一、pyspider框架

(1)安装

pip3 install pyspider

(2)启动

#pyspider

注意：pyspider命令默认会以all模式运行所有的组件，方便调试。作者建议在线上模式分开部署各各组件，详情请查看部署章节

运行成功后用浏览器打开http://localhost:5000/访问控制台

（3）自带代码剖析

#!/usr/bin/env python

# -\*- encoding: utf-8 -\*-

# Created on 2015-10-08 12:45:44

# Project: test

from pyspider.libs.base\_handler import \*

class Handler(BaseHandler):

crawl\_config = {

}

@every(minutes=24 \* 60)

def on\_start(self):

self.crawl('http://scrapy.org/', callback=self.index\_page)

@config(age=10 \* 24 \* 60 \* 60)

def index\_page(self, response):

for each in response.doc('a[href^="http"]').items():

self.crawl(each.attr.href, callback=self.detail\_page)

@config(priority=2)

def detail\_page(self, response):

return {

"url": response.url,

"title": response.doc('title').text(),

}

代码简单分析：

def on\_start(self) 方法是入口代码。当在web控制台点击run按钮时会执行此方法。

self.crawl(url, callback=self.index\_page)这个方法是调用API生成一个新的爬取任务，这个任务被添加到待抓取队列。

def index\_page(self, response) 这个方法获取一个Response对象。 response.doc是pyquery对象的一个扩展方法。pyquery是一个类似于jQuery的对象选择器。

def detail\_page(self, response)返回一个结果集对象。这个结果默认会被添加到resultdb数据库（如果启动时没有指定数据库默认调用sqlite数据库）。你也可以重写on\_result(self,result)方法来指定保存位置。

更多知识：

@every(minutes=24\*60, seconds=0) 这个设置是告诉scheduler（调度器）on\_start方法每天执行一次。

@config(age=10 \* 24 \* 60 \* 60) 这个设置告诉scheduler（调度器）这个request（请求）过期时间是10天，10天内再遇到这个请求直接忽略。这个参数也可以在self.crawl(url, age=10\*24\*60\*60) 和 crawl\_config中设置。

@config(priority=2) 这个是优先级设置。数字越大越先执行

（4）执行任务

a.当完成脚本编写，调试无误后，请先保存你的脚本，然后返回到控制台首页。

b.直接点击status那栏，把状态由ToDO改成debug或running。

c.最后点击项目最右边那个run按钮启动项目。

d.当progress那栏有数据显示说明启动成功。到现在就可以点最右侧的results查看结果了。

（5）**self.crawl(url,\*\*kwargs)**

参数:

url需要被抓取的url或url列表．

callback这个参数用来指定爬取内容后需要哪个方法来处理内容．一般解析为 response. \_default: \_\_call\_\_ \_　如下面调用方法：

**def on\_start(self):**

**self.crawl('http://scrapy.org/', callback=self.index\_page)**

**self.crawl还有以下可选参数**

**age**本参数用来指定任务的有效期，在有效期内不会重复抓取．默认值是-1（永远不过期，意思是只抓一次）

@config(age=10 \* 24 \* 60 \* 60)

def index\_page(self, response):

...

解析：每一个回调index\_page的任务有效期是10天，在10天之内再遇到这个任务都会被忽略掉（除非有强制抓取参数才不会忽略）

**priority**这个参数用来指定任务的优先级，数值越大越先被执行．默认值为0．

def index\_page(self):

self.crawl('http://www.example.org/page2.html', callback=self.index\_page)

self.crawl('http://www.example.org/233.html', callback=self.detail\_page,priority=1)

**exetime** 多长时间后开始爬取，默认值为0，立即执行

import time def on\_start(self):

self.crawl('http://www.example.org/', callback=self.callback,exetime=time.time()+30\*60)

