有随机数库、jieba 分词、pandas 库等：

```
import random
import jieba
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
import gensim
from gensim.models import Word2Vec
from sklearn.preprocessing import scale
import multiprocessing
```

第二步，加载停用词字典，停用词词典为 stopwords.txt 文件，可以根据场景自己在该文本里面添加要去除的词（比如冠词、人称、数字等特定词）：

```
#加载停用词
stopwords=pd.read_csv('stopwords.txt',index_col=False,quoting=3,sep="\t",names=['stopword'], encoding='utf-8')
stopwords=stopwords['stopword'].values
```

第三步，加载语料，语料是4个已经分好类的 csv 文件，直接用 pandas 加载即可，加载之后可以首先删除 nan 行，并提取要分词的 content 列转换为 list 列表：

```
    #加载语料
    laogong_df = pd.read_csv('beilaogongda.csv', encoding='utf-8', sep=',')
    laopo_df = pd.read_csv('beilaogongda.csv', encoding='utf-8', sep=',')
    erzi_df = pd.read_csv('beierzida.csv', encoding='utf-8', sep=',')
    nver_df = pd.read_csv('beinverda.csv', encoding='utf-8', sep=',')
    #删除语料的nan行
    laogong_df.dropna(inplace=True)
    laopo_df.dropna(inplace=True)
    erzi_df.dropna(inplace=True)
    nver_df.dropna(inplace=True)
    #转换
    laogong = laogong_df.segment.values.tolist()
    laopo = laopo_df.segment.values.tolist()
    erzi = erzi_df.segment.values.tolist()
    nver = nver_df.segment.values.tolist()
```

2. 分词和去停用词。

第一步，定义分词、去停用词和批量打标签的函数，函数包含两个参数：`content_lines` 参数为语料列表；sentences 参数为预先定义的 list，用来存储分词并打标签后的结果：

```
    #定义分词和打标签函数preprocess_text
    #参数content_lines即为上面转换的list
    #参数sentences是定义的空list，用来储存打标签之后的数据
    #参数category 是类型标签
    def preprocess_text(content_lines, sentences):
        for line in content_lines:
            try:
                segs=jieba.lcut(line)
                segs = [v for v in segs if not str(v).isdigit()]#去数字
                segs = list(filter(lambda x:x.strip(), segs))   #去左右空格
                segs = list(filter(lambda x:len(x)>1, segs)) #长度为1的字符
                segs = list(filter(lambda x:x not in stopwords, segs)) #去掉停用词
                sentences.append(" ".join(segs))# 打标签
            except Exception:
                print(line)
                continue
```

第二步，调用函数、生成训练数据，根据我提供的司法语料数据，分为报警人被老公打，报警人被老婆打，报警人被儿子打，报警人被女儿打，标签分别为0、1、2、3，具体如下：

```
    sentences = []
    preprocess_text(laogong, sentences)
    preprocess_text(laopo, sentences)
    preprocess_text(erzi, sentences)
    preprocess_text(nver, sentences)
```

第三步，将得到的数据集打散，生成更可靠的训练集分布，避免同类数据分布不均匀：

```
random.shuffle(sentences)
```

第四步，我们在控制台输出前10条数据，观察一下（因为上面做了随机打散，你看到的前10条可能不一样）：

```
    for sentence in sentences[:10]:
        print(sentence[0], sentence[1])  #下标0是词列表，1是标签
```

得到的结果聚类和分类是不同的，这里没有标签：

```
报警 老公 东西 持械 人无事 民警 到场 民警 携带 防护 设备
家庭 纠纷 报警 儿子 人无事 民警 到场
称其 小姑 打电话 其称 老公 不清 报警 人称 稍后 民警 到场
报警 老公 打伤 民警 到场
报警 人称 儿子 持械 人无事 民警 携带 防护 设备
女儿 民警 携带 防护 装备 民警 到场
家庭 纠纷 报警 人称 儿子 持械 人伤 无需 救护 民警 到场
家庭 矛盾 报警 儿子 民警 到场
报警 人称 丈夫 持械 人伤 无需 民警 注意安全 民警 到场
报警 人称 丈夫 人伤 无需 救护车 民警 到场
```

3. 抽取词向量特征。

抽取特征，将文本中的词语转换为词频矩阵，统计每个词语的 `tf-idf` 权值，获得词在对应文本中的 `tf-idf` 权重：

```
    #将文本中的词语转换为词频矩阵 矩阵元素a[i][j] 表示j词在i类文本下的词频
    vectorizer = TfidfVectorizer(sublinear_tf=True, max_df=0.5)
    #统计每个词语的tf-idf权值
    transformer = TfidfTransformer()
```

```
    # 第一个fit_transform是计算tf-idf 第二个fit_transform是将文本转为词频矩阵
    tfidf = transformer.fit_transform(vectorizer.fit_transform(sentences))
    # 获取词袋模型中的所有词语
    word = vectorizer.get_feature_names()
    # 将tf-idf矩阵抽取出来，元素w[i][j]表示j词在i类文本中的tf-idf权重
    weight = tfidf.toarray()
    #查看特征大小
    print ('Features length: ' + str(len(word)))
```

4. 实战 `TF-IDF` 的中文文本 `K-means` 聚类

第一步，使用 `k-means++` 来初始化模型，当然也可以选择随机初始化，即 `init="random"`，然后通过 PCA 降维把上面的权重 weight 降到10维，进行聚类模型训练：

```
    numClass=4 #聚类分几簇
    clf = KMeans(n_clusters=numClass, max_iter=10000, init="k-means++", tol=1e-6)  #这里也可以选择随机初始化init="random"
    pca = PCA(n_components=10)  # 降维
    TnewData = pca.fit_transform(weight)  # 载入N维
    s = clf.fit(TnewData)
```

第二步，定义聚类结果可视化函数 `plot_cluster(result,newData,numClass)`，该函数包含3个参数，其中 result 表示聚类拟合的结果集；newData 表示权重 weight 降维的结果，这里需要降维到2维，即平面可视化；numClass 表示聚类分为几簇，绘制代码第一部分绘制结果 newData，第二部分绘制聚类的中心点：

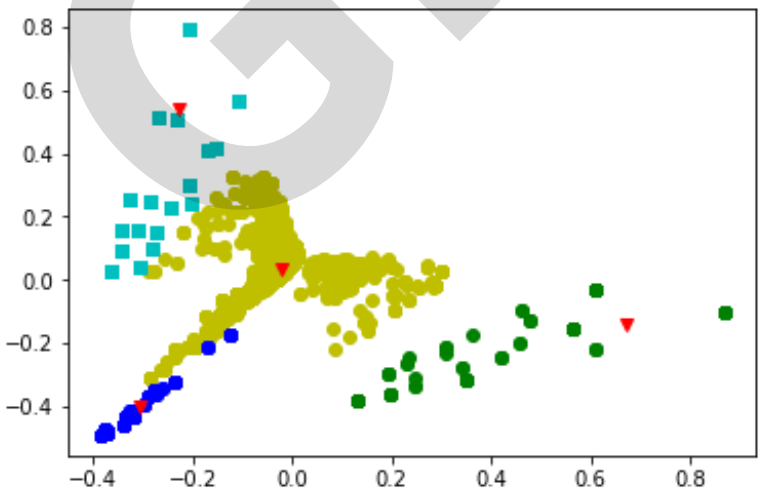
```
    def plot_cluster(result,newData,numClass):
        plt.figure(2)
        Lab = [[] for i in range(numClass)]
        index = 0
        for labi in result:
            Lab[labi].append(index)
            index += 1
        color = ['oy', 'ob', 'og', 'cs', 'ms', 'bs', 'ks', 'ys', 'yv', 'mv', 'bv', 'kv', 'gv', 'y^', 'm^', 'b^', 'k^',
                 'g^'] * 3
        for i in range(numClass):
            x1 = []
            y1 = []
            for ind1 in newData[Lab[i]]:
```

```
                    # print ind1
                    try:
                        y1.append(ind1[1])
                        x1.append(ind1[0])
                    except:
                        pass
                plt.plot(x1, y1, color[i])

            #绘制初始中心点
            x1 = []
            y1 = []
            for ind1 in clf.cluster_centers_:
                try:
                    y1.append(ind1[1])
                    x1.append(ind1[0])
                except:
                    pass
            plt.plot(x1, y1, "rv") #绘制中心
            plt.show()
```

第三步，对数据降维到2维，然后获得结果，最后绘制聚类结果图：

```
        pca = PCA(n_components=2)   # 输出两维
        newData = pca.fit_transform(weight)   # 载入N维
        result = list(clf.predict(TnewData))
        plot_cluster(result,newData,numClass)
```

第四步，得到的聚类结果图，4个中心点和4个簇，我们看到结果还比较好，簇的边界很清楚：



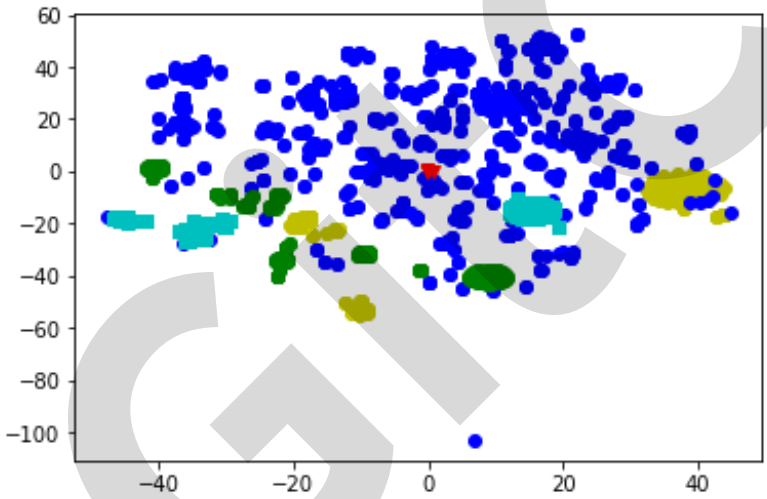
第五步，上面演示的可视化过程，降维使用了 PCA，我们还可以试试 TSNE，两者同为降维工具，主要区别在于，所在的包不同（也即机制和原理不同）：

```
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

因为原理不同，导致 TSNE 保留下的属性信息，更具代表性，也即最能体现样本间的差异，但是 TSNE 运行极慢，PCA 则相对较快，下面看看 TSNE 运行的可视化结果：

```
from sklearn.manifold import TSNE
ts =TSNE(2)
newData = ts.fit_transform(weight)
result = list(clf.predict(TnewData))
plot_cluster(result,newData,numClass)
```

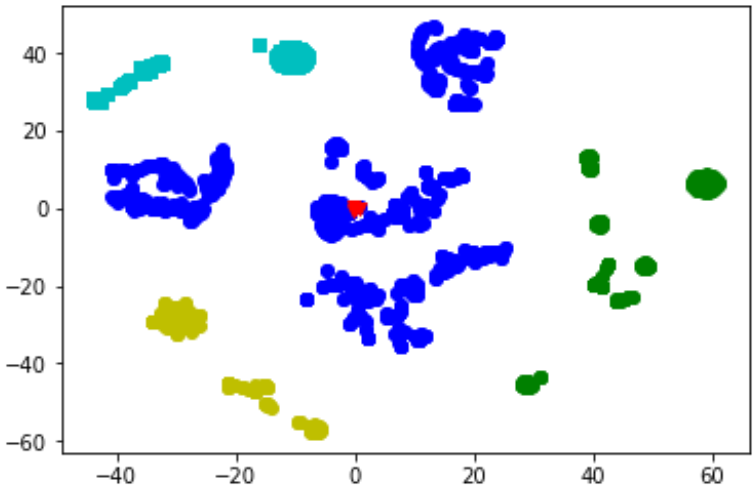
得到的可视化结果，为一个中心点，不同簇落在围绕中心点的不同半径之内，我们看到在这里结果并不是很好：



第六步，为了更好的表达和获取更具有代表性的信息，在展示（可视化）高维数据时，更为一般的处理，常常先用 PCA 进行降维，再使用 TSNE：

```
from sklearn.manifold import TSNE
newData = PCA(n_components=4).fit_transform(weight)   # 载入N维
newData =TSNE(2).fit_transform(newData)
result = list(clf.predict(TnewData))
plot_cluster(result,newData,numClass)
```

得到的可视化结果，不同簇落在围绕中心点的不同半径之内：



总结

上面通过真实小案例，对司法数据一步步实现中文短文本聚类，从优化和提高模型准确率来说，主要有两方面可以尝试：

1. 特征向量的构建，除了词袋模型，可以考虑使用 word2vec 和 doc2vec 等；
2. 模型上可以采用基于密度的 DBSCAN、层次聚类等算法。

第08课：从自然语言处理角度看 HMM 和 CRF

近几年在自然语言处理领域中，HMM（隐马尔可夫模型）和 CRF（条件随机场）算法常常被用于分词、句法分析、命名实体识别、词性标注等。由于两者之间有很大的共同点，所以在很多应用上往往是重叠的，但在命名实体、句法分析等领域 CRF 似乎更胜一筹。通常来说如果做自然语言处理，这两个模型应该都要了解，下面我们来看看本文的内容。

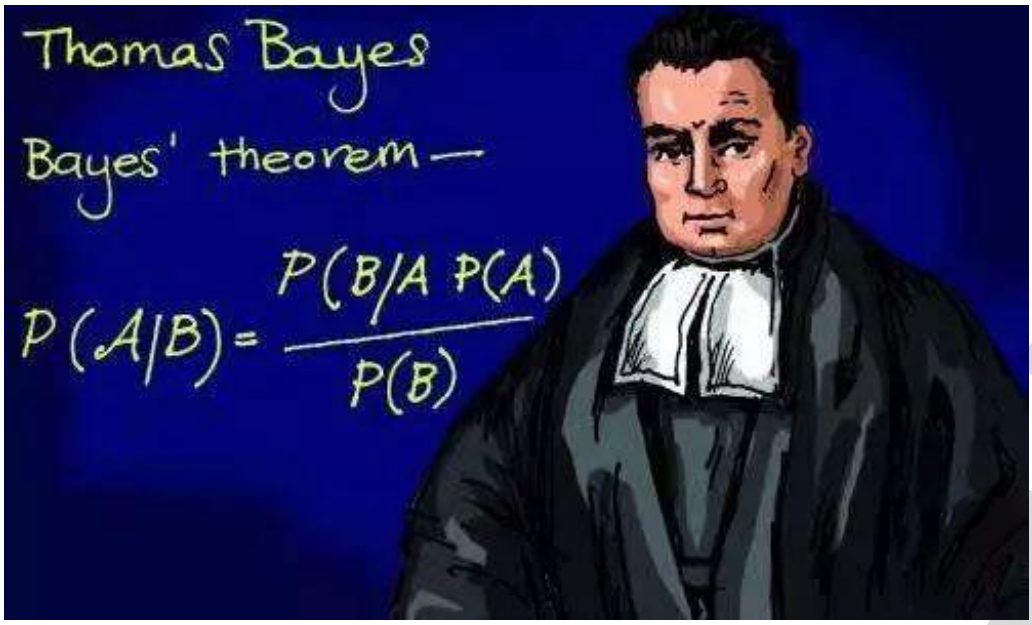
从贝叶斯定义理解生成式模型和判别式模型

理解 HMM（隐马尔可夫模型）和 CRF（条件随机场）模型之前，我们先来看两个概念：生成式模型和判别式模型。

在机器学习中，生成式模型和判别式模型都用于有监督学习，有监督学习的任务就是从数据中学习一个模型（也叫分类器），应用这一模型，对给定的输入 X 预测相应的输出 Y。这个模型的一般形式为：决策函数 Y=f(X) 或者条件概率分布 P(Y|X)。

首先，简单从贝叶斯定理说起，若记 P(A)、P(B) 分别表示事件 A 和事件 B 发生的概率，则 P(A|B) 表示事件 B 发生的情况下事件 A 发生的概率；P(AB)表示事件 A 和事件 B 同时发生的概率。

根据贝叶斯公式可以得出：



生成式模型：估计的是联合概率分布，P(Y, X)=P(Y|X)*P(X)，由联合概率密度分布 P(X,Y)，然后求出条件概率分布 P(Y|X) 作为预测的模型，即生成模型公式为：P(Y|X)= P(X,Y)/ P(X)。基本思想是首先建立样本的联合概率密度模型 P(X,Y)，然后再得到后验概率 P(Y|X)，再利用它进行分类，其主要关心的是给定输入 X 产生输出 Y 的生成关系。

判别式模型：估计的是条件概率分布， P(Y|X)，是给定观测变量 X 和目标变量 Y 的条件模型。由数据直接学习决策函数 Y=f(X) 或者条件概率分布 P(Y|X) 作为预测的模型，其主要关心的是对于给定的输入 X，应该预测什么样的输出 Y。

所以，HMM 使用隐含变量生成可观测状态，其生成概率有标注集统计得到，是一个生成模型。其他常见的生成式模型有：Gaussian、 Naive Bayes、Mixtures of multinomials 等。

而 CRF 就像一个反向的隐马尔可夫模型（HMM），通过可观测状态判别隐含变量，其概率亦通过标注集统计得来，是一个判别模型。其他常见的判别式模型有：K 近邻法、感知机、决策树、逻辑斯谛回归模型、最大熵模型、支持向量机、提升方法等。

HMM（隐马尔可夫模型）和 CRF（条件随机场）的理论部分，推荐看周志华老师的西瓜书《机器学习》。

动手实战：基于 HMM 训练自己的 Python 中文分词器

模型介绍

HMM 模型是由一个“五元组”组成的集合：

- StatusSet：状态值集合，状态值集合为 (B, M, E, S)，其中 B 为词的首个字，M 为词中间的字，E 为词语中最后一个字，S 为单个字，B、M、E、S 每个状态代表的是该字在词语中的位置。

  举个例子，对“中国的人工智能发展进入高潮阶段”，分词可以标注为：“中B国E的S人B工E智B能E发B展E进B入E高B潮E阶B段E”，最后的分词结果为：['中国', '的', '人工', '智能', '发展', '进入', '高潮', '阶段']。
- ObservedSet：观察值集合，观察值集合就是所有语料的汉字，甚至包括标点符号所组成的集合。
- TransProbMatrix：转移概率矩阵，状态转移概率矩阵的含义就是从状态 X 转移到状态 Y 的概率，是一个4×4的矩阵，即 {B,E,M,S}×{B,E,M,S}。
- EmitProbMatrix：发射概率矩阵，发射概率矩阵的每个元素都是一个条件概率，代表 P(Observed[i]|Status[j]) 概率。
- InitStatus：初始状态分布，初始状态概率分布表示句子的第一个字属于 {B,E,M,S} 这四种状态的概率。

将 HMM 应用在分词上，要解决的问题是：参数（ObservedSet、TransProbMatrix、EmitRobMatrix、InitStatus）已知的情况下，求解状态值序列。

解决这个问题的最有名的方法是 Viterbi 算法。

语料准备

本次训练使用的预料 `syj_trainCorpus_utf8.txt` 是我爬取的短文本处理生成的。整个语料大小 264M，包含1116903条数据，UTF-8 编码，词与词之间用空格隔开，用来训练分词模型。

语料已上传到 CSDN 资源上，下载地址请点击：中文自然语言处理中文分词训练语料 。

语料格式，用空格隔开的：

> 如果 继续 听任 资产阶级 自由化 的 思潮 泛滥 ，
>
> 党 就 失去 了 凝聚力 和 战斗力 ，

> 怎么 能 成为 全国 人民 的 领导 核心 ？
>
> 中国 又 会 成为 一盘散沙 ，
>
> 那 还有 什么 希望 ？

编码实现

（1）预定义

首先引出库，这两个库的作用是用来模型保存的：

```
    import pickle
    import json
```

接下来定义 HMM 中的状态，初始化概率，以及中文停顿词：

```
    STATES = {'B', 'M', 'E', 'S'}
    EPS = 0.0001
    #定义停顿标点
    seg_stop_words = {" ",", ",",。 ","“”","〝', "？", "！", "：", "《",
"》 ", "、 ", "；", ".", "\ ", "′", "—", "，", ".", "?", "!", "`", "~",
"@", "#", "$", "%", "^", "&", "*", "(", ")", "-", "_", "+", "=", "[", "]
", "{", "}", '"', "'", "<", ">", "\\", "|" "\r", "\n","\t"}
```

（2）面向对象封装成类

首先，将 HMM 模型封装为独立的类 `HMM_Model`，下面先给出类的结构定义：

```
    class HMM_Model:
        def __init__(self):
            pass
        #初始化
        def setup(self):
            pass
         #模型保存
        def save(self, filename, code):
            pass
        #模型加载
        def load(self, filename, code):
            pass
```

```
        #模型训练
        def do_train(self, observes, states):
            pass
        #HMM计算
        def get_prob(self):
            pass
        #模型预测
        def do_predict(self, sequence):
            pass
```

第一个方法 `__init__()` 是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法，其中定义了数据结构和初始变量，实现如下：

```
    def __init__(self):
            self.trans_mat = {}
            self.emit_mat = {}
            self.init_vec = {}
            self.state_count = {}
            self.states = {}
            self.inited = False
```

其中的数据结构定义：

- `trans_mat`：状态转移矩阵，`trans_mat[state1][state2]` 表示训练集中由 state1 转移到 state2 的次数。
- `emit_mat`：观测矩阵，`emit_mat[state][char]` 表示训练集中单字 char 被标注为 state 的次数。
- `init_vec`：初始状态分布向量，`init_vec[state]` 表示状态 state 在训练集中出现的次数。
- `state_count`：状态统计向量，`state_count[state]`表示状态 state 出现的次数。
- `word_set`：词集合，包含所有单词。

第二个方法 setup()，初始化第一个方法中的数据结构，具体实现如下：

```
        #初始化数据结构
        def setup(self):
            for state in self.states:
```

```
                    # build trans_mat
                    self.trans_mat[state] = {}
                    for target in self.states:
                        self.trans_mat[state][target] = 0.0
                    self.emit_mat[state] = {}
                    self.init_vec[state] = 0
                    self.state_count[state] = 0
                self.inited = True
```

第三个方法 save()，用来保存训练好的模型，filename 指定模型名称，默认模型名称为 hmm.json，这里提供两种格式的保存类型，JSON 或者 pickle 格式，通过参数 code 来决定，code 的值为 `code='json'` 或者 `code = 'pickle'`，默认为 `code='json'`，具体实现如下：

```
    #模型保存
    def save(self, filename="hmm.json", code='json'):
        fw = open(filename, 'w', encoding='utf-8')
        data = {
            "trans_mat": self.trans_mat,
            "emit_mat": self.emit_mat,
            "init_vec": self.init_vec,
            "state_count": self.state_count
        }
        if code == "json":
            txt = json.dumps(data)
            txt = txt.encode('utf-8').decode('unicode-escape')
            fw.write(txt)
        elif code == "pickle":
            pickle.dump(data, fw)
        fw.close()
```

第四个方法 load()，与第三个 save() 方法对应，用来加载模型，filename 指定模型名称，默认模型名称为 hmm.json，这里提供两种格式的保存类型，JSON 或者 pickle 格式，通过参数 code 来决定，code 的值为 `code='json'` 或者 `code = 'pickle'`，默认为 `code='json'`，具体实现如下：

```
    #模型加载
    def load(self, filename="hmm.json", code="json"):
        fr = open(filename, 'r', encoding='utf-8')
        if code == "json":
```

```
            txt = fr.read()
            model = json.loads(txt)
        elif code == "pickle":
            model = pickle.load(fr)
        self.trans_mat = model["trans_mat"]
        self.emit_mat = model["emit_mat"]
        self.init_vec = model["init_vec"]
        self.state_count = model["state_count"]
        self.inited = True
        fr.close()
```

第五个方法 `do_train()`，用来训练模型，因为使用的标注数据集， 因此可以使用更简单的监督学习算法，训练函数输入观测序列和状态序列进行训练， 依次更新各矩阵数据。类中维护的模型参数均为频数而非频率， 这样的设计使得模型可以进行在线训练，使得模型随时都可以接受新的训练数据继续训练，不会丢失前次训练的结果。具体实现如下：

```
    #模型训练
    def do_train(self, observes, states):
        if not self.inited:
            self.setup()

        for i in range(len(states)):
            if i == 0:
                self.init_vec[states[0]] += 1
                self.state_count[states[0]] += 1
            else:
                self.trans_mat[states[i - 1]][states[i]] += 1
                self.state_count[states[i]] += 1
                if observes[i] not in self.emit_mat[states[i]]:
                    self.emit_mat[states[i]][observes[i]] = 1
                else:
                    self.emit_mat[states[i]][observes[i]] += 1
```

第六个方法 `get_prob()`，在进行预测前，需将数据结构的频数转换为频率，具体实现如下：

```
    #频数转频率
    def get_prob(self):
        init_vec = {}
        trans_mat = {}
        emit_mat = {}
        default = max(self.state_count.values())
```

```
7.
8.          for key in self.init_vec:
9.              if self.state_count[key] != 0:
10.                 init_vec[key] = float(self.init_vec[key]) /
    self.state_count[key]
11.             else:
12.                 init_vec[key] = float(self.init_vec[key]) / default
13.
14.         for key1 in self.trans_mat:
15.             trans_mat[key1] = {}
16.             for key2 in self.trans_mat[key1]:
17.                 if self.state_count[key1] != 0:
18.                     trans_mat[key1][key2] = float(self.trans_mat[key1][
    key2]) / self.state_count[key1]
19.                 else:
20.                     trans_mat[key1][key2] = float(self.trans_mat[key1][
    key2]) / default
21.
22.         for key1 in self.emit_mat:
23.             emit_mat[key1] = {}
24.             for key2 in self.emit_mat[key1]:
25.                 if self.state_count[key1] != 0:
26.                     emit_mat[key1][key2] = float(self.emit_mat[key1][ke
    y2]) / self.state_count[key1]
27.                 else:
28.                     emit_mat[key1][key2] = float(self.emit_mat[key1][ke
    y2]) / default
29.         return init_vec, trans_mat, emit_mat
```

第七个方法 `do_predict()` ，预测采用 Viterbi 算法求得最优路径， 具体实现如下：

```
1.      #模型预测
2.      def do_predict(self, sequence):
3.          tab = [{}]
4.          path = {}
5.          init_vec, trans_mat, emit_mat = self.get_prob()
6.
7.          # 初始化
8.          for state in self.states:
9.              tab[0][state] = init_vec[state] * emit_mat[state].get(seque
    nce[0], EPS)
10.             path[state] = [state]
11.
12.         # 创建动态搜索表
```

```
            for t in range(1, len(sequence)):
                tab.append({})
                new_path = {}
                for state1 in self.states:
                    items = []
                    for state2 in self.states:
                        if tab[t - 1][state2] == 0:
                            continue
                        prob = tab[t - 1][state2] * trans_mat[state2].get(st
ate1, EPS) * emit_mat[state1].get(sequence[t], EPS)
                        items.append((prob, state2))
                    best = max(items)
                    tab[t][state1] = best[0]
                    new_path[state1] = path[best[1]] + [state1]
                path = new_path

            # 搜索最有路径
            prob, state = max([(tab[len(sequence) - 1][state], state) for s
tate in self.states])
            return path[state]
```

上面实现了类 `HMM_Model` 的7个方法，接下来我们来实现分词器，这里先定义两个函数，这两个函数是独立的，不在类中。

（1）定义一个工具函数

对输入的训练语料中的每个词进行标注，因为训练数据是空格隔开的，可以进行转态标注，该方法用在训练数据的标注，具体实现如下：

```
    def get_tags(src):
        tags = []
        if len(src) == 1:
            tags = ['S']
        elif len(src) == 2:
            tags = ['B', 'E']
        else:
            m_num = len(src) - 2
            tags.append('B')
            tags.extend(['M'] * m_num)
            tags.append('S')
        return tags
```

（2）定义一个工具函数

根据预测得到的标注序列将输入的句子分割为词语列表，也就是预测得到的状态序列，解析成一个 list 列表进行返回，具体实现如下：

```
    def cut_sent(src, tags):
        word_list = []
        start = -1
        started = False

        if len(tags) != len(src):
            return None

        if tags[-1] not in {'S', 'E'}:
            if tags[-2] in {'S', 'E'}:
                tags[-1] = 'S'
            else:
                tags[-1] = 'E'

        for i in range(len(tags)):
            if tags[i] == 'S':
                if started:
                    started = False
                    word_list.append(src[start:i])
                word_list.append(src[i])
            elif tags[i] == 'B':
                if started:
                    word_list.append(src[start:i])
                start = i
                started = True
            elif tags[i] == 'E':
                started = False
                word = src[start:i+1]
                word_list.append(word)
            elif tags[i] == 'M':
                continue
        return word_list
```

最后，我们来定义分词器类 HMMSoyoger，继承 `HMM_Model` 类并实现中文分词器训练、分词功能，先给出 HMMSoyoger 类的结构定义：

```
    class HMMSoyoger(HMM_Model):
```

```
        def __init__(self, *args, **kwargs):
            pass
        #加载训练数据
        def read_txt(self, filename):
            pass
        #模型训练函数
        def train(self):
            pass
        #模型分词预测
        def lcut(self, sentence):
            pass
```

第一个方法 init()，构造函数，定义了初始化变量，具体实现如下：

```
    def __init__(self, *args, **kwargs):
            super(HMMSoyoger, self).__init__(*args, **kwargs)
            self.states = STATES
            self.data = None
```

第二个方法 `read_txt()`，加载训练语料，读入文件为 txt，并且 UTF-8 编码，防止中文出现乱码，具体实现如下：

```
    #加载语料
    def read_txt(self, filename):
            self.data = open(filename, 'r', encoding="utf-8")
```

第三个方法 train()，根据单词生成观测序列和状态序列，并通过父类的 `do_train()` 方法进行训练，具体实现如下：

```
    def train(self):
            if not self.inited:
                self.setup()

            for line in self.data:
                line = line.strip()
                if not line:
                    continue

               #观测序列
                observes = []
                for i in range(len(line)):
```

```
                    if line[i] == " ":
                        continue
                    observes.append(line[i])

                #状态序列
                words = line.split(" ")

                states = []
                for word in words:
                    if word in seg_stop_words:
                        continue
                    states.extend(get_tags(word))
                #开始训练
                if(len(observes) >= len(states)):
                    self.do_train(observes, states)
                else:
                    pass
```

第四个方法 lcut()，模型训练好之后，通过该方法进行分词测试，具体实现如下：

```
    def lcut(self, sentence):
            try:
                tags = self.do_predict(sentence)
                return cut_sent(sentence, tags)
            except:
                return sentence
```

通过上面两个类和两个方法，就完成了基于 HMM 的中文分词器编码，下面我们来进行模型训练和测试。

训练模型

首先实例化 HMMSoyoger 类，然后通过 `read_txt()` 方法加载语料，再通过 train() 进行在线训练，如果训练语料比较大，可能需要等待一点时间，具体实现如下：

```
    soyoger = HMMSoyoger()
    soyoger.read_txt("syj_trainCorpus_utf8.txt")
    soyoger.train()
```

模型测试

模型训练完成之后，我们就可以进行测试：

```
    soyoger.lcut("中国的人工智能发展进入高潮阶段。")
```

得到结果为：

```
['中国', '的', '人工', '智能', '发展', '进入', '高潮', '阶段', '。']
```

```
soyoger.lcut("中文自然语言处理是人工智能技术的一个重要分支。")
```

得到结果为：

```
['中文', '自然', '语言', '处理', '是人', '工智', '能技', '术的', '一个', '重要', '分支', '。']
```

可见，最后的结果还是不错的，如果想取得更好的结果，可自行制备更大更丰富的训练数据集。

基于 CRF 的开源中文分词工具 Genius 实践

Genius 是一个基于 CRF 的开源中文分词工具，采用了 Wapiti 做训练与序列标注，支持 Python 2.x、Python 3.x。

安装

（1）下载源码

在 Github 上下载源码地址，解压源码，然后通过 `python setup.py install` 安装。

（2）Pypi 安装

通过执行命令：`easy_install genius` 或者 `pip install genius` 安装。

分词

首先引入 Genius，然后对 text 文本进行分词。

```
import genius
text = u"""中文自然语言处理是人工智能技术的一个重要分支。"""
```

```
seg_list = genius.seg_text(
    text,
    use_combine=True,
    use_pinyin_segment=True,
    use_tagging=True,
    use_break=True
)
print(' '.join([word.text for word in seg_list])
```

其中，`genius.seg_text` 函数接受5个参数，其中 text 是必填参数：

- text 第一个参数为需要分词的字。
- `use_break` 代表对分词结构进行打断处理，默认值 True。
- `use_combine` 代表是否使用字典进行词合并，默认值 False。
- `use_tagging` 代表是否进行词性标注，默认值 True。
- `use_pinyin_segment` 代表是否对拼音进行分词处理，默认值 True。

总结

本文首先通过贝叶斯定理，理解了判别式模型和生成式模型的区别，接着通过动手实战——基于 HMM 训练出自己的 Python 中文分词器，并进行了模型验证，最后给出一个基于 CRF 的开源中文分词工具。

参考文献

1. Genius
2. 周志华《机器学习》

第09课：一网打尽神经序列模型之 RNN 及其变种 LSTM、GRU

首先，我们来思考下，当人工神经网络从浅层发展到深层；从全连接到卷积神经网络。在此过程中，人类在图片分类、语音识别等方面都取得了非常好的结果，那么我们为什么还需要循环神经网络呢？

因为，上面提到的这些网络结构的层与层之间是全连接或部分连接的，但在每层之间的节点是无连接的，这样的网络结构并不能很好的处理序列数据。

序列数据的处理，我们从语言模型 N-gram 模型说起，然后着重谈谈 RNN，并通过 RNN 的变种 LSTM 和 GRU 来实战文本分类。

语言模型 N-gram 模型

通过前面的课程，我们了解到一般自然语言处理的传统方法是将句子处理为一个词袋模型（Bag-of-Words，BoW），而不考虑每个词的顺序，比如用朴素贝叶斯算法进行垃圾邮件识

别或者文本分类。在中文里有时候这种方式没有问题，因为有些句子即使把词的顺序打乱，还是可以看懂这句话在说什么，比如：

> T：研究表明，汉字的顺序并不一定能影响阅读，比如当你看完这句话后。
>
> F：研表究明，汉字的序顺并不定一能影阅响读，比如当你看完这句话后。

但有时候不行，词的顺序打乱，句子意思就变得让人不可思议了，例如：

> T：我喜欢吃烧烤。
>
> F：烧烤喜欢吃我。

那么，有没有模型是考虑句子中词与词之间的顺序的呢？有，语言模型中的 N-gram 就是一种。

N-gram 模型是一种语言模型（Language Model，LM），是一个基于概率的判别模型，它的输入是一句话（词的顺序序列），输出是这句话的概率，即这些词的联合概率（Joint Probability）。
使用 N-gram 语言模型思想，一般是需要知道当前词以及前面的词，因为一个句子中每个词的出现并不是独立的。比如，如果第一个词是“空气”，接下来的词是“很”，那么下一个词很大概率会是“新鲜”。类似于我们人的联想，N-gram 模型知道的信息越多，得到的结果也越准确。

在前面课程中讲解的文本分类中，我们曾用到基于 sklearn 的词袋模型，尝试加入抽取 `2-gram` 和 `3-gram` 的统计特征，把词库的量放大，获得更强的特征。

通过 ngram_range 参数来控制，代码如下：

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer(
    analyzer='word', # tokenise by character ngrams
    ngram_range=(1,4),  # use ngrams of size 1 and 2
    max_features=20000,  # keep the most common 1000 ngrams
)
```

