Tutorial: 抽取公司实体间的股权交易关系

0. 环境准备

0.1. deepdive安装

下载CNdeepdive，运行install.sh，选择1安装deepdive。

配置环境变量，deepdive的可执行文件一般安装在~/local/bin文件夹下。 在~/.bash_profile下添加如下内容并保存：

```
export PATH="/root/local/bin:$PATH"
```

然后执行source ~/.bash_profile设置环境变量。

0.2. postgresql安装

运行

```
 bash <(curl -fsSL git.io/getdeepdive) postgres
```

安装postgresql。

0.3.nlp环境安装

运行nlp_setup.sh，配置中文standford nlp环境。

0.4. 项目框架搭建

建立自己的项目文件夹***transaction***，在本地postgresql中为项目建立数据库，再在项目文件夹下建立数据库配置文件:

```
 echo "postgresql://$USER@$HOSTNAME:5432/db_name" >db.url
```

再在transaction下分别建立输入数据文件夹***input***，脚本文件夹***udf***，用户配置文件app.ddlog，模型配置文件deepdive.conf， 可参照给定的transaction文件夹样例格式。

（PS：transaction文件夹中是已经建立完毕的项目，后面所需的脚本和数据文件都可以直接复制）

deepdive定义了很多自己的语法规则和自动化脚本，导入数据库的过程一般为deepdive do db_name指令，用户通过配置app.ddlog指示数据流。

1. 实验步骤

1.1 . 先验数据导入

我们需要从知识库中获取已知具有交易关系的实体对，来作为训练数据。本项目采用的数据从国泰安数据库（http://www.gtarsc.com）中公司关系-股权交易模块中下载。

(1). 通过匹配有交易的股票代码对和代码-公司对，过滤出存在交易关系的公司对，存入transaction_dbdata.csv中。将csv文件放入input/文件夹下。

(2). 在app.ddlog中定义相应的数据表

```
@source
transaction_dbdata(
```

```
    @key
    company1_name text,
    @key
    company2_name text
).
```

(3). 命令行生成postgresql数据表

```
$ deepdive compile && deepdive do transaction_dbdata
```

- 在执行app.ddlog前，如果有改动，需要先执行deepdive compile编译才能生效
- 对于不依赖于其他表的表格，deepdive会自动去input文件夹下找到同名csv文件，在postgresql里建表导入
- 运行命令时，deepdive会在当前命令行里生成一个执行计划文件，和vi语法一样，审核后使用:wq保存并执行。

1.2. 待抽取文章导入

(1). 准备待抽取的文章（示例使用上市公司公告），命名为articles.csv，放在input文件夹下。

(2). 在app.ddlog中建立对应的articles表。

```
articles(
     id         text,
     content    text
).
```

(3). 同理，执行命令行，导入文章到postgresql中。

```
$ deepdive do articles
```

deepdive可以直接查询数据库数据，用query语句或者deepdive sql "sql语句"进行数据库操作。进行查询id指令，检验导入是否成功：

```
$ deepdive query '?- articles(id, _).'

id
------------
1201835868
1201835869
1201835883
1201835885
1201835927
1201845343
1201835928
1201835930
1201835934
1201841180
:
```

1。3. 用nlp模块进行文本处理

deepdive默认采用standford nlp进行文本处理。输入文本数据，nlp模块将以句子为单位，返回每句的分词、lemma、pos、NER和句法分析的结果，为后续特征抽取做准备。我们将这些结果存入sentences表中。（1).

在app.ddlog文件中定义sentences表，用于存放nlp结果：

```
sentences(
    doc_id         text,
    sentence_index int,
    sentence_text  text,
    tokens         text[],
    lemmas         text[],
   pos_tags       text[],
    ner_tags       text[],
    doc_offsets    int[],
    dep_types      text[],
    dep_tokens     int[]
).
```

(2). 定义NLP处理的函数nlp_markup

```
function nlp_markup over (
    doc_id  text,
    content text
) returns rows like sentences
implementation "udf/nlp_markup.sh" handles tsv lines.
```

- 声明一个ddlog函数，这个函数输入文章的doc_id和content，输出按sentences表的字段格式
- 函数调用udf/nlp_markup.sh调用nlp模块，这里可以自由发挥
- nlp_markup.sh的脚本内容见transaction示例代码中的udf/文件夹，它调用udf/bazzar/parser下的run.sh实现。

注意： 此处需要重新编译nlp代码模块

复制transaction/udf/的目录下的bazzar文件夹到你自己项目的udf/中。这个模块需要重新编译。进入bazzar/parser目录下，执行编译命令：

```
sbt/sbt stage
```

编译完成后会在target中生成可执行文件。

---

(3). 使用如下语法调用nlp_markup函数，从articles表中读取输入，输出存放在sentences表中。

```
sentences += nlp_markup(doc_id, content) :-
articles(doc_id, content).
```