解析：30分钟后开始爬取

**retries** 任务执行失败后重试次数. default: 3

**itag**任务标记值，此标记会在抓取时对比，如果这个值发生改变，不管有效期有没有到都会重新抓取新内容．多数用来动态判断内容是否修改或强制重爬．默认值是：None.

def index\_page(self, response):

for item in response.doc('.item').items():

self.crawl(item.find('a').attr.url, callback=self.detail\_page, itag=item.find('.update-time').text())

本实例中使用页面中update-time元素的值当成itag来判断内容是否有更新．

class Handler(BaseHandler):

crawl\_config = { 'itag': 'v223' }

修改全局参数itag，使所有任务都重新执行（需要点run按钮来启动任务）

**auto\_recrawl** 当为真时，每天重新爬取

def on\_start(self):

self.crawl('http://www.example.org/', callback=self.callback,age=5\*60\*60, auto\_recrawl=True)

**method HTTP**请求方法设置，默认值: GET

**params**把一个字典参数附加到url参数里，如 ：

def on\_start(self):

self.crawl('http://httpbin.org/get', callback=self.callback,params={'a': 123, 'b': 'c'})

self.crawl('http://httpbin.org/get?a=123&b=c', callback=self.callback)

解析：这两个是相同的任务

**data**这个参数会附加到ＵＲＬ请求的body里，如果是字典会经过form-encoding编码再附加.

def on\_start(self):

self.crawl('http://httpbin.org/post', callback=self.callback,method='POST', data={'a': 123, 'b': 'c'})

**files** dictionary of {field: {filename: 'content'}} files to multipart upload.`

**headers**自定义请求头（字典类型）．

**cookies**自定义请求的cookies（字典类型）．

**connect\_timeout**指定请求时链接超时时间,单位秒，默认值：20.

**timeout**请求内容里最大等待秒数．默认值：120．

**allow\_redirects**遇到30x状态码时是否重新请求跟随．默认是：True.

**validate\_cert**遇到HTTPS类型的URL时是否验证证书，默认值：True.

**proxy**设置代理服务器，格式如 username:password@hostname:port .暂时只支持http代理

class Handler(BaseHandler):

crawl\_config = { 'proxy': 'localhost:8080' }

Handler.crawl\_config里配置proxy参数会对整个项目生效，本项目的所有任务都会使用代理爬取．

**taskid**唯一性的taskid用来区别不同的任务．默认taskid是由URL经过md5计算得出．你也可以使用def get\_taskid(self, task)方法覆盖系统自带的来自定义任务id.如：

import json from pyspider.libs.utils

import md5string

def get\_taskid(self, task):

return md5string(task['url']+json.dumps(task['fetch'].get('data', '')))

（6）Response

Response对象的方法及成员参考

**Response.url** 返回最后的URL地址.

**Response.text**请求响应的文本格式内容

如果Response.encoding 是 None 或 chardet 模块可用, 响应内容会自动被解析为指定的编码.

**Response.content**请求响应的二进制格式内容，未做编码解析

**Response.doc**本方法会调用PyQuery库用返回的内容生成一个PyQuery对象以方便使用，生成对象时默认已经把里面的所有链接格式化成绝对链接，可直接分析使用．

**Response.etree**本方法会调用lxml库用返回的内容生成一个lxml对象以方便使用．

**Response.json**本方法会调用JSON相关库来解析返回的内容.

**Response.status\_code**

**Response.orig\_url**

If there is any redirection during the request, here is the url you just submit via self.crawl.

**Response.headers**请求响应的头信息，dict格式

**Response.cookies**

**Response.errorfetch**的报错信息

**Response.time**抓取使用的时间

**Response.ok**如果状态码是200并且没有错误信息这个值就是True.用来判断是否请求成功．

**Response.encoding**

Encoding of Response.content.

If Response.encoding is None, encoding will be guessed by header or content or chardet(if available).

Set encoding of content manually will overwrite the guessed encoding.