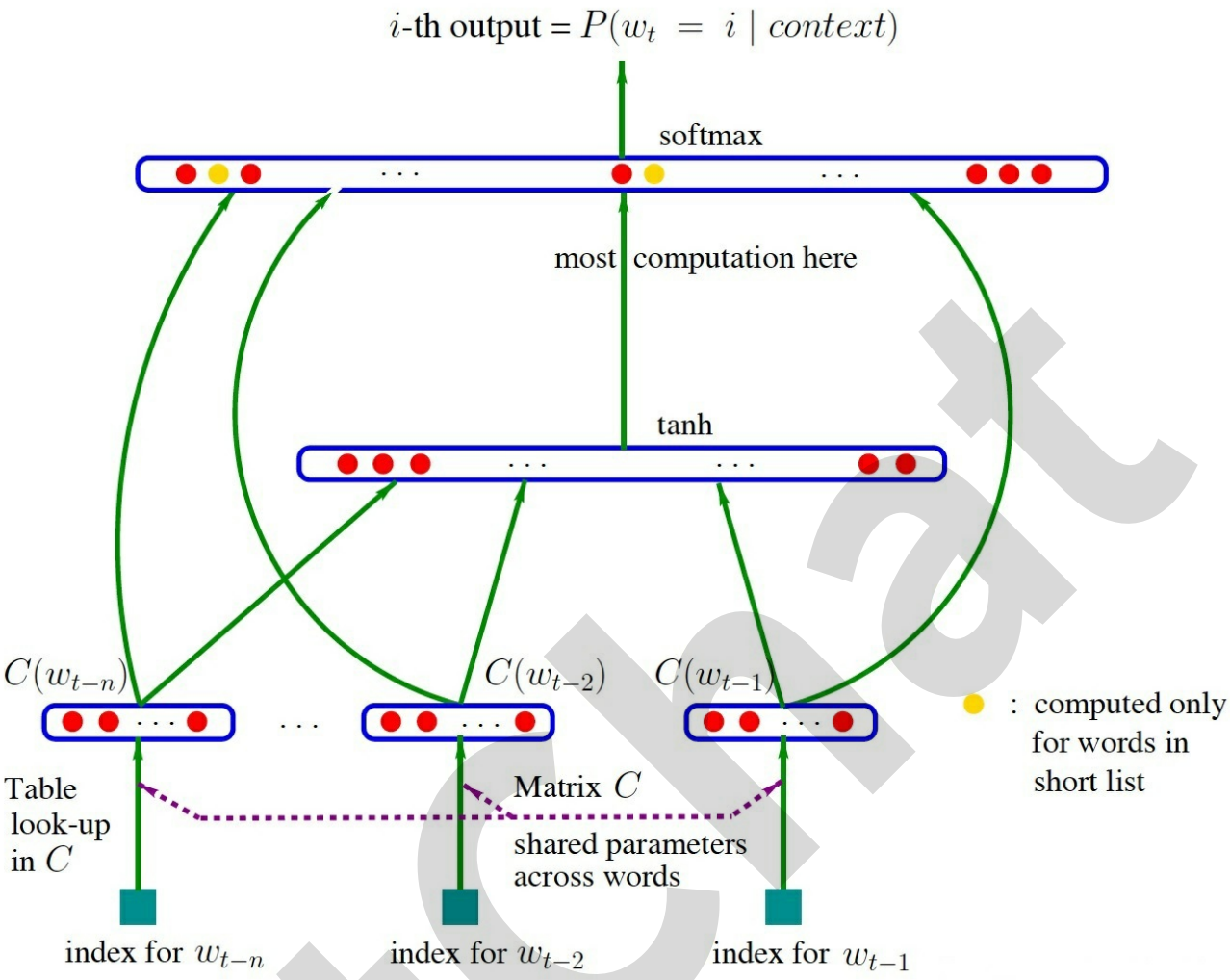
因此，N-gram 模型，在自然语言处理中主要应用在如词性标注、垃圾短信分类、分词器、机器翻译和语音识别、语音识别等领域。

然而 N-gram 模型并不是完美的，它存在如下优缺点：

- 优点：包含了前 N-1 个词所能提供的全部信息，这些词对于当前词的出现概率具有很强的约束力；
- 缺点：需要很大规模的训练文本来确定模型的参数，当 N 很大时，模型的参数空间过大。所以常见的 N 值一般为1，2，3等。还有因数据稀疏而导致的数据平滑问题，解决方法主要是拉普拉斯平滑和内插与回溯。

所以，根据 N-gram 的优缺点，它的进化版 NNLM（Neural Network based Language Model）诞生了。

NNLM 由 Bengio 在2003年提出，它是一个很简单的模型，由四层组成，输入层、嵌入层、隐层和输出层，模型结构如下图（来自百度图片）：



NNLM 接收的输入是长度为 N 的词序列，输出是下一个词的类别。首先，输入是词序列的 index 序列，例如词“我”在字典（大小为|V|）中的 index 是10，词“是”的 index 是23，“小明”的 index 是65，则句子“我是小明”的 index 序列就是 10、 23、65。嵌入层（Embedding）是一个大小为 `|V|×K` 的矩阵，从中取出第10、23、65行向量拼成 3×K 的矩阵就是 Embedding 层的输出了。隐层接受拼接后的 Embedding 层输出作为输入，以 tanh 为激活函数，最后送入带 softmax 的输出层，输出概率。

NNLM 最大的缺点就是参数多，训练慢，要求输入定长 N 这一点很不灵活，同时不能利用完整的历史信息。

因此，针对 NNLM 存在的问题，Mikolov 在2010年提出了 RNNLM，有兴趣可以阅读相关论文，其结构实际上是用 RNN 代替 NNLM 里的隐层，这样做的好处，包括减少模型参数、提高训练速度、接受任意长度输入、利用完整的历史信息。同时，RNN 的引入意味着可以使用 RNN 的其他变体，像 LSTM、BLSTM、GRU 等等，从而在序列建模上进行更多更丰

富的优化。

以上，从词袋模型说起，引出语言模型 N-gram 以及其优化模型 NNLM 和 RNNLM，后续内容从 RNN 说起，来看看其变种 LSTM 和 GRU 模型如何处理类似序列数据。

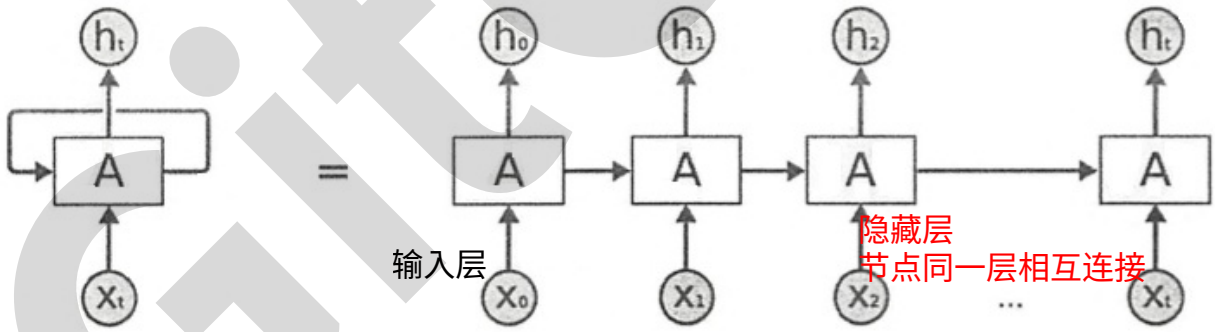
RNN 以及变种 LSTM 和 GRU 原理

RNN 为序列数据而生

RNN 称为循环神经网路，因为这种网络有“记忆性”，主要应用在自然语言处理（NLP）和语音领域。RNN 具体的表现形式为网络会对前面的信息进行记忆并应用于当前输出的计算中，即隐藏层之间的节点不再无连接而是有连接的，并且隐藏层的输入不仅包括输入层的输出还包括上一时刻隐藏层的输出。

理论上，RNN 能够对任何长度的序列数据进行处理，但由于该网络结构存在“梯度消失”问题，所以在实际应用中，解决梯度消失的方法有：梯度裁剪（Clipping Gradient）和 LSTM（`Long Short-Term Memory`）。

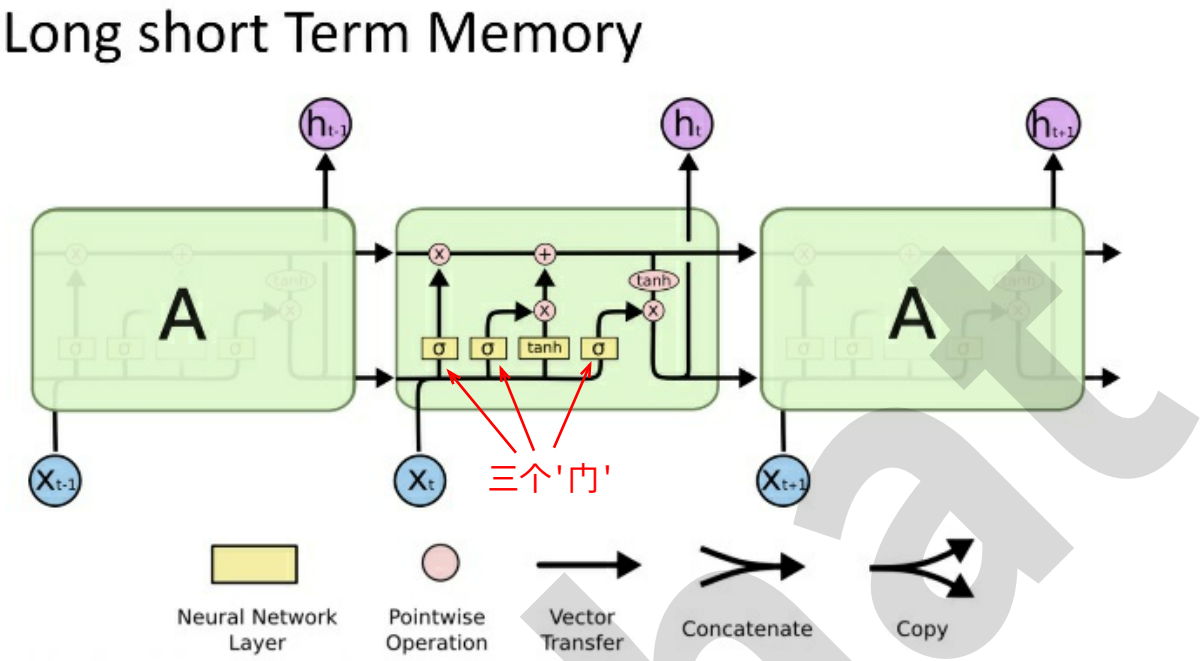
下图是一个简单的 RNN 经典结构：



RNN 包含输入单元（Input Units），输入集标记为 $\{x_0, x_1, \ldots, x_t, x_t \ldots\}$；输出单元（Output Units）的输出集则被标记为 $\{y_0, y_1, \ldots, y_t, \ldots\}$；RNN 还包含隐藏单元（Hidden Units），我们将其输出集标记为 $\{h_0, h_1, \ldots, h_t, \ldots\}$，这些隐藏单元完成了最为主要的工作。

LSTM 结构

LSTM 在1997年由“Hochreiter & Schmidhuber”提出，目前已经成为 RNN 中的标准形式，用来解决上面提到的 RNN 模型存在“长期依赖”的问题。

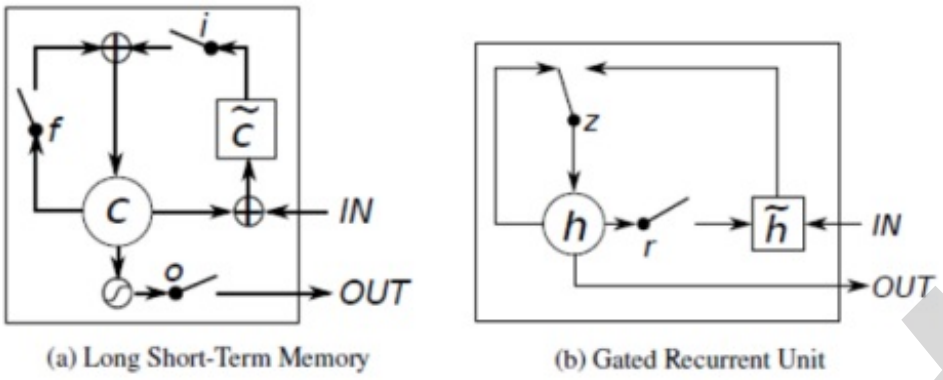


LSTM 通过三个“门”结构来控制不同时刻的状态和输出。所谓的“门”结构就是使用了 Sigmoid 激活函数的全连接神经网络和一个按位做乘法的操作，Sigmoid 激活函数会输出一个0~1之间的数值，这个数值代表当前有多少信息能通过“门”，0表示任何信息都无法通过，1表示全部信息都可以通过。其中，“遗忘门”和“输入门”是 LSTM 单元结构的核心。下面我们来详细分析下三种“门”结构。

- 遗忘门，用来让 LSTM “忘记”之前没有用的信息。它会根据当前时刻节点的输入 X_t、上一时刻节点的状态 C_{t-1} 和上一时刻节点的输出 h_{t-1} 来决定哪些信息将被遗忘。
- 输入门，LSTM 来决定当前输入数据中哪些信息将被留下来。在 LSTM 使用遗忘门“忘记”部分信息后需要从当前的输入留下最新的记忆。输入门会根据当前时刻节点的输入 X_t、上一时刻节点的状态 C_{t-1} 和上一时刻节点的输出 h_{t-1} 来决定哪些信息将进入当前时刻节点的状态 C_t，模型需要记忆这个最新的信息。
- 输出门，LSTM 在得到最新节点状态 C_t 后，结合上一时刻节点的输出 h_{t-1} 和当前时刻节点的输入 X_t 来决定当前时刻节点的输出。

GRU 结构

GRU（Gated Recurrent Unit）是2014年提出来的新的 RNN 架构，它是简化版的 LSTM。下面是 LSTM 和 GRU 的结构比较图（来自于网络）：



(a) Long Short-Term Memory (b) Gated Recurrent Unit

Figure 1: Illustration of (a) LSTM and (b) gated recurrent units. (a) i, f and o are the input, forget and output gates, respectively. c and $\tilde{c}$ denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and $\tilde{h}$ are the activation and the candidate activation.

在超参数均调优的前提下，据说效果和 LSTM 差不多，但是参数少了1/3，不容易过拟合。如果发现 LSTM 训练出来的模型过拟合比较严重，可以试试 GRU。

实战基于 Keras 的 LSTM 和 GRU 文本分类

上面讲了那么多，但是 RNN 的知识还有很多，比如双向 RNN 等，这些需要自己去学习，下面，我们来实战一下基于 LSTM 和 GRU 的文本分类。

本次开发使用 Keras 来快速构建和训练模型，使用的数据集还是第06课使用的司法数据。

整个过程包括：

1. 语料加载
2. 分词和去停用词
3. 数据预处理
4. 使用 LSTM 分类
5. 使用 GRU 分类

第一步，引入数据处理库，停用词和语料加载：

```
#引入包
import random
import jieba
import pandas as pd

```

```
    #加载停用词
    stopwords=pd.read_csv('stopwords.txt',index_col=False,quoting=3,sep="\t",names=['stopword'], encoding='utf-8')
    stopwords=stopwords['stopword'].values

    #加载语料
    laogong_df = pd.read_csv('beilaogongda.csv', encoding='utf-8', sep=',')
    laopo_df = pd.read_csv('beilaogongda.csv', encoding='utf-8', sep=',')
    erzi_df = pd.read_csv('beierzida.csv', encoding='utf-8', sep=',')
    nver_df = pd.read_csv('beinverda.csv', encoding='utf-8', sep=',')
    #删除语料的nan行
    laogong_df.dropna(inplace=True)
    laopo_df.dropna(inplace=True)
    erzi_df.dropna(inplace=True)
    nver_df.dropna(inplace=True)
    #转换
    laogong = laogong_df.segment.values.tolist()
    laopo = laopo_df.segment.values.tolist()
    erzi = erzi_df.segment.values.tolist()
    nver = nver_df.segment.values.tolist()
```

第二步，分词和去停用词：

```
    #定义分词和打标签函数preprocess_text
    #参数content_lines即为上面转换的list
    #参数sentences是定义的空list，用来储存打标签之后的数据
    #参数category 是类型标签
    def preprocess_text(content_lines, sentences, category):
        for line in content_lines:
            try:
                segs=jieba.lcut(line)
                segs = [v for v in segs if not str(v).isdigit()]#去数字
                segs = list(filter(lambda x:x.strip(), segs)) #去左右空格
                segs = list(filter(lambda x:len(x)>1, segs))#长度为1的字符
                segs = list(filter(lambda x:x not in stopwords, segs)) #去掉停用词
                sentences.append((" ".join(segs), category))# 打标签
            except Exception:
                print(line)
                continue

```

```
#调用函数、生成训练数据
sentences = []    在外面定义，是把所有类型放在一个
preprocess_text(laogong, sentences,0)
preprocess_text(laopo, sentences, 1)
preprocess_text(erzi, sentences, 2)
preprocess_text(nver, sentences, 3)
```

第三步，先打散数据，使数据分布均匀，然后获取特征和标签列表：

```
#打散数据，生成更可靠的训练集
random.shuffle(sentences)

#控制台输出前10条数据，观察一下
for sentence in sentences[:10]:
    print(sentence[0], sentence[1])
#所有特征和对应标签
all_texts = [ sentence[0] for sentence in sentences]
all_labels = [ sentence[1] for sentence in sentences]
```

第四步，使用 LSTM 对数据进行分类：

```
#引入需要的模块
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.layers import Dense, Input, Flatten, Dropout
from keras.layers import LSTM, Embedding,GRU
from keras.models import Sequential

#预定义变量
MAX_SEQUENCE_LENGTH = 100     #最大序列长度
EMBEDDING_DIM = 200     #embdding 维度
VALIDATION_SPLIT = 0.16     #验证集比例
TEST_SPLIT = 0.2     #测试集比例
#keras的sequence模块文本序列填充
tokenizer = Tokenizer()
tokenizer.fit_on_texts(all_texts)
sequences = tokenizer.texts_to_sequences(all_texts)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)
labels = to_categorical(np.asarray(all_labels))
print('Shape of data tensor:', data.shape)
```

```
23.     print('Shape of label tensor:', labels.shape)
24.
25.     #数据切分
26.     p1 = int(len(data)*(1-VALIDATION_SPLIT-TEST_SPLIT))
27.     p2 = int(len(data)*(1-TEST_SPLIT))
28.     x_train = data[:p1]
29.     y_train = labels[:p1]
30.     x_val = data[p1:p2]
31.     y_val = labels[p1:p2]
32.     x_test = data[p2:]
33.     y_test = labels[p2:]
34.
35.     #LSTM训练模型
36.     model = Sequential()
37.     model.add(Embedding(len(word_index) + 1, EMBEDDING_DIM, input_lengt
    h=MAX_SEQUENCE_LENGTH))
38.     model.add(LSTM(200, dropout=0.2, recurrent_dropout=0.2))
39.     model.add(Dropout(0.2))                    第一个dropout是x和hidden之间的dropout,
40.     model.add(Dense(64, activation='relu'))    第二个是hidden-hidden之间的dropout
41.     model.add(Dense(labels.shape[1], activation='softmax'))
42.     model.summary()
43.     #模型编译
44.     model.compile(loss='categorical_crossentropy',
45.                   optimizer='rmsprop',
46.                   metrics=['acc'])
47.     print(model.metrics_names)
48.     model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=
    10, batch_size=128)
49.     model.save('lstm.h5')
50.     #模型评估
51.     print(model.evaluate(x_test, y_test))
```

训练过程结果为：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_13 (Embedding)     (None, 100, 200)          78400
_________________________________________________________________
lstm_11 (LSTM)               (None, 200)               320800
_________________________________________________________________
dropout_10 (Dropout)         (None, 200)               0
_________________________________________________________________
dense_11 (Dense)             (None, 64)                12864
_________________________________________________________________
dense_12 (Dense)             (None, 4)                 260
=================================================================
Total params: 412,324
Trainable params: 412,324
Non-trainable params: 0
_________________________________________________________________
['loss', 'acc']
Train on 1102 samples, validate on 275 samples
Epoch 1/10
1102/1102 [==============================] - 15s 14ms/step - loss: 1.3522 - acc: 0.3358 - val_loss: 1.2505 - val_acc: 0.3964
Epoch 2/10
1102/1102 [==============================] - 13s 12ms/step - loss: 1.1428 - acc: 0.5327 - val_loss: 3.5307 - val_acc: 0.3527
Epoch 3/10
1102/1102 [==============================] - 13s 12ms/step - loss: 1.1563 - acc: 0.6443 - val_loss: 0.6789 - val_acc: 0.7418
Epoch 4/10
1102/1102 [==============================] - 13s 12ms/step - loss: 0.5988 - acc: 0.7160 - val_loss: 0.5245 - val_acc: 0.7564
Epoch 5/10
1102/1102 [==============================] - 14s 13ms/step - loss: 0.4740 - acc: 0.7260 - val_loss: 0.4130 - val_acc: 0.7418
Epoch 6/10
1102/1102 [==============================] - 15s 14ms/step - loss: 0.4165 - acc: 0.7287 - val_loss: 0.4021 - val_acc: 0.7491
Epoch 7/10
1102/1102 [==============================] - 14s 13ms/step - loss: 0.3789 - acc: 0.7541 - val_loss: 0.3611 - val_acc: 0.7164
Epoch 8/10
1102/1102 [==============================] - 14s 12ms/step - loss: 0.3711 - acc: 0.7414 - val_loss: 0.3551 - val_acc: 0.7636
Epoch 9/10
1102/1102 [==============================] - 13s 12ms/step - loss: 0.3608 - acc: 0.7559 - val_loss: 0.3536 - val_acc: 0.7309
Epoch 10/10
1102/1102 [==============================] - 13s 12ms/step - loss: 0.3928 - acc: 0.7441 - val_loss: 0.3682 - val_acc: 0.7600
345/345 [==============================] - 1s 4ms/step
[0.3640817756238191, 0.7449275744125362]
```

第五步，使用 GRU 进行文本分类，上面就是完整的使用 LSTM 进行 文本分类，如果使用 GRU 只需要改变模型训练部分：

```
    model = Sequential()
    model.add(Embedding(len(word_index) + 1, EMBEDDING_DIM, input_length=MAX_SEQUENCE_LENGTH))
    model.add(GRU(200, dropout=0.2, recurrent_dropout=0.2))
    model.add(Dropout(0.2))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(labels.shape[1], activation='softmax'))
    model.summary()

    model.compile(loss='categorical_crossentropy',
                  optimizer='rmsprop',
                  metrics=['acc'])
    print(model.metrics_names)
    model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=10, batch_size=128)
    model.save('lstm.h5')

    print(model.evaluate(x_test, y_test))
```

训练过程结果：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_12 (Embedding)     (None, 100, 200)          78400
_________________________________________________________________
gru_4 (GRU)                  (None, 200)               240600
_________________________________________________________________
dropout_9 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_9 (Dense)              (None, 64)                12864
_________________________________________________________________
dense_10 (Dense)             (None, 4)                 260
=================================================================
Total params: 332,124
Trainable params: 332,124
Non-trainable params: 0
_________________________________________________________________
['loss', 'acc']
Train on 1102 samples, validate on 275 samples
Epoch 1/10
1102/1102 [==============================] - 11s 10ms/step - loss: 1.3378 - acc: 0.3848 - val_loss: 1.1833 - val_acc: 0.4836
Epoch 2/10
1102/1102 [==============================] - 10s 9ms/step - loss: 0.9942 - acc: 0.5771 - val_loss: 0.8684 - val_acc: 0.6364
Epoch 3/10
1102/1102 [==============================] - 10s 9ms/step - loss: 0.6594 - acc: 0.6797 - val_loss: 0.5194 - val_acc: 0.7200
Epoch 4/10
1102/1102 [==============================] - 10s 9ms/step - loss: 0.4273 - acc: 0.7468 - val_loss: 0.3908 - val_acc: 0.7418
Epoch 5/10
1102/1102 [==============================] - 10s 9ms/step - loss: 0.3745 - acc: 0.7450 - val_loss: 0.3793 - val_acc: 0.7382
Epoch 6/10
1102/1102 [==============================] - 10s 9ms/step - loss: 0.4281 - acc: 0.7296 - val_loss: 0.3945 - val_acc: 0.6909
Epoch 7/10
1102/1102 [==============================] - 10s 9ms/step - loss: 0.3795 - acc: 0.7505 - val_loss: 0.3803 - val_acc: 0.7018
Epoch 8/10
1102/1102 [==============================] - 11s 10ms/step - loss: 0.3841 - acc: 0.7468 - val_loss: 0.3751 - val_acc: 0.7164
Epoch 9/10
1102/1102 [==============================] - 11s 10ms/step - loss: 0.3602 - acc: 0.7414 - val_loss: 0.3744 - val_acc: 0.7418
Epoch 10/10
1102/1102 [==============================] - 10s 9ms/step - loss: 0.3592 - acc: 0.7523 - val_loss: 0.3756 - val_acc: 0.7382
345/345 [==============================] - 1s 4ms/step
[0.34260822119920148, 0.77101449206255479]
```

总结

本文从词袋模型谈起，旨在引出语言模型 N-gram 以及其优化模型 NNLM 和 RNNLM，并通过 RNN 以及其变种 LSTM 和 GRU 模型，理解其如何处理类似序列数据的原理，并实战基于 LSTM 和 GRU 的中文文本分类。

参考文献：

1. Hinton 神经网络公开课编程题2——神经概率语言模型（NNLM）
2. Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients

[help me with Html]

第10课：动手实战基于 CNN 的电影推荐系统

本文从深度学习卷积神经网络入手，基于 Github 的开源项目来完成 MovieLens 数据集的电影推荐系统。

什么是推荐系统呢？

什么是推荐系统呢？首先我们来看看几个常见的推荐场景。

如果你经常通过豆瓣电影评分来找电影，你会发现下图所示的推荐：

热门推荐 7 / 7

HISFF X 豆瓣 邀您投片

短片是创作实验最好的切入口，我们邀请爱好短片的你、有才的你，来，投，片！

最受欢迎的影评 更多热门影评» 新片影评»

三个傻瓜？并不

酒街 评论 《三傻大闹宝莱坞》 ★★★★★

电影奉献了个理想人物兰乔，他不是傻瓜，他应该是励志英雄，他的存在具有划时代的意义，当然对于社会等级制度比较严谨的印度来说。 电影故事情节很流畅，"消音器"与三个傻瓜立誓10年后看谁发展最棒，十年之约之... (全文)

一名临床医生的话

治疗你的左半边 评论 《我不是药神》 ★★★★☆

我看哭了，片子里的事儿我以前的工作中大致都遇到过，但还是看哭了，因为生命的苦难与尊严的不堪一击。 对于药品贵这事儿，其实无解。如果强行要求药厂低售价，药厂高额的研发成本收不回来倒闭了，以后谁再研发新... (全文)

如果你喜欢购物，根据你的选择和购物行为，平台会给你推荐相似商品：

在互联网的很多场景下都可以看到推荐的影子。因为推荐可以帮助用户和商家满足不同的需求：

- 对用户而言：找到感兴趣的东西，帮助发现新鲜、有趣的事物。
- 对商家而言：提供个性化服务，提高信任度和粘性，增加营收。

常见的推荐系统主要包含两个方面的内容，基于用户的推荐系统（UserCF）和基于物品的推荐系统（ItemCF）。两者的区别在于，UserCF 给用户推荐那些和他有共同兴趣爱好的用户喜欢的商品，而 ItemCF 给用户推荐那些和他之前喜欢的商品类似的商品。这两种方式都会遭遇冷启动问题。

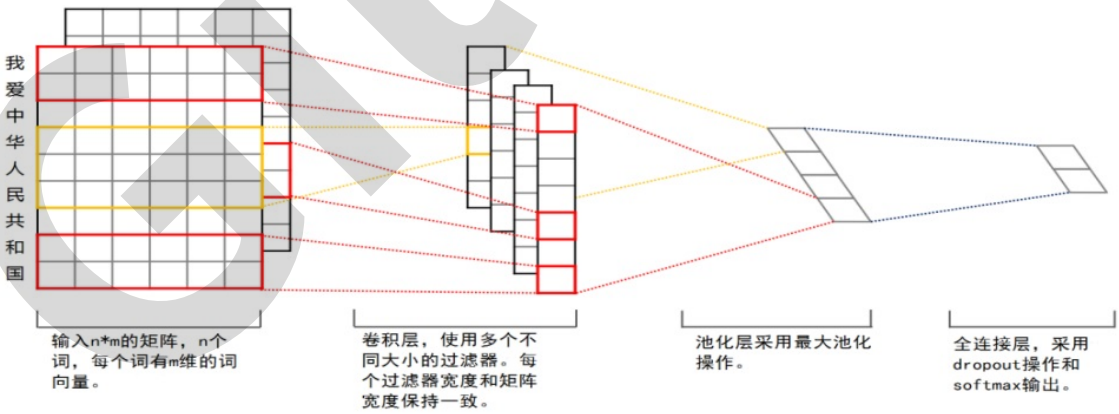
下面是 UserCF 和 ItemCF 的对比：

	UserCF	ItemCF
性能	适用于用户较少的场合，如果用户很多，计算用户相似度矩阵代价很大	适用于物品数明显小于用户数的场合，如果物品很多（网页），计算物品相似度矩阵代价很大
领域	时效性较强，用户个性化兴趣不太明显的领域	长尾物品丰富，用户个性化需求强烈的领域
实时性	用户有新行为，不一定造成推荐结果的立即变化	用户有新行为，一定会导致推荐结果的实时变化
冷启动	在新用户对很少的物品产生行为后，不能立即对他进行个性化推荐，因为用户相似度表是每隔一段时间离线计算的	新用户只要对一个物品产生行为，就可以给他推荐和该物品相关的其他物品
	新物品上线后一段时间，一旦有用户对物品产生行为，就可以将新物品推荐给和对它产生行为的用户兴趣相似的其他用户	但没有办法在不离线更新物品相似度表的情况下将新物品推荐给用户
推荐理由	很难提供令用户信服的推荐解释	利用用户的历史行为给用户做推荐解释，可以令用户比较信服

CNN 是如何应用在文本处理上的？

提到卷积神经网络（CNN），相信大部分人首先想到的是图像分类，比如 MNIST 手写体识别，CAFRI10 图像分类。CNN 已经在图像识别方面取得了较大的成果，随着近几年的不断发展，在文本处理领域，基于文本挖掘的文本卷积神经网络被证明是有效的。

首先，来看看 CNN 是如何应用到 NLP 中的，下面是一个简单的过程图：



和图像像素处理不一样，自然语言通常是一段文字，那么在特征矩阵中，矩阵的每一个行向量（比如 word2vec 或者 doc2vec）代表一个 Token，包括词或者字符。如果一段文字包含有 n 个词，每个词有 m 维的词向量，那么我们可以构造出一个 `n*m` 的词向量矩阵，在 NLP 处理过程中，让过滤器宽度和矩阵宽度保持一致整行滑动。

动手实战基于 CNN 的电影推荐系统

将 CNN 的技术应用到自然语言处理中并与电影推荐相结合，来训练一个基于文本的卷积神经网络，实现电影个性化推荐系统。

首先感谢作者 chengstone 的分享，源码请访问下面网址：

- Github

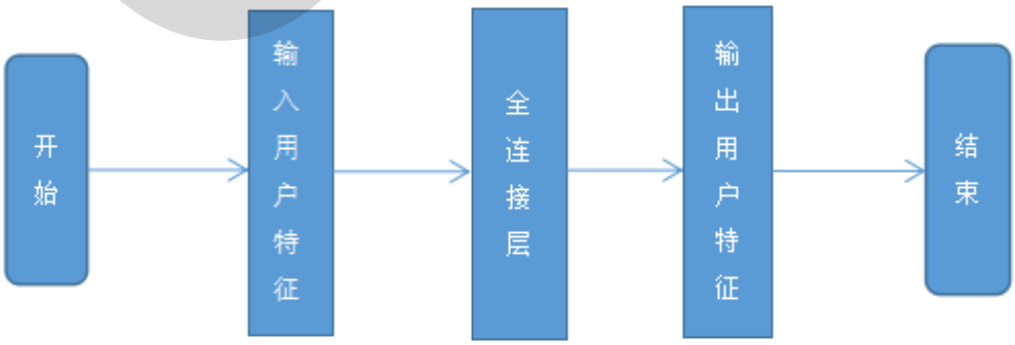
在验证了 CNN 应用在自然语言处理上是有效的之后，从推荐系统的个性化推荐入手，在文本上，把 CNN 成果应用到电影的个性化推荐上。并在特征工程中，对训练集和测试集做了相应的特征处理，其中有部分字段是类型性变量，特征工程上可以采用 `one-hot` 编码，但是对于 UserID、MovieID 这样非常稀疏的变量，如果使用 `one-hot`，那么数据的维度会急剧膨胀，对于这份数据集来说是不合适的。

具体算法设计如下：

1. 定义用户嵌入矩阵。

用户的特征矩阵主要是通过用户信息嵌入网络来生成的，在预处理数据的时候，我们将 UserID、MovieID、性别、年龄、职业特征全部转成了数字类型，然后把这个数字当作嵌入矩阵的索引，在网络的第一层就使用嵌入层，这样数据输入的维度保持在（N，32）和（N，16）。然后进行全连接层，转成（N，128）的大小，再进行全连接层，转成（N，200）的大小，这样最后输出的用户特征维度相对比较高，也保证了能把每个用户所带有的特征充分携带并通过特征表达。

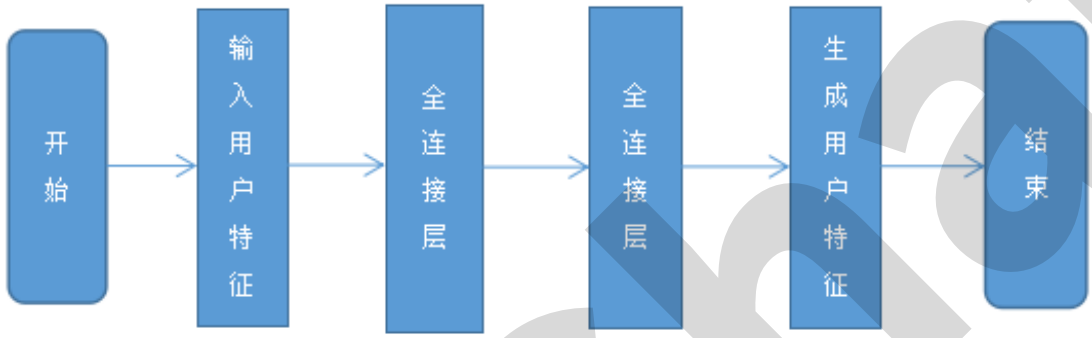
具体流程如下：



2. 生成用户特征。

生成用户特征是在用户嵌入矩阵网络输出结果的基础上，通过2层全连接层实现的。第一个全连接层把特征矩阵转成（N，128）的大小，再进行第二次全连接层，转成（N，200）的大小，这样最后输出的用户特征维度相对比较高，也保证了能把每个用户所带有的特征充分携带并通过特征表达。