(4). 编译并执行deepdive compile和deepdive do sentences两个命令，生成sentences数据表。

执行以下命令来查询生成结果：

```
deepdive query '
doc_id, index, tokens, ner_tags | 5
?- sentences(doc_id, index, text, tokens, lemmas, pos_tags, ner_tags, _, _, _).
'
```

可以看到id为1201734370文章的前五句的解析结果。

tips: *可以看到sentences给出的plan中包含articles表的执行。plan中前面有冒号的行表示默认已经执行，不*

会重做，否则将要生成。如果articles有更新，需要重新deepdive redo articles或者用deepdive mark todo articles来将articles标记为未执行，这样在生成sentences的过程中就会默认更新articles了。

**注意： 这一步跑的会非常慢，可能需要四五个小时。大家可以减少articles的行数，来缩短时间，完成demo。

1.4. 实体抽取及候选实体对生成

这一步，我们要抽取文本中的候选实体（公司），并生成候选实体对。

(1). 首先在app.ddlog中定义实体数据表：

```
company_mention(
    mention_id     text,
    mention_text   text,
    doc_id         text,
    sentence_index int,
    begin_index    int,
    end_index      int
).
```

每个实体都是表中的一列数据，同时存储了实体在句中的起始位置和结束位置。

(2). 再定义实体抽取的函数：

```
function map_company_mention over (
    doc_id         text,
    sentence_index int,
    tokens         text[],
    ner_tags       text[]
) returns rows like company_mention
implementation "udf/map_company_mention.py" handles tsv lines.
```

- map_company_mention.py见样例。这个脚本遍历每个数据库中的句子，找出连续的NER标记为ORG的序列，再做其它过滤处理，其它脚本也要复制过去。这个脚本是一个生成函数，用yield语句返回输出行。

(3). 然后在app.ddlog中写调用函数，从sentences表中输入，输出到company_mention中。

```
company_mention += map_company_mention(
doc_id, sentence_index, tokens, ner_tags
) :-
sentences(doc_id, sentence_index, _, tokens, _, _, ner_tags, _, _, _).
```

(4). 最后编译并执行：

```
$ deepdive compile && deepdive do company_mention
```

(5). 下面生成实体对，即要预测关系的两个公司。在这一步我们将实体表做笛卡尔积，同时按自定义脚本过滤一些不符合形成交易条件的公司。定义数据表如下：

```
transaction_candidate(
    p1_id   text,
    p1_name text,
    p2_id   text,
    p2_name text
).
```

(6). 统计每个句子的实体数：

```
num_company(doc_id, sentence_index, COUNT(p)) :-
company_mention(p, _, doc_id, sentence_index, _, _).
```

(7). 定义过滤函数：

```
function map_transaction_candidate over (
    p1_id         text,
    p1_name       text,
    p2_id         text,
    p2_name      text
) returns rows like transaction_candidate
implementation "udf/map_transaction_candidate.py" handles tsv lines.
```

(8). 描述函数的调用：

```
transaction_candidate += map_transaction_candidate(p1, p1_name, p2, p2_name) :-
num_company(same_doc, same_sentence, num_p),
company_mention(p1, p1_name, same_doc, same_sentence, p1_begin, _),
company_mention(p2, p2_name, same_doc, same_sentence, p2_begin, _),
num_p < 5,
p1_name != p2_name,
p1_begin != p2_begin.
```

一些简单的过滤操作可以直接通过app.ddlog中的数据库语法执行，比如p1_name != p2_name，过滤掉两个相同实体组成的实体对。

（PS：此处如果报路径错误，请将transform.py中company_full_short.csv的相对路径改为绝对路径。）

(9). 编译并执行：

```
$ deepdive compile && deepdive do transaction_candidate
```

生成候选实体表。

1.5. 特征提取

这一步我们抽取候选实体对的文本特征。

(1). 定义特征表：

```
transaction_feature(
    p1_id   text,
    p2_id   text,
    feature text
```

).

这里的feature列是实体对间一系列文本特征的集合。

(2). 生成feature表需要的输入为实体对表和文本表，输入和输出属性在app.ddlog中定义如下：

```
 function extract_transaction_features over (
    p1_id          text,
    p2_id          text,
    p1_begin_index int,
    p1_end_index   int,
```

```
    p2_begin_index int,
    p2_end_index   int,
    doc_id         text,
    sent_index     int,
    tokens         text[],
    lemmas         text[],
    pos_tags       text[],
    ner_tags       text[],
    dep_types      text[],
    dep_tokens     int[]
) returns rows like transaction_feature
implementation "udf/extract_transaction_features.py" handles tsv lines.
```