**Response.save**

The object saved by self.crawl API

**Response.js\_script\_result**

content returned by JS script

**Response.raise\_for\_status()**

Raise HTTPError if status code is not 200 or Response.error exists.

（7）项目

大多数情况下，一个项目就是你针对一个网站写的一个爬虫脚本．

项目是相对独立的但是你可以导入其它项目或其它项目的模块．

项目有五个状态:TODO,STOP,CHECKING,DEBUG,RUNNING

TODO- 当一个脚本刚刚被创建时的状态

STOP- 你可以设置项目状态为STOP让项目停止运行

CHECKING- 当一个运行中的项目被编辑时项目状态会被自动设置成此状态并停止运行．

DEBUG/RUNNING- 这两状态都会运行爬虫，但是他们之间是有区别的．一般来说调试阶段用DEBUG状态，线上用RUNNING状态．

爬虫的抓取速度根据网上流行的token-bucket来控制.

rate- 每秒执行多少个请求．

burst- 设置并发数,如：rate/burst = 0.1/3,这个的意思是爬虫10秒爬一个页面．但是开始时前三个任务会同时时行，不会等10秒，第四个任务爬取前会等10秒．

项目删除：把group设置成delete并把项目状态设置成STOP，２４小时后系统会自动删除此项目．

（8）任务

什么是任务

任务是调度器调度的最基本单元

任务属性

一个任务有唯一的任务ID叫taskid(默认是：md5(url),不过可以重写get\_taskid(self,task)方法指定自己项目生成taskid方法。)

不同项目之间任务不冲突。

任务的四个状态:

* active（运行）
* failed（失败）
* success（成功）
* bad - （暂时没用着）

只有当任务状态为运行（active）时才会被调度。

根据优先级执行任务．

Schedule（调度器）

* 当遇到一个新任务（之前没有遇到过的链接）时：
* 如果设置了exetime并且还没到时间，任务会被添加到等待队列.否则会接受添加到执行队列
* 当任务已经在队列中:
* 如果没有设置force\_update就会忽略
* 当遇到一个之前爬过的任务：
* 如果设置了age,并且last\_crawl\_age+age<now任务会被接受并添加到执行队列，否则被忽略．
* 如果设置了itag且值不等于之前的值会添加到执行队列重新爬取，否则被忽略

（9）环境

脚本执行环境

变量

self.project\_name　　项目名

self.projectinformation 　当前项目信息

self.response　　响应信息

self.task　　　任务信息

关于脚本

脚本的名称不重要，但是你必需有一个类继承BaseHandler．

每个方法默认都会传三个参数用来获取相关信息分别是：自身self，请求信息response，项目信息task.如：def callback(self, response, task)

默认情况下响应代码不等于200的都会被忽略，但是你可以设置@catch\_status\_code\_error参数来改变默认参数

关于环境

日志和异常都会被捕获打印输出

你可以导入其它项目或项目的模块使用

ＷＥＢ视图：Web view

在web控制台以浏览器模式显示页面内容

ＨＴＭＬ视图：HTML view

在回调函数时显示ＨＴＭＬ代码

跟踪视图：Follows view

view the callbacks that can be made from the current callback index\_page follows view will show the detail\_page callbacks that can be executed.

消息视图：Messages view

显示来自 self.send\_message API接口的消息.

ＣＳＳ选择器助力：Enable CSS Selector Helper

ＣＳＳ选择器助手可以在web视图中通过单击获取节点的css规则，并自动插件到脚本．

（10）结果

通常Pyspider在WebUI控制台上查看和下载爬取结果非常方便，但这种办法不是在所有的项目中好用．

**结果数据库**

结果数据库resultdb仅针对结果预览，不适合大规模存储。如果你想从修改抓取的数据保存方式，可以使用数据库API，数据库API可以帮助你连接并查询数据．

from [pyspider](http://www.pyspider.cn/)

.database import connect\_database

resultdb = connect\_database("<your resutldb connection url>") for project in resultdb: for result in resultdb.select(project):

assert result['taskid']

assert result['url']

assert result['result']

result['result']就是你的爬虫脚本爬回来的数据．

**结果保存组件**

在产品环境中，您可能需要把结果发送到您自己系统的接口上以方便第三方程序处理，而不是将其存储到resultdb。强烈建议重写resultworker。

from pyspider.result import ResultWorker Class MyResultWorker(ResultWorker):

def on\_result(self, task, result):

assert task['taskid']

assert task['project']

assert task['url']

assert result # your processing code goes here

result就是你的爬虫脚本爬取的数据.

