THIS EXAMPLE IS STALE. NEEDS REVAMP!

Text chunking example

Introduction

In this document, we will describe an example application of text chunking using DeepDive to demonstrate how to use *categorical factors* with *categorical variables*. This example assumes a working installation of DeepDive and basic knowledge of how to build an application in DeepDive. Please go through the tutorial with the spouse example application before preceding.

Text chunking consists of dividing a text in syntactically correlated parts of words. For example, the following sentence:

> He reckons the current account deficit will narrow to only # 1.8 billion in September .

can be divided as follows:

```
[NP He ] [VP reckons ] [NP the current account deficit ] [VP will narrow ] [PP to ] [NP only # 1.8 billion ] [PP in ] [NP September ] .
```

Text chunking is an intermediate step towards full parsing. It was the shared task for CoNLL-2000. Training and test data for this task is derived from the Wall Street Journal corpus (WSJ), which includes words, part-of-speech tags, and chunking tags.

In the example, we will predicate chunk label for each word. We include three inference rules, corresponding to *logistic regression*, *linear-chain conditional random field* (CRF), and *skip-chain conditional random field*. The features and rules we use are very simple, just to illustrate how to use categorical variables and categorical factors in DeepDive to build applications.

Running the example

The complete example is under the `examples/chunking` directory.

```
cd examples/chunking/
```

The structure of this directory is as follows:

- `input/` contains training and testing data.
- `udf/` contains extractor for extracting training data and features.
- `result/` contains evaluation scripts and sample results.

To run this example, use the following command:

```
deepdive compile && deepdive run
```

Then run the following to evaluate the results:

```
result/eval.sh
```

Example walkthrough

The application performs the following high-level steps:

1. Data preprocessing: load training and test data into database.
2. Feature extraction: extract surrounding words and their part-of-speech tags as features.
3. Statistical inference and learning.
4. Evaluation of the results.

1. Data preprocessing

The train and test data consist of words, their part-of-speech tag and the chunk tags as derived from the WSJ corpus. The raw data is first copied into table `words_raw` by `input/init_words_raw.sh` script. Then it is processed to convert the chunk labels to integer indexes, based on predefined mappings in the `tags` table. This process is defined in `app.ddlog` using the following code:

```
words(sent_id, word_id, word, pos, true_tag, tag_id) :-
  words_raw(sent_id, word_id, word, pos, true_tag),
  tags(tag, tag_id),
  if true_tag = "B-UCP" then ""
  else if true_tag = "I-UCP" then ""
  else if strpos(true_tag, "-") > 0 then
     split_part(true_tag, "-", 2)
  else if true_tag = "O" then "O"
  else ""
```

```
  end = tag.
```

The input table words_raw looks like

```
 word_id |    word     | pos | tag  | id
---------+------------+-----+------+----
       1 | Confidence | NN  | B-NP |
[...]
```

The output table words looks like

```
 sent_id | word_id |    word     | pos | true_tag | tag  | id
---------+---------+------------+-----+----------+------+----
       1 |       1 | Confidence | NN  | B-NP     |    0 |  0
[...]
```

2. Feature extraction

To predict chunking label, we need to add features. We use three simple features: the word itself, its part-of-speech tag, and the part-of-speech tag of its previous word. We add an extractor in app.ddlog:

```
function ext_features
  over (word_id1 bigint, word1 text, pos1 text, word2 text, pos
2 text)
  returns rows like word_features
  implementation "udf/ext_features.py" handles tsv lines.


word_features +=
  ext_features(word_id1, word1, pos1, word2, pos2) :-
  words(sent_id, word_id1, word1, pos1, _, _),
  words(sent_id, word_id2, word2, pos2, _, _),
  [word_id1 = word_id2 + 1],
  word1 IS NOT NULL.
```

where the input is generating 2-grams from words table, which looks like:

```
 w1.word_id | w1.word | w1.pos | w2.word | w2.pos
-----------+---------+--------+---------+--------
```

```
        15 | figures | NNS     | trade    | NN
[...]
```

The output will look like:

```
 word_id |    feature     | id
---------+--------------+----
      15 | word=figures |
      15 | pos=NNS      |
      15 | prev_pos=NN  |
[...]
```

The user-defined function can be in `udf/ext_features.py`.