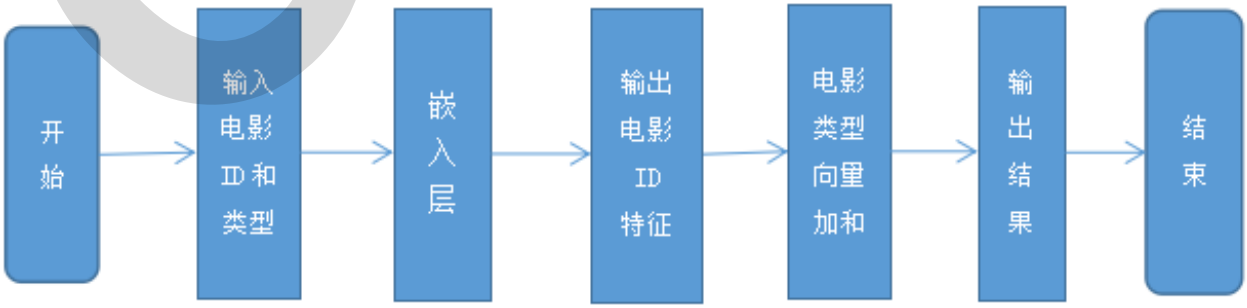
具体流程如下：



3. 定义电影 ID 嵌入矩阵。

通过电影 ID 和电影类型分别生成电影 ID 和电影类型特征，电影类型的多个嵌入向量做加和输出。电影 ID 的实现过程和上面一样，但是对于电影类型的处理相较于上面，稍微复杂一点。因为电影类型有重叠性，一个电影可以属于多个类别，当把电影类型从嵌入矩阵索引出来之后是一个（N，32）形状的矩阵，因为有多个类别，这里采用的处理方式是矩阵求和，把类别加上去，变成（1，32）形状，这样使得电影的类别信息不会丢失。

具体流程如下：

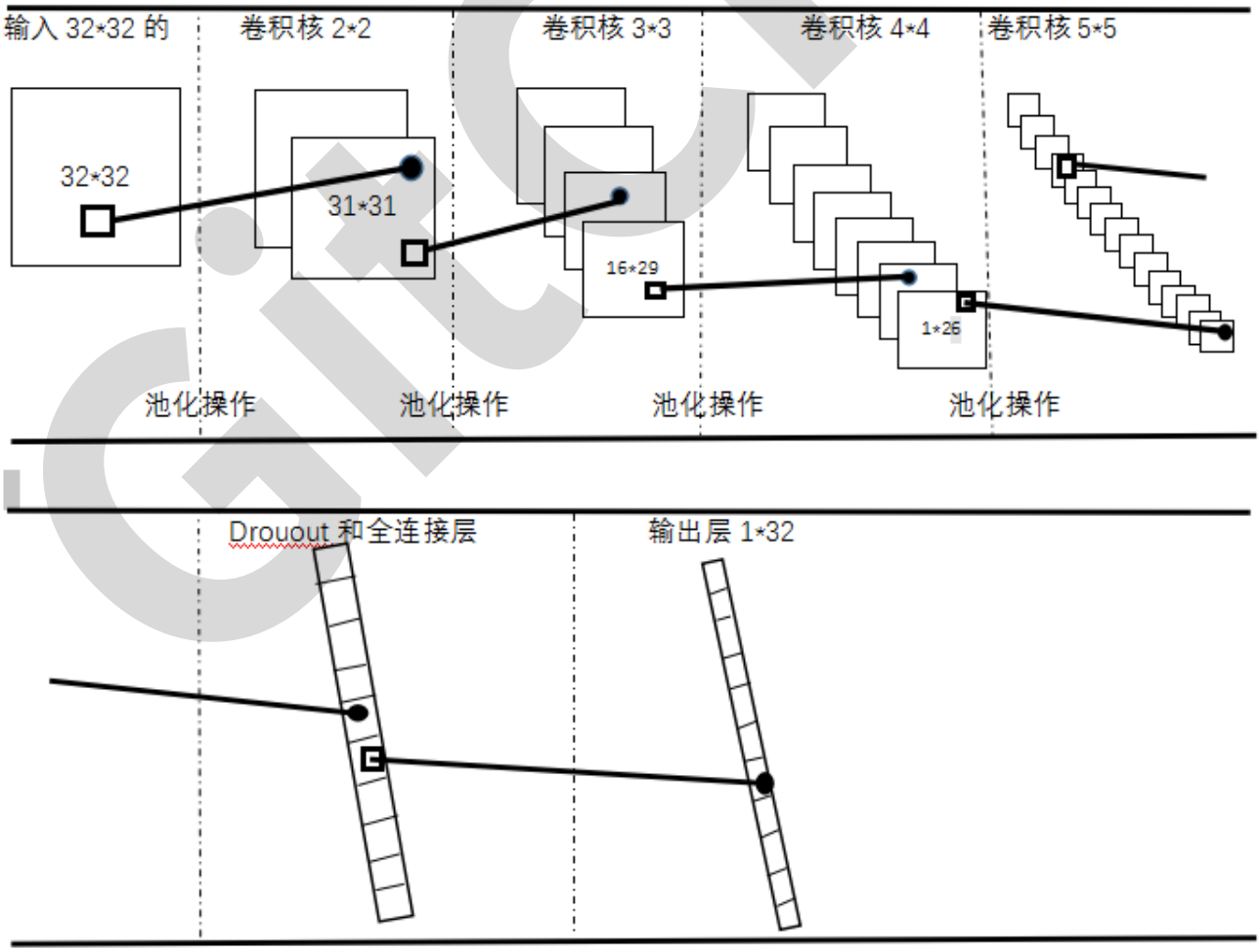


4. 文本卷积神经网络设计。

文本卷积神经网络和单纯的 CNN 网络结构有点不同，因为自然语言通常是一段文字与图片像素组成的矩阵是不一样的。在电影文本特征矩阵中，矩阵的每一个行构成的行向量代表一个 Token，包括词或者字符。如果一段文字有 n 个词，每个词有 m 维的词向量，那么我们可以构造出一个 `n*m` 的矩阵。而且 NLP 处理过程中，会有多个不同大小的过滤器串行执行，且过滤器宽度和矩阵宽度保持一致，是整行滑动。在执行完卷积操作之后采用了 ReLU 激活函数，然后采用最大池化操作，最后通过全连接并 Dropout 操作和 Softmax 输出。这里电影名称的处理比较特殊，并没有采用循环神经网络，而采用的是文本在 CNN 网络上的应用。

对于电影数据集，我们对电影名称做 CNN 处理，其大致流程，从嵌入矩阵中得到电影名对应的各个单词的嵌入向量，由于电影名称比较特殊一点，名称长度有一定限制，这里过滤器大小使用时，就选择2、3、4、5长度。然后对文本嵌入层使用滑动2、3、4、5个单词尺寸的卷积核做卷积和最大池化，然后 Dropout 操作，全连接层输出。

具体流程如下：



具体过程描述：

（1）首先输入一个 `32*32` 的矩阵；

（2）第一次卷积核大小为 `2*2`，得到 `31*31` 的矩阵，然后通过 `[1,14,1,1]` 的 `max-pooling` 操作，得到的矩阵为 `18*31`；

（3）第二次卷积核大小为 `3*3`，得到 `16*29的矩阵，然后通过[1,13,1,1]` 的 `max-pooling` 操作，得到的矩阵为 `4*29`；

（4）第三次卷积核大小 `4*4`，得到 `1*26` 的矩阵，然后通过 `[1,12,1,1]` 的 `max-pooling` 操作，得到的矩阵为 `1*26`；

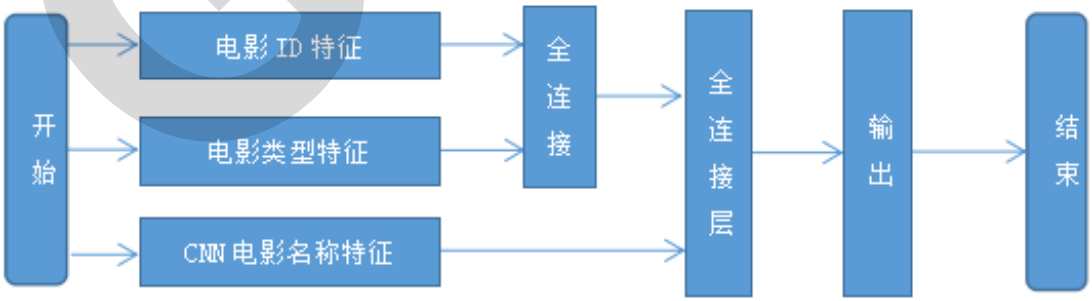
（5）第四次卷积核大小 `5*5`，得到 `1*22` 的矩阵，然后通过 `[1,11,1,1]` 的 `max-pooling` 操作，得到的矩阵为 `1*22`；

（6）最后通过 Dropout 和全连接层，`len(window_sizes) * filter_num =32`，得到 `1*32` 的矩阵。

5. 电影各层做一个全连接层。

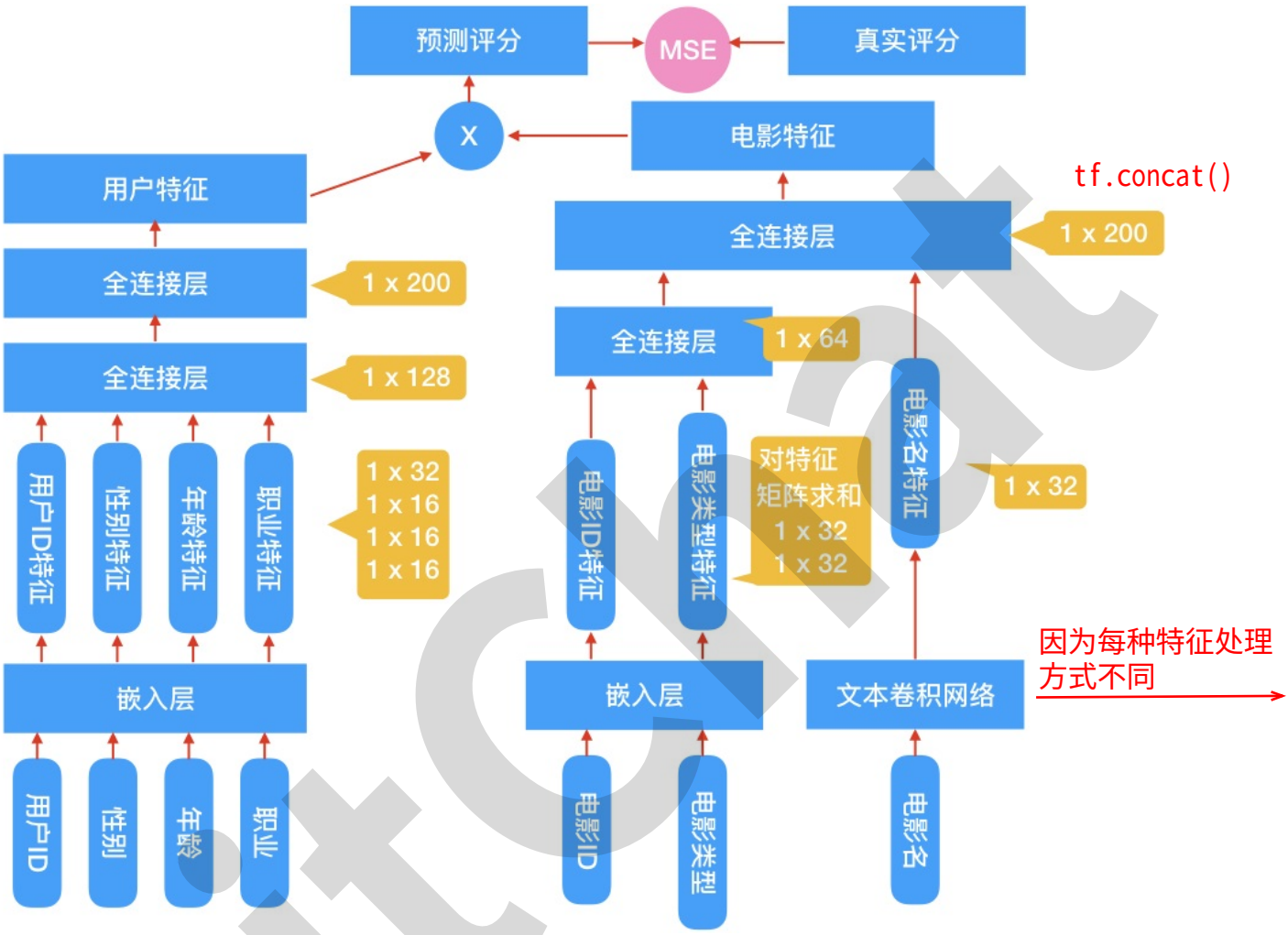
将上面几步生成的特征向量，通过2个全连接层连接在一起，第一个全连接层是电影 ID 特征和电影类型特征先全连接，之后再和 CNN 生成的电影名称特征全连接，生成最后的特征集。

具体流程如下：



6. 完整的基于 CNN 的电影推荐流程。

把以上实现的模块组合成整个算法，将网络模型作为回归问题进行训练，得到训练好的用户特征矩阵和电影特征矩阵进行推荐。



基于 CNN 的电影推荐系统代码调参过程

在训练过程中，我们需要对算法预先设置一些超参数，这里给出的最终的设置结果：

```
        # 设置迭代次数
        num_epochs = 5
        # 设置BatchSize大小
        batch_size = 256
        #设置dropout保留比例
        dropout_keep = 0.5
        # 设置学习率
        learning_rate = 0.0001
        # 设置每轮显示的batches大小
        show_every_n_batches = 20
```

首先对数据集进行划分，按照 4:1 的比例划分为训练集和测试集，下面给出的是算法模型最终训练集合测试集使用的划分结果：

```
    #将数据集分成训练集和测试集，随机种子不固定
    train_X,test_X, train_y, test_y = train_test_split(features,
                                                       targets_values,
                                                       test_size = 0.3,
                                                       random_state = 0)
```

接下来是具体模型训练过程。训练过程，要不断调参，根据经验调参粒度可以选择从粗到细分阶段进行。

调参过程对比：

（1）第一步，先固定，`learning_rate=0.01` 和 `num_epochs=10`，测试 `batch_size=128` 对迭代时间和 Loss 的影响；

（2）第二步，先固定，`learning_rate=0.01` 和 `num_epochs=10`，测试 `batch_size=256` 对迭代时间和 Loss 的影响；

（3）第三步，先固定，`learning_rate=0.01` 和 `num_epochs=10`，测试 `batch_size=512` 对迭代时间和 Loss 的影响；

（4）第四步，先固定，`learning_rate=0.01` 和 `num_epochs=5`，测试 `batch_size=128` 对迭代时间和 Loss 的影响；

（5）第五步，先固定，`learning_rate=0.01` 和 `num_epochs=5`，测试 `batch_size=256` 对迭代时间和 Loss 的影响；

（6）第六步，先固定，`learning_rate=0.01` 和 `num_epochs=5`，测试 `batch_size=512` 对迭代时间和 Loss 的影响；

（7）第七步，先固定，`batch_size=256` 和 `num_epochs=5`，测试 `learning_rate=0.001` 对 Loss 的影响；

（8）第八步，先固定，`batch_size=256` 和 `num_epochs=5`，测试 `learning_rate=0.0005` 对 Loss 的影响；

（9）第九步，先固定，`batch_size=256` 和 `num_epochs=5`，测试 `learning_rate=0.0001` 对 Loss 的影响；

（10）第十步，先固定，`batch_size=256` 和 `num_epochs=5`，测试 `learning_rate=0.00005` 对 Loss 的影响。

得到的调参结果对比表如下：

序号	learning_rate	batch_size	num_epochs	耗时（单位秒）	test_loss
1	0.01	128	10	2064.75	1.021
2	0.01	256	10	1488.35	0.971
3	0.01	512	10	1214.63	0.975
4	0.01	128	5	1072.42	0.969
5	0.01	256	5	759.86	0.985
6	0.01	512	5	688.92	0.962
7	0.001	256	5	783.31	0.861
8	0.0005	256	5	782.45	0.869
9	0.0001	256	5	788.34	0.857
10	0.00005	256	5	792.20	0.915

通过上面（1）-（6）步调参比较，在 `learning_rate`、`batch_size` 相同的情况下，`num_epochs` 对于训练时间影响较大；而在 `learning_rate`、`num_epochs` 相同情况下，`batch_size` 对 Loss 的影响较大，`batch_size` 选择512，Loss 有抖动情况，权衡之下，最终确定后续调参固定采用 `batch_size=256`、`num_epochs=5` 的超参数值，后续（7）-（10）步，随着 `learning_rate` 逐渐减小，发现 Loss 是先逐渐减小，而在 `learning_rate=0.00005` 时反而增大，最终选择出学习率为 `learning_rate=0.0001` 的超参数值。

基于 CNN 的电影推荐系统电影推荐

在上面，完成模型训练验证之后，实际来进行推荐电影，这里使用生产的用户特征矩阵和电影特征矩阵做电影推荐，主要有三种方式的推荐。

1. 推荐同类型的电影。

思路是：计算当前看的电影特征向量与整个电影特征矩阵的余弦相似度，取相似度最大的 `top_k` 个，这里加了些随机选择在里面，保证每次的推荐稍稍有些不同。

```
    def recommend_same_type_movie(movie_id_val, top_k = 20):

        loaded_graph = tf.Graph()  #
        with tf.Session(graph=loaded_graph) as sess:  #
            # Load saved model
            loader = tf.train.import_meta_graph(load_dir + '.meta')
            loader.restore(sess, load_dir)

            norm_movie_matrics = tf.sqrt(tf.reduce_sum(tf.square(movie_matrics), 1, keep_dims=True))
            normalized_movie_matrics = movie_matrics / norm_movie_matrics

            #推荐同类型的电影
            probs_embeddings = (movie_matrics[movieid2idx[movie_id_val]]).reshape([1, 200])
            probs_similarity = tf.matmul(probs_embeddings, tf.transpose(normalized_movie_matrics))
            sim = (probs_similarity.eval())
            print("您看的电影是：{}".format(movies_orig[movieid2idx[movie_id_val]]))
            print("以下是给您的推荐：")
            p = np.squeeze(sim)
            p[np.argsort(p)[:-top_k]] = 0
            p = p / np.sum(p)
            results = set()
            while len(results) != 5:
                c = np.random.choice(3883, 1, p=p)[0]
                results.add(c)
            for val in (results):
                print(val)
                print(movies_orig[val])
            return result
```

2. 推荐您喜欢的电影。

思路是：使用用户特征向量与电影特征矩阵计算所有电影的评分，取评分最高的 `top_k` 个，同样加了些随机选择部分。

```
    def recommend_your_favorite_movie(user_id_val, top_k = 10):

        loaded_graph = tf.Graph()  #
        with tf.Session(graph=loaded_graph) as sess:  #
            # Load saved model
            loader = tf.train.import_meta_graph(load_dir + '.meta')
            loader.restore(sess, load_dir)

            #推荐您喜欢的电影
            probs_embeddings = (users_matrics[user_id_val-1]).reshape([1, 200])
            probs_similarity = tf.matmul(probs_embeddings, tf.transpose(movie_matrics))
            sim = (probs_similarity.eval())

            print("以下是给您的推荐：")
            p = np.squeeze(sim)
            p[np.argsort(p)[:-top_k]] = 0
            p = p / np.sum(p)
            results = set()
            while len(results) != 5:
                c = np.random.choice(3883, 1, p=p)[0]
                results.add(c)
            for val in (results):
                print(val)
                print(movies_orig[val])

            return results
```

3. 看过这个电影的人还看了（喜欢）哪些电影。

（1）首先选出喜欢某个电影的 `top_k` 个人，得到这几个人的用户特征向量；

（2）然后计算这几个人对所有电影的评分 ；

（3）选择每个人评分最高的电影作为推荐；

（4）同样加入了随机选择。

```
    def recommend_other_favorite_movie(movie_id_val, top_k = 20):
        loaded_graph = tf.Graph()  #
        with tf.Session(graph=loaded_graph) as sess:  #
            # Load saved model
            loader = tf.train.import_meta_graph(load_dir + '.meta')
            loader.restore(sess, load_dir)
            probs_movie_embeddings = (movie_matrics[movieid2idx[movie_id_val]]).reshape([1, 200])
            probs_user_favorite_similarity = tf.matmul(probs_movie_embeddings, tf.transpose(users_matrics))
            favorite_user_id = np.argsort(probs_user_favorite_similarity.eval())[0][-top_k:]

            print("您看的电影是：{}".format(movies_orig[movieid2idx[movie_id_val]]))

            print("喜欢看这个电影的人是：{}".format(users_orig[favorite_user_id-1]))
            probs_users_embeddings = (users_matrics[favorite_user_id-1]).reshape([-1, 200])
            probs_similarity = tf.matmul(probs_users_embeddings, tf.transpose(movie_matrics))
            sim = (probs_similarity.eval())
            p = np.argmax(sim, 1)
            print("喜欢看这个电影的人还喜欢看：")
            results = set()
            while len(results) != 5:
                c = p[random.randrange(top_k)]
                results.add(c)
            for val in (results):
                print(val)
                print(movies_orig[val])
            return results
```


基于 CNN 的电影推荐系统不足

这里讨论一下基于上述方法所带来的不足：

1. 由于一个新的用户在刚开始的时候并没有任何行为记录，所以系统会出现冷启动的问题；

2. 由于神经网络是一个黑盒子过程，我们并不清楚在反向传播的过程中的具体细节，也不知道每一个卷积层抽取的特征细节，所以此算法缺乏一定的可解释性；
3. 一般来说，在工业界，用户的数据量是海量的，而卷积神经网络又要耗费大量的计算资源，所以进行集群计算是非常重要的。但是由于本课程所做实验环境有限，还是在单机上运行，所以后期可以考虑在服务器集群上全量跑数据，这样获得的结果也更准确。

总结

上面通过 Github 上一个开源的项目，梳理了 CNN 在文本推荐上的应用，并通过模型训练调参，给出一般的模型调参思路，最后建议大家自己把源码下载下来跑跑模型，效果更好。

参考文献及推荐阅读

1. 推荐系统
2. Deep Convolutional Neural Networks for Sentiment Analysis of ShortTexts,CND Santos ,M Gattit ,2014.
3. 推荐系统实践，p50-60，p120-130，项亮。

第11课：动手实战基于 LSTM 轻松生成各种古诗

目前循环神经网络（RNN）已经广泛用于自然语言处理中，可以处理大量的序列数据，可以说是最强大的神经网络模型之一。人们已经给 RNN 找到了越来越多的事情做，比如画画和写诗，微软的小冰都已经出版了一本诗集了。

而其实训练一个能写诗的神经网络并不难，下面我们就介绍如何简单快捷地建立一个会写诗的网络模型。

本次开发环境如下：

- Python 3.6
- Keras 环境
- Jupyter Notebook

整个过程分为以下步骤完成：

1. 语料准备

2. 语料预处理
3. 模型参数配置
4. 构建模型
5. 训练模型
6. 模型作诗
7. 绘制模型网络结构图

下面一步步来构建和训练一个会写诗的模型。

第一，语料准备。一共四万多首古诗，每行一首诗，标题在预处理的时候已经去掉了。

第二，文件预处理。首先，机器并不懂每个中文汉字代表的是什么，所以要将文字转换为机器能理解的形式，这里我们采用 One-Hot 的形式，这样诗句中的每个字都能用向量来表示，下面定义函数 `preprocess_file()` 来处理。

```
    puncs = [']', '[', ' (', ') ', '{', '}', ':', '《', '》 ']
    def preprocess_file(Config):
        # 语料文本内容
        files_content = ''
        with open(Config.poetry_file, 'r', encoding='utf-8') as f:
            for line in f:
                # 每行的末尾加上"]"符号代表一首诗结束
                for char in puncs:
                    line = line.replace(char, "")
                files_content += line.strip() + "]"

        words = sorted(list(files_content))
        words.remove(']')
        counted_words = {}
        for word in words:
            if word in counted_words:
                counted_words[word] += 1
            else:
                counted_words[word] = 1

        # 去掉低频的字
        erase = []
        for key in counted_words:
            if counted_words[key] <= 2:
                erase.append(key)
        for key in erase:
```

```
            del counted_words[key]
        del counted_words[']']
        wordPairs = sorted(counted_words.items(), key=lambda x: -x[1])

        words, _ = zip(*wordPairs)
        # word到id的映射
        word2num = dict((c, i + 1) for i, c in enumerate(words))
        num2word = dict((i, c) for i, c in enumerate(words))
        word2numF = lambda x: word2num.get(x, 0)
        return word2numF, num2word, words, files_content
```

在每行末尾加上 `]` 符号是为了标识这首诗已经结束了。我们给模型学习的方法是，给定前六个字，生成第七个字，所以在后面生成训练数据的时候，会以6的跨度，1的步长截取文字，生成语料。如果出现了 `]` 符号，说明 `]` 符号之前的语句和之后的语句是两首诗里面的内容，两首诗之间是没有关联关系的，所以我们后面会舍弃掉包含 `]` 符号的训练数据。

第三，模型参数配置。预先定义模型参数和加载语料以及模型保存名称，通过类 Config 实现。

```
class Config(object):
    poetry_file = 'poetry.txt'
    weight_file = 'poetry_model.h5'
    # 根据前六个字预测第七个字
    max_len = 6
    batch_size = 512
    learning_rate = 0.001
```

第四，构建模型，通过 PoetryModel 类实现，类的代码结构如下：

```
    class PoetryModel(object):
        def __init__(self, config):
            pass

        def build_model(self):
            pass

        def sample(self, preds, temperature=1.0):
            pass

        def generate_sample_result(self, epoch, logs):
            pass
```

```
13.
14.         def predict(self, text):
15.             pass
16.
17.         def data_generator(self):
18.             pass
19.         def train(self):
20.             pass
```

类中定义的方法具体实现功能如下：

（1）init 函数定义，通过加载 Config 配置信息，进行语料预处理和模型加载，如果模型文件存在则直接加载模型，否则开始训练。

```
1.      def __init__(self, config):
2.              self.model = None
3.              self.do_train = True
4.              self.loaded_model = False
5.              self.config = config
6.
7.              # 文件预处理
8.              self.word2numF, self.num2word, self.words, self.files_conte
    nt = preprocess_file(self.config)
9.              if os.path.exists(self.config.weight_file):
10.                 self.model = load_model(self.config.weight_file)
11.                 self.model.summary()
12.             else:
13.                 self.train()
14.             self.do_train = False
15.             self.loaded_model = True
```

（2）`build_model` 函数主要用 Keras 来构建网络模型，这里使用 LSTM 的 GRU 来实现，当然直接使用 LSTM 也没问题。

```
1.      def build_model(self):
2.              '''建立模型'''
3.              input_tensor = Input(shape=(self.config.max_len,))
4.              embedd = Embedding(len(self.num2word)+1, 300, input_length=
    self.config.max_len)(input_tensor)
5.              lstm = Bidirectional(GRU(128, return_sequences=True))(embed
    d)
6.              dropout = Dropout(0.6)(lstm)
```

```
            lstm = Bidirectional(GRU(128, return_sequences=True))(embed
d)
            dropout = Dropout(0.6)(lstm)
            flatten = Flatten()(lstm)
            dense = Dense(len(self.words), activation='softmax')(flatte
n)
            self.model = Model(inputs=input_tensor, outputs=dense)
            optimizer = Adam(lr=self.config.learning_rate)
            self.model.compile(loss='categorical_crossentropy', optimiz
er=optimizer, metrics=['accuracy'])
```

（3）sample 函数，在训练过程的每个 epoch 迭代中采样。

```
    def sample(self, preds, temperature=1.0):
        '''
        当temperature=1.0时，模型输出正常
        当temperature=0.5时，模型输出比较open
        当temperature=1.5时，模型输出比较保守
        在训练的过程中可以看到temperature不同，结果也不同
        '''
        preds = np.asarray(preds).astype('float64')
        preds = np.log(preds) / temperature
        exp_preds = np.exp(preds)
        preds = exp_preds / np.sum(exp_preds)
        probas = np.random.multinomial(1, preds, 1)
        return np.argmax(probas)
```

（4）训练过程中，每个 epoch 打印出当前的学习情况。

```
    def generate_sample_result(self, epoch, logs):
        print("\n==================Epoch {}====================".f
ormat(epoch))
        for diversity in [0.5, 1.0, 1.5]:
            print("------------Diversity {}--------------".format(di
versity))
            start_index = random.randint(0, len(self.files_content)
- self.config.max_len - 1)
            generated = ''
            sentence = self.files_content[start_index: start_index
+ self.config.max_len]
            generated += sentence
            for i in range(20):
                x_pred = np.zeros((1, self.config.max_len))
```

```
                    for t, char in enumerate(sentence[-6:]):
                        x_pred[0, t] = self.word2numF(char)

                    preds = self.model.predict(x_pred, verbose=0)[0]
                    next_index = self.sample(preds, diversity)
                    next_char = self.num2word[next_index]
                    generated += next_char
                    sentence = sentence + next_char
                print(sentence)
```

（5）predict 函数，用于根据给定的提示，来进行预测。

根据给出的文字，生成诗句，如果给的 text 不到四个字，则随机补全。

```
    def predict(self, text):
            if not self.loaded_model:
                return
            with open(self.config.poetry_file, 'r', encoding='utf-8') as f:
                file_list = f.readlines()
            random_line = random.choice(file_list)
            # 如果给的text不到四个字，则随机补全
            if not text or len(text) != 4:
                for _ in range(4 - len(text)):
                    random_str_index = random.randrange(0, len(self.words))
                    text += self.num2word.get(random_str_index) if self.num2word.get(random_str_index) not in [',', '。',
', '] else self.num2word.get(
                        random_str_index + 1)
            seed = random_line[-(self.config.max_len):-1]
            res = ''
            seed = 'c' + seed
            for c in text:
                seed = seed[1:] + c
                for j in range(5):
                    x_pred = np.zeros((1, self.config.max_len))
                    for t, char in enumerate(seed):
                        x_pred[0, t] = self.word2numF(char)
                    preds = self.model.predict(x_pred, verbose=0)[0]
                    next_index = self.sample(preds, 1.0)
                    next_char = self.num2word[next_index]
                    seed = seed[1:] + next_char
```

```
                res += seed
            return res
```

（6）`data_generator` 函数，用于生成数据，提供给模型训练时使用。

```
    def data_generator(self):
        i = 0
        while 1:
            x = self.files_content[i: i + self.config.max_len]
            y = self.files_content[i + self.config.max_len]
            puncs = [']', '[', ' (', ') ', '{', '}', ':', '《', '》' , ':']
            if len([i for i in puncs if i in x]) != 0:
                i += 1
                continue
            if len([i for i in puncs if i in y]) != 0:
                i += 1
                continue
            y_vec = np.zeros(
                shape=(1, len(self.words)),
                dtype=np.bool
            )
            y_vec[0, self.word2numF(y)] = 1.0
            x_vec = np.zeros(
                shape=(1, self.config.max_len),
                dtype=np.int32
            )
            for t, char in enumerate(x):
                x_vec[0, t] = self.word2numF(char)
            yield x_vec, y_vec
            i += 1
```

（7）train 函数，用来进行模型训练，其中迭代次数 `number_of_epoch` ，是根据训练语料长度除以 `batch_size` 计算的，如果在调试中，想用更小一点的 `number_of_epoch` ，可以自定义大小，把 train 函数的第一行代码注释即可。

```
    def train(self):
        #number_of_epoch = len(self.files_content) // self.config.batch_size
        number_of_epoch = 10
        if not self.model:
            self.build_model()
```

```
            self.model.summary()
            self.model.fit_generator(
                generator=self.data_generator(),
                verbose=True,
                steps_per_epoch=self.config.batch_size,
                epochs=number_of_epoch,
                callbacks=[
keras.callbacks.ModelCheckpoint(self.config.weight_file, save_weights_only=False),
LambdaCallback(on_epoch_end=self.generate_sample_result)
                ]
            )
```

第五，整个模型构建好以后，接下来进行模型训练。

```
    model = PoetryModel(Config)
```

训练过程中的第1-2轮迭代：

```
Epoch 1/10
512/512 [==============================] - 106s 207ms/step - loss: 7.4376 - acc: 0.1445

==================Epoch 0==================
----------Diversity 0.5----------
别数子，握手子风意不慢。夕玉原卷猿。填楚旋裁意。若化
----------Diversity 1.0----------
。寥寥夜含风。音矗犟惊群。下移舞朱初洞露已旎嫂鑫逊绁
----------Diversity 1.5----------
响佩环，银台克疾压群绚郜沫寚鞞珍蒙醝撞栽幉缪蜿醻浇韶
Epoch 2/10
512/512 [==============================] - 112s 219ms/step - loss: 6.9103 - acc: 0.1680

==================Epoch 1==================
----------Diversity 0.5----------
猿屡啸，林静迈咤泌不好好食再供彩浸菊险劳呈凰好好夜好
----------Diversity 1.0----------
俗与时未安闲青觑众以搓辫善馀凿肢圮寺倍狻辱食马庭愁婀
----------Diversity 1.5----------
作闲身上古城瘼杼剃螭翚下枋迈铍骳枳凋尘侏培簭豑玙柯菖
```

训练过程中的第9-10轮迭代：

```
Epoch 9/10
512/512 [==============================] - 110s 214ms/step - loss: 5.9472 - acc: 0.1777

==========Epoch 8==========
----------Diversity 0.5----------
。到头诗卷须。欲会洲月岂。汉萝炉向夜。物清岁见晓。物
----------Diversity 1.0----------
相见，相见复绛悠不裹人卢海道莫力宁相石一舰悟辐异遣下
----------Diversity 1.5----------
候暖麴蘖调，剩惆蚊襟水不示黯龙贫节朐晓见扶次侃瓦撇惆
Epoch 10/10
512/512 [==============================] - 109s 213ms/step - loss: 5.8819 - acc: 0.1875

==========Epoch 9==========
----------Diversity 0.5----------
贫。]凤凰衔处。符向三诗烟。徒辛三徒绵。应近群莫珑。
----------Diversity 1.0----------
恩及夏台。将旦声同嘉。嘉此镐沾华。妇客郑官烟。改源无
----------Diversity 1.5----------
群才。收兵铸通箸巧岱缁忿庾发滹没霪许纾楼蟠芹螭施关枕
```

虽然训练过程写出的诗句不怎么能看得懂，但是可以看到模型从一开始标点符号都不会用 ，到最后写出了有一点点模样的诗句，能看到模型变得越来越聪明了。

第六，模型作诗，模型迭代10次之后的测试，首先输入几个字，模型根据输入的提示，做出诗句。

```
1.      text = input("text:")
2.      sentence = model.predict(text)
3.      print(sentence)
```