你需要把自己的保存结果的脚本（如：my\_result\_worder.py）保存在pyspider目录，在启动pyspider时添加参数result\_worker和--result-cls调用自己的类．如：pyspider result\_worker --result-cls=my\_result\_worder. 这样来启动结果处理类．或修改配置文件，如下：

{

... "result\_worker": { "result\_cls": "my\_result\_worder. MyResultWorker" } ...

}

配置文件使用方法请见： [Please refer to Deployment](http://docs.pyspider.org/en/latest/Deployment)

**设计自己的数据库结构**

默认存储在数据库中的结果被编码为JSON格式不方便分析查看和第三方程序使用，所以强烈建议设计自己的数据库，并重写resultworker。

**提示：**

**如果在一个回调或页面上返回多个结果?**

系统默认使用taskid(url)来对任务去重，所以在同一个网页上如果返回多个结果，后边的会把之前的结果覆盖掉．

有一种解决方案是使用信息接口生成新的taskid来保证同一个页面上的多个数据能被单独保存，如：

def detail\_page(self, response): for li in response.doc('li'):

self.send\_message(self.project\_name, {

...

}, url=response.url+"#"+li('a.product-sku').text()) def on\_message(self, project, msg): return msg

二、机器学习

（1）fisher判别分析

**原理**

费歇（FISHER）判别思想是投影，使多维问题简化为一维问题来处理。选择一个适当的投影轴,使所有的样品点都投影到这个轴上得到一个投影值。对这个投影轴的方向的要求是：使每一类内的投影值所形成的类内离差尽可能小，而不同类间的投影值所形成的类间离差尽可能大。

（2）决策树

**决策树（Decision Trees ，DTs）**是一种无监督的学习方法，用于分类和回归。它对数据中蕴含的决策规则建模，以预测目标变量的值。

某些情况，例如下面的例子，决策树通过学习模拟一个包含一系列是否判断的正弦曲线。树越深，决策树的规则和拟合越复杂。

决策树的一些优点：

* 易于理解和解释。数可以可视化。
* 几乎不需要数据预处理。其他方法经常需要数据标准化，创建虚拟变量和删除缺失值。决策树还不支持缺失值。
* 使用树的花费（例如预测数据）是训练数据点(data points)数量的对数。
* 可以同时处理数值变量和分类变量。其他方法大都适用于分析一种变量的集合。
* 可以处理多值输出变量问题。
* 使用白盒模型。如果一个情况被观察到，使用逻辑判断容易表示这种规则。相反，如果是黑盒模型（例如人工神经网络），结果会非常难解释。
* 可以使用统计检验检验模型。这样做被认为是提高模型的可行度。
* 即使对真实模型来说，假设无效的情况下，也可以较好的适用。

决策树的一些缺点：

* 决策树学习可能创建一个过于复杂的树，并不能很好的预测数据。也就是过拟合。修剪机制（现在不支持），设置一个叶子节点需要的最小样本数量，或者数的最大深度，可以避免过拟合。
* 决策树可能是不稳定的，因为即使非常小的变异，可能会产生一颗完全不同的树。这个问题通过decision trees with an ensemble来缓解。
* 学习一颗最优的决策树是一个NP-完全问题under several aspects of optimality and even for simple concepts。因此，传统决策树算法基于启发式算法，例如贪婪算法，即每个节点创建最优决策。这些算法不能产生一个全家最优的决策树。对样本和特征随机抽样可以降低整体效果偏差。
* 概念难以学习，因为决策树没有很好的解释他们，例如，XOR, parity or multiplexer problems.
* 如果某些分类占优势，决策树将会创建一棵有偏差的树。因此，建议在训练之前，先抽样使样本均衡。