- 函数调用extract_transaction_features.py来抽取特征。这里调用了deepdive自带的ddlib库，得到各种POS/NER/词序列的窗口特征。此处也可以自定义特征。

(3).把sentences表和mention表做join，得到的结果输入函数，输出到transaction_feature表中。

```
transaction_feature += extract_transaction_features(
p1_id, p2_id, p1_begin_index, p1_end_index, p2_begin_index, p2_end_index,
doc_id, sent_index, tokens, lemmas, pos_tags, ner_tags, dep_types, dep_tokens
) :-
company_mention(p1_id, _, doc_id, sent_index, p1_begin_index, p1_end_index),
company_mention(p2_id, _, doc_id, sent_index, p2_begin_index, p2_end_index),
sentences(doc_id, sent_index, _, tokens, lemmas, pos_tags, ner_tags, _, dep_types,
dep_tokens).
```

(4). 然后编译并执行，生成特征数据库：

```
$ deepdive compile && deepdive do transaction_feature
```

执行如下语句，查看生成结果：

```
deepdive query '| 20 ?- transaction_feature(_, _, feature).'
```

---

feature
————————————————————————————————————

WORD_SEQ_[郴州市 城市 建设 投资 发展 集团 有限 公司]

LEMMA_SEQ_[郴州市 城市 建设 投资 发展 集团 有限 公司]

NER_SEQ_[ORG ORG ORG ORG ORG ORG ORG ORG]

POS_SEQ_[NR NN NN NN NN NN JJ NN]

W_LEMMA_L_1_R_1_[为]_[提供]

W_NER_L_1_R_1_[O]_[O]

W_LEMMA_L_1_R_2_[为]_[提供 担保]

W_NER_L_1_R_2_[O]_[O O]

W_LEMMA_L_1_R_3_[为]_[提供 担保 公告]

W_NER_L_1_R_3_[O]_[O O O]

W_LEMMA_L_2_R_1_[公司 为]_[提供]

W_NER_L_2_R_1_[ORG O]_[O]

W_LEMMA_L_2_R_2_[公司 为]_[提供 担保]

W_NER_L_2_R_2_[ORG O]_[O O]

W_LEMMA_L_2_R_3_[公司 为]_[提供 担保 公告]

W_NER_L_2_R_3_[ORG O]_[O O O]

W_LEMMA_L_3_R_1_[有限 公司 为]_[提供]

W_NER_L_3_R_1_[ORG ORG O]_[O]

W_LEMMA_L_3_R_2_[有限 公司 为]_[提供 担保]

W_NER_L_3_R_2_[ORG ORG O]_[O O]

(20 rows)

:

现在，我们已经有了想要判定关系的实体对和它们的特征集合。

1.6. 样本打标

这一步，我们希望在候选实体对中标出部分正负例。

- 利用已知的实体对和候选实体对关联
- 利用规则打部分正负标签

(1). 首先在app.ddlog里定义transaction_label表，存储监督数据：

```
@extraction
transaction_label(
    @key
    @references(relation="has_transaction", column="p1_id", alias="has_transaction")
    p1_id   text,
    @key
    @references(relation="has_transaction", column="p2_id", alias="has_transaction")
    p2_id   text,
    @navigable
    label   int,
    @navigable
    rule_id text
).
```

rule_id代表在标记决定相关性的规则名称。label为正值表示正相关，负值表示负相关。绝对值越大，相关性越大。

(2). 初始化定义，复制transaction_candidate表，label均定义为零。

```
 transaction_label(p1, p2, 0, NULL) :- transaction_candidate(p1, _, p2, _).
```

(3).将前面准备的db数据导入transaction_label表中，rule*id标记为"from*\dbdata"。因为国泰安的数据比较官方，可以基于较高的权重，这里设为3。在app.ddlog中定义如下：

```
transaction_label(p1,p2, 3, "from_dbdata") :-
    transaction_candidate(p1, p1_name, p2, p2_name), transaction_dbdata(n1, n2),
```

```
    [ lower(n1) = lower(p1_name), lower(n2) = lower(p2_name) ;
      lower(n2) = lower(p1_name), lower(n1) = lower(p2_name) ].
```

(4). 如果只利用下载的实体对，可能和未知文本中提取的实体对重合度较小，不利于特征参数推导。因此可以通过一些逻辑规则，对未知文本进行预标记。

```
 function supervise over (
    p1_id text, p1_begin int, p1_end int,
    p2_id text, p2_begin int, p2_end int,
    doc_id          text,
    sentence_index int,
    sentence_text  text,
    tokens          text[],
    lemmas          text[],
    pos_tags        text[],
    ner_tags        text[],
    dep_types       text[],
    dep_tokens      int[]
) returns (
    p1_id text, p2_id text, label int, rule_id text
)
implementation "udf/supervise_transaction.py" handles tsv lines.
```