比如输入：小雨，模型做出的诗句为：

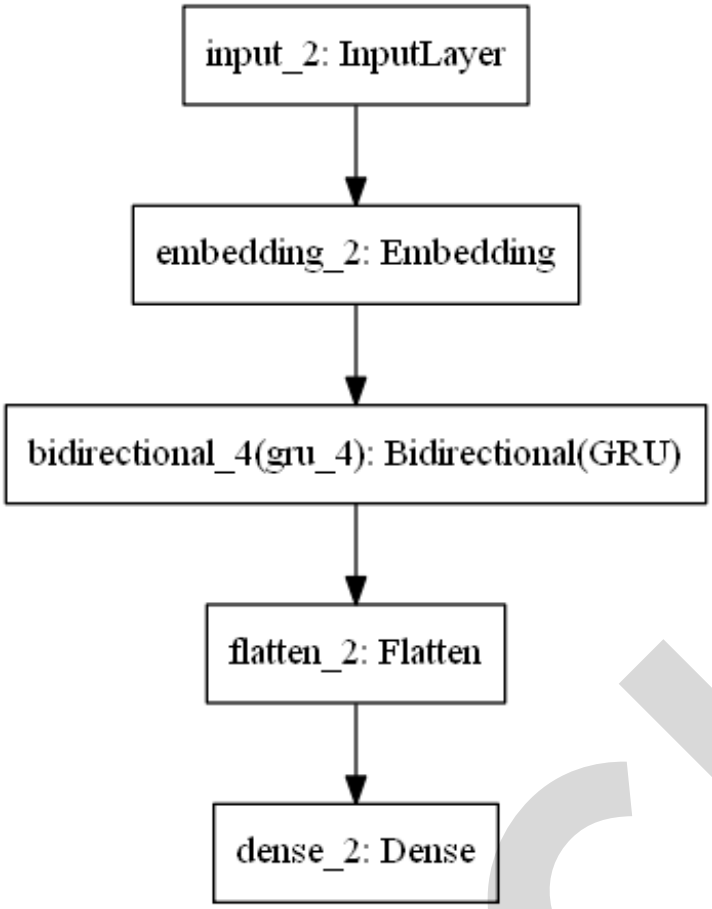
> 输入：text：小雨
>
> 结果：小妃侯里守。雨封即客寥。俘剪舟过槽。傲老槟冬绛。

第七，绘制网络结构图。

模型结构绘图，采用 Keras自带的功能实现：

```
1.      plot_model(model.model, to_file='model.png')
```

得到的模型结构图如下：



本节使用 LSTM 的变形 GRU 训练出一个能作诗的模型，当然大家可以替换训练语料为歌词或者小说，让机器人自动创作不同风格的歌曲或者小说。

参考文献以及推荐阅读：

1. 基于 Keras 和 LSTM 的文本生成

第12课：完全基于情感词典的文本情感分析

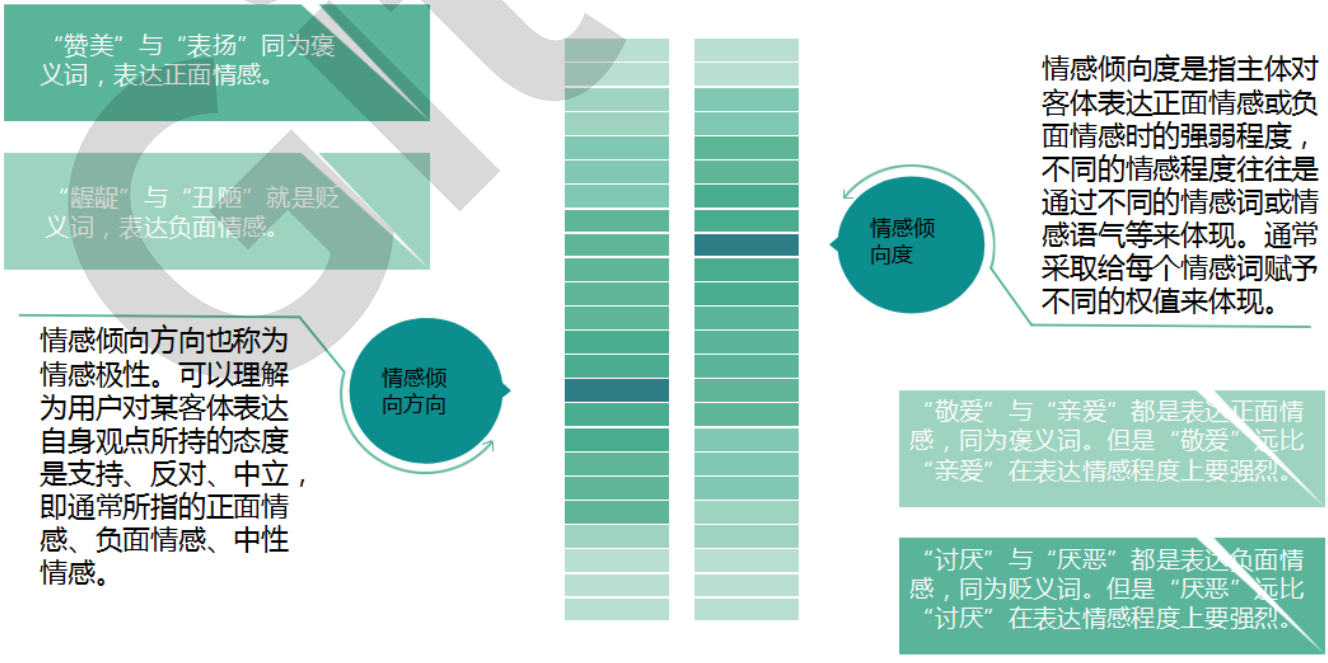
目前情感分析在中文自然语言处理中比较火热，很多场景下，我们都需要用到情感分析。比如，做金融产品量化交易，需要根据爬取的舆论数据来分析政策和舆论对股市或者基金期货的态度；电商交易，根据买家的评论数据，来分析商品的预售率等等。

下面我们通过以下几点来介绍中文自然语言处理情感分析：

1. 中文情感分析方法简介；
2. SnowNLP 快速进行评论数据情感分析；
3. 基于标注好的情感词典来计算情感值；
4. pytreebank 绘制情感树；
5. 股吧数据情感分类。

中文情感分析方法简介

情感倾向可认为是主体对某一客体主观存在的内心喜恶，内在评价的一种倾向。它由两个方面来衡量：一个情感倾向方向，一个是情感倾向度。



目前，情感倾向分析的方法主要分为两类：一种是基于情感词典的方法；一种是基于机器学习的方法，如基于大规模语料库的机器学习。前者需要用到标注好的情感词典；后者则需要大量的人工标注的语料作为训练集，通过提取文本特征，构建分类器来实现情感的分类。

文本情感分析的分析粒度可以是词语、句子、段落或篇章。

段落篇章级情感分析主要是针对某个主题或事件进行情感倾向判断，一般需要构建对应事件的情感词典，如电影评论的分析，需要构建电影行业自己的情感词典，这样效果会比通用情感词典更好；也可以通过人工标注大量电影评论来构建分类器。句子级的情感分析大多通过计算句子里包含的所有情感词的值来得到。

篇章级的情感分析，也可以通过聚合篇章中所有的句子的情感倾向来计算得出。因此，针对句子级的情感倾向分析，既能解决短文本的情感分析，同时也是篇章级文本情感分析的基础。

中文情感分析的一些难点，比如句子是由词语根据一定的语言规则构成的，应该把句子中词语的依存关系纳入到句子情感的计算过程中去，不同的依存关系，进行情感倾向计算是不一样的。文档的情感，根据句子对文档的重要程度赋予不同权重，调整其对文档情感的贡献程度等。

SnowNLP 快速进行评论数据情感分析

如果有人问，有没有比较快速简单的方法能判断一句话的情感倾向，那么 SnowNLP 库就是答案。

SnowNLP 主要可以进行中文分词、词性标注、情感分析、文本分类、转换拼音、繁体转简体、提取文本关键词、提取摘要、分割句子、文本相似等。

需要注意的是，用 SnowNLP 进行情感分析，官网指出进行电商评论的准确率较高，其实是因为它的语料库主要是电商评论数据，但是可以自己构建相关领域语料库，替换单一的电商评论语料，准确率也挺不错的。

1. SnowNLP 安装。

（1）使用 pip 安装：

```
pip install snownlp==0.11.1
```

（2）使用 Github 源码安装。

首先，下载 SnowNLP 的 Github 源码并解压，在解压目录，通过下面命令安装：

```
python  setup.py install
```

以上方式，二选一安装完成之后，就可以引入 SnowNLP 库使用了。

```
from snownlp import SnowNLP
```

2. 评论语料获取情感值。

首先，SnowNLP 对情感的测试值为0到1，值越大，说明情感倾向越积极。下面我们通过 SnowNLP 测试在京东上找的好评、中评、差评的结果。

首先，引入 SnowNLP 库：

```
from snownlp import SnowNLP
```

（1）测试一条京东的好评数据：

```
SnowNLP(u'本本已收到，体验还是很好，功能方面我不了解，只看外观还是很不错很薄，很轻，也有质感。').sentiments
```

得到的情感值很高，说明买家对商品比较认可，情感值为：

```
0.999950702449061
```

（2）测试一条京东的中评数据：

```
SnowNLP(u'屏幕分辨率一般，送了个极丑的鼠标。').sentiments
```

得到的情感值一般，说明买家对商品看法一般，甚至不喜欢，情感值为：

```
0.032514028834003 23
```

（3）测试一条京东的差评数据：

```
SnowNLP(u'很差的一次购物体验，细节做得极差了，还有发热有点严重啊，散热不行，用起来就是烫得厉害，很垃圾！！！').sentiments
```

得到的情感值一般，说明买家对商品不认可，存在退货嫌疑，情感值为：

```
0.0036849517156107847
```

以上就完成了简单快速的情感值计算，对评论数据是不是很好用呀！！！

使用 SnowNLP 来计算情感值，官方推荐的是电商评论数据计算准确度比较高，难道非评论数据就不能使用 SnowNLP 来计算情感值了吗？当然不是，虽然 SnowNLP 默认提供的模型是用评论数据训练的，但是它还支持我们根据现有数据训练自己的模型。

首先我们来看看自定义训练模型的**源码 Sentiment 类**，代码定义如下：

```
class Sentiment(object):

    def __init__(self):
        self.classifier = Bayes()

    def save(self, fname, iszip=True):
        self.classifier.save(fname, iszip)

    def load(self, fname=data_path, iszip=True):
        self.classifier.load(fname, iszip)

    def handle(self, doc):
        words = seg.seg(doc)
        words = normal.filter_stop(words)
        return words

    def train(self, neg_docs, pos_docs):
        data = []
        for sent in neg_docs:
            data.append([self.handle(sent), 'neg'])
```

```
            for sent in pos_docs:
                data.append([self.handle(sent), 'pos'])
            self.classifier.train(data)

        def classify(self, sent):
            ret, prob = self.classifier.classify(self.handle(sent))
            if ret == 'pos':
                return prob
            return 1-prob
```

通过源代码，我们可以看到，可以使用 train方法训练数据，并使用 save 方法和 load 方法保存与加载模型。下面训练自己的模型，训练集 pos.txt 和 neg.txt 分别表示积极和消极情感语句，两个 TXT 文本中每行表示一句语料。

下面代码进行自定义模型训练和保存：

```
from snownlp import sentiment
sentiment.train('neg.txt', 'pos.txt')
sentiment.save('sentiment.marshal')
```

基于标注好的情感词典来计算情感值

这里我们使用一个行业标准的情感词典——玻森情感词典，来自定义计算一句话、或者一段文字的情感值。

整个过程如下：

1. 加载玻森情感词典；
2. jieba 分词；
3. 获取句子得分。

首先引入包：

```
import pandas as pd
import jieba
```

接下来加载情感词典：

```
df = pd.read_table("bosonnlp//BosonNLP_sentiment_score.txt",sep= "
",names=['key','score'])
```

查看一下情感词典前5行：

	key	score
0	最尼玛	-6.704000
1	扰民	-6.497564
2	fuck...	-6.329634
3	RNM	-6.218613
4	wcnmlgb	-5.967100

将词 key 和对应得分 score 转成2个 list 列表，目的是找到词 key 的时候，能对应获取到 score 值：

```
key = df['key'].values.tolist()
score = df['score'].values.tolist()
```

定义分词和统计得分函数：

```
def getscore(line):
    segs = jieba.lcut(line)  #分词
    score_list  = [score[key.index(x)] for x in segs if(x in key)]
    return  sum(score_list)  #计算得分
```

最后来进行结果测试：

```
line = "今天天气很好，我很开心"
print(round(getscore(line),2))

line = "今天下雨，心情也受到影响。"
print(round(getscore(line),2))
```

获得的情感得分保留2位小数：

> 5.26

> -0.96

pytreebank 绘制情感树

1. 安装 pytreebank。

在 Github 上下载 pytreebank 源码，解压之后，进入解压目录命令行，执行命令：

```
python setup.py install
```

最后通过引入命令，判断是否安装成功：

```
import pytreebank
```

提示，如果在 Windows 下安装之后，报错误：

```
UnicodeDecodeError: 'gbk' codec can't decode byte 0x92 in position 247
83: illegal multibyte sequence
```

这是由于编码问题引起的，可以在安装目录下报错的文件中报错的代码地方加个 `encoding='utf-8'` 编码：

```
import_tag( "script",
contents=format_replacements(open(scriptname,encoding='utf-8').read(),
replacements), type="text/javascript" )
```

2. 绘制情感树。

首先引入 pytreebank 包：

```
    import pytreebank
```

然后，加载用来可视化的 JavaScript 和 CSS 脚本：

```
pytreebank.LabeledTree.inject_visualization_javascript()
```

绘制情感树，把句子首先进行组合再绘制图形：

```
line = '(4 (0 你) (3 (2 是) (3 (3 谁) (2 的)) (2 谁))))'
pytreebank.create_tree_from_string(line).display()
```

得到的情感树如下：

股吧数据情感分类

历经89天的煎熬之后，7月15日中兴终于盼来了解禁，在此首先恭喜中兴，解禁了，希望再踏征程。

但在7月15日之前，随着中美贸易战不断升级，中兴股价又上演了一场“跌跌不休”的惨状，我以中美贸易战背景下中兴通讯在股吧解禁前一段时间的评论数据，来进行情感数据人工打标签和分类。其中，把消极 、中性 、积极分别用0、1、2来表示。

整个文本分类流程主要包括以下6个步骤：

- 中文语料；
- 分词；
- 复杂规则；
- 特征向量；
- 算法建模；

- 情感分析。

本次分类算法采用 CNN，首先引入需要的包：

```
import pandas as pd
import numpy as np
import jieba
import random
import keras
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalMaxPooling1D
from keras.datasets import imdb
from keras.models import model_from_json
from keras.utils import np_utils
import matplotlib.pyplot as plt
```

继续引入停用词和语料文件：

```
dir = "D://ProgramData//PythonWorkSpace//chat//chat8//"
stopwords=pd.read_csv(dir
+"stopwords.txt",index_col=False,quoting=3,sep="\t",names=
['stopword'], encoding='utf-8')
stopwords=stopwords['stopword'].values
```

```
df_data1 = pd.read_csv(dir+"data1.csv",encoding='utf-8')
df_data1.head()
```

下图展示数据的前5行：

	Id	title	time	content	replay	all_replay	time.1	label
0	334	蝴蝶效应啊，国家队就不能伸出援手吗	2018/6/22 9:34	蝴蝶效应啊，国家队就不能伸出援手吗	（2）	\r\n 救不起呀\r\n ...	2018-06-22 09:45:00	0
1	341	根据港股跌幅计算，中兴通讯今天明天必开板。	2018/6/22 9:42	根据港股跌幅计算，中兴通讯今天明天必开板。	（3）	\r\n 看看还有多少封单，别忽悠了，...	2018-06-22 09:47:31	0
2	344	博傻开始	2018/6/22 9:35	今天半仓，明天低开全仓	（25）	\r\n 今天打不开吧？这么大的压单\...	2018-06-22 09:40:28	0
3	345	奎恩	2018/6/22 11:58	110万资金惨遭七连跌的杀戮，只剩下51万，赤裸裸的屠杀	（2）	\r\n 能剩1万算你牛！\r\n ...	2018-06-22 15:38:35	0
4	346	亏大了——被平仓——也平不掉	2018/6/22 12:46	000063：5万自己的加5万荣资的，现在都是卷商的还卖不掉，还在吹我加钱吖	（3）	\r\n 偷鸡不成蚀把米啊，你想着赚更...	2018-06-22 12:59:25	0

接着进行数据预处理，把消极、中性、积极分别为0、1、2的预料分别拿出来：

```
#把内容有缺失值的删除
df_data1.dropna(inplace=True)
#抽取文本数据和标签
data_1 = df_data1.loc[:,['content','label']]
#把消极  中性  积极分别为0、1、2的预料分别拿出来
data_label_0 = data_1.loc[data_1['label'] ==0,:]
data_label_1 = data_1.loc[data_1['label'] ==1,:]
data_label_2 = data_1.loc[data_1['label'] ==2,:]
```

接下来，定义中文分词函数：

```
#定义分词函数
def preprocess_text(content_lines, sentences, category):
    for line in content_lines:
        try:
            segs=jieba.lcut(line)
            segs = filter(lambda x:len(x)>1, segs)
            segs = [v for v in segs if not str(v).isdigit()]#去数字
            segs = list(filter(lambda x:x.strip(), segs)) #去左右空格
            segs = filter(lambda x:x not in stopwords, segs)
            temp = " ".join(segs)
            if(len(temp)>1):
                sentences.append((temp, category))
        except Exception:
            print(line)
```

```
            continue
```

生成训练的分词数据，并进行打散，使其分布均匀：

```
#获取数据
data_label_0_content = data_label_0['content'].values.tolist()
data_label_1_content = data_label_1['content'].values.tolist()
data_label_2_content = data_label_2['content'].values.tolist()
#生成训练数据
sentences = []
preprocess_text(data_label_0_content, sentences, 0)
preprocess_text(data_label_1_content, sentences, 1)
preprocess_text(data_label_2_content, sentences,2)
#我们打乱一下顺序，生成更可靠的训练集
random.shuffle(sentences)
```

对数据集进行切分，按照训练集合测试集7:3的比例：

```
#所以把原数据集分成训练集的测试集，咱们用sklearn自带的分割函数。
from sklearn.model_selection import train_test_split
x, y = zip(*sentences)
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.3,random_state=1234)
```

然后，对特征构造词向量：

```
#抽取特征，我们对文本抽取词袋模型特征
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer(
    analyzer='word', #tokenise by character ngrams
    max_features=4000,  #keep the most common 1000 ngrams
)
vec.fit(x_train)
```

定义模型参数：

```
# 设置参数
max_features = 5001
maxlen = 100
batch_size = 32
```

```
embedding_dims = 50
filters = 250
kernel_size = 3
hidden_dims = 250
epochs = 10
nclasses = 3
```

输入特征转成 Array 和标签处理，打印训练集和测试集的 shape：

```
x_train = vec.transform(x_train)
x_test = vec.transform(x_test)
x_train = x_train.toarray()
x_test = x_test.toarray()   分类-》One-Hot编码
y_train = np_utils.to_categorical(y_train,nclasses)
y_test = np_utils.to_categorical(y_test,nclasses)
x_train = sequence.pad_sequences(x_train, maxlen=maxlen) 只取100个信息
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
```

定义一个绘制 Loss 曲线的类：

自定义：回调函数，方便画图观察

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = {'batch':[], 'epoch':[]}
        self.accuracy = {'batch':[], 'epoch':[]}
        self.val_loss = {'batch':[], 'epoch':[]}
        self.val_acc = {'batch':[], 'epoch':[]}

    def on_batch_end(self, batch, logs={}):
        self.losses['batch'].append(logs.get('loss'))
        self.accuracy['batch'].append(logs.get('acc'))
        self.val_loss['batch'].append(logs.get('val_loss'))
        self.val_acc['batch'].append(logs.get('val_acc'))

    def on_epoch_end(self, batch, logs={}):
        self.losses['epoch'].append(logs.get('loss'))
        self.accuracy['epoch'].append(logs.get('acc'))
        self.val_loss['epoch'].append(logs.get('val_loss'))
        self.val_acc['epoch'].append(logs.get('val_acc'))

    def loss_plot(self, loss_type):
        iters = range(len(self.losses[loss_type]))
```

```
            plt.figure()
            # acc
            plt.plot(iters, self.accuracy[loss_type], 'r', label='train acc
')
            # loss
            plt.plot(iters, self.losses[loss_type], 'g', label='train loss'
)
            if loss_type == 'epoch':
                # val_acc
                plt.plot(iters, self.val_acc[loss_type], 'b', label='val ac
c')
                # val_loss
                plt.plot(iters, self.val_loss[loss_type], 'k', label='val l
oss')
            plt.grid(True)
            plt.xlabel(loss_type)
            plt.ylabel('acc-loss')
            plt.legend(loc="upper right")
            plt.show()
```

然后，初始化上面类的对象，并作为模型的回调函数输入，训练模型：

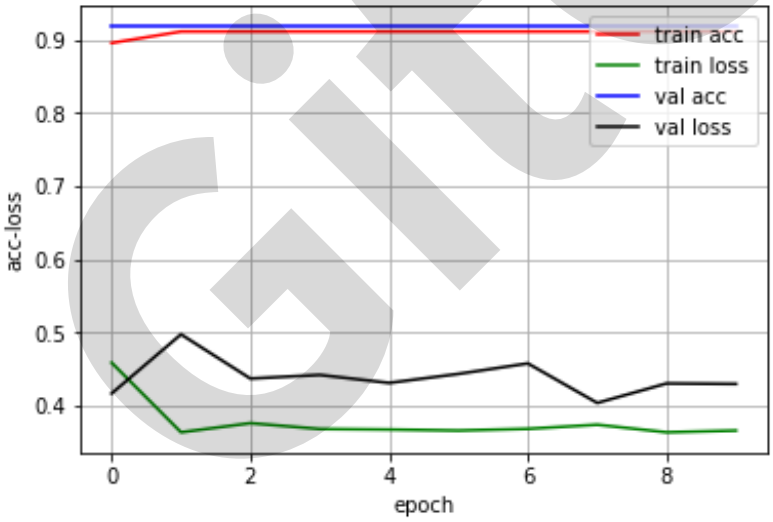
```
history = LossHistory()
print('Build model...')
model = Sequential()

model.add(Embedding(max_features,
                    embedding_dims,
                    input_length=maxlen))
model.add(Dropout(0.5))
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1))
model.add(GlobalMaxPooling1D())
model.add(Dense(hidden_dims))
model.add(Dropout(0.5))
model.add(Activation('relu'))
model.add(Dense(nclasses))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

```
model.fit(x_train, y_train,
                batch_size=batch_size,
                epochs=epochs,
                validation_data=(x_test, y_test),callbacks=[history])
```

得到的模型迭代次数为10轮的训练过程：

```
Build model...
Train on 2049 samples, validate on 683 samples
Epoch 1/10
2049/2049 [==============================] - 7s 4ms/step - loss: 0.4584 - acc: 0.8956 - val_loss: 0.4164 - val_acc: 0.9180
Epoch 2/10
2049/2049 [==============================] - 8s 4ms/step - loss: 0.3633 - acc: 0.9112 - val_loss: 0.4972 - val_acc: 0.9180
Epoch 3/10
2049/2049 [==============================] - 7s 4ms/step - loss: 0.3758 - acc: 0.9112 - val_loss: 0.4368 - val_acc: 0.9180
Epoch 4/10
2049/2049 [==============================] - 7s 3ms/step - loss: 0.3679 - acc: 0.9112 - val_loss: 0.4417 - val_acc: 0.9180
Epoch 5/10
2049/2049 [==============================] - 7s 4ms/step - loss: 0.3671 - acc: 0.9112 - val_loss: 0.4308 - val_acc: 0.9180
Epoch 6/10
2049/2049 [==============================] - 7s 4ms/step - loss: 0.3657 - acc: 0.9112 - val_loss: 0.4431 - val_acc: 0.9180
Epoch 7/10
2049/2049 [==============================] - 7s 4ms/step - loss: 0.3681 - acc: 0.9112 - val_loss: 0.4574 - val_acc: 0.9180
Epoch 8/10
2049/2049 [==============================] - 7s 4ms/step - loss: 0.3737 - acc: 0.9112 - val_loss: 0.4034 - val_acc: 0.9180
Epoch 9/10
2049/2049 [==============================] - 7s 3ms/step - loss: 0.3633 - acc: 0.9112 - val_loss: 0.4301 - val_acc: 0.9180
Epoch 10/10
2049/2049 [==============================] - 7s 3ms/step - loss: 0.3657 - acc: 0.9112 - val_loss: 0.4294 - val_acc: 0.9180
```

最后绘制 Loss 图像： 画图：history.loss_plot('epoch')



关于本次分类，这里重点讨论的一个知识点就是数据分布不均匀的情况，我们都知道，本次贸易战中兴公司受影响很大，导致整个股票价格处于下跌趋势，所以整个舆论上，大多数评论都是消极的态度，导致数据分布极不均匀。

那数据分布不均匀一般怎么处理呢？从以下几个方面考虑：

- 数据采样，包括上采样、下采样和综合采样；
- 改变分类算法，在传统分类算法的基础上对不同类别采取不同的加权方式，使得模型更看重少数类；
- 采用合理的性能评价指标；
- 代价敏感。

总结，本文通过第三方、基于词典等方式计算中文文本情感值，以及通过情感树来进行可视化，然而这些内容只是情感分析的入门知识，情感分析还涉及句法依存等，最后通过一个 CNN 分类模型，提供一种有监督的情感分类思路。

参考文献及推荐阅读：

1. 基于情感词典的中文自然语言处理情感分析（上）
2. 基于情感词典的中文自然语言处理情感分析（下）

第13课：动手制作自己的简易聊天机器人

自动问答简介

自动聊天机器人，也称为自动问答系统，由于所使用的场景不同，叫法也不一样。自动问答（Question Answering，QA）是指利用计算机自动回答用户所提出的问题以满足用户知识需求的任务。不同于现有搜索引擎，问答系统是信息服务的一种高级形式，系统返回用户的不再是基于关键词匹配排序的文档列表，而是精准的自然语言答案。近年来，随着人工智能的飞速发展，自动问答已经成为倍受关注且发展前景广泛的研究方向。

自动问答主要研究的内容和关键科学问题如下：

1. **问句理解**：给定用户问题，自动问答首先需要理解用户所提问题。用户问句的语义理解包含词法分析、句法分析、语义分析等多项关键技术，需要从文本的多个维度理解其中包含的语义内容。
2. **文本信息抽取**：自动问答系统需要在已有语料库、知识库或问答库中匹配相关的信息，并抽取出相应的答案。
3. **知识推理**：自动问答中，由于语料库、知识库和问答库本身的覆盖度有限，并不是所有问题都能直接找到答案。这就需要在已有的知识体系中，通过知识推理的手段获取这些隐含的答案。

纵观自动问答研究的发展态势和技术现状，以下研究方向或问题将可能成为未来整个领域和行业重点关注的方向：基于深度学习的端到端自动问答，多领域、多语言的自动问答，面向问答的深度推理，篇章阅读理解、对话等。

基于 Chatterbot 制作中文聊天机器人

ChatterBot 是一个构建在 Python 上，基于一系列规则和机器学习算法完成的聊天机器人，具有结构清晰，可扩展性好，简单实用的特点。

Chatterbot 安装有两种方式：

- 使用 `pip install chatterbot` 安装；
- 直接在 Github Chatterbot 下载这个项目，通过 `python setup.py install` 安装，其中 examples 文件夹中包含几个例子，可以根据例子加深自己的理解。

安装过程如果出现错误，主要是需要安装这些依赖库：

```
chatterbot-corpus>=1.1,<1.2
mathparse>=0.1,<0.2
nltk>=3.2,<4.0
pymongo>=3.3,<4.0
python-dateutil>=2.6,<2.7
python-twitter>=3.0,<4.0
sqlalchemy>=1.2,<1.3
pint>=0.8.1
```

1. 手动设置一点语料，体验基于规则的聊天机器人回答。

```
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer
Chinese_bot = ChatBot("Training demo") #创建一个新的实例
Chinese_bot.set_trainer(ListTrainer)
Chinese_bot.train([
    '亲，在吗？',
    '亲，在呢',
    '这件衣服的号码大小标准吗？',
    '亲，标准呢，请放心下单吧。',
    '有红色的吗？',
    '有呢，目前有白红蓝3种色调。',
```

```
])
```

下面进行测试：

```
# 测试一下
question = '亲，在吗'
print(question)
response = Chinese_bot.get_response(question)
print(response)
print("\n")
question = '有红色的吗？'
print(question)
response = Chinese_bot.get_response(question)
print(response)
```

从得到的结果可以看出，这应该完全是基于规则的判断：

> 亲，在吗
>
> 亲，在呢
>
> 有红色的吗？
>
> 有呢，目前有白红蓝3种色调。

2. 训练自己的语料。

本次使用的语料来自 QQ 群的聊天记录，导出的 QQ 聊天记录稍微处理一下即可使用，整个过程如下。

（1）首先载入语料，第二行代码主要是想把每句话后面的换行 `\n` 去掉。

```
lines = open("QQ.txt","r",encoding='gbk').readlines()
sec = [ line.strip() for line in lines]
```

（2）接下来就可以训练模型了，由于整个语料比较大，训练过程也比较耗时。

```
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer
```

```
Chinese_bot = ChatBot("Training")
Chinese_bot.set_trainer(ListTrainer)
Chinese_bot.train(sec)
```

这里需要注意，如果训练过程很慢，可以在第一步中加入如下代码，即只取前1000条进行训练：

```
sec = sec[0:1000]
```

（3）最后，对训练好的模型进行测试，可见训练数据是 QQ 群技术对话，也看得出程序员们都很努力，整体想的都是学习。

```
In [12]: Chinese_bot.get_response("努力学习")
Out[12]: <Statement text:那我这个本科生能出去干什么..>

In [13]: Chinese_bot.get_response("做开发呀")
Out[13]: <Statement text:什么都不会>

In [14]: Chinese_bot.get_response("可以学啊")
Out[14]: <Statement text:那些自学的是学习不好的大部分>

In [17]: Chinese_bot.get_response("话不能这样说吧")
Out[17]: <Statement text:数学可以读研转IT啊>
```

以上只是简单的 Chatterbot 演示，如果想看更好的应用，推荐看官方文档。

基于 Seq2Seq 制作中文聊天机器人

前面，我们在第09课《一网打尽神经序列模型之 RNN 及其变种 LSTM、GRU》中讲了序列数据处理模型，从 `N-gram` 语言模型到 RNN 及其变种。这里我们讲另外一个基于深度学习的 Seq2Seq 模型。

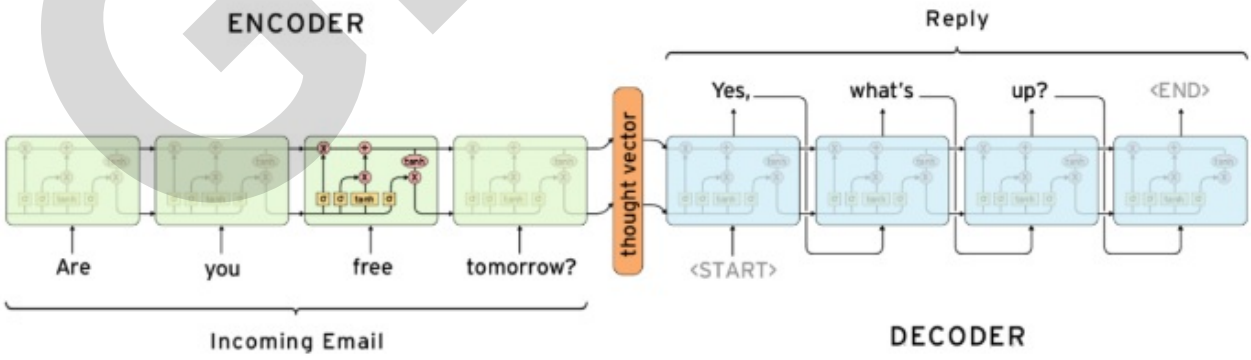
从 RNN 结构说起，根据输出和输入序列不同数量 RNN ，可以有多种不同的结构，不同结构自然就有不同的引用场合。

- One To One 结构，仅仅只是简单的给一个输入得到一个输出，此处并未体现序列的特征，例如图像分类场景。

- One To Many 结构，给一个输入得到一系列输出，这种结构可用于生产图片描述的场景。
- Many To One 结构，给一系列输入得到一个输出，这种结构可用于文本情感分析，对一些列的文本输入进行分类，看是消极还是积极情感。
- Many To Many 结构，给一系列输入得到一系列输出，这种结构可用于翻译或聊天对话场景，将输入的文本转换成另外一系列文本。
- 同步 Many To Many 结构，它是经典的 RNN 结构，前一输入的状态会带到下一个状态中，而且每个输入都会对应一个输出，我们最熟悉的应用场景是字符预测，同样也可以用于视频分类，对视频的帧打标签。

在 Many To Many 的两种模型中，第四和第五种是有差异的，经典 RNN 结构的输入和输出序列必须要等长，它的应用场景也比较有限。而第四种，输入和输出序列可以不等长，这种模型便是 Seq2Seq 模型，即 Sequence to Sequence。它实现了从一个序列到另外一个序列的转换，比如 Google 曾用 Seq2Seq 模型加 Attention 模型实现了翻译功能，类似的还可以实现聊天机器人对话模型。经典的 RNN 模型固定了输入序列和输出序列的大小，而 Seq2Seq 模型则突破了该限制。

Seq2Seq 属于 `Encoder-Decoder` 结构，这里看看常见的 `Encoder-Decoder` 结构。基本思想就是利用两个 RNN，一个 RNN 作为 Encoder，另一个 RNN 作为 Decoder。Encoder 负责将输入序列压缩成指定长度的向量，这个向量就可以看成是这个序列的语义，这个过程称为编码，如下图，获取语义向量最简单的方式就是直接将最后一个输入的隐状态作为语义向量。也可以对最后一个隐含状态做一个变换得到语义向量，还可以将输入序列的所有隐含状态做一个变换得到语义变量。



具体理论知识这里不再赘述，下面重点看看，如何通过 Keras 实现一个 `LSTM_Seq2Seq` 自动问答机器人。

1. 语料准备。

语料我们使用 Tab 键 `\t` 把问题和答案区分，每一对为一行。其中，语料为爬虫爬取的工程机械网站的问答。

2. 模型构建和训练。

第一步，引入需要的包：

```
from keras.models import Model
from keras.layers import Input, LSTM, Dense
import numpy as np
import pandas as pd
```

第二步，定义模型超参数、迭代次数、语料路径：

```
#Batch size 的大小
batch_size = 32
# 迭代次数epochs
epochs = 100
# 编码空间的维度Latent dimensionality
latent_dim = 256
# 要训练的样本数
num_samples = 5000
#设置语料的路径
data_path = 'D://nlp//ch13//files.txt'
```