**session会话**

**NLTK语言处理**

**一、similar**

* 用来识别文章中和搜索词相似的词语，可以用在搜索引擎中的相关度识别功能中。
* text1.similar("monstrous")
* 查询出了text1中与monstrous相关的所有词语

**二、common\_contexts**

* 用来识别2个关键词相似的词语。
* text2.common\_contexts(["monstrous","very"])

**三、generate**

* 用来自动生成文章。
* text3.generate()

**四、len**

* 可以用于判断重复词密度
* from \_\_future\_\_ import division
* len(text3) / len(set(text3))

**五、count**

* 可以用于判断关键词密度。
* text3.count('smote') / len(text3)

**六、nltk.collocations.BigramCollocationFinder**

* 能够在给定的text里找到可能的词组。
* 它的原理是：
* 为每个字建立frequency distribution，即某个字出现的概率。然后从这个概率分布里找到哪一对字的概率类似，从而找到可能的词组。
* 利用这一原理，我们可以根据类别搜集大量的词料库，并对其进行Bigram查找，只要数据量足够大，就能基本准确地自动地建立起我们自己的词库
* 在中文分词时，这个词库的好坏将直接决定搜索的准确性
* 附一个Bigram的实现：
* from nltk.corpus import webtext
* from nltk.corpus import stopwords
* from nltk.collocations import BigramAssocMeasures, BigramCollocationFinder
* filter\_stopwords = lambda x: len(x) < 3 or x in stopwords.words("english")
* words = (w.lower() for w in webtext.words("grail.txt"))
* bcf = BigramCollocationFinder.from\_words(words)
* bcf.apply\_word\_filter(filter\_stopwords)
* print bcf.nbest(BigramAssocMeasures.likelihood\_ratio, 10)

**七、利用概率分布进行关联规则挖掘**

* 电子商务网站上，某个商品页经常会有“购买该商品的用户还购买了”的推荐panel
* 本文利用nltk的概率分布算法FreqDist实现该功能
* 首先，用户购买的信息都存放在数据库里，把这些数据导出来，例如对每个人，都生成如下的tuple
* goods(USERi) = (goodsId1, goodsId2, ..., goodsIdN)
* 表示某个用户i购买了1,2..N种商品
* 之后，需要做个排序，例如用户A购买了(4,5) 2种商品， 用户B购买了(5, 4)两种商品，应该说他们的购买特征是相同的，如果不排序，就会有(4, 5) != (5, 4)
* 第二步，把所有用户购买的商品形成list

goods\_sets = [(goods1, goods2, goodsN), (goods4, goods8), ...]

* 第三步，利用nltk计算goods\_sets概率分布

goods\_fd = nltk.FreqDist(goods\_sets)

* 第四步，结果出来了

找最经常一起购买的商品组合前10位：

* goods\_fd[:10]

找最不经常一起购买的商品组合10个：

* goods\_fd[-10:]

**八、通过nltk的机器学习方法实现论坛垃圾帖的过滤**

**九、Chunking**

实体识别最基本的技术就是chunking，即分块，可以理解为把多个token组成词组。

Noun Phrase Chunking

我们就先以名词词组从chunking为例，即NP-chunking

>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]

>>> grammar = "NP: {<DT>?<JJ>\*<NN>}" #Tag Patterns，定语（0或1个）形容词（任意个）名词（1个）

>>> cp = nltk.RegexpParser(grammar)

>>> result = cp.parse(sentence)