3. Statistical learning and inference

We will predicate the chunk tag for each word, which corresponds to `tag` column of `words` table. The variables are declared in `app.ddlog`:

```
tag?(word_id bigint) Categorical(13).
```

Here, we have 13 types of chunk tags `NP, VP, PP, ADJP, ADVP, SBAR, O, PRT, CONJP, INTJ, LST, B, null` according to CoNLL-2000 task description. We have three rules, logistic regression, linear-chain CRF, and skip-chain CRF. The logistic regression rule is:

```
@weight(f)
tag(word_id) :- word_features(word_id, f).
```

To express conditional random field, just use the `Multinomial` factor to link variables that could interact with each other. For more information about CRF, see [this tutorial on CRF](). The following rule links labels of neighboring words:

```
@weight("?")
Multinomial(tag(word_id_1), tag(word_id_2)) :-
  words(_, word_id_1, _, _, _, _),
  words(_, word_id_2, _, _, _, _),
  word_id_2=word_id_1+1.
```

It is similar with skip-chain CRF, where we have skip edges that link labels of identical words.

```
@weight("?")
Multinomial(tag(word_id_1), tag(word_id_2)) :-
```

```
  words(sent_id, word_id_1, word, _, _, tag),
  words(sent_id, word_id_2, word, _, _, _),
  tag IS NOT NULL,
  word_id_1<word_id_2.
```

We also specify the holdout variables according to task description about training and test data in `deepdive.conf`.

```
# Specify a holdout fraction
deepdive.calibration.holdout_query: """
    INSERT INTO dd_graph_variables_holdout(variable_id)
    SELECT dd_id
    FROM dd_variables_chunk
    WHERE word_id > 220663
"""
#deepdive.sampler.sampler_cmd: "numbskull"
deepdive.sampler.sampler_args: "-l 100 -i 100 --sample_evidence
"
```

4. Evaluation results

Running the following script will give the evaluation results.

```
result/eval.sh
```

Below are the results for using different rules. We can see that by adding CRF rules, we get better results both for precision and recall.

Logistic regression

```
  processed 47377 tokens with 23852 phrases; found: 23642 phras
es; correct: 19156.
  accuracy:  89.56%; precision:  81.03%; recall:  80.31%; FB1:  
80.67
              ADJP: precision:  50.40%; recall:  42.92%; FB1:   4
6.36  373
```

```
                 ADVP: precision:  69.21%; recall:  71.13%; FB1:  7
0.16  890

                CONJP: precision:   0.00%; recall:   0.00%; FB1:
0.00  13

                 INTJ: precision: 100.00%; recall:  50.00%; FB1:  6
6.67  1

                  LST: precision:   0.00%; recall:   0.00%; FB1:
0.00  0

                   NP: precision:  79.88%; recall:  77.52%; FB1:  7
8.68  12055

                   PP: precision:  90.51%; recall:  89.59%; FB1:  9
0.04  4762

                  PRT: precision:  66.39%; recall:  76.42%; FB1:  7
1.05  122

                 SBAR: precision:  83.51%; recall:  71.96%; FB1:  7
7.31  461

                   VP: precision:  79.48%; recall:  84.71%; FB1:  8
2.01  4965
```

LR + linear-chain CRF

```
  processed 47377 tokens with 23852 phrases; found: 22996 phras
es; correct: 19746.

  accuracy:  91.58%; precision:  85.87%; recall:  82.79%; FB1:
84.30

                     : precision:   0.00%; recall:   0.00%; FB1:
0.00  1

                 ADJP: precision:  75.74%; recall:  69.86%; FB1:  7
2.68  404

                 ADVP: precision:  76.47%; recall:  73.56%; FB1:  7
4.99  833

                CONJP: precision:  25.00%; recall:  22.22%; FB1:  2
3.53  8

                 INTJ: precision:  50.00%; recall:  50.00%; FB1:  5
0.00  2
```

```
                 LST: precision:   0.00%; recall:   0.00%; FB1:
0.00  0

                  NP: precision:  82.22%; recall:  77.19%; FB1:  7
9.63  11662

                  PP: precision:  93.43%; recall:  94.26%; FB1:  9
3.84  4854

                 PRT: precision:  66.67%; recall:  69.81%; FB1:  6
8.20  111

                SBAR: precision:  84.93%; recall:  74.77%; FB1:  7
9.52  471

                  VP: precision:  90.37%; recall:  90.21%; FB1:  9
0.29  4650
```

LR + linear-chain CRF + skip-chain CRF

```
  processed 47377 tokens with 23852 phrases; found: 22950 phras
es; correct: 19794.

  accuracy:  91.79%; precision:  86.25%; recall:  82.99%; FB1:
84.59

                    : precision:   0.00%; recall:   0.00%; FB1:
0.00  1

                ADJP: precision:  75.25%; recall:  68.72%; FB1:  7
1.84  400

                ADVP: precision:  76.29%; recall:  73.56%; FB1:  7
4.90  835

               CONJP: precision:  30.00%; recall:  33.33%; FB1:  3
1.58  10

                INTJ: precision: 100.00%; recall:  50.00%; FB1:  6
6.67  1

                 LST: precision:   0.00%; recall:   0.00%; FB1:
0.00  0

                  NP: precision:  82.96%; recall:  77.54%; FB1:  8