第三步，把语料向量化：

```
#把数据向量话
input_texts = []
target_texts = []
input_characters = set()
target_characters = set()

with open(data_path, 'r', encoding='utf-8') as f:
    lines = f.read().split('\n')
for line in lines[: min(num_samples, len(lines) - 1)]:
    #print(line)
    input_text, target_text = line.split('\t')
```

```
        # We use "tab" as the "start sequence" character
        # for the targets, and "\n" as "end sequence" character.
        target_text = target_text[0:100]
        target_text = '\t' + target_text + '\n'
        input_texts.append(input_text)
        target_texts.append(target_text)

        for char in input_text:
            if char not in input_characters:
                input_characters.add(char)
        for char in target_text:
            if char not in target_characters:
                target_characters.add(char)

input_characters = sorted(list(input_characters))
target_characters = sorted(list(target_characters))
num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)
max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])

print('Number of samples:', len(input_texts))
print('Number of unique input tokens:', num_encoder_tokens)
print('Number of unique output tokens:', num_decoder_tokens)
print('Max sequence length for inputs:', max_encoder_seq_length)
print('Max sequence length for outputs:', max_decoder_seq_length)

input_token_index = dict(
    [(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict(
    [(char, i) for i, char in enumerate(target_characters)])

encoder_input_data = np.zeros(
    (len(input_texts), max_encoder_seq_length,
num_encoder_tokens),dtype='float32')
decoder_input_data = np.zeros(
    (len(input_texts), max_decoder_seq_length,
num_decoder_tokens),dtype='float32')
decoder_target_data = np.zeros(
    (len(input_texts), max_decoder_seq_length,
num_decoder_tokens),dtype='float32')

for i, (input_text, target_text) in enumerate(zip(input_texts,
target_texts)):
    for t, char in enumerate(input_text):
```

```
            encoder_input_data[i, t, input_token_index[char]] = 1.
        for t, char in enumerate(target_text):
            # decoder_target_data is ahead of decoder_input_data by one tim
    estep
            decoder_input_data[i, t, target_token_index[char]] = 1.
            if t > 0:
                # decoder_target_data will be ahead by one timestep
                # and will not include the start character.
                decoder_target_data[i, t - 1, target_token_index[char]] = 1
    .
```

第四步，`LSTM_Seq2Seq` 模型定义、训练和保存：

```
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
# 输出 `encoder_outputs`
encoder_states = [state_h, state_c]

# 状态 `encoder_states`
decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=Tr
ue)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

# 定义模型
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# 训练
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
model.fit([encoder_input_data, decoder_input_data],
decoder_target_data,
          batch_size=batch_size,
          epochs=epochs,
          validation_split=0.2)
# 保存模型
model.save('s2s.h5')
```

第五步，Seq2Seq 的 Encoder 操作：

```
encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs,
    [decoder_outputs] + decoder_states)
```

第六步，把索引和分词转成序列：

```
reverse_input_char_index = dict(
    (i, char) for char, i in input_token_index.items())
reverse_target_char_index = dict(
    (i, char) for char, i in target_token_index.items())
```

第七步，定义预测函数，先使用预模型预测，然后编码成汉字结果：

```
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq)
    #print(states_value)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # Populate the first character of target sequence with the start c
haracter.
    target_seq[0, 0, target_token_index['\t']] = 1.

    # Sampling loop for a batch of sequences
    # (to simplify, here we assume a batch of size 1).
    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value)

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
```

```
            sampled_char = reverse_target_char_index[sampled_token_index]
            decoded_sentence += sampled_char
            if (sampled_char == '\n' or
               len(decoded_sentence) > max_decoder_seq_length):
                stop_condition = True

            # Update the target sequence (of length 1).
            target_seq = np.zeros((1, 1, num_decoder_tokens))
            target_seq[0, 0, sampled_token_index] = 1.
            # 更新状态
            states_value = [h, c]
        return decoded_sentence
```

3. 模型预测。

首先，定义一个预测函数：

```
def predict_ans(question):
        inseq = np.zeros((len(question), max_encoder_seq_length, num_encoder_tokens),dtype='float16')
        decoded_sentence = decode_sequence(inseq)
        return decoded_sentence
```

然后就可以预测了：

```
print('Decoded sentence:', predict_ans("挖机履带掉了怎么装上去"))
```

总结

本文我们首先基于 Chatterbot 制作了中文聊天机器人，并用 QQ 群对话语料自己尝试训练。然后通过 LSTM 和 Seq2Seq 模型，根据爬取的语料，训练了一个自动问答的模型，通过以上两种方式，我们们对自动问答有了一个简单的入门。

参考文献及推荐阅读：

1. 《中文信息处理发展报告（2016）》
2. ChatterBot 文档
3. ChatterBot 的 GitHub

4. Sutskever, Vinyals and Le (2014)
5. 漫谈四种神经网络序列解码模型
6. 怎样导出 QQ 群里的所有聊天记录？
7. Keras 中的LSTM_Seq2Seq例子

第14课：动手实战中文命名实体提取

命名实体识别（Named EntitiesRecognition，NER）是自然语言处理的一个基础任务。其目的是识别语料中人名、地名、组织机构名等命名实体，比如，2015年中国国家海洋局对124个国际海底地理实体的命名。

由于命名实体数量不断增加，通常不可能在词典中穷尽列出，且其构成方法具有各自的一些规律性，因而，通常把对这些词的识别从词汇形态处理（如汉语切分）任务中独立处理，称为命名实体识别。

命名实体识别技术是信息抽取、信息检索、机器翻译、问答系统等多种自然语言处理技术必不可少的组成部分。

常见的命名实体识别方法综述

命名实体是命名实体识别的研究主体，一般包括三大类（实体类、时间类和数字类）和七小类（人名、地名、机构名、时间、日期、货币和百分比）命名实体。评判一个命名实体是否被正确识别包括两个方面：实体的边界是否正确和实体的类型是否标注正确。

命名实体识别的主要技术方法分为：基于规则和词典的方法、基于统计的方法、二者混合的方法等。

1.基于规则和词典的方法。

基于规则的方法多采用语言学专家手工构造规则模板，选用特征包括统计信息、标点符号、关键字、指示词和方向词、位置词（如尾字）、中心词等方法，以模式和字符串相匹配为主要手段，这类系统大多依赖于知识库和词典的建立。基于规则和词典的方法是命名实体识别中最早使用的方法，一般而言，当提取的规则能比较精确地反映语言现象时，基于规则的方法性能要优于基于统计的方法。但是这些规则往往依赖于具体语言、领域和文本风格，编制过程耗时且难以涵盖所有的语言现象，特别容易产生错误，系统可移植性不好，对于不同的系统需要语言学专家重新书写规则。基于规则的方法的另外一个缺点是代价太大，存在系统建设周期长、移植性差而且需要建立不同领域知识库作为辅助以提高系统识别能力等问题。

2.基于统计的方法。

基于统计机器学习的方法主要包括隐马尔可夫模型（HiddenMarkovMode，HMM）、最大熵（MaxmiumEntropy，ME）、支持向量机（Support VectorMachine，SVM）、条件随机场（ConditionalRandom Fields，CRF）等。

在基于统计的这四种学习方法中，最大熵模型结构紧凑，具有较好的通用性，主要缺点是训练时间长复杂性高，有时甚至导致训练代价难以承受，另外由于需要明确的归一化计算，导致开销比较大。而条件随机场为命名实体识别提供了一个特征灵活、全局最优的标注框架，但同时存在收敛速度慢、训练时间长的问题。一般说来，最大熵和支持向量机在正确率上要比隐马尔可夫模型高一些，但隐马尔可夫模型在训练和识别时的速度要快一些，主要是由于在利用 Viterbi 算法求解命名实体类别序列时的效率较高。隐马尔可夫模型更适用于一些对实时性有要求以及像信息检索这样需要处理大量文本的应用，如短文本命名实体识别。

基于统计的方法对特征选取的要求较高，需要从文本中选择对该项任务有影响的各种特征，并将这些特征加入到特征向量中。依据特定命名实体识别所面临的主要困难和所表现出的特性，考虑选择能有效反映该类实体特性的特征集合。主要做法是通过对训练语料所包含的语言信息进行统计和分析，从训练语料中挖掘出特征。有关特征可以分为具体的单词特征、上下文特征、词典及词性特征、停用词特征、核心词特征以及语义特征等。

基于统计的方法对语料库的依赖也比较大，而可以用来建设和评估命名实体识别系统的大规模通用语料库又比较少。

3.混合方法。

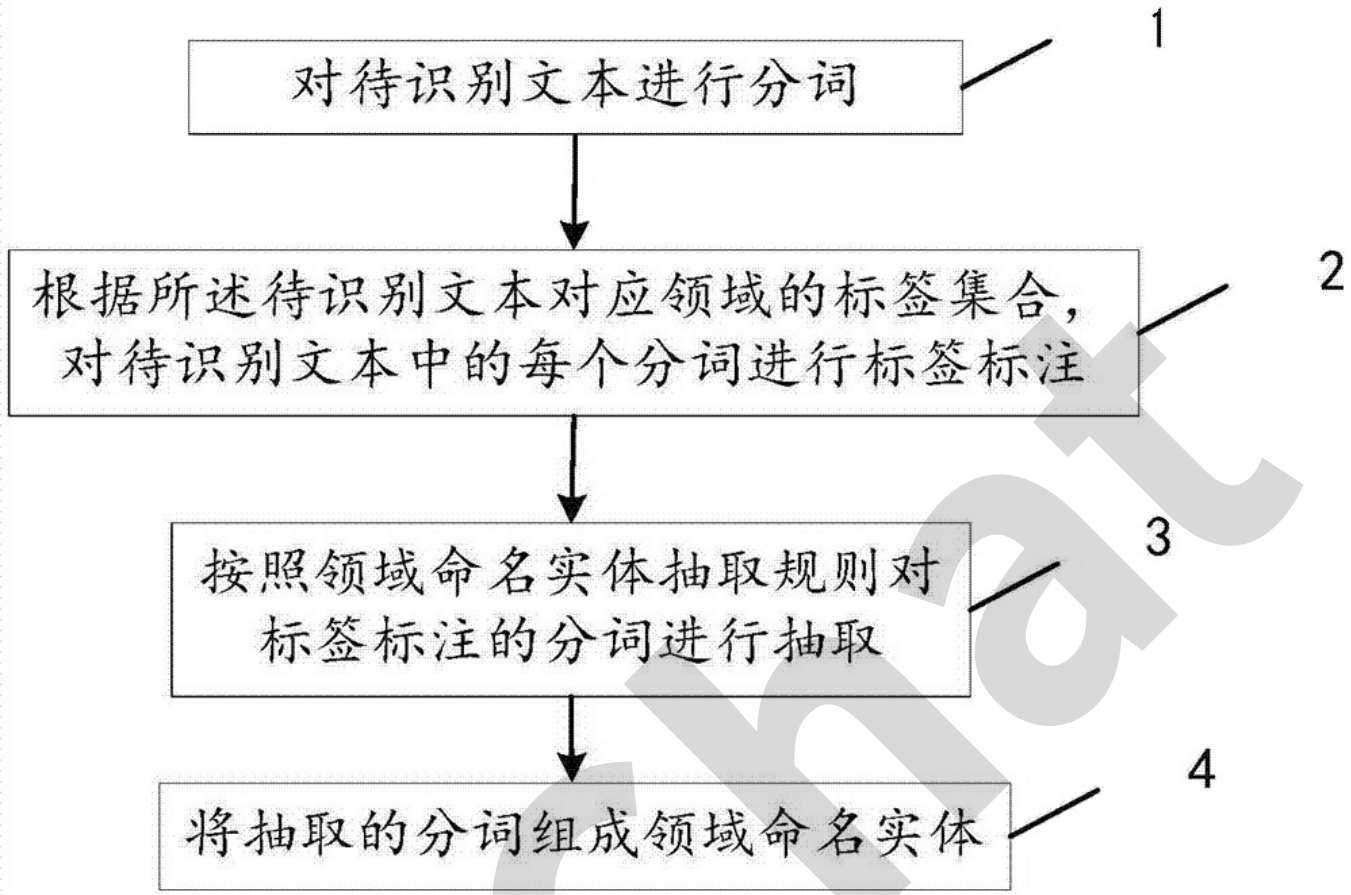
自然语言处理并不完全是一个随机过程，单独使用基于统计的方法使状态搜索空间非常庞大，必须借助规则知识提前进行过滤修剪处理。目前几乎没有单纯使用统计模型而不使用规则知识的命名实体识别系统，在很多情况下是使用混合方法：

1. 统计学习方法之间或内部层叠融合。
2. 规则、词典和机器学习方法之间的融合，其核心是融合方法技术。在基于统计的学习方法中引入部分规则，将机器学习和人工知识结合起来。
3. 将各类模型、算法结合起来，将前一级模型的结果作为下一级的训练数据，并用这些训练数据对模型进行训练，得到下一级模型。

命名实体识别的一般流程

如下图所示，一般的命名实体流程主要分为四个步骤：

1. 对需要进行提取的文本语料进行分词；
2. 获取需要识别的领域标签，并对分词结果进行标签标注；
3. 对标签标注的分词进行抽取；
4. 将抽取的分词组成需要的领域的命名实体。



动手实战命名实体识别

命名实体的类别，我们在第01课《中文自然语言处理的完整机器处理流程》中已经给出了，这里不再赘述，下面通过 jieba 分词包和 pyhanlp 来实战命名实体识别和提取。

1.jieba 进行命名实体识别和提取。

第一步，引入 jieba 包：

```
import jieba
import jieba.analyse
import jieba.posseg as posg
```

第二步，使用 jieba 进行词性切分，allowPOS 指定允许的词性，这里选择名词 n 和地名 ns：

```
sentence=u'''上线三年就成功上市，拼多多上演了互联网企业的上市奇迹，却也放大平台上存在的诸多问题，拼多多在美国上市。'''
```

```
kw=jieba.analyse.extract_tags(sentence,topK=10,withWeight=True,allowPOS=('n','ns'))
    for item in kw:
        print(item[0],item[1])
```

在这里，我们可以得到打印出来的结果：

> 上市 1.437080435586
>
> 上线 0.820694551317
>
> 奇迹 0.775434839431
>
> 互联网 0.712189275429
>
> 平台 0.6244340485550001
>
> 企业 0.422177218495
>
> 美国 0.415659623166
>
> 问题 0.39635135730800003

可以看得出，上市和上线应该是动词，这里给出的结果不是很准确。接下来，我们使用 textrank 算法来试试：

```
    kw=jieba.analyse.textrank(sentence,topK=20,withWeight=True,allowPOS=('ns','n'))
    for item in kw:
        print(item[0],item[1])
```

这次得到的结果如下，可见，两次给出的结果还是不一样的。

> 上市 1.0
>
> 奇迹 0.572687398431635
>
> 企业 0.5710407272273452

> 互联网 0.5692560484441649
>
> 上线 0.23481844682115297
>
> 美国 0.23481844682115297

2.pyhanlp 进行命名实体识别和提取。

第一步，引入pyhanlp包：

```
from pyhanlp import *
```

第二步，进行词性切分：

```
sentence=u'''上线三年就成功上市,拼多多上演了互联网企业的上市奇迹,却也放大平台上存在的诸多问题，拼多多在美国上市。'''
analyzer = PerceptronLexicalAnalyzer()
segs = analyzer.analyze(sentence)
arr = str(segs).split(" ")
```

第三步，定义一个函数，从得到的结果中，根据词性获取指定词性的词：

```
def get_result(arr):
    re_list = []
    ner = ['n','ns']
    for x in arr:
        temp = x.split("/")
        if(temp[1] in ner):
            re_list.append(temp[0])
    return re_list
```

第四步，我们获取结果：

```
result = get_result(arr)
print(result)
```

得到的结果如下，可见比 jieba 更准确：

```
['互联网', '企业', '奇迹', '平台', '问题', '美国']
```

总结

本文对命名实体识别的方法进行了总结，并给出一般的处理流程，最后通过简单的 jieba 分词和 pyhanlp 分词根据词性获取实体对象，后续大家也可以尝试通过哈工大和斯坦福的包来处理，下篇我们通过条件随机场 CRF 来训练一个命名实体识别模型。

参考文献及推荐阅读

1. 命名实体识别研究[J]，国防科技大学计算机学院-张晓艳、王挺、陈火旺。
2. 命名实体识别

第15课：基于 CRF 的中文命名实体识别模型实现

命名实体识别在越来越多的场景下被应用，如自动问答、知识图谱等。非结构化的文本内容有很多丰富的信息，但找到相关的知识始终是一个具有挑战性的任务，命名实体识别也不例外。

前面我们用隐马尔可夫模型（HMM）自己尝试训练过一个分词器，其实 HMM 也可以用来训练命名实体识别器，但在本文，我们讲另外一个算法——条件随机场（CRF），来训练一个命名实体识别器。

浅析条件随机场（CRF）

条件随机场（Conditional Random Fields，简称 CRF）是给定一组输入序列条件下另一组输出序列的条件概率分布模型，在自然语言处理中得到了广泛应用。

首先，我们来看看什么是随机场。“随机场”的名字取的很玄乎，其实理解起来不难。随机场是由若干个位置组成的整体，当按照某种分布给每一个位置随机赋予一个值之后，其全体就叫做随机场。

还是举词性标注的例子。假如我们有一个十个词形成的句子需要做词性标注。这十个词每个词的词性可以在我们已知的词性集合（名词，动词……)中去选择。当我们为每个词选择完词性后，这就形成了一个随机场。

了解了随机场，我们再来看看马尔科夫随机场。马尔科夫随机场是随机场的特例，它假设随机场中某一个位置的赋值仅仅与和它相邻的位置的赋值有关，和与其不相邻的位置的赋值无关。

继续举十个词的句子词性标注的例子。如果我们假设所有词的词性只和它相邻的词的词性有关时，这个随机场就特化成一个马尔科夫随机场。比如第三个词的词性除了与自己本身的位置有关外，还只与第二个词和第四个词的词性有关。

理解了马尔科夫随机场，再理解 CRF 就容易了。CRF 是马尔科夫随机场的特例，它假设马尔科夫随机场中只有 X 和 Y 两种变量，X 一般是给定的，而 Y 一般是在给定 X 的条件下我们的输出。这样马尔科夫随机场就特化成了条件随机场。

在我们十个词的句子词性标注的例子中，X 是词，Y 是词性。因此，如果我们假设它是一个马尔科夫随机场，那么它也就是一个 CRF。

对于 CRF，我们给出准确的数学语言描述：设 X 与 Y 是随机变量，P(Y|X) 是给定 X 时 Y 的条件概率分布，若随机变量 Y 构成的是一个马尔科夫随机场，则称条件概率分布 P(Y|X) 是条件随机场。

基于 CRF 的中文命名实体识别模型实现

在常规的命名实体识别中，通用场景下最常提取的是时间、人物、地点及组织机构名，因此本模型也将提取以上四种实体。

1.开发环境。

本次开发所选用的环境为：

- `Sklearn_crfsuite`
- Python 3.6
- Jupyter Notebook

2.数据预处理。

本模型使用人民日报1998年标注数据，进行预处理。语料库词性标记中，对应的实体词依次为 t、nr、ns、nt。对语料需要做以下处理：

- 将语料全角字符统一转为半角；
- 合并语料库分开标注的姓和名，例如：`温/nr 家宝/nr`；
- 合并语料库中括号中的大粒度词，例如：`[国家/n 环保局/n]nt`；
- 合并语料库分开标注的时间，例如：`（/w 一九九七年/t 十二月/t 三十一日/t ）/w`。

首先引入需要用到的库：

```
import re
import sklearn_crfsuite
from sklearn_crfsuite import metrics
from sklearn.externals import joblib
```

数据预处理，定义 CorpusProcess 类，我们还是先给出类实现框架：

```
class CorpusProcess(object):

    def __init__(self):
        """初始化"""
        pass

    def read_corpus_from_file(self, file_path):
        """读取语料"""
        pass

    def write_corpus_to_file(self, data, file_path):
        """写语料"""
        pass

    def q_to_b(self,q_str):
        """全角转半角"""
        pass

    def b_to_q(self,b_str):
        """半角转全角"""
        pass

    def pre_process(self):
        """语料预处理 """
        pass

    def process_k(self, words):
        """处理大粒度分词,合并语料库中括号中的大粒度分词,类似：[国家/n  环保局/n]nt """
        pass

    def process_nr(self, words):
        """ 处理姓名，合并语料库分开标注的姓和名，类似：温/nr  家宝/nr"""
        pass

    def process_t(self, words):
        """处理时间,合并语料库分开标注的时间词，类似： （/w  一九九七年/t  十二月/t  三十一日/t  ）/w   """
        pass

    def pos_to_tag(self, p):
        """由词性提取标签"""
```

```
        pass

    def tag_perform(self, tag, index):
        """标签使用BIO模式"""
        pass

    def pos_perform(self, pos):
        """去除词性携带的标签先验知识"""
        pass

    def initialize(self):
        """初始化 """
        pass

    def init_sequence(self, words_list):
        """初始化字序列、词性序列、标记序列 """
        pass

    def extract_feature(self, word_grams):
        """特征选取"""
        pass

    def segment_by_window(self, words_list=None, window=3):
        """窗口切分"""
        pass

    def generator(self):
        """训练数据"""
        pass
```

由于整个代码实现过程较长，我这里给出重点步骤，最后会在 **Github 上连同语料代码一同给出**，下面是关键过程实现。

对语料中的句子、词性，实体分类标记进行区分。标签采用“BIO”体系，即实体的第一个字为 `B_*`，其余字为 `I_*`，非实体字统一标记为 O。大部分情况下，标签体系越复杂，准确度也越高，但这里采用简单的 BIO 体系也能达到相当不错的效果。这里模型采用 `tri-gram` 形式，所以在字符列中，要在句子前后加上占位符。

```
def init_sequence(self, words_list):
            """初始化字序列、词性序列、标记序列 """
            words_seq = [[word.split(u'/')[0] for word in words] for wo
rds in words_list]
```

```
            pos_seq = [[word.split(u'/')[1] for word in words] for
words in words_list]
            tag_seq = [[self.pos_to_tag(p) for p in pos] for pos in pos
_seq]
            self.pos_seq = [[[pos_seq[index][i] for _ in range(len(word
s_seq[index][i]))]
                             for i in range(len(pos_seq[index]))] for ind
ex in range(len(pos_seq))]
            self.tag_seq = [[[self.tag_perform(tag_seq[index][i], w) fo
r w in range(len(words_seq[index][i]))]
                             for i in range(len(tag_seq[index]))] for ind
ex in range(len(tag_seq))]
            self.pos_seq = [[u'un']+[self.pos_perform(p) for pos in pos
_seq for p in pos]+[u'un'] for pos_seq in self.pos_seq]
            self.tag_seq = [[t for tag in tag_seq for t in tag] for tag
_seq in self.tag_seq]
            self.word_seq = [[u'<BOS>']+[w for word in word_seq for w i
n word]+[u'<EOS>'] for word_seq in words_seq]
```

处理好语料之后，紧接着进行模型定义和训练，定义 `CRF_NER` 类，我们还是采用先给出类实现框架，再具体讲解其实现：

```
    class CRF_NER(object):
        def __init__(self):
            """初始化参数"""
            pass

        def initialize_model(self):
            """初始化"""
            pass

        def train(self):
            """训练"""
            pass

        def predict(self, sentence):
            """预测"""
            pass
        def load_model(self):
            """加载模型 """
            pass
        def save_model(self):
            """保存模型"""
```

```
            pass
```

在 `CRF_NER` 类中，分别完成了语料预处理和模型训练、保存、预测功能，具体实现如下。

第一步，init 函数实现了模型参数定义和 CorpusProcess 的实例化和语料预处理：

```
    def __init__(self):
            """初始化参数"""
            self.algorithm = "lbfgs"
            self.c1 ="0.1"
            self.c2 = "0.1"
            self.max_iterations = 100 #迭代次数
            self.model_path = dir + "model.pkl"
            self.corpus = CorpusProcess()  #Corpus 实例
            self.corpus.pre_process()  #语料预处理
            self.corpus.initialize()  #初始化语料
            self.model = None
```

第二步，给出模型定义，了解 `sklearn_crfsuite.CRF` 详情可查该文档。

```
    def initialize_model(self):
            """初始化"""
            algorithm = self.algorithm
            c1 = float(self.c1)
            c2 = float(self.c2)
            max_iterations = int(self.max_iterations)
            self.model = sklearn_crfsuite.CRF(algorithm=algorithm, c1=c
1, c2=c2,

max_iterations=max_iterations, all_possible_transitions=True)
```

第三步，模型训练和保存，分为训练集和测试集：

```
    def train(self):
            """训练"""
            self.initialize_model()
            x, y = self.corpus.generator()
            x_train, y_train = x[500:], y[500:]
            x_test, y_test = x[:500], y[:500]
            self.model.fit(x_train, y_train)
            labels = list(self.model.classes_)
```

```
            labels.remove('O')
            y_predict = self.model.predict(x_test)
            metrics.flat_f1_score(y_test, y_predict, average='weighted', labels=labels)
            sorted_labels = sorted(labels, key=lambda name: (name[1:], name[0]))
            print(metrics.flat_classification_report(y_test, y_predict, labels=sorted_labels, digits=3))
            self.save_model()
```

第四至第六步中 predict、`load_model`、`save_model` 方法的实现，大家可以在文末给出的地址中查看源码，这里就不堆代码了。

最后，我们来看看模型训练和预测的过程和结果：

```
    ner = CRF_NER()
    model = ner.train()
```

经过模型训练，得到的准确率和召回率如下：

	precision	recall	f1-score	support
B_LOC	0.944	0.827	0.882	266
I_LOC	0.892	0.801	0.844	1203
B_ORG	0.941	0.913	0.927	682
I_ORG	0.932	0.869	0.899	997
B_PER	0.985	0.918	0.951	440
I_PER	0.983	0.939	0.961	824
B_T	0.993	0.993	0.993	444
I_T	0.995	0.995	0.995	1099
avg / total	0.953	0.904	0.928	5955

进行模型预测，其结果还不错，如下：

```
ner.predict(u'新华社北京十二月三十一日电(中央人民广播电台记者刘振英、新华社记者张宿堂)今天是一九九七年的最后一天。')
'新华社 北京 十二月三十一日 中央人民广播电台 刘振英 新华社 张宿堂 今天 一九九七年'
```

```
ner.predict(u'一九四九年，国庆节，毛泽东同志在天安门城楼上宣布中国共产党从此站起来了！')
'一九四九年 国庆节 毛泽东 天安门城 中国共产党'
```

基于 CRF 的中文命名实体识别模型实现先讲到这儿，项目源码和涉及到的语料，大家可以到：Github 上查看。

总结

本文浅析了条件随机场，并使用 `sklearn_crfsuite.CRF` 模型，对人民日报1998年标注数据进行了模型训练和预测，以帮助大家加强对条件随机场的理解。

参考资料及推荐阅读

1. 条件随机场（CRF）
2. 条件随机场CRF（一）从随机场到线性链条件随机场
3. 命名实体：基于 CRF 的中文命名实体识别模型
4. 条件随机场（CRF）理论及应用

第16课：动手实战中文句法依存分析

句法分析是自然语言处理（NLP）中的关键技术之一，其基本任务是确定句子的句法结构或者句子中词汇之间的依存关系。主要包括两方面的内容：一是确定语言的语法体系，即对语言中合法句子的语法结构给予形式化的定义；另一方面是句法分析技术，即根据给定的语法体系，自动推导出句子的句法结构，分析句子所包含的句法单位和这些句法单位之间的关系。

句法分析被用在很多场景中，比如搜索引擎用户日志分析和关键词识别，比如信息抽取、自动问答、机器翻译等其他自然语言处理相关的任务。

语法体系

句法分析需要遵循某一语法体系，根据该体系的语法确定语法树的表示形式，我们看下面这个句子：

> 西门子将努力参与中国的三峡工程建设。

用可视化的工具 Stanford Parser 来看看句法分析的整个过程：

Your query

西门子将努力参与中国的三峡工程建设。

Segmentation

西门子 将 努力 参与 中国 的 三峡 工程 建设 。

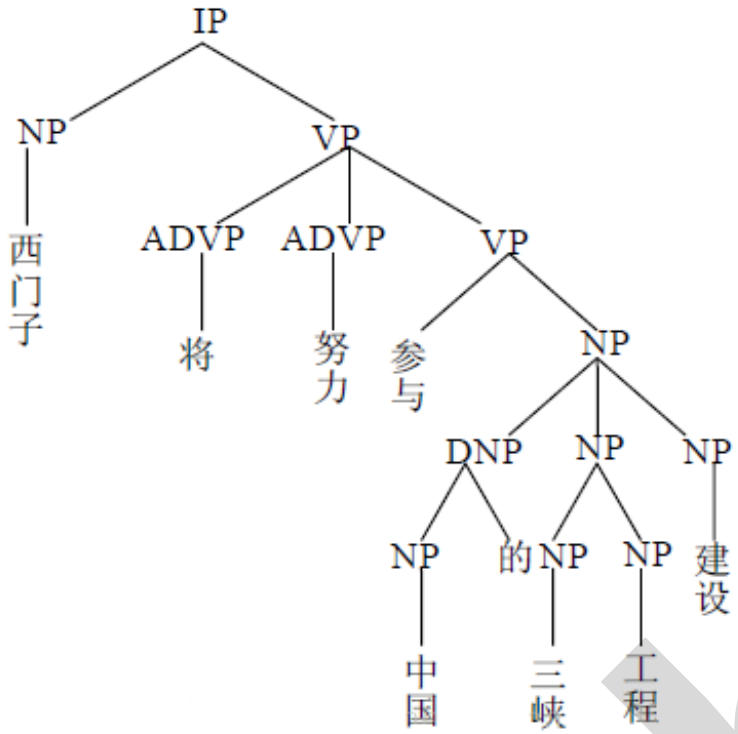
Tagging

西门子/NR 将/AD 努力/AD 参与/VV 中国/NR 的/DEG 三峡/NR 工程/NN 建设/NN 。/PU

Parse

```
(ROOT
  (IP
    (NP (NR 西门子))
    (VP
      (ADVP (AD 将))
      (ADVP (AD 努力))
      (VP (VV 参与)
        (NP
          (DNP
            (NP (NR 中国))
            (DEG 的))
          (NP
            (NP (NR 三峡))
            (NP (NN 工程)))
          (NP (NN 建设)))))
    (PU 。)))
```

短语结构树由终节点、非终结点以及短语标记三部分组成。句子分裂的语法规则为若干终结点构成一个短语，作为非终结点参与下一次规约，直至结束。如下图：



句法分析技术

依存句法分析

依存句法

依存句法（Dependency Parsing， DP）通过分析语言单位内成分之间的依存关系揭示其句法结构。

直观来讲，依存句法的目的在于分析识别句子中的“主谓宾”、“定状补”这些语法成分，并分析各成分之间的关系。

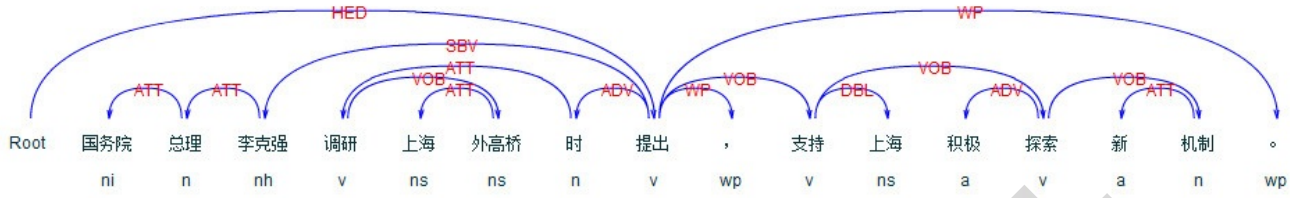
依存句法的结构没有非终结点，词与词之间直接发生依存关系，构成一个依存对，其中一个是核心词，也叫支配词，另一个叫修饰词，也叫从属词。

依存关系用一个有向弧表示，叫做依存弧。依存弧的方向为由从属词指向支配词，当然反过来也是可以的，按个人习惯统一表示即可。

例如，下面这个句子：

> 国务院总理李克强调研上海外高桥时提出，支持上海积极探索新机制。

依存句法的分析结果见下（利用哈工大 LTP）：



从分析结果中我们可以看到，句子的核心谓词为“提出”，主语是“李克强”，提出的宾语是“支持上海……”，“调研……时”是“提出”的（时间）状语，“李克强”的修饰语是“国务院总理”，“支持”的宾语是“探索新机制”。

有了上面的依存句法分析结果，我们就可以比较容易的看到，“提出者”是“李克强”，而不是“上海”或“外高桥”，即使它们都是名词，而且距离“提出”更近。

依存关系

依存句法通过分析语言单位内成分之前的依存关系解释其句法结构，主张句子中核心动词是支配其他成分的中心成分。而它本身却不受其他任何成分的支配，所有受支配成分都以某种关系从属于支配者。

在20世纪70年代，Robinson 提出依存句法中关于依存关系的四条公理，在处理中文信息的研究中，中国学者提出了依存关系的第五条公理，分别如下：

1. 一个句子中只有一个成分是独立的；
2. 句子的其他成分都从属于某一成分；
3. 任何一个成分都不能依存于两个或两个以上的成分；
4. 如果成分 A 直接从属成分 B，而成分 C 在句子中位于 A 和 B 之间，那么，成分 C 或者从属于 A，或者从属于 B，或者从属于 A 和 B 之间的某一成分；
5. 中心成分左右两边的其他成分相互不发生关系。

句子成分之间相互支配与被支配、依存与被依存的现象，普遍存在于汉语的词汇（合成语）、短语、单句、段落、篇章等能够独立运用和表达的语言之中，这一特点体现了依存关系的普遍性。依存句法分析可以反映出句子各成分之间的语义修饰关系，它可以获得长距离的搭配信息，并与句子成分的物理位置无关。