>>> print result

上面的这个方法就是用Regular Expressions来表示tag pattern，从而找到NP-chunking

再给个例子，tag patterns可以加上多条，可以变的更复杂

grammar = r"""NP: {<DT|PP/$>?<JJ>\*<NN>} # chunk determiner/possessive, adjectives and nouns

{<NNP>+} # chunk sequences of proper nouns

"""

cp = nltk.RegexpParser(grammar)

sentence = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"), ("her", "PP$"), ("long", "JJ"), ("golden", "JJ"), ("hair", "NN")]

>>> print cp.parse(sentence)

下面给个例子看看怎么从语料库中找到匹配的词性组合，

>>> cp = nltk.RegexpParser('CHUNK: {<V.\*> <TO> <V.\*>}') ＃找‘动词 to 动词’的组合

>>> brown = nltk.corpus.brown

>>> for sent in brown.tagged\_sents():

... tree = cp.parse(sent)

... for subtree in tree.subtrees():

... if subtree.node == 'CHUNK': print subtree

...

**十、POS tagging词性标注**

>>> text = nltk.word\_tokenize("And now for something completely different")

>>> nltk.pos\_tag(text)

Automatic Tagging

下面就来讲讲各种自动标注的方法，因为tag要根据词的context，所以tag是以sentense为单位的，而不是word为单位，因为如果以词为单位，一个句子的结尾词会影响到下个句子开头词的tag，这样是不合理的，以句子为单位可以避免这样的错误，让context的影响不会越过sentense。

我们就用brown corpus作为例子，

>>> from nltk.corpus import brown

>>> brown\_tagged\_sents = brown.tagged\_sents(categories='news')

>>> brown\_sents = brown.sents(categories='news')

可以分布取出标注过的句子集合， 未标注的句子集合，分别用做标注算法的验证集和测试集。

The Default Tagger

The simplest possible tagger assigns the same tag to each token.

>>> raw = 'I do not like green eggs and ham, I do not like them Sam I am!'

>>> tokens = nltk.word\_tokenize(raw)

>>> default\_tagger = nltk.DefaultTagger('NN')

>>> default\_tagger.tag(tokens)

>>> patterns = [

... (r'.\*ing$', 'VBG'), # gerunds

... (r'.\*ed$', 'VBD'), # simple past

... (r'.\*es$', 'VBZ'), # 3rd singular present

... (r'.\*ould$', 'MD'), # modals

... (r'.\*/'s$', 'NN$'), # possessive nouns

... (r'.\*s$', 'NNS'), # plural nouns

... (r'^-?[0-9]+(.[0-9]+)?$', 'CD'), # cardinal numbers

... (r'.\*', 'NN') # nouns (default)

... ]

>>> regexp\_tagger = nltk.RegexpTagger(patterns)

>>> regexp\_tagger.tag(brown\_sents[3])

[('``', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative', 'NN'), ('handful', 'NN'),

('of', 'NN'), ('such', 'NN'), ('reports', 'NNS'), ('was', 'NNS'), ('received', 'VBD'),

("''", 'NN'), (',', 'NN'), ('the', 'NN'), ('jury', 'NN'), ('said', 'NN'), (',', 'NN'),

('``', 'NN'), ('considering', 'VBG'), ('the', 'NN'), ('widespread', 'NN'), ...]

这个Tagger，进步了一点，就是你可以定义一些正则文法的规则，满足规则就tag成相应的词性，否则还是default

这个方法开始有点实用价值了， 就是通过统计训练corpus里面最常用的词，最有可能出现的词性是什么，来进行词性标注。

>>> fd = nltk.FreqDist(brown.words(categories='news'))

>>> cfd = nltk.ConditionalFreqDist(brown.tagged\_words(categories='news'))

>>> most\_freq\_words = fd.keys()[:100]

>>> likely\_tags = dict((word, cfd[word].max()) for word in most\_freq\_words)

>>> baseline\_tagger = nltk.UnigramTagger(model=likely\_tags)