- 输入候选实体对的关联文本，定义打标函数
- 函数调用udf/supervise_transaction.py，规则名称和所占的权重定义在脚本中。在app.ddlog中定义标记函数。

(5). 调用标记函数，将规则抽到的数据写入transaction_label表中。

```
transaction_label += supervise(
p1_id, p1_begin, p1_end,
p2_id, p2_begin, p2_end,
doc_id, sentence_index, sentence_text,
tokens, lemmas, pos_tags, ner_tags, dep_types, dep_token_indexes
) :-
transaction_candidate(p1_id, _, p2_id, _),
company_mention(p1_id, p1_text, doc_id, sentence_index, p1_begin, p1_end),
company_mention(p2_id, p2_text, _, _, p2_begin, p2_end),
sentences(
    doc_id, sentence_index, sentence_text,
    tokens, lemmas, pos_tags, ner_tags, _, dep_types, dep_token_indexes
).
```

(6). 不同的规则可能覆盖了相同的实体对，从未给出不同甚至相反的label。建立transaction*label*resolved表，统一实体对间的label。利用label求和，在多条规则和知识库标记的结果中，为每对实体做vote。

```
 transaction_label_resolved(p1_id, p2_id, SUM(vote)) :-transaction_label(p1_id, p2_id,
vote, rule_id).
```

(7). 执行以下命令，得到最终标签。

```
$ deepdive do transaction_label_resolved
```

2. 模型构建

通过1的步骤，我们已经得到了所有前期需要准备的数据。下面可以构建模型了。

2.1 变量表定义

(1). 定义最终存储的表格，『? 』表示此表是用户模式下的变量表，即需要推导关系的表。这里我们预测的是公司间是狗存在交易关系。

```
@extraction
has_transaction?(
    p1_id text,
    p2_id text
).
```

(2). 根据打标的结果，灌入已知的变量

```
has_transaction(p1_id, p2_id) = if l > 0 then TRUE
                  else if l < 0 then FALSE
                  else NULL end :- transaction_label_resolved(p1_id, p2_id, l).
```

此时变量表中的部分变量label已知，成为了先验变量。

(3). 最后编译执行决策表：

```
$ deepdive compile && deepdive do has_transaction
```

2.2 因子图构建

(1). 指定特征

将每一对has_transaction中的实体对和特征表连接起来，通过特征factor的连接，全局学习这些特征的权重。在app.ddlog中定义：

```
@weight(f)
has_transaction(p1_id, p2_id) :-
    transaction_candidate(p1_id, _, p2_id, _),
    transaction_feature(p1_id, p2_id, f).
```

(2). 指定变量间的依赖性

我们可以指定两张变量表间遵守的规则，并给这个规则以权重。比如c1和c2有交易，可以推出c2和c1也有交易。这是一条可以确保的定理，因此给予较高权重：

```
 @weight(3.0)
 has_transaction(p1_id, p2_id) => has_transaction(p2_id, p1_id) :-
    transaction_candidate(p1_id, _, p2_id, _).
```

变量表间的依赖性使得deepdive很好地支持了多关系下的抽取。

(3). 最后，编译，并生成最终的概率模型：

```
$ deepdive compile && deepdive do probabilities
```

查看我们预测的公司间交易关系概率：

```
$ deepdive sql "SELECT p1_id, p2_id, expectation FROM has_transaction_label_inference
ORDER BY random() LIMIT 20"
```

p1_id	p2_id	expectation
1201778739_118_170_171	1201778739_118_54_60	0
1201778739_54_30_35	1201778739_54_8_11	0.035
1201759193_1_26_31	1201759193_1_43_48	0.07
1201766319_65_331_331	1201766319_65_159_163	0
1201761624_17_30_35	1201761624_17_9_14	0.188
1201743500_3_0_5	1201743500_3_8_14	0.347
1201789764_3_16_21	1201789764_3_75_76	0
1201778739_120_26_27	1201778739_120_29_30	0.003
1201752964_3_21_21	1201752964_3_5_10	0.133
1201775403_1_83_88	1201775403_1_3_6	0
1201778793_15_5_6	1201778793_15_17_18	0.984
1201773262_2_85_88	1201773262_2_99_99	0.043
1201734457_24_19_20	1201734457_24_28_29	0.081
1201752964_22_48_50	1201752964_22_9_10	0.013
1201759216_5_38_44	1201759216_5_55_56	0.305
1201755097_4_18_22	1201755097_4_52_57	1
1201750746_2_0_5	1201750746_2_20_26	0.034
1201759186_4_45_46	1201759186_4_41_43	0.005
1201734457_18_7_11	1201734457_18_13_18	0.964
1201759263_36_18_20	1201759263_36_33_36	0.002

至此，我们的交易关系抽取就基本完成了。更多详细说明请见http://deepdive.stanford.edu