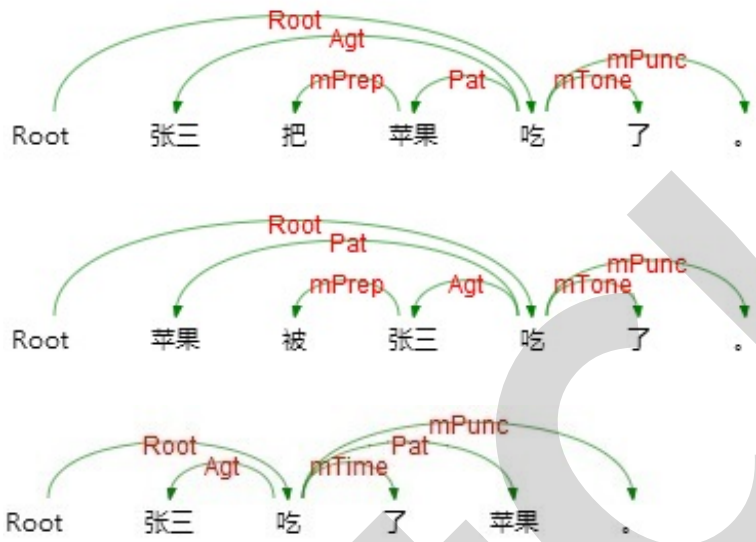
依存句法分析标注关系（共14种）及含义如下表所示：

关系类型	Tag	Description	Example
主谓关系	SBV	subject-verb	我送她一束花 (我 <-- 送)
动宾关系	VOB	直接宾语，verb-object	我送她一束花 (送 --> 花)
间宾关系	IOB	间接宾语，indirect-object	我送她一束花 (送 --> 她)
前置宾语	FOB	前置宾语，fronting-object	他什么书都读 (书 <-- 读)
兼语	DBL	double	他请我吃饭 (请 --> 我)
定中关系	ATT	attribute	红苹果 (红 <-- 苹果)
状中结构	ADV	adverbial	非常美丽 (非常 <-- 美丽)
动补结构	CMP	complement	做完了作业 (做 --> 完)
并列关系	COO	coordinate	大山和大海 (大山 --> 大海)
介宾关系	POB	preposition-object	在贸易区内 (在 --> 内)
左附加关系	LAD	left adjunct	大山和大海 (和 <-- 大海)
右附加关系	RAD	right adjunct	孩子们 (孩子 --> 们)
独立结构	IS	independent structure	两个单句在结构上彼此独立
核心关系	HED	head	指整个句子的核心

语义依存分析

语义依存分析（Semantic Dependency Parsing， SDP），分析句子各个语言单位之间的语义关联，并将语义关联以依存结构呈现。使用语义依存刻画句子语义，好处在于不需要去抽象词汇本身，而是通过词汇所承受的语义框架来描述该词汇，而论元的数目相对词汇来说数量总是少了很多。

语义依存分析目标是跨越句子表层句法结构的束缚，直接获取深层的语义信息。例如以下三个句子，用不同的表达方式表达了同一个语义信息，即张三实施了一个吃的动作，吃的动作是对苹果实施的。



语义依存分析不受句法结构的影响，将具有直接语义关联的语言单元直接连接依存弧并标记上相应的语义关系。这也是语义依存分析与依存句法分析的重要区别。

语义依存关系分为三类，分别是主要语义角色，每一种语义角色对应存在一个嵌套关系和反关系；事件关系，描述两个事件间的关系；语义依附标记，标记说话者语气等依附性信息。

关系类型	Tag	Description	Example
施事关系	Agt	Agent	我送她一束花 (我 <-- 送)
当事关系	Exp	Experiencer	我跑得快 (跑 --> 我)
感事关系	Aft	Affection	我思念家乡 (思念 --> 我)
领事关系	Poss	Possessor	他有一本好读 (他 <-- 有)
受事关系	Pat	Patient	他打了小明 (打 --> 小明)
客事关系	Cont	Content	他听到鞭炮声 (听 --> 鞭炮声)
成事关系	Prod	Product	他写了本小说 (写 --> 小说)
源事关系	Orig	Origin	我军缴获敌人四辆坦克 (缴获 --> 坦克)

涉事关系	Datv	Dative	他告诉我个秘密 (告诉 --> 我)
比较角色	Comp	Comitative	他成绩比我好 (他 --> 我)
属事角色	Belg	Belongings	老赵有俩女儿 (老赵 <-- 有)
类事角色	Clas	Classification	他是中学生 (是 --> 中学生)
依据角色	Accd	According	本庭依法宣判 (依法 <-- 宣判)
缘故角色	Reas	Reason	他在愁女儿婚事 (愁 --> 婚事)
意图角色	Int	Intention	为了金牌他拼命努力 (金牌 <-- 努力)
结局角色	Cons	Consequence	他跑了满头大汗 (跑 --> 满头大汗)
方式角色	Mann	Manner	球慢慢滚进空门 (慢慢 <-- 滚)
工具角色	Tool	Tool	她用砂锅熬粥 (砂锅 <-- 熬粥)
材料角色	Malt	Material	她用小米熬粥 (小米 <-- 熬粥)
时间角色	Time	Time	唐朝有个李白 (唐朝 <-- 有)
空间角色	Loc	Location	这房子朝南 (朝 --> 南)
历程角色	Proc	Process	火车正在过长江大桥 (过 --> 大桥)
趋向角色	Dir	Direction	部队奔向南方 (奔 --> 南)
范围角色	Sco	Scope	产品应该比质量 (比 --> 质量)
数量角色	Quan	Quantity	一年有365天 (有 --> 天)
数量数组	Qp	Quantity-phrase	三本书 (三 --> 本)
频率角色	Freq	Frequency	他每天看书 (每天 <-- 看)
顺序角色	Seq	Sequence	他跑第一 (跑 --> 第一)
描写角色	Desc(Feat)	Description	他长得胖 (长 --> 胖)
宿主角色	Host	Host	住房面积 (住房 <-- 面积)
名字修饰角色	Nmod	Name-modifier	果戈里大街 (果戈里 <-- 大街)
时间修饰角色	Tmod	Time-modifier	星期一上午 (星期一 <-- 上午)
反角色	r + main role		打篮球的小姑娘 (打篮球 <-- 姑娘)
嵌套角色	d + main role		爷爷看见孙子在跑 (看见 --> 跑)
并列关系	eCoo	event Coordination	我喜欢唱歌和跳舞 (唱歌 --> 跳舞)
选择关系	eSelt	event Selection	您是喝茶还是喝咖啡 (茶 --> 咖啡)
等同关系	eEqu	event Equivalent	他们三个人一起走 (他们 --> 三个人)
先行关系	ePrec	event Precedent	首先，先
顺承关系	eSucc	event Successor	随后，然后

递进关系	eProg	event Progression	况且，并且
转折关系	eAdvt	event adversative	却，然而
原因关系	eCau	event Cause	因为，既然
结果关系	eResu	event Result	因此，以致
推论关系	eInf	event Inference	才，则
条件关系	eCond	event Condition	只要，除非
假设关系	eSupp	event Supposition	如果，要是
让步关系	eConc	event Concession	纵使，哪怕
手段关系	eMetd	event Method	
目的关系	ePurp	event Purpose	为了，以便
割舍关系	eAban	event Abandonment	与其，也不
选取关系	ePref	event Preference	不如，宁愿
总括关系	eSum	event Summary	总而言之
分叙关系	eRect	event Recount	例如，比方说
连词标记	mConj	Recount Marker	和，或
的字标记	mAux	Auxiliary	的，地，得
介词标记	mPrep	Preposition	把，被
语气标记	mTone	Tone	吗，呢
时间标记	mTime	Time	才，曾经
范围标记	mRang	Range	都，到处
程度标记	mDegr	Degree	很，稍微
频率标记	mFreq	Frequency Marker	再，常常
趋向标记	mDir	Direction Marker	上去，下来
插入语标记	mPars	Parenthesis Marker	总的来说，众所周知
否定标记	mNeg	Negation Marker	不，没，未
情态标记	mMod	Modal Marker	幸亏，会，能
标点标记	mPunc	Punctuation Marker	，。！
重复标记	mPept	Repetition Marker	走啊走 (走 --> 走)
多数标记	mMaj	Majority Marker	们，等
实词虚化标记	mVain	Vain Marker	
离合标记	mSepa	Seperation Marker	吃了个饭 (吃 --> 饭) 洗了个澡 (洗 --> 澡)
根节点	Root	Root	全句核心节点

Pyhanlp 实战依存句法

最后，我们通过 Pyhanlp 库实现依存句法的实战练习。这个过程中，我们选用 Dependency Viewer 工具进行可视化展示。可视化时， txt 文档需要采用 UTF-8 编码。

首先，引入包，然后可直接进行分析：

```
from pyhanlp import *
sentence = "徐先生还具体帮助他确定了把画雄鹰、松鼠和麻雀作为主攻目标。"
print(HanLP.parseDependency(sentence))
```

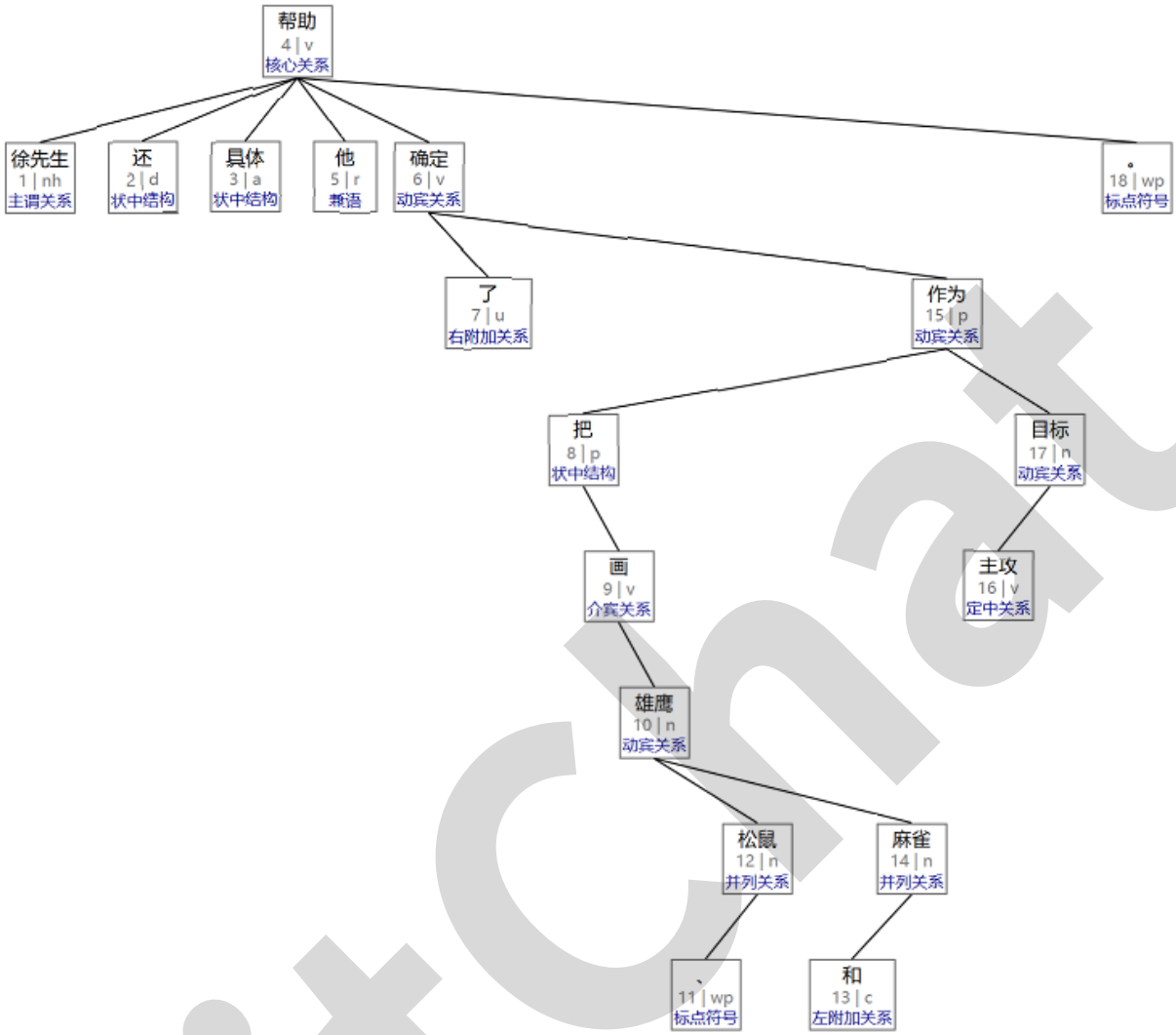
得到的结果：

```
1	徐先生	徐先生	nh	nr	_	4	主谓关系	_	_
2	还	还	d	d	_	4	状中结构	_	_
3	具体	具体	a	a	_	4	状中结构	_	_
4	帮助	帮助	v	v	_	0	核心关系	_	_
5	他	他	r	rr	_	4	兼语	_	_
6	确定	确定	v	v	_	4	动宾关系	_	_
7	了	了	u	ule	_	6	右附加关系	_	_
8	把	把	p	pba	_	15	状中结构	_	_
9	画	画	v	v	_	8	介宾关系	_	_
10	雄鹰	雄鹰	n	n	_	9	动宾关系	_	_
11	、	、	wp	w	_	12	标点符号	_	_
12	松鼠	松鼠	n	n	_	10	并列关系	_	_
13	和	和	c	cc	_	14	左附加关系	_	_
14	麻雀	麻雀	n	n	_	10	并列关系	_	_
15	作为	作为	p	p	_	6	动宾关系	_	_
16	主攻	主攻	v	vn	_	17	定中关系	_	_
17	目标	目标	n	n	_	15	动宾关系	_	_
18	。	。	wp	w	_	4	标点符号	_	_
```

然后，我们将结果保存在 txt 文件中：

```
f = open("D://result.txt",'a+')
print((HanLP.parseDependency(sentence )),file = f)
```

最后，通过 Dependency Viewer 工具进行可视化，如果出现乱码，记得把 txt 文档保存为 UTF-8 式即可，得到的可视化结果如下图所示：



总结

HanLP可以实现句子的依存关系处理，
接着可以使用CRF，确定句子的具体关系

本文，首先为大家介绍了语法体系，以及如何根据语法体系确定一个句子的语法树，为后面的句法分析奠定基础。

接着，介绍了依存句法，它的目的是通过分析语言单位内成分之间的依存关系揭示其句法结构，随之讲解了依存句法中的五大依存关系。

最后，进一步介绍了区别于依存句法的语义依存，其目的是分析句子各个语言单位之间的语义关联，并将语义关联以依存结构呈现。

文章结尾，通过 Pyhanlp 实战以及可视化，带大家进一步加深对中文依存句法的了解。

参考资料以及推荐阅读：

1. 中文依存句法分析概述及应用
2. LTP 依存分析模块所使用的依存关系标记含义
3. 依存句法解析
4. 依存分析：中文依存句法分析简介
5. 依存句法分析与语义依存分析的区别
6. pyltp:the python extension for LTP
7. Dependency Viewer

第17课：基于 CRF 的中文句法依存分析模型实现

句法分析是自然语言处理中的关键技术之一，其基本任务是确定句子的句法结构或者句子中词汇之间的依存关系。主要包括两方面的内容，一是确定语言的语法体系，即对语言中合法句子的语法结构给予形式化的定义；另一方面是句法分析技术，即根据给定的语法体系，自动推导出句子的句法结构，分析句子所包含的句法单位和这些句法单位之间的关系。

依存关系本身是一个树结构，每一个词看成一个节点，依存关系就是一条有向边。本文主要通过清华大学的句法标注语料库，来实现基于 CRF 的中文句法依存分析模型。

清华大学句法标注语料库

清华大学的句法标注语料，包括训练集（train.conll）和开发集合文件（dev.conll）。训练集大小 5.41M，共185541条数据。测试集大小为 578kb，共19302条数据。

语料本身格式如下图所示：

```
1  1  坚决  坚决  a  ad  _  2  方式
2  2  惩治  惩治  v  v  _  0  核心成分
3  3  贪污  贪污  v  v  _  7  限定
4  4  贿赂  贿赂  n  n  _  3  连接依存
5  5  等  等  u  udeng  _  3  连接依存
6  6  经济  经济  n  n  _  7  限定
7  7  犯罪  犯罪  v  vn  _  2  受事
```

通过上图，我们可以看出，每行语料包括有8个标签，分别是 ID、FROM、lEMMA、CPOSTAG、POSTAG、FEATS、HEAD、DEPREL。详细介绍如下图：

```
1   ID  当前词在句子中的序号，1开始.
2   FORM    当前词语或标点
3   LEMMA   当前词语（或标点）的原型或词干，在中文中，此列与FORM相同
4   CPOSTAG 当前词语的词性（粗粒度）
5   POSTAG  当前词语的词性（细粒度）
6   FEATS   句法特征，在本次评测中，此列未被使用，全部以下划线代替。
7   HEAD    当前词语的中心词
8   DEPREL  当前词语与中心词的依存关系
```

模型的实现

通过上面对句法依存关键技术的定义，我们明白了，句法依存的基本任务是确定句子的句法结构或者句子中词汇之间的依存关系。同时，我们也对此次模型实现的语料有了基本了解。

有了这些基础内容，我们便可以开始着手开发了。

本模型的实现过程，我们将主要分为训练集和测试集数据预处理、语料特征生成、模型训练及预测三大部分来实现，最终将通过模型预测得到正确的预测结果。

本次实战演练，我们选择以下模型和软件：

- Sklearn_crfsuite
- Python3.6
- Jupyter Notebook

训练集和测试集数据预处理

由于上述给定的语料，在模型中，我们不能直接使用，必须先经过预处理，把上述语料格式重新组织成具有词性、方向和距离的格式。

首先，我们通过一个 Python 脚本 `get_parser_train_test_input.py`，生成所需要的训练集和测试集，执行如下命令即可：

```
1.      cat train.conll | python get_parser_train_test_input.py > train.dat
    a
2.      cat dev.conll | python get_parser_train_test_input.py > dev.data
```

上面的脚本通过 cat 命令和管道符把内容传递给脚本进行处理。这里需要注意的是，脚本需要

在 Linux 环境下执行，且语料和脚本应放在同一目录下。

`get_parser_train_test_input.py` 这一脚本的目的，就是重新组织语料，组织成可以使用 CRF 算法的格式，具有词性、方向和距离的格式。我们认为，如果词 A 依赖词 B，A 就是孩子，B 就是父亲。按照这种假设得到父亲节点的粗词性和详细词性，以及和依赖次之间的距离。

我们打开该脚本，看看它的代码，如下所示，重要的代码给出了注释。

```
    #coding=utf-8
    '''词A依赖词B，A就是孩子，B就是父亲'''
    import sys

    sentence = ["Root"]
    def do_parse(sentence):
        if len(sentence) == 1:return
        for line in sentence[1:]:
            line_arr = line.strip().split("\t")
            c_id = int(line_arr[0])
            f_id = int(line_arr[6])
            if f_id == 0:
                print("\t".join(line_arr[2:5])+"\t" + "0_Root")
                continue
            f_post,f_detail_post = sentence[f_id].strip().split("\t")[3:5] #得到父亲节点的粗词性和详细词性
            c_edge_post = f_post #默认是依赖词的粗粒度词性，但是名词除外；名词取细粒度词性
            if f_post == "n":
                c_edge_post = f_detail_post
            #计算是第几个出现这种词行
            diff = f_id - c_id #确定要走几步
            step = 1 if f_id > c_id  else -1 #确定每一步方向
            same_post_num = 0 #中间每一步统计多少个一样的词性
            cmp_idx = 4 if f_post == "n" else 3  #根据是否是名词决定取的是粗or详细词性
            for i in range(0, abs(diff)):
                idx = c_id + (i+1)*step
                if sentence[idx].strip().split("\t")[cmp_idx] == c_edge_post:
                    same_post_num += step

            print("\t".join(line_arr[2:5])+"\t" + "%d_%s"%(same_post_num, c_edge_post))
```

```
            print("")

        for line in sys.stdin:
            line = line.strip()
            line_arr = line.split("\t")
            if  line == "" or line_arr[0] == "1":
                do_parse(sentence)
                sentence = ["Root"]
            if line =="":continue
            sentence.append(line)
```

整个脚本按行读入，每行按 Tab 键分割，首先得到父亲节点的词性，然后根据词性是否是名词 n 进行判断，默认是依赖词的粗粒度词性，如果是名词取细粒度词性。

脚本处理完，数据集的格式如下：

```
坚决    a    ad   1_v
惩治    v    v    0_Root
贪污    v    v    1_v
贿赂    n    n    -1_v
等  u    udeng    -1_v
经济    n    n    1_v
犯罪    v    vn   -2_v
```

根据依存文法，决定两个词之间依存关系的主要有两个因素：方向和距离。正如上图中第四列类别标签所示，该列可以定义为以下形式：

> [+|-]dPOS

其中，`[+|-]` 表示中心词在句子中相对坐标轴的方向；POS 代表中心词具有的词性类别；d 表示与中心词词性相同的词的数量，即距离。

语料特征生成

语料特征提取，主要采用 N-gram 模型来完成。这里我们使用 3-gram 完成提取，将词性与词语两两进行匹配，分别返回特征集合和标签集合，需要注意整个语料采用的是 UTF-8 编码格式。

整个编码过程中，我们首先需要引入需要的库，然后对语料进行读文件操作。语料采用 UTF-8 编码格式，以句子为单位，按 Tab 键作分割处理，从而实现句子 3-gram 模型的特征提取。具体实现如下。

```
    import sklearn_crfsuite
    from sklearn_crfsuite import metrics
    from sklearn.externals import joblib
```

首先引入需要用到的库，如上面代码所示。其目的是使用模型 `sklearn_crfsuite .CRF`，metrics 用来进行模型性能测试，joblib 用来保存和加载训练好的模型。

接着，定义包含特征处理方法的类，命名为 CorpusProcess，类结构定义如下：

```
class CorpusProcess(object):

    def __init__(self):
        """初始化"""
        pass

    def read_corpus_from_file(self, file_path):
        """读取语料"""
        pass

    def write_corpus_to_file(self, data, file_path):
        """写语料"""
        pass

    def process_sentence(self,lines):
        """处理句子"""
        pass

    def initialize(self):
        """语料初始化"""
        pass

    def generator(self, train=True):
        """特征生成器"""
        pass

    def extract_feature(self, sentences):
        """提取特征"""
        pass
```

下面介绍下 CorpusProcess 类中各个方法的具体实现。

第1步， 实现 init 构造函数，目的初始化预处理好的语料的路径：

```
    def __init__(self):
            """初始化"""
            self.train_process_path =  dir +  "data//train.data"    #预处理之后的训练集
            self.test_process_path =  dir +  "data//dev.data"  #预处理之后的测试集
```

这里的路径可以自定义，这里的语料之前已经完成了预处理过程。

第2-3步，`read_corpus_from_file` 方法和 `write_corpus_to_file` 方法，分别定义了语料文件的读和写操作：

```
    def read_corpus_from_file(self, file_path):
        """读取语料"""
        f = open(file_path, 'r',encoding='utf-8')
        lines = f.readlines()
        f.close()
        return lines

    def write_corpus_to_file(self, data, file_path):
        """写语料"""
        f = open(file_path, 'w')
        f.write(str(data))
        f.close()
```

这一步，主要用 open 函数来实现语料文件的读和写。

第4-5步，`process_sentence` 方法和 initialize 方法，用来处理句子和初始化语料，把语料按句子结构用 list 存储起来，存储到内存中：

```
    def process_sentence(self,lines):
            """处理句子"""
            sentence = []
            for line in lines:
                if not line.strip():
                    yield sentence
                    sentence = []
```

```
                else:
                    lines = line.strip().split(u'\t')
                    result = [line for line in lines]
                    sentence.append(result)

        def initialize(self):
            """语料初始化"""
            train_lines = self.read_corpus_from_file(self.train_process_path)
            test_lines = self.read_corpus_from_file(self.test_process_path)
            self.train_sentences = [sentence for sentence in self.process_sentence(train_lines)]
            self.test_sentences = [sentence for sentence in self.process_sentence(test_lines)]
```

这一步，通过 `process_sentence` 把句子收尾的空格去掉，然后通过 initialize 函数调用上面 `read_corpus_from_file` 方法读取语料，分别加载训练集和测试集。

第6步，特征生成器，分别用来指定生成训练集或者测试集的特征集：

```
    def generator(self, train=True):
        """特征生成器"""
        if train:
            sentences = self.train_sentences
        else:
            sentences = self.test_sentences
        return self.extract_feature(sentences)
```

这一步，对训练集和测试集分别处理，如果参数 train 为 True，则表示处理训练集，如果是 False，则表示处理测试集。

第7步，特征提取，简单的进行 `3-gram` 的抽取，将词性与词语两两进行匹配，分别返回特征集合和标签集合：

```
    def extract_feature(self, sentences):
            """提取特征"""
            features, tags = [], []
            for index in range(len(sentences)):
                feature_list, tag_list = [], []
                for i in range(len(sentences[index])):
```

```
                    feature = {"w0": sentences[index][i][0],
                               "p0": sentences[index][i][1],
                               "w-1": sentences[index][i-1][0] if i != 0 else "BOS",
                               "w+1": sentences[index][i+1][0] if i != len(sentences[index])-1 else "EOS",
                               "p-1": sentences[index][i-1][1] if i != 0 else "un",
                               "p+1": sentences[index][i+1][1] if i != len(sentences[index])-1 else "un"}
                    feature["w-1:w0"] = feature["w-1"]+feature["w0"]
                    feature["w0:w+1"] = feature["w0"]+feature["w+1"]
                    feature["p-1:p0"] = feature["p-1"]+feature["p0"]
                    feature["p0:p+1"] = feature["p0"]+feature["p+1"]
                    feature["p-1:w0"] = feature["p-1"]+feature["w0"]
                    feature["w0:p+1"] = feature["w0"]+feature["p+1"]
                    feature_list.append(feature)
                    tag_list.append(sentences[index][i][-1])
                features.append(feature_list)
                tags.append(tag_list)
            return features, tags
```

经过第6步，确定处理的是训练集还是测试集之后，通过 `extract_feature` 对句子进行特征抽取，使用 3-gram 模型，得到特征集合和标签集合的对应关系。

模型训练及预测

在完成特征工程和特征提取之后，接下来，我们要进行模型训练和预测，要预定义模型需要的一些参数，并初始化模型对象，进而完成模型训练和预测，以及模型的保存与加载。

首先，我们定义模型 ModelParser 类，进行初始化参数、模型初始化，以及模型训练、预测、保存和加载，类的结构定义如下：

```
    class ModelParser(object):

        def __init__(self):
            """初始化参数"""
            pass

        def initialize_model(self):
            """模型初始化"""
            pass

```

```
        def train(self):
            """训练"""
            pass

        def predict(self, sentences):
            """模型预测"""
            pass

        def load_model(self, name='model'):
            """加载模型 """
            pass

        def save_model(self, name='model'):
            """保存模型"""
            pass
```

接下来，我们分析 ModelParser 类中方法的具体实现。

第1步，init 方法实现算法模型参数和语料预处理 CorpusProcess 类的实例化和初始化：

```
    def __init__(self):
            """初始化参数"""
            self.algorithm = "lbfgs"
            self.c1 = 0.1
            self.c2 = 0.1
            self.max_iterations = 100
            self.model_path = "model.pkl"
            self.corpus = CorpusProcess()  #初始化CorpusProcess类
            self.corpus.initialize()  #语料预处理
            self.model = None
```

这一步，init 方法初始化参数以及 CRF 模型的参数，算法选用 LBFGS，c1 和 c2 分别为0.1，最大迭代次数100次。然后定义模型保存的文件名称，以及完成对 CorpusProcess 类 的初始化。

第2-3步，`initialize_model` 方法和 train 实现模型定义和训练：

```
    def initialize_model(self):
            """模型初始化"""
            algorithm = self.algorithm
            c1 = float(self.c1)
```

```
            c2 = float(self.c2)
            max_iterations = int(self.max_iterations)
            self.model = sklearn_crfsuite.CRF(algorithm=algorithm, c1=c1, c2=c2,
max_iterations=max_iterations, all_possible_transitions=True)

        def train(self):
            """训练"""
            self.initialize_model()
            x_train, y_train = self.corpus.generator()
            self.model.fit(x_train, y_train)
            labels = list(self.model.classes_)
            x_test, y_test = self.corpus.generator(train=False)
            y_predict = self.model.predict(x_test)
            metrics.flat_f1_score(y_test, y_predict, average='weighted', labels=labels)
            sorted_labels = sorted(labels, key=lambda name: (name[1:], name[0]))
            print(metrics.flat_classification_report(y_test, y_predict, labels=sorted_labels, digits=3))
            self.save_model()
```

这一步，`initialize_model` 方法实现了 `sklearn_crfsuite.CRF` 模型的初始化。然后在 train 方法中，先通过 fit 方法训练模型，再通过 `metrics.flat_f1_score` 对测试集进行 F1 性能测试，最后将模型保存。

第4-6步，分别实现模型预测、保存和加载方法，具体代码大家可以访问 Github。

最后，实例化类，并进行模型训练：

```
    model = ModelParser()
    model.train()
```

对模型进行预测，预测数据输入格式为三维，表示完整的一句话：

```
[[['坚决', 'a', 'ad', '1_v'],
  ['惩治', 'v', 'v', '0_Root'],
  ['贪污', 'v', 'v', '1_v'],
```

```
['贿赂', 'n', 'n', '-1_v'],
['等', 'u', 'udeng', '-1_v'],
['经济', 'n', 'n', '1_v'],
['犯罪', 'v', 'vn', '-2_v']]]
```

模型预测的结果如下图所示：

```
In [81]: sen =[['坚决', 'a', 'ad', '1_v'],
           ['惩治', 'v', 'v', '0_Root'],
           ['贪污', 'v', 'v', '1_v'],
           ['贿赂', 'n', 'n', '-1_v'],
           ['等', 'u', 'udeng', '-1_v'],
           ['经济', 'n', 'n', '1_v'],
           ['犯罪', 'v', 'vn', '-2_v']]]

In [82]: model.predict(sen)
Out[82]: [['1_v', '0_Root', '2_n', '-1_v', '-1_v', '1_v', '-2_v']]
```

预测的结果，和原始语料预处理得到的标签格式保持一致。

语料和代码下载，请访问：Github。

总结

本文通过清华大学的句法标注语料库，实现了基于 CRF 的中文句法依存分析模型。借此实例，相信大家对句法依存已有了一个完整客观的认识。

参考文献及推荐阅读

1. 使用 CoNLL 2002 数据的英文句法依存分析
2. CRF++ 依存句法分析
3. 依存分析：基于序列标注的中文依存句法分析模型实现

第18课：模型部署上线的几种服务发布方式

在前面所有的模型训练和预测中，我们训练好的模型都是直接通过控制台或者 Jupyter Notebook 来进行预测和交互的，在一个系统或者项目中使用这种方式显然不可能，那在 Web 应用中如何使用我们训练好的模型呢？本文将通过以下四个方面对该问题进行讲解：

1. 微服务架构简介；
2. 模型的持久化与加载方式；
3. Flask 和 Bottle 微服务框架；
4. Tensorflow Serving 模型部署和服务。

微服务架构简介

微服务是指开发一个单个小型的但有业务功能的服务，每个服务都有自己的处理和轻量通讯机制，可以部署在单个或多个服务器上。微服务也指一种松耦合的、有一定的有界上下文的面向服务架构。也就是说，如果每个服务都要同时修改，那么它们就不是微服务，因为它们紧耦合在一起；如果你需要掌握一个服务太多的上下文场景使用条件，那么它就是一个有上下文边界的服务，这个定义来自 DDD 领域驱动设计。

相对于单体架构和 SOA，它的主要特点是组件化、松耦合、自治、去中心化，体现在以下几个方面：

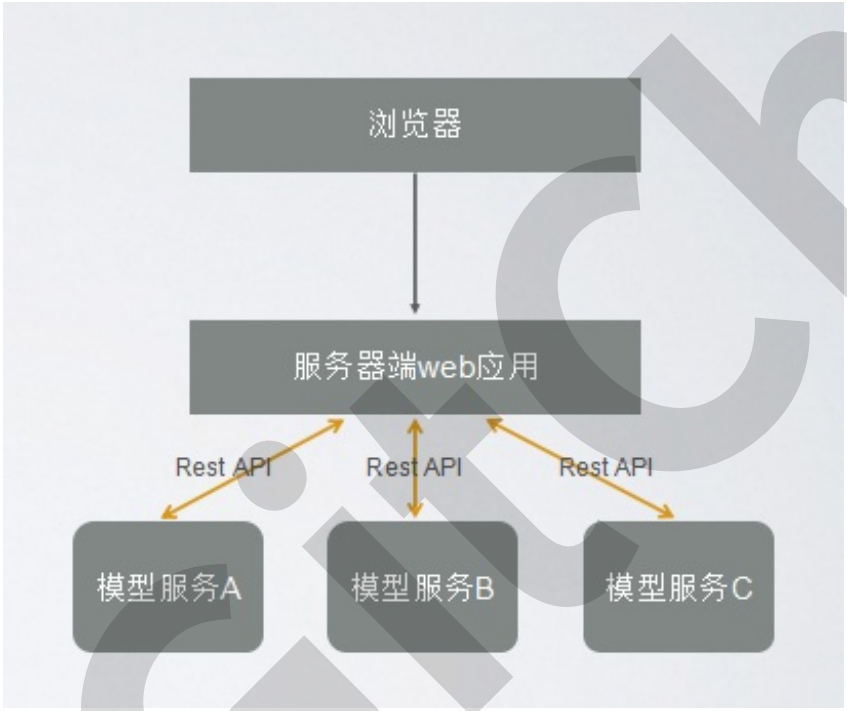
1. 一组小的服务：服务粒度要小，而每个服务是针对一个单一职责的业务能力的封装，专注做好一件事情；
2. 独立部署运行和扩展：每个服务能够独立被部署并运行在一个进程内。这种运行和部署方式能够赋予系统灵活的代码组织方式和发布节奏，使得快速交付和应对变化成为可能。
3. 独立开发和演化：技术选型灵活，不受遗留系统技术约束。合适的业务问题选择合适的技术可以独立演化。服务与服务之间采取与语言无关的 API 进行集成。相对单体架构，微服务架构是更面向业务创新的一种架构模式。
4. 独立团队和自治：团队对服务的整个生命周期负责，工作在独立的上下文中，自己决策自己治理，而不需要统一的指挥中心。团队和团队之间通过松散的社区部落进行衔接。

由此，我们可以看到整个微服务的思想，与我们现在面对信息爆炸、知识爆炸做事情的思路是相通的：通过解耦我们所做的事情，分而治之以减少不必要的损耗，使得整个复杂的系统和组织能够快速地应对变化。

我们为什么采用微服务呢？

> “让我们的系统尽可能快地响应变化”
>
> ——Rebecca Parson

下面是一个简单的微服务模型架构设计：



模型的持久化与加载方式

开发过 J2EE 应用的人应该对持久化的概念很清楚。通俗得讲，就是临时数据（比如内存中的数据，是不能永久保存的）持久化为持久数据（比如持久化至数据库中，能够长久保存）。

那我们训练好的模型一般都是存储在内存中，这个时候就需要用到持久化方式，在 Python 中，常用的模型持久化方式有三种，并且都是以文件的方式持久化。

1.JSON（JavaScript Object Notation）格式。

JSON 是一种轻量级的数据交换格式，易于人们阅读和编写。使用 JSON 函数需要导入 JSON 库：

```
1.  import json
```

它拥有两个格式处理函数：

- json.dumps：将 Python 对象编码成 JSON 字符串；
- json.loads：将已编码的 JSON 字符串解码为 Python 对象。

下面看一个例子。

首先我们创建一个 List 对象 data，然后把 data 编码成 JSON 字符串保存在 data.json 文件中，之后再读取 data.json 文件中的字符串解码成 Python 对象，代码如下：

```
In [37]: import json

In [38]: data = [ { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4, 'e' : 5 } ]
         print(type(data))   #data是一个list对象
         data_dumps = json.dumps(data)
         print(type(data_dumps))  #编码成字符串
         f = open("data.json", 'w')
         f.write(data_dumps)
         f.close()

         <class 'list'>
         <class 'str'>

In [39]: f = open("data.json", 'r')
         data =f.readlines()
         print(type(data[0]))   #data是一个字符串
         json_loads = json.loads(data[0])
         print(type(json_loads))  #字符串解码成一个list对象
         print(json_loads)

         <class 'str'>
         <class 'list'>
         [{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}]
```