这段code就是从corpus中取出top 100的词，然后找到这100个词出现次数最多的词性，然后形成likely\_tags的字典，然后将这个字典作为model传个unigramTagger，unigramTagger就是一元的tagger，即不考虑前后context的一种简单的tagger，这个方法有个最大的问题，你只指定了top 100词的词性，那么其他的词怎么办

好，前面的default tagger有用了，baseline\_tagger = nltk.UnigramTagger(model=likely\_tags, backoff=nltk.DefaultTagger('NN'))，这样就可以部分解决这个问题， 不知道的就用default tagger来标注，这个方法的准确性完全取决于这个model的大小，这儿取了top100的词，可能准确性不高，但是随着你取的词的增多，这个准确率会不断提高。

N-Gram Tagging

上面给出的lookup tagger就是用的Unigram tagger， 现在给出Unigram tagger更一般的用法

>>> from nltk.corpus import brown

>>> brown\_tagged\_sents = brown.tagged\_sents(categories='news')

>>> brown\_sents = brown.sents(categories='news')

>>> unigram\_tagger = nltk.UnigramTagger(brown\_tagged\_sents) ＃Training

>>> unigram\_tagger.tag(brown\_sents[2007])

你可以来已标注的语料库对Unigram tagger进行训练

n元就是要考虑context，即考虑前n-1个word的tag，来给当前的word进行tagging

就n元tagger的特例二元tagger作为例子

>>> bigram\_tagger = nltk.BigramTagger(train\_sents)

>>> bigram\_tagger.tag(brown\_sents[2007])

这样有个问题，如果tag的句子中的某个词的context在训练集里面没有，哪怕这个词在训练集中有，也无法对他进行标注，还是要通过backoff来解决这样的问题

>>> t0 = nltk.DefaultTagger('NN')

>>> t1 = nltk.UnigramTagger(train\_sents, backoff=t0)

>>> t2 = nltk.BigramTagger(train\_sents, backoff=t1)

Transformation-Based Tagging

n-gram tagger存在的问题是，model会占用比较大的空间，还有就是在考虑context时，只会考虑前面词的tag，而不会考虑词本身。

而要介绍的这种tagger可以比较好的解决这些问题，用存储rule来代替model，这样可以节省大量的空间，同时在rule中不限制仅考虑tag，也可以考虑word本身。

那么Brill tagging的原理从底下这个例子就可以了解

(1) replace NN with VB when the previous word is TO;

(2) replace TO with IN when the next tag is NNS.

Phrase to increase grants to states for vocational rehabilitation

Unigram TO NN NNS TO NNS IN JJ NN

Rule 1 VB

Rule 2 IN

Output TO VB NNS IN NNS IN JJ NN

第一步用unigram tagger对所有词做一遍tagging，这里面可能有很多不准确的

下面就用rule来纠正第一步中guess错的那些词的tag，最终得到比较准确的tagging

十一、**nltk提供了names的corpus**

>>> from nltk.corpus import names

>>> names.words('male.txt') ＃男性的name的列表

>>> names.words('female.txt') ＃女性的name的列表

这里简单的假设这个名字的性别和最后一个字母相关，那么就把最后一个字母作为每个test case的特征

>>> def gender\_features(word):

... return {'last\_letter': word[-1]}

>>> gender\_features('Shrek')

{'last\_letter': 'k'}

所以就定义如上的特征抽取函数，并用它来生成我们的训练集和测试集

>>> from nltk.corpus import names

>>> import random

>>> names = ([(name, 'male') for name in names.words('male.txt')] +

... [(name, 'female') for name in names.words('female.txt')])

>>> random.shuffle(names) ＃原来的name是按字母排序的，为了达到比较好的训练效果，必须打乱顺序，随机化

>>> featuresets = [(gender\_features(n), g) for (n,g) in names]

>>> train\_set, test\_set = featuresets[500:], featuresets[:500] ＃把特征集一部分作为train集，一部分用来测试