2. pickle 模块

pickle 提供了一个简单的持久化功能。可以将对象以文件的形式存放在磁盘上。pickle 模块只能在 Python 中使用，Python 中几乎所有的数据类型（列表、字典、集合、类等）都可以用 pickle 来序列化。pickle 序列化后的数据，可读性差，人一般无法识别。

使用的时候需要引入库：

```
import pickle
```

它有以下两个方法：

- pickle.dump(obj, file[, protocol])：序列化对象，并将结果数据流写入到文件对象中。参数 protocol 是序列化模式，默认值为0，表示以文本的形式序列化。protocol 的值还可以是1或2，表示以二进制的形式序列化。
- pickle.load(file)：反序列化对象。将文件中的数据解析为一个 Python 对象。

我们继续延用上面的例子。实现的不同点在于，这次文件打开时用了 `with...as...` 语法，使用 pickle 保存结果，文件保存为 data.pkl，代码如下。

```
In [46]: import pickle

In [47]: data = [ { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4, 'e' : 5 } ]
         with open('data.pkl', 'wb') as f:
             pickle.dump(data, f)

In [48]: with open('data.pkl', 'rb') as f:
             model = pickle.load(f)
```

3. sklearn 中的 joblib 模块。

使用 joblib，首先需要引入包：

```
from sklearn.externals import joblib
```

使用方法如下，基本和 JSON、pickle一样，这里不再详细讲解。第17课中，进行模型保存时使用的就是这种方式，可以看代码，回顾一下。

```
joblib.dump(model, model_path)   #模型保存
joblib.load(model_path)   #模型加载
```

Flask 和 Bottle 微服务框架

模型的微服务器

通过上面，我们对微服务和 Python 中三种模型持久化和加载方式有了基本了解。下面我们看看，Python 中如何把模型发布成一个微服务的。

这里给出两个微服务框架 Bottle 和 Flask。

Bottle 是一个非常小巧但高效的微型 Python Web 框架，它被设计为仅仅只有一个文件的 Python 模块，并且除 Python 标准库外，它不依赖于任何第三方模块。

Bottle 本身主要包含以下四个模块，依靠它们便可快速开发微 Web 服务：

- 路由（Routing）：将请求映射到函数，可以创建十分优雅的 URL；
- 模板（Templates）：可以快速构建 Python 内置模板引擎，同时还支持 Mako、Jinja2、Cheetah 等第三方模板引擎；
- 工具集（Utilites）：用于快速读取 form 数据，上传文件，访问 Cookies，Headers 或者其它 HTTP 相关的 metadata；
- 服务器（Server）：内置 HTTP 开发服务器，并且支持 paste、fapws3、 bjoern、Google App Engine、Cherrypy 或者其它任何 WSGI HTTP 服务器。

Flask 也是一个 Python 编写的 Web 微框架，可以让我们使用 Python 语言快速实现一个网站或 Web 服务。并使用方式和 Bottle 相似，Flask 依赖 Jinja2 模板和 Werkzeug WSGI 服务。Werkzeug 本质是 Socket 服务端，其用于接收 HTTP 请求并对请求进行预处理，然后触发 Flask 框架，开发人员基于 Flask 框架提供的功能对请求进行相应的处理，并返回给用户，如果返回给用户的内容比较复杂时，需要借助 Jinja2 模板来实现对模板的处理，即将模板和数据进行渲染，将渲染后的字符串返回给用户浏览器。

Bottle 和 Flask 在使用上相似，而且 Flask 的文档资料更全，发布的服务更稳定，因此下面重点以 Flask 为例，来说明模型的微服务发布过程。

如果大家想进一步了解这两个框架，可以参考说明文档。

1.安装。

对 Bottle 和 Flask 进行安装，分别执行如下命令即可安装成功：

```
pip install bottle
pip install Flask
```

安装好之后，分别进入需要的包就可以写微服务程序了。这两个框架在使用时，用法、语法结构都差不多，网上 Flask 的中文资料相对多一些，所以这里用 Flask 来举例。

2. 第一个最小的 Flask 应用。

第一个最小的 Flask 应用看起来会是这样:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

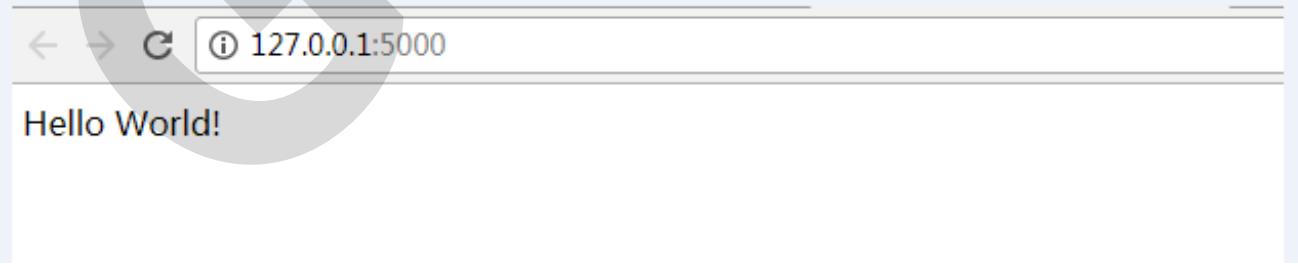
把它保存为 hello.py（或是类似的），然后用 Python 解释器来运行：

```
python hello.py
```

或者直接在 Jupyter Notebook 里面执行，都没有问题。服务启动将在控制台打印如下消息：

> Running on http://127.0.0.1:5000/

意思就是，可以通过 localhost 和 5000 端口，在浏览器访问：



这时我们就得到了服务在浏览器上的返回结果，于是也成功构建了与浏览器交互的服务。

如果要修改服务对应的 IP 地址和端口怎么办？只需要修改这行代码，即可修改 IP 地址和端口：

```
app.run(host='192.168.31.19',port=8088)
```

3. Flask 发布一个预测模型。

首先，我们这里使用第17课保存的模型“model.pkl”。如果不使用浏览器，常规的控制台交互，我们这样就可以实现：

```
    from sklearn.externals import joblib
    model_path = "D://达人课//中文自然语言处理入门实战课程//ch18//model.pkl"
    model = joblib.load(model_path)
    sen =[['坚决', 'a', 'ad', '1_v'],
                ['惩治', 'v', 'v', '0_Root'],
                ['贪污', 'v', 'v', '1_v'],
                ['贿赂', 'n', 'n', '-1_v'],
                ['等', 'u', 'udeng', '-1_v'],
                ['经济', 'n', 'n', '1_v'],
                ['犯罪', 'v', 'vn', '-2_v']]]
    print(model.predict(sen))
```

如果你现在有个需求，要求你的模型和浏览器进行交互，那 Flask 就可以实现。

在第一个最小的 Flask 应用基础上，我们增加模型预测接口，这里注意：**启动之前把 IP 地址修改为自己本机的地址或者服务器工作站所在的 IP 地址。**

完整的代码如下，首先在启动之前先把模型预加载到内存中，然后重新定义 predict 函数，接受一个参数 sen：

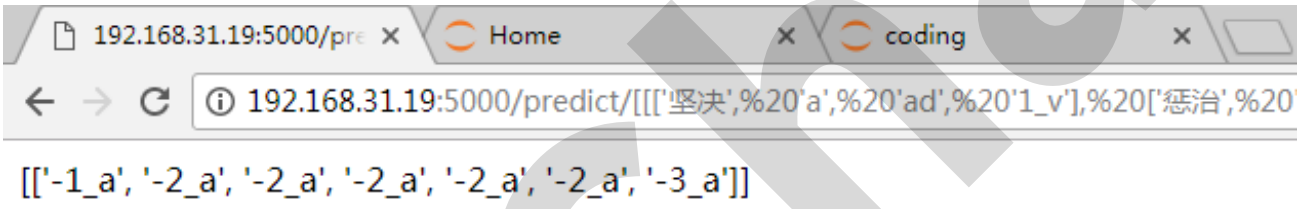
```
    from sklearn.externals import joblib
    from flask import Flask,request
    app = Flask(__name__)

    @app.route('/')
    def hello_world():
        return 'Hello World!'

    @app.route('/predict/<sen>')
    def predict(sen):
        result = model.predict(sen)
        return str(result)

```

```
14.     if __name__ == '__main__':
15.         model_path = "D://ch18//model.pkl"
16.         model = joblib.load(model_path)
17.         app.run(host='192.168.31.19')
```

启动 Flask 服务之后，在浏览器地址中输入：

> http://192.168.31.19:5000/predict/[[['坚决', 'a', 'ad', '1_v'], ['惩治', 'v', 'v', '0_Root'], ['贪污', 'v', 'v', '1_v'], ['贿赂', 'n', 'n', '-1_v'], ['等', 'u', 'udeng', '-1_v'], ['经济', 'n', 'n', '1_v'], ['犯罪', 'v', 'vn', '-2_v']]]

得到预测结果，这样就完成了微服务的发布，并实现了模型和前端浏览器的交互。



Tensorflow Serving 模型部署和服务

TensorFlow Serving 是一个用于机器学习模型 Serving 的高性能开源库。它可以将训练好的机器学习模型部署到线上，使用 gRPC 作为接口接受外部调用。更加让人眼前一亮的是，它支持模型热更新与自动模型版本管理。这意味着一旦部署 TensorFlow Serving 后，你再也不需要为线上服务操心，只需要关心你的线下模型训练。

同样，TensorFlow Serving 可以将模型部署在移动端，如安卓或者 iOS 系统的 App 应用上。关于 Tensorflow Serving 模型部署和服务，这里不在列举示例，直接参考文末的推荐阅读。

总结

本节对微服务架构做了简单介绍，并介绍了三种机器学习模型持久化和加载的方式，接着介绍了 Python 的两个轻量级微服务框架 Bottle 和 Flask。随后，我们通过 Flask 制作了一个简单

的微服务预测接口，实现模型的预测和浏览器交互功能，最后简单介绍了 TensorFlow Servin 模型的部署和服务功能。

学完上述内容，读者可轻易实现自己训练的模型和 Web 应用的结合，提供微服务接口，实现模型上线应用。

参考文献以及推荐阅读

1. Bottle 文档
2. Flask 文档
3. 面向机器智能的 TensorFlow 实践：产品环境中模型的部署
4. Tensorflow Serving 服务部署与访问（Python + Java）

第19课：知识挖掘与知识图谱概述

搜索技术日新月异，如今它不再是搜索框中输入几个单词那么简单了。不仅输入方式多样化，并且还要在非常短的时间内给出一个精准而又全面的答案。目前，谷歌给出的解决方案就是——知识图谱（Knowledge Graph）。

知识图谱能做什么？

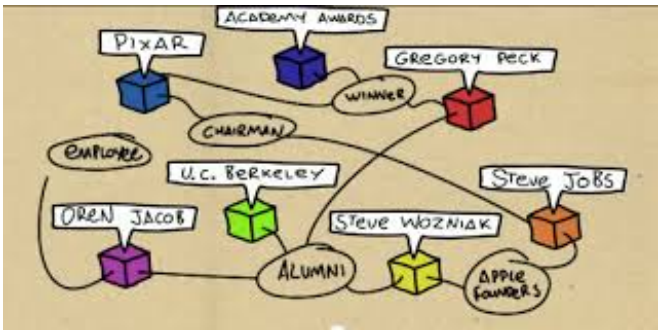
知识图谱想做的，就是在不同数据（来自现实世界）之间建立联系，从而带给我们更有意义的搜索结果。

比如，在上图中，用 Google 搜索自然语言处理，右侧会显示研究领域和相关概念。点击这些知识点，又可以深入了解；再比如，搜索一个人名时，右侧会给出此人的生平、背景、居住位置、作品等信息。

这就是知识图谱，它不再是单一的信息，而是一个多元的信息网络。

知识图谱的源头

知识图谱的雏形好几年前就已出现，一家名为 Metaweb 的小公司，将现实世界中实体（人或事）的各种数据信息存储在系统中，并在数据之间建立起联系，从而发展出有别于传统关键词搜索的技术。



谷歌认为这一系统很有发展潜力，于2010年收购了 Metaweb。那时 Metawab 已经存储了1200万个节点（Reference Point，相当于一个词条或者一个页面），谷歌收购后的两年中，大大加速这一进程，现已有超过5.7亿个节点并在它们之间建了180亿个有效连接（这可是一个相当大的数字，维基百科英文版也才有大约400万个节点）。

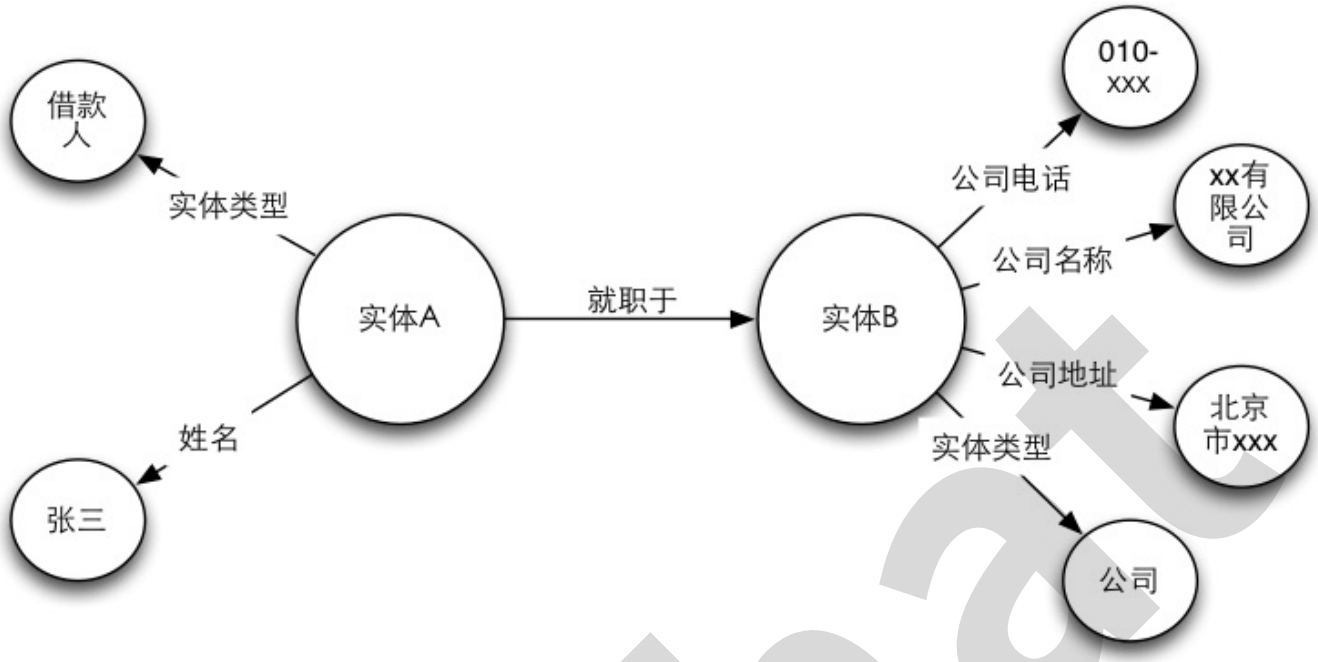
知识图谱的通用表示方法

本质上，知识图谱是一种揭示实体之间关系的语义网络 ，可以对现实世界的事物及其相互关系进行形式化地描述 。现在的知识图谱已被用来泛指各种大规模的知识库 。

三元组是知识图谱的一种通用表示方式，即 $G = (E, R, S)$，其中
$E = e_1, e_2, \dots, e_{|E|}$ 是知识库中的实体集合，共包含 $|E|$ 种不同实体，
$R = r_1, r_2, \dots, r_{|E|}$ 是知识库中的关系集合，共包含 $|R|$ 种不同关系，
$S \subseteq E \times R \times E$ 代表知识库中的三元组集合。

三元组的基本形式主要包括实体 A、关系、实体 B 和概念、属性、属性值等，实体是知识图谱中的最基本元素，不同的实体间存在不同的关系。概念主要指集合、类别、对象类型、事物的种类，例如人物、地理等；属性主要指对象可能具有的属性、特征、特性、特点以及参数，例如国籍、生日等；属性值主要指对象指定属性的值，例如中国、1988—09—08等。每个实体（概念的外延）可用一个全局唯一确定的 ID 来标识，每个属性—属性值对可用来刻画实体的内在特性，而关系可用来连接两个实体，刻画它们之间的关联。

如下图是实体 A 与实体 B 组成的一个简单三元组形式。



知识图谱的架构

知识图谱的架构主要包括自身的逻辑结构以及体系架构，分别说明如下。

1. 知识图谱的逻辑结构。

知识图谱在逻辑上可分为模式层与数据层两个层次，数据层主要是由一系列的事实组成，而知识将以事实为单位进行存储。如果用（实体 A，关系，实体 B）、（实体、属性，属性值）这样的三元组来表达事实，可选择图数据库作为存储介质，例如开源的 Neo4j、Twitter 的 FlockDB、Sones 的 GraphDB 等。模式层构建在数据层之上，主要是通过本体库来规范数据层的一系列事实表达。本体是结构化知识库的概念模板，通过本体库而形成的知识库不仅层次结构较强，并且冗余程度较小。

2. 知识图谱的体系架构。

知识图谱的体系架构是指其构建模式结构，如图下图所示。

知识图谱主要有自顶向下与自底向上两种构建方式。自顶向下指的是先为知识图谱定义好本体与数据模式，再将实体加入到知识库。该构建方式需要利用一些现有的结构化知识库作为其基础知识库，例如 Freebase 项目就是采用这种方式，它的绝大部分数据是从维基百科中得到的。自底向上指的是从一些开放链接数据中提取出实体，选择其中置信度较高的加入到知识库，再构建顶层的本体模式。目前，大多数知识图谱都采用自底向上的方式进行构建，其中最典型就是 Google 的 Knowledge Vault。

知识图谱的关键技术

大规模知识库的构建与应用需要多种智能信息处理技术的支持。这就涉及到当下异常火爆的人工智能中的自然语言处理（NLP）技术。

所谓自然语言，就是我们平时所说的话（包括语音或文字），但这些话计算机如何能“理解”？过程很复杂，下面是其中的几个关键步骤。

1. 知识抽取。

知识抽取技术，可以从一些公开的半结构化、非结构化的数据中提取出实体、关系、属性等知识要素。

知识抽取主要包含实体抽取、关系抽取、属性抽取等，涉及到的 NLP 技术有命名实体识别、句法依存、实体关系识别等。

2. 知识表示。

知识表示形成的综合向量对知识库的构建、推理、融合以及应用均具有重要的意义。

基于三元组的知识表示形式受到了人们广泛的认可，但是其在计算效率、数据稀疏性等方面却面临着诸多问题。近年来，以深度学习为代表的表示学习技术取得了重要的进展，可以将实体的语义信息表示为稠密低维实值向量，进而在低维空间中高效计算实体、关系及其之间的复杂语义关联。

知识表示学习主要包含的 NLP 技术有语义相似度计算、复杂关系模型，知识代表模型如距离模型、双线性模型、神经张量模型、矩阵分解模型、翻译模型等。

3.知识融合。

由于知识图谱中的知识来源广泛，存在知识质量良莠不齐、来自不同数据源的知识重复、知识间的关联不够明确等问题，所以必须要进行知识的融合。知识融合是高层次的知识组织，使来自不同知识源的知识在同一框架规范下进行异构数据整合、消歧、加工、推理验证、更新等步骤，达到数据、信息、方法、经验以及人的思想的融合，形成高质量的知识库。

在知识融合过程中，实体对齐、知识加工是两个重要的过程。

4.知识推理。

知识推理则是在已有的知识库基础上进一步挖掘隐含的知识，从而丰富、扩展知识库。在推理的过程中，往往需要关联规则的支持。由于实体、实体属性以及关系的多样性，人们很难穷举

所有的推理规则，一些较为复杂的推理规则往往是手动总结的。对于推理规则的挖掘，主要还是依赖于实体以及关系间的丰富情况。知识推理的对象可以是实体、实体的属性、实体间的关系、本体库中概念的层次结构等。

知识推理方法主要可分为基于逻辑的推理与基于图的推理两种类别。

大规模开放知识库

互联网的发展为知识工程提供了新的机遇。从一定程度上看，是互联网的出现帮助突破了传统知识工程在知识获取方面的瓶颈。从1998年 Tim Berners Lee 提出语义网至今，涌现出大量以互联网资源为基础的新一代知识库。这类知识库的构建方法可以分为三类：互联网众包、专家协作和互联网挖掘，如下图所示：

知识图谱库名称	机构	特点、构建手段	应用产品
FreeBase	MetaWeb(2010年被谷歌收购)	• 实体、语义类、属性、关系； • 自动+人工：部分数据从维基百科等数据源抽取而得到；另一部分数据来自人工协同编辑 • https://developers.google.com/freebase/	Google Search Engine，Google Now
Knowledge Vault（谷歌知识图谱）	Google	• 实体、语义类、属性、关系； • 超大规模数据库；源自维基百科、Freebase、《世界各国纪实年鉴》 • https://research.google.com/pubs/pub45634	Google Search Engine，Google Now
DBpedia	莱比锡大学、柏林自由大学、OpenLink Software	• 实体、语义类、属性、关系 • 从维基百科抽取 •	DBPedia
维基数据(Wikidata)	维基媒体基金会(Wikimedia Foundation)	• 实体、语义类、属性、关系,与维基百科紧密结合 • 人工（协同编辑）	Wikipedia
Wolfram Alpha	沃尔夫勒姆公司(Wolfram Research)	• 实体、语义类、属性、关系, 知识计算 • 部分知识来自于Mathematica；其它知识来自于各个垂直网站	Apple Siri
Bing Satori	Microsoft	• 实体、语义类、属性、关系, 知识计算 • 自动+人工	Bing Search Engine, Microsoft Cortana
YAGO	马克斯·普朗克研究所	• 自动：从维基百科、WordNet和GeoNames提取信息	YAGO
Facebook Social Graph	Facebook	• Facebook 社交网络数据	Social Graph Search
百度知识图谱	百度	• 搜索结构化数据	百度搜索
搜狗知立方	搜狗	• 搜索结构化数据	搜狗搜索
ImageNet	斯坦福大学	• 搜索引擎 • 亚马逊 AMT	计算机视觉相关应用

下面介绍几个知名的中文知识图谱资源：

- OpenKG.CN：中文开放知识图谱联盟旨在通过建设开放的社区来促进中文知识图谱数据的开放与互联，促进中文知识图谱工具的标准化和技术普及。
- Zhishi.me ：Zhishi.me 是中文常识知识图谱。主要通过从开放的百科数据中抽取结构化数据，已融合了百度百科，互动百科以及维基百科中的中文数据。
- CN-DBPeidia：CN-DBpedia 是由复旦大学知识工场实验室研发并维护的大规模通用领域结构化百科。
- cnSchema.org: cnSchema.org 是一个基于社区维护的开放的知识图谱 Schema 标准。cnSchema 的词汇集包括了上千种概念分类、数据类型、属性和关系等常用概念定义，以支持知识图谱数据的通用性、复用性和流动性。

知识图谱的典型应用

知识图谱为互联网上海量、异构、动态的大数据表达、组织、管理以及利用提供了一种更为有效的方式，使得网络的智能化水平更高，更加接近于人类的认知思维。

基于大规模开放知识库或知识图谱的应用，目前尚处在持续不断的发展与探索的阶段。下面列出了一些国内外比较出色的应用。

1. 语义检索。

谷歌公司通过建立 Google Knowledge Graph，实现了对知识的体系化组织与展示，试图从用户搜索意图感知、以及查询扩展的角度，直接提供给用户想要的知识。

2. 智能问答。

IBM 公司通过搭建知识图谱，并通过自然语言处理和机器学习等技术，开发出了 Watson 系统。在2011年2月的美国问答节目《Jeopardy!》上，Watson 战胜了这一节目的两位冠军选手，可与1996年同样来自 IBM 的“深蓝”战胜国际象棋大师卡斯帕罗夫产生的影响相提并论，被认为是人工智能历史上的一个里程碑。

3. 领域专家快速生成。

构建面向特定领域、特定主题的大规模知识库是实现对某一领域深度分析和计算的重要基础，OpenKN 通过实现端到端的开放知识库构建工具集，实现了在给定部分种子（Seed）的情况下，从无到有的生成领域知识库，进而形成领域专家。

4. 行业生态深度分析与预测。

利用开放大数据可以帮助企业发现潜伏在数据中的威胁，将结构化网络日志、文本数据、开源和第三方数据整合进一个单一的环境，屏蔽可疑的信号与噪声，有效保护用户网络，可在信用卡欺诈行为识别、医疗行业疾病预测、电商商品推荐、强化组织数据安全、不一致性验证、异常分析、金融量化交易、法律分析服务等多方面提供有价值的服务。

知识图谱的前景与挑战

在关注到知识图谱在自然语言处理、人工智能等领域展现巨大潜力的同时，也不难发现知识图谱中的知识获取、知识表示、知识推理等技术依然面临着一些困难与挑战，在未来的一段时间内，知识图谱将是大数据智能的前沿研究问题，有很多重要的开放性问题亟待学术界和产业界协力解决。我们认为，未来知识图谱研究有以下几个重要挑战：

- 知识类型与表示。知识图谱主要采用（实体1、关系、实体2）三元组的形式来表示知识，

这种方法可以较好地表示很多事实性知识。然而，人类知识类型多样，面对很多复杂知识，三元组就束手无策了。例如，人们的购物记录信息、新闻事件等，包含大量实体及其之间的复杂关系，更不用说人类大量的涉及主观感受、主观情感和模糊的知识了。

- 知识获取。如何从互联网大数据萃取知识，是构建知识图谱的重要问题。目前已经提出各种知识获取方案，并已成功抽取大量有用的知识。但在抽取知识的准确率、覆盖率和效率等方面，都仍不如人意，有极大的提升空间。
- 知识融合。来自不同数据的抽取知识可能存在大量噪音和冗余，或者使用了不同的语言。如何将这些知识有机融合起来，建立更大规模的知识图谱，是实现大数据智能的必由之路。
- 知识应用。目前大规模知识图谱的应用场景和方式还比较有限，如何有效实现知识图谱的应用，利用知识图谱实现深度知识推理，提高大规模知识图谱计算效率，需要人们不断锐意发掘用户需求，探索更重要的应用场景，提出新的应用算法。

总结

本文对知识图谱的起源、定义、架构、大规模知识库、应用以及未来挑战等内容，进行了全面阐述。

知识抽取、知识表示、知识融合以及知识推理为构建知识图谱的四大核心技术，本文就当前产业界的需求介绍了它在智能搜索、深度问答、社交网络以及一些垂直行业中的实际应用。此外，还总结了目前知识图谱面临的主要挑战，并对其未来的研究方向进行了展望。

知识图谱的重要性不仅在于它是一个拥有强大语义处理能力与开放互联能力的知识库，并且还是一把开启智能机器大脑的钥匙，能够打开 Web3.0 时代的知识宝库，为相关学科领域开启新的发展方向。

参考资料以及推荐阅读

1. 柳絮飞.《知识图谱：谷歌打造未来搜索》，电脑爱好者，2013年。
2. 徐增林，盛泳潘，贺丽荣，王雅芳.《知识图谱技术综述》，电子科技大学统计机器智能与学习实验室，2016年7月。
3. 知识图谱——机器大脑中的知识库
4. 人工智能2.0时代的开放知识计算

第20课：Neo4j 从入门到构建一个简单知识图谱

Neo4j 对于大多数人来说，可能是比较陌生的。其实，Neo4j 是一个图形数据库，就像传统的关系数据库中的 Oracel 和 MySQL一样，用来持久化数据。Neo4j 是最近几年发展起来的新技术，属于 NoSQL 数据库中的一种。

本文主要从 Neo4j 为什么被用来做知识图谱，Neo4j 的简单安装，在 Neo4j 浏览器中创建节点和关系，Neo4j 的 Python 接口操作以及用 Neo4j 构建一个简单的农业知识图谱五个方面来讲。

Neo4j 为什么被用来做知识图谱

从第19课《知识挖掘与知识图谱概述》中，我们已经明白，知识图谱是一种基于图的数据结构，由节点和边组成。其中节点即实体，由一个全局唯一的 ID 标示，关系（也称属性）用于连接两个节点。通俗地讲，知识图谱就是把所有不同种类的信息连接在一起而得到一个关系网络，提供了从“关系”的角度去分析问题的能力。

而 Neo4j 作为一种经过特别优化的图形数据库，有以下优势：

- **数据存储**：不像传统数据库整条记录来存储数据，Neo4j 以图的结构存储，可以存储图的节点、属性和边。属性、节点都是分开存储的，属性与节点的关系构成边，这将大大有助于提高数据库的性能。
- **数据读写**：在 Neo4j 中，存储节点时使用了 `Index-free Adjacency` 技术，即每个节点都有指向其邻居节点的指针，可以让我们在时间复杂度为 O(1) 的情况下找到邻居节点。另外，按照官方的说法，在 Neo4j 中边是最重要的，是 `First-class Entities`，所以单独存储，更有利于在图遍历时提高速度，也可以很方便地以任何方向进行遍历。
- **资源丰富**：Neo4j 作为较早的一批图形数据库之一，其文档和各种技术博客较多。
- **同类对比**：Flockdb 安装过程中依赖太多，安装复杂；Orientdb，Arangodb 与 Neo4j 做对比，从易用性来说都差不多，但是从稳定性来说，neo4j 是最好的。

综合上述以及因素，我认为 Neo4j 是做知识图谱比较简单、灵活、易用的图形数据库。

Neo4j 的简单安装

Neo4j 是基于 Java 的图形数据库，运行 Neo4j 需要启动 JVM 进程，因此必须安装 Java SE 的 JDK。从 Oracle 官方网站下载 Java SE JDK，选择版本 JDK8 以上版本即可。

下面简单介绍下 Neo4j 在 Linux 和 Windows 的安装过程。首先去官网下载对应版本。解压之后，Neo4j 应用程序有如下主要的目录结构：

- bin 目录：用于存储 Neo4j 的可执行程序；
- conf 目录：用于控制 Neo4j 启动的配置文件；
- data 目录：用于存储核心数据库文件；
- plugins 目录：用于存储 Neo4j 的插件。

Linux 系统下的安装

通过 tar 解压命令解压到一个目录下：

```
tar -xzvf neo4j-community-3.3.1-unix.tar.gz
```

然后进入 Neo4j 解压目录：

```
cd /usr/local/neo4j/neo4j-community-3.1.0
```

通过启动命令，可以实现启动、控制台、停止服务：

```
bin/neo4j  start/console/stop（启动/控制台/停止）
```

通过 `cypher-shell` 命令，可以进入命令行：

```
bin/cypher-shell
```

Windows 系统下的安装

启动 DOS 命令行窗口，切换到解压目录 bin 下，以管理员身份运行命令，分别为启动服务、停止服务、重启服务和查询服务的状态：

```
bin\neo4j start
```

输入:neo4j.bat consol,成功启动即可

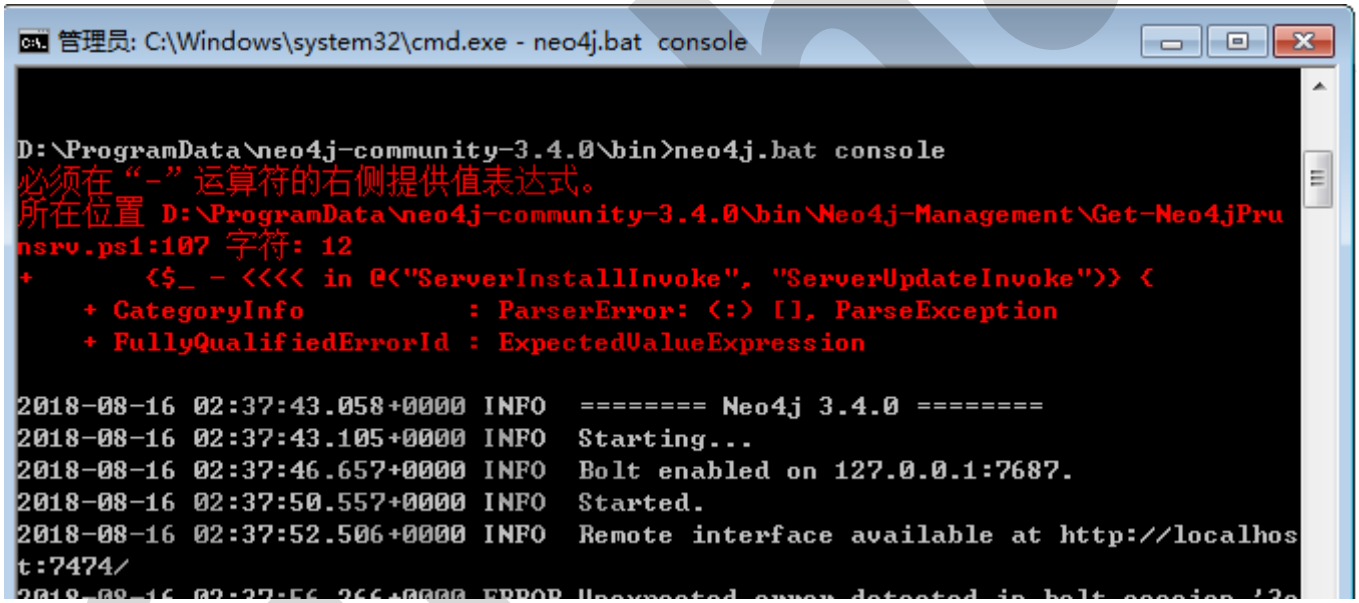
```
bin\neo4j stop
bin\neo4j restart
bin\neo4j status
```

把 Neo4j 安装为服务（Windows Services），可通过以下命令：

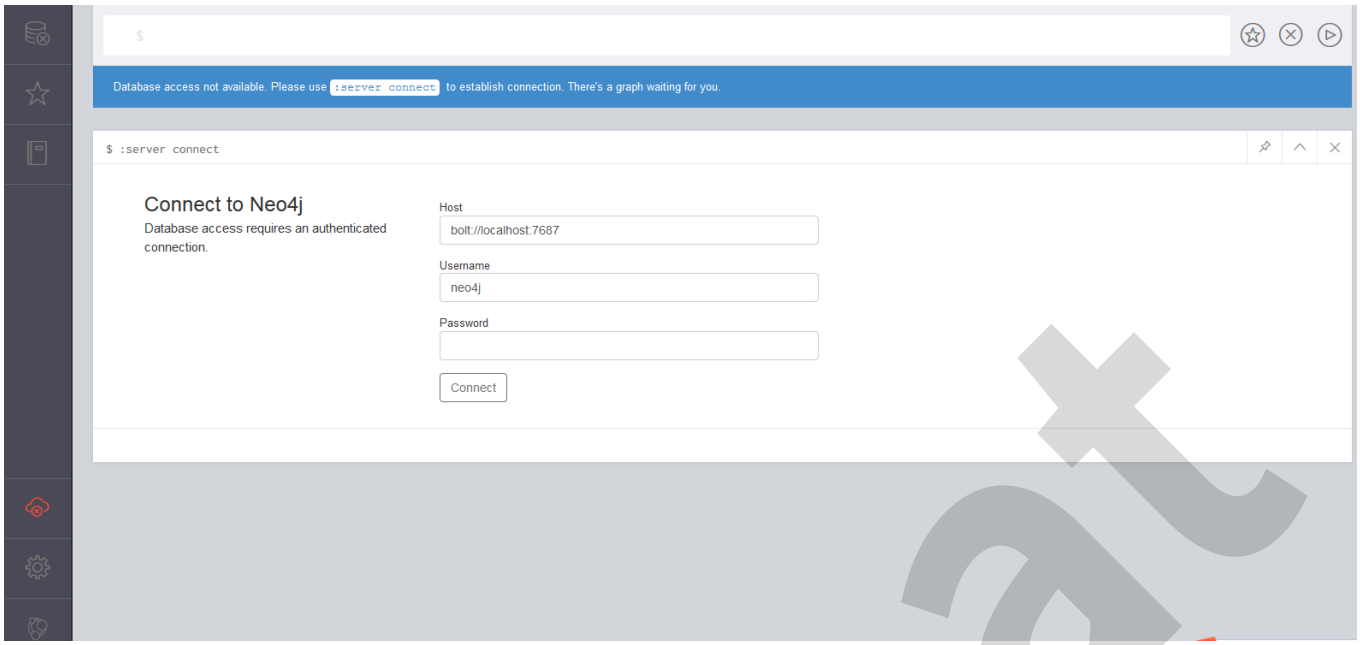
```
bin\neo4j install-service
bin\neo4j uninstall-service
```

Neo4j 的配置文档存储在 conf 目录下，Neo4j 通过配置文件 neo4j.conf 控制服务器的工作。默认情况下，不需要进行任意配置，就可以启动服务器。

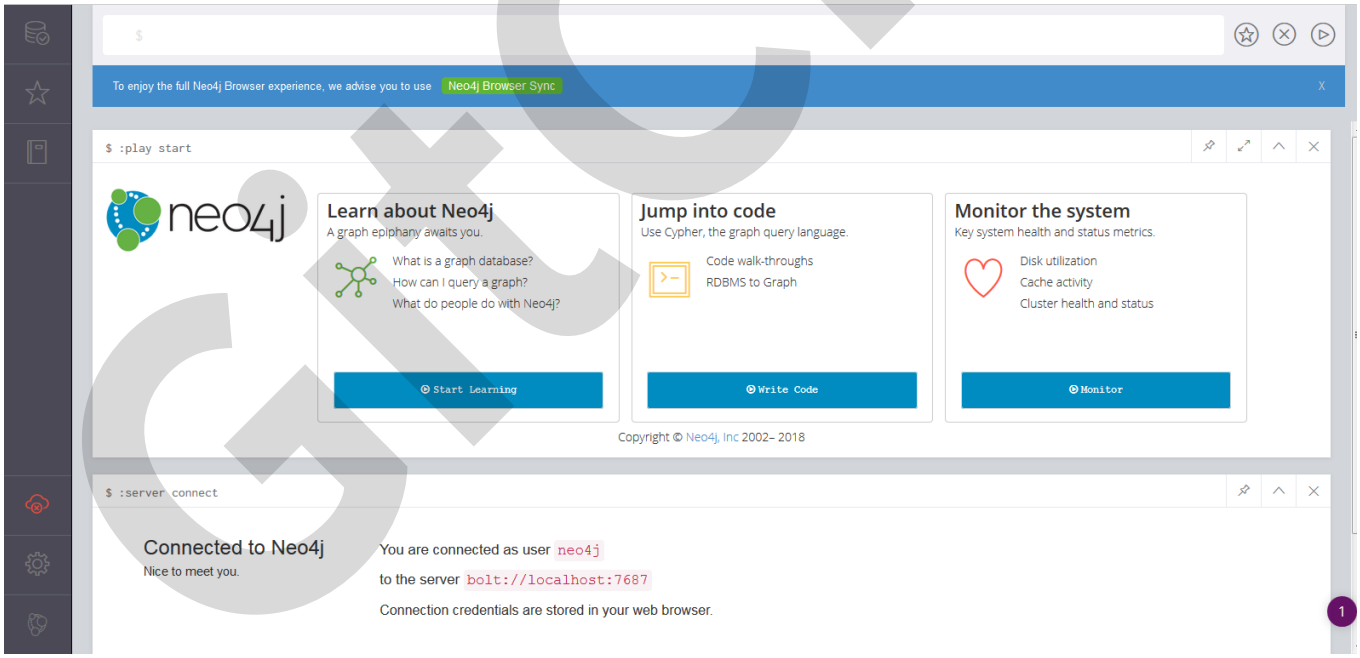
下面我们在 Windows 环境下启动 Neo4j：



Neo4j 服务器具有一个集成的浏览器，在一个运行的服务器实例上访问：http://localhost:7474/，打开浏览器，显示启动页面：



默认的 Host 是 `bolt://localhost:7687`，默认的用户是 neo4j，其默认的密码是 neo4j，第一次成功登录到 Neo4j 服务器之后，需要重置密码。访问 Graph Database 需要输入身份验证，Host 是 Bolt 协议标识的主机。登录成功后界面：



到此为止，我们就完成了 Neo4j 的基本安装过程，更详细的参数配置，可以参考官方文档。

在 Neo4j 浏览器中创建节点和关系

下面，我们简单编写 Cypher 命令，Cypher 命令可以通过 Neo4j 教程学习，在浏览器中通过

Neo4j 创建两个节点和两个关系。

在 `$` 命令行中，编写 Cypher 脚本代码，点击 Play 按钮完成创建，依次执行下面的语句：

```
CREATE (n:Person { name: 'Andres', title: 'Developer' }) return n;
```

作用是创建一个 Person，并包含属性名字和职称。

下面这条语句也创建了一个 Person 对象，属性中只是名字和职称不一样。

```
CREATE (n:Person { name: 'Vic', title: 'Developer' }) return n;
```

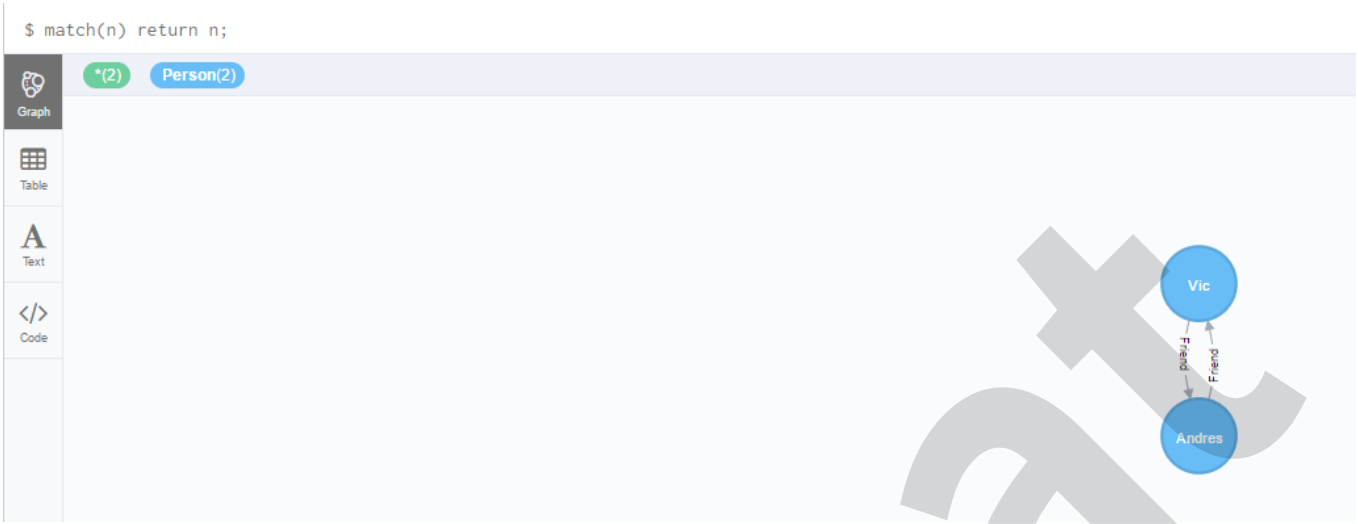
紧接着，通过下面两行命令进行两个 Person 的关系匹配：

```
match(n:Person{name:"Vic"}),(m:Person{name:"Andres"}) create (n)-[r:Friend]->(m) return r;

match(n:Person{name:"Vic"}),(m:Person{name:"Andres"}) create (n)<-[r:Friend]-(m) return r;
```

最后，在创建完两个节点和关系之后，查看数据库中的图形：

```
match(n) return n;
```

如下图，返回两个 Person 节点，以及其关系网，两个 Person 之间组成 Friend 关系：



Neo4j 的 Python 操作

既然 Neo4j 作为一个图库数据库，那我们在项目中使用的时候，必然不能通过上面那种方式完成任务，一般都要通过代码来完成数据的持久化操作。其中，对于 Java 编程者来说，可通过 Spring Data Neo4j 达到这一目的。

而对于 Python 开发者来说，Py2neo 库也可以完成对 Neo4j 的操作，操作过程如下。

首先 安装 Py2neo。Py2neo 的安装过程非常简单，在命令行通过下面命令即可安装成功。

```
pip install py2neo
```

安装好之后，我们来看一下简单的图关系构建，看下面代码：

```
from py2neo.data import Node, Relationship
a = Node("Person", name="Alice")
b = Node("Person", name="Bob")
ab = Relationship(a, "KNOWS", b)
```

第一行代码，首先引入 Node 和 Relationship 对象，紧接着，创建 a 和 b 节点对象，最后一行匹配 a 和 b 之间的工作雇佣关系。接着来看看 ab 对象的内容是什么：

```
print(ab)
```

通过 print 打印出 ab 的内容：

```
(Alice)-[:KNOWS {}]->(Bob)
```

通过这样，就完成了 Alice 和 Bob 之间的工作关系，如果有多组关系将构建成 Person 之间的一个关系网。

了解更多 Py2neo 的使用方法，建议查看官方文档。

用 Neo4j 构建一个简单的农业知识图谱

我们来看一个基于开源语料的简单农业知识图谱，由于过程比较繁杂，数据和知识图谱数据预处理过程这里不再赘述，下面，我们重点看基于 Neo4j 来创建知识图谱的过程。

整个过程主要包含以下步骤：

- 环境准备
- 语料准备
- 语料加载
- 知识图谱查询展示

Neo4j 环境准备。

根据上面对 Neo4j 环境的介绍，这里默认你已经搭建好 Neo4j 的环境，并能正常访问，如果没有环境，请自行搭建好 Neo4j 的可用环境。

数据语料介绍。

本次提供的语料是已经处理好的数据，包含6个 csv 文件，文件内容和描述如下。

- attributes.csv：文件大小 2M，内容是通过互动百科页面得到的部分实体的属性，包含字段：Entity、AttributeName、Attribute，分别表示实体、属性名称、属性值。文件前5行结构如下：

```
Entity,AttributeName,Attribute
密度板,别名,纤维板
葡萄蔓枯病,主要为害部位,枝蔓
坎德拉,性别,男
坎德拉,国籍,法国
```

```
坎德拉,场上位置,后卫
```

- `hudong_pedia.csv`：文件大小 94.6M，内容是已经爬好的农业实体的百科页面的结构化数据，包含字段：title、url、image、openTypeList、detail、baseInfoKeyList、baseInfoValueList，分别表示名称、百科 URL 地址、图片、分类类型、详情、关键字、依据来源。文件前2行结构如下：

```
"title","url","image","openTypeList","detail","baseInfoKeyList","baseInfoValueList"
"菊糖","http://www.baike.com/wiki/菊糖","http://a0.att.hudong.com/72/85/20200000013920144736851207227_s.jpg","健康科学##分子生物学##化学品##有机物##科学##自然科学##药品##药学名词##药物中文名称列表","[药理作用] 诊断试剂 人体内不含菊糖，静注后，不被机体分解、结合、利用和破坏，经肾小球滤过，通过测定血中和尿中的菊糖含量，可以准确计算肾小球的滤过率。菊糖广泛存在于植物组织中,约有3.6万种植物中含有菊糖,尤其是菊芋、菊苣块根中含有丰富的菊糖[6,8]。菊芋(Jerusalem artichoke)又名洋姜,多年生草本植物,在我国栽种广泛,其适应性广、耐贫瘠、产量高、易种植,一般亩产菊芋块茎为2 000～4 000 kg,菊芋块茎除水分外,还含有15%～20%的菊糖,是加工生产菊糖及其制品的良好原料。","中文名：","菊糖"
"密度板","http://www.baike.com/wiki/密度板","http://a0.att.hudong.com/64/31/20200000013920144728317993941_s.jpg","居家##巧克力包装##应用科学##建筑材料##珠宝盒##礼品盒##科学##糖果盒##红酒盒##装修##装饰材料##隔断##首饰盒","密度板（英文：Medium Density Fiberboard (MDF)）也称纤维板，是以木质纤维或其他植物纤维为原料，施加脲醛树脂或其他适用的胶粘剂制成的人造板材。按其密度的不同，分为高密度板、中密度板、低密度板。密度板由于质软耐冲击，也容易再加工，在国外是制作家私的一种良好材料，但由于国家关于高密度板的标准比国际标准低数倍，所以，密度板在中国的使用质量还有待提高。","中文名：##全称：##别名：##主要材料：##分类：##优点：","密度板##中密度板纤维板##纤维板##以木质纤维或其他植物纤维##高密度板、中密度板、低密度板##表面光滑平整、材质细密性能稳定"
```

- `hudong_pedia2.csv`：文件大小 41M，内容结构和 `hudong_pedia.csv` 文件保持一致，只是增加数据量，作为 `hudong_pedia.csv` 数据的补充。
- `new_node.csv`：文件大小 2.28M，内容是节点名称和标签，包含字段：title、lable，分别表示节点名称、标签，文件前5行结构如下：

```
title,lable
药物治疗,newNode
膳食纤维,newNode
Boven Merwede,newNode
亚美尼亚苏维埃百科全书,newNode
```

- `wikidata_relation.csv`：文件大小 1.83M，内容是实体和关系，包含字段 HudongItem1、relation、HudongItem2，分别表示实体1、关系、实体2，文件前5行结构如下：

```
HudongItem1,relation,HudongItem2
菊糖,instance of,化合物
菊糖,instance of,多糖
瓦尔,instance of,河流
菊糖,subclass of,食物
瓦尔,origin of the watercourse,莱茵河
```

- `wikidata_relation2.csv`：大小 7.18M，内容结构和 `wikidata_relation.csv` 一致，作为 `wikidata_relation.csv` 数据的补充。

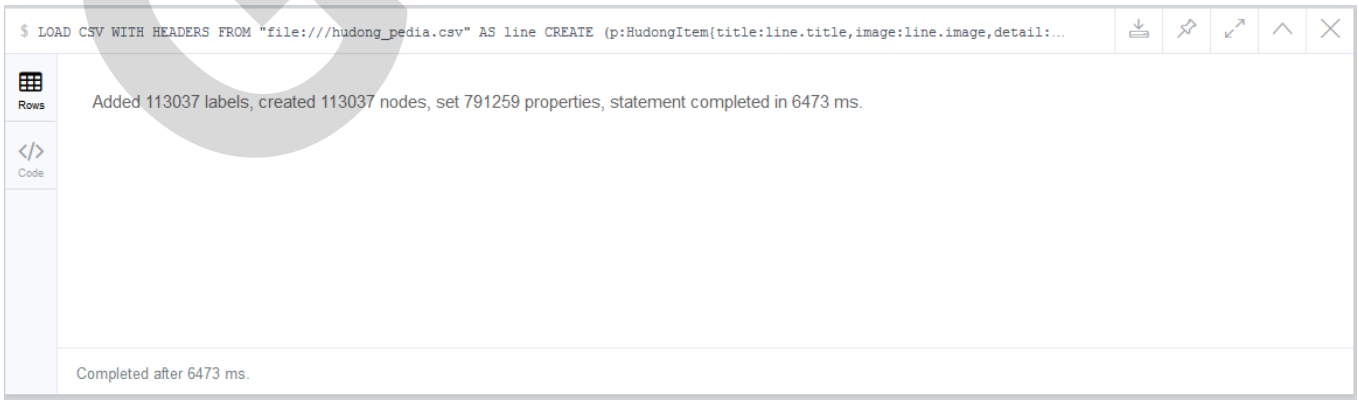
语料加载。

语料加载，利用 Neo4j 的 `LOAD CSV WITH HEADERS FROM...` 功能进行加载，具体操作过程如下。

首先，依次执行以下命令：

```
// 将hudong_pedia.csv 导入
LOAD CSV WITH HEADERS  FROM "file:///hudong_pedia.csv" AS line
CREATE (p:HudongItem{title:line.title,image:line.image,detail:line.detail,url:line.url,openTypeList:line.openTypeList,baseInfoKeyList:line.baseInfoKeyList,baseInfoValueList:line.baseInfoValueList})
```

执行成功之后，控制台显示成功：



上面这张图，表示数据加载成功，并显示加载的数据条数和耗费的时间。

```
// 新增了hudong_pedia2.csv
LOAD CSV WITH HEADERS  FROM "file:///hudong_pedia2.csv" AS line
CREATE (p:HudongItem{title:line.title,image:line.image,detail:line.detail,url:line.url,openTypeList:line.openTypeList,baseInfoKeyList:line.baseInfoKeyList,baseInfoValueList:line.baseInfoValueList})

// 创建索引
CREATE CONSTRAINT ON (c:HudongItem)
ASSERT c.title IS UNIQUE
```

以上命令的意思是，将 `hudong_pedia.csv` 和 `hudong_pedia2.csv` 导入 Neo4j 作为结点，然后对 titile 属性添加 UNIQUE（唯一约束/索引）。

注意： 如果导入的时候出现 Neo4j JVM 内存溢出错误，可以在导入前，先把 Neo4j 下的 `conf/neo4j.conf` 中的 `dbms.memory.heap.initial_size` 和 `dbms.memory.heap.max_size` 调大点。导入完成后再把值改回去即可。

下面继续执行数据导入命令：

```
// 导入新的节点
LOAD CSV WITH HEADERS FROM "file:///new_node.csv" AS line
CREATE (:NewNode { title: line.title })

//添加索引
CREATE CONSTRAINT ON (c:NewNode)
ASSERT c.title IS UNIQUE

//导入hudongItem和新加入节点之间的关系
LOAD CSV  WITH HEADERS FROM "file:///wikidata_relation2.csv" AS line
MATCH (entity1:HudongItem{title:line.HudongItem}) ,(entity2:NewNode{title:line.NewNode})
CREATE (entity1)-[:RELATION { type: line.relation }]->(entity2)

LOAD CSV  WITH HEADERS FROM "file:///wikidata_relation.csv" AS line
MATCH (entity1:HudongItem{title:line.HudongItem1}) ,(entity2:HudongItem{title:line.HudongItem2})
CREATE (entity1)-[:RELATION { type: line.relation }]->(entity2)
```

执行完这些命令后，我们导入 `new_node.csv` 新节点，并对 titile 属性添加 UNIQUE（唯一约束/索引），导入 `wikidata_relation.csv` 和 `wikidata_relation2.csv`，并给节点之间

创建关系。

紧接着，继续导入实体属性，并创建实体之间的关系：

```
LOAD CSV WITH HEADERS FROM "file:///attributes.csv" AS line
MATCH (entity1:HudongItem{title:line.Entity}),
(entity2:HudongItem{title:line.Attribute})
CREATE (entity1)-[:RELATION { type: line.AttributeName }]->(entity2);

LOAD CSV WITH HEADERS FROM "file:///attributes.csv" AS line
MATCH (entity1:HudongItem{title:line.Entity}), (entity2:NewNode{title:l
ine.Attribute})
CREATE (entity1)-[:RELATION { type: line.AttributeName }]->(entity2);

LOAD CSV WITH HEADERS FROM "file:///attributes.csv" AS line
MATCH (entity1:NewNode{title:line.Entity}),
(entity2:NewNode{title:line.Attribute})
CREATE (entity1)-[:RELATION { type: line.AttributeName }]->(entity2);

LOAD CSV WITH HEADERS FROM "file:///attributes.csv" AS line
MATCH (entity1:NewNode{title:line.Entity}), (entity2:HudongItem{title:l
ine.Attribute})
CREATE (entity1)-[:RELATION { type: line.AttributeName }]->(entity2)
```

这里注意，建索引的时候带了 label，因此只有使用 label 时才会使用索引，这里我们的实体有两个 label，所以一共做 `2*2=4` 次。当然也可以建立全局索引，即对于不同的 label 使用同一个索引。

以上过程，我们就完成了语料加载，并创建了实体之间的关系和属性匹配，下面我们来看看 Neo4j 图谱关系展示。

知识图谱查询展示

最后通过 cypher 语句查询来看看农业图谱展示。

首先，展示 HudongItem 实体，执行如下命令：

```
MATCH (n:HudongItem) RETURN n LIMIT 25
```

对 HudongItem 实体进行查询，返回结果的25条数据，结果如下图：

接着，展示 NewNode 实体，执行如下命令：

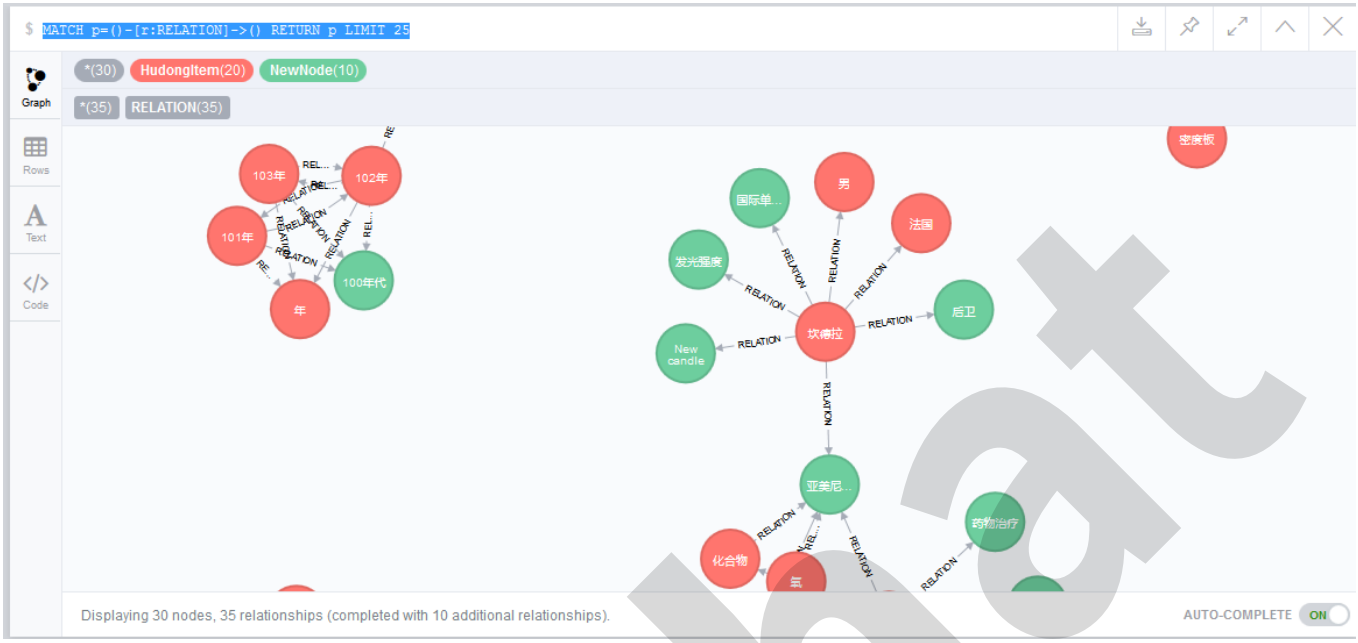
```
1.  MATCH (n:NewNode) RETURN n LIMIT 25
```

对 NewNode 实体进行查询，返回结果的25条数据，结果如下图：

之后，展示 RELATION 直接的关系，执行如下命令：

```
1.  MATCH p=()-[r:RELATION]->() RETURN p LIMIT 25
```

展示实体属性关系，结果如下图：



总结

本节内容到此结束，回顾下整篇文章，主要讲了以下内容：

1. 解释了 Neo4j 被用来做知识图谱的原因；
2. Neo4j 的简单安装以及在 Neo4j 浏览器中创建节点和关系；
3. Neo4j 的 Python 接口操作及使用；
4. 从五个方面讲解了如何使用 Neo4j 构建一个简单的农业知识图谱。

最后，强调一句，知识图谱未来会通过自然语言处理技术和搜索技术结合应用会越来越广，工业界所出的地位也会越来越重要。

参考文献及推荐阅读

1. Neo4j 官网
2. The Neo4j Developer Manual
3. Neo4j 教程
4. Py2neo——Neo4j & Python 的配合使用

第21课：中文自然语言处理的应用、现状和未来

自然语言理解和自然语言生成是自然语言处理的两大内核，机器翻译是自然语言理解方面最早的研究工作。自然语言处理的主要任务是：研究表示语言能力和语言应用的模型，建立和实现计算框架并提出相应的方法不断地完善模型，根据这样的语言模型设计有效地实现自然语言通信的计算机系统，并研讨关于系统的评测技术，最终实现用自然语言与计算机进行通信。目前，具有一定自然语言处理能力的典型应用包括计算机信息检索系统、多语种翻译系统等。

微软创始人比尔·盖茨曾经表示，“语言理解是人工智能领域皇冠上的明珠”。

语言是逻辑思维和交流的工具，宇宙万物中，只有人类才具有这种高级功能。要实现人与计算机间采用自然语言通信，必须使计算机同时具备自然语言理解和自然语言生成两大功能。

因此，NLP 作为人工智能的一个子领域，其主要目的就包括两个方面：自然语言理解，让计算机理解自然语言文本的意义；自然语言生成，让计算机能以自然语言文本来表达给定的意图、思想等。自然语言是人类智慧的结晶，自然语言处理是人工智能中最为困难的问题之一，而对自然语言处理的研究也是充满魅力和挑战的。

NLP 领域发展现状如何？

近年来，自然语言处理处于快速发展阶段。各种词表、语义语法词典、语料库等数据资源的日

益丰富，词语切分、词性标注、句法分析等技术的快速进步，各种新理论、新方法、新模型的出现推动了自然语言处理研究的繁荣。互联网与移动互联网和世界经济社会一体化的潮流对自然语言处理技术的迫切需求，为自然语言处理研究发展提供了强大的市场动力。

我国直到上世纪80年代中期才开始较大规模和较系统的自然语言处理研究，尽管较国际水平尚有较大差距，但已经有了比较稳定的研究内容，包括语料库、知识库等数据资源建设，词语切分、句法分析等基础技术，以及信息检索、机器翻译等应用技术。

当前国内外出现了一批基于 NLP 技术的应用系统，例如 IBM 的 Watson 在电视问答节目中战胜人类冠军；苹果公司的 Siri 个人助理被大众广为测试；谷歌、微软、百度等公司纷纷发布个人智能助理；科大讯飞牵头研发高考机器人……但相比于性能趋于饱和的计算机视觉和语音识别技术，自然语言处理因技术难度太大、应用场景太复杂，研究成果还未达到足够的高度。

自然语言处理中句子级分析技术

目前，自然语言处理的对象有词、句子、篇章和段落、文本等，但是大多归根到底在句子的处理上，自然语言处理中的自然语言句子级分析技术，可以大致分为词法分析、句法分析、语义分析三个层面。

第一层面的词法分析包括汉语分词和词性标注两部分。和大部分西方语言不同，汉语书面语词语之间没有明显的空格标记，文本中的句子以字串的形式出现。因此汉语自然语言处理的首要工作就是要将输入的字串切分为单独的词语，然后在此基础上进行其他更高级的分析，这一步骤称为分词。

除了分词，词性标注也通常认为是词法分析的一部分。给定一个切好词的句子，词性标注的目的是为每一个词赋予一个类别，这个类别称为词性标记，比如，名词（Noun）、动词（Verb）、形容词（Adjective）等。一般来说，属于相同词性的词，在句法中承担类似的角色。

第二个层面的句法分析是对输入的文本句子进行分析以得到句子的句法结构的处理过程。对句法结构进行分析，一方面是语言理解的自身需求，句法分析是语言理解的重要一环，另一方面也为其它自然语言处理任务提供支持。例如句法驱动的统计机器翻译需要对源语言或目标语言（或者同时两种语言）进行句法分析；语义分析通常以句法分析的输出结果作为输入以便获得更多的指示信息。

根据句法结构表示形式的不同，最常见的句法分析任务可以分为以下三种：

1. 短语结构句法分析，该任务也被称作成分句法分析，作用是识别出句子中的短语结构以及短语之间的层次句法关系；
2. 依存句法分析，作用是识别句子中词汇与词汇之间的相互依存关系；
3. 深层文法句法分析，即利用深层文法，例如词汇化树邻接文法、词汇功能文法、组合范畴文法等，对句子进行深层的句法以及语义分析。

上述几种句法分析任务比较而言，依存句法分析属于浅层句法分析。其实现过程相对简单，比较适合在多语言环境下的应用，但是依存句法分析所能提供的信息也相对较少。深层文法句法分析可以提供丰富的句法和语义信息，但是采用的文法相对复杂，分析器的运行复杂度也较高，这使得深层句法分析当前不适合处理大规模数据。短语结构句法分析介于依存句法分析和深层文法句法分析之间。

第三个层面是语义分析。语义分析的最终目的是理解句子表达的真实语义。但是，语义应该采用什么表示形式一直困扰着研究者们，至今这个问题也没有一个统一的答案。

语义角色标注是目前比较成熟的浅层语义分析技术。基于逻辑表达的语义分析也得到学术界的长期关注。出于机器学习模型复杂度、效率的考虑，自然语言处理系统通常采用级联的方式，即分词、词性标注、句法分析、语义分析分别训练模型。实际使用时，给定输入句子，逐一使用各个模块进行分析，最终得到所有结果。

深度学习背景下的自然语言处理

近年来，随着研究工作的深入，研究者们开始从传统机器学习转向深度学习。2006年开始，有人利用深层神经网络在大规模无标注语料上无监督的为每个词学到了一个分布式表示，形式上把每个单词表示成一个固定维数的向量，当作词的底层特征。在此特征基础上，完成了词性标注、命名实体识别和语义角色标注等多个任务，后来有人利用递归神经网络完成了句法分析、情感分析和句子表示等多个任务，这也为语言表示提供了新的思路。

面向自然语言处理的深度学习研究工作，目前尚处于起步阶段，尽管已有的深度学习算法模型如循环神经网络、递归神经网络和卷积神经网络等已经有较为显著的应用，但还没有重大突破。围绕适合自然语言处理领域的深度学习模型构建等研究应该有着非常广阔的空间。

在当前已有的深度学习模型研究中，难点是在模型构建过程中参数的优化调整方面。主要有深

度网络层数、正则化问题及网络学习速率等，可能的解决方案比如有采用多核机提升网络训练速度，针对不同应用场合，选择合适的优化算法等。

自然语言处理未来的研究方向

纵观自然语言处理技术研究发展的态势和现状，以下研究方向或问题将可能成为自然语言处理未来研究必须攻克的堡垒：

1. 词法和句法分析方面：包括多粒度分词、新词发现、词性标注等；
2. 语义分析方面：包括词义消歧、非规范文本的语义分析。其中，非规范划化文本主要指社交平台上比较口语化、弱规范甚至不规范的短文本，因其数据量巨大和实时性而具有研究和应用价值，被广泛用于舆情监控、情感分析和突发事件发现等任务；
3. 语言认知模型方面：比如使用深度神经网络处理自然语言，建立更有效、可解释的语言计算模型，例如，词嵌入的发现。还有目前词的表示是通过大量的语料库学习得到的，如何通过基于少量样本来发现新词、低频词也急需探索；
4. 知识图谱方面：如何构建能够融合符号逻辑和表示学习的大规模高精度的知识图谱；
5. 文本分类与聚类方面：通过有监督、半监督和无监督学习，能够准确进行分类和聚类。当下大多数语料都是没有标签的，未来在无监督或者半监督方面更有需求；
6. 信息抽取方面：对于多源异构信息，如何准确进行关系、事件的抽取等。信息抽取主要从

面向开放域的可扩展信息抽取技术、自学习与自适应和自演化的信息抽取系统以及面向多源异构数据的信息融合技术方向发展；

7. 情感分析方面：包括基于上下文感知的情感分析、跨领域跨语言情感分析、基于深度学习的端到端情感分析、情感解释、反讽分析、立场分析等；
8. 自动文摘方面：如何表达要点信息？如何评估信息单元的重要性？这些都要随着语义分析、篇章理解、深度学习等技术快速发展；
9. 信息检索方面：包括意图搜索、语义搜索等，都将有可能出现在各种场景的垂直领域，将以知识化推理为检索运行方式，以自然语言多媒体交互为手段的智能化搜索与推荐技术；
10. 自动问答方面：包括深度推理问答、多轮问答等各种形式的自动问答系统；
11. 机器翻译方面：包括面向小数据的机器翻译、非规范文本的机器翻译和篇章级机器翻译等。

总结

本文，从 NLP 的概念出发，首先指出了自然语言处理的两大内核：自然语言理解和自然语言生成；然后简单介绍了国内外 NLP 研究发展现状；紧接着重点介绍了最常用、应用最广的自然语言处理中句子级分析技术，最后在深度学习背景下，指出了自然语言处理未来可能遇到的挑战和重点研究方向，为后期的学习提供指导和帮助。

关于本课程结束寄语

首先，非常感谢 GitChat 给我们提供这样一个学习平台，非常感谢编辑老师的辛苦指导，非常感谢各位同学能够报名参加本课程。

近年来，深度学习正在逐渐的填平领域鸿沟，继在图像和语音领域取得的巨大成功后，深度学习在同属人类认知范畴的自然语言处理方面也在不断取得更好的效果。

在《中文自然语言处理入门实战》达人课中，相信各位同学通过一些小数据量的“简易版”实例，已经体会到了中文自然语言处理的精妙，并完成了中文自然语言处理从0到1的过程。但如何更好地把这些技术应用在工业生产中，不仅需要过硬的技术更要求熟练掌握相关核心算法的原理，做到知其然并知其所以然。

因此，我计划在《中文自然语言处理入门实战》课程的基础上，推出《中文自然语言处理核心算法精要剖析》课程，本课程共包含28节，包含了中文自然语言处理核心算法精要，有针对性的面向想要深入学习中文自然语言处理和想要从事 NLP 相关工作的人，重点学习这些核心算法的原理，注重理论与实战结合，助力在中文自然语言处理上理解的更深、做的更好。

对《中文自然语言处理核心算法精要剖析》感兴趣的同学，请继续关注我，课程后期将尽快上线。

参考以及推荐阅读

- 今日头条李航：深度学习 NLP 的现有优势与未来挑战