>>> classifier = nltk.NaiveBayesClassifier.train (train\_set) ＃用训练集来训练bayes分类器

>>> classifier.classify (gender\_features('Trinity')) ＃训练完就可以用这个分类器来实际进行分类工作了'female'

用测试集来测试

>>> print nltk.classify.accuracy (classifier, test\_set) ＃用测试集来测试这个分类器，nltk提供accuracy接口

0.758

现在只考虑了最后一个字母这个特征，准确率是75％，显然还有很大的提升空间。

>>> classifier.show\_most\_informative\_features (5) ＃这个接口有意思， 你可以显示出区分度最高的几个

nltk接口很贴心，还考虑到你内存太小，放不下所有的feature集合，提供这个接口来当用到时，实时的计算feature

>>> from nltk.classify import apply\_features

>>> train\_set = apply\_features (gender\_features, names[500:])

>>> test\_set = apply\_features(gender\_features, names[:500])

>>> def gender\_features(word):

... return {'suffix1': word[-1:],

... 'suffix2': word[-2:]}

但是如果把特征增加为，首字母，尾字母，并统计每个字符的出现次数，反而会导致overfitting，测试准确性反而不如之前只考虑尾字母的情况

def gender\_features2(name):

features = {}

features["firstletter"] = name[0].lower()

features["lastletter"] = name[–1].lower()

for letter in 'abcdefghijklmnopqrstuvwxyz':

features["count(%s)" % letter] = name.lower().count(letter)

features["has(%s)" % letter] = (letter in name.lower())

return features

>>> gender\_features2('John')

{'count(j)': 1, 'has(d)': False, 'count(b)': 0, ...}

>>> featuresets = [(gender\_features2(n), g) for (n,g) in names]

>>> train\_set, test\_set = featuresets[500:], featuresets[:500]

>>> classifier = nltk.NaiveBayesClassifier.train(train\_set)

>>> print nltk.classify.accuracy(classifier, test\_set)

比如对于文本分类，可以选取是否包含特征词汇作为文本特征

all\_words = nltk.FreqDist(w.lower() for w in movie\_reviews.words())

word\_features = all\_words.keys()[:2000] ＃找出出现频率较高的特征词，虽然这个找法不太合理

def document\_features(document):

document\_words = set(document)

features = {}

for word in word\_features:

features['contains(%s)' % word] = (word in document\_words)

return features

>>> print document\_features(movie\_reviews.words('pos/cv957\_8737.txt'))

{'contains(waste)': False, 'contains(lot)': False, ...}

对于pos tagging，我们也可以用分类的方法去解决

比如我们可以通过词的后缀来判断它的词性， 这边就以是否包含常见的词的后缀作为特征

>>> def pos\_features(word):

... features = {}

... for suffix in common\_suffixes:

... features['endswith(%s)' % suffix] = word.lower().endswith(suffix)

... return features

当然这个特征选取的比较简单，那么改进一下，根据后缀，并考虑context，即前一个词和词性，一起作为特征，这样考虑就比较全面了。后缀之所以要考虑3种情况，是因为一般表示词性的后缀，最多3个字符，s，er，ing

def pos\_features(sentence, i, history):

features = {"suffix(1)": sentence[i][-1:],

"suffix(2)": sentence[i][-2:],

"suffix(3)": sentence[i][-3:]}

if i == 0:

features["prev-word"] = "<START>"

features["prev-tag"] = "<START>"

else:

features["prev-word"] = sentence[i-1]

features["prev-tag"] = history[i-1] ＃history里面存放了句子里面每个词的词性

return features

那么分类器，除了bayes外，nltk还有decision tree， Maximum Entropy classifier就不具体说了

还有对于大规模数据处理， pure python的分类器的效率相对是比较底下的，所以必须用高效的语言如c语言实现的分类器， NLTK也支持这样的分类器的package，可以参考NLTK的web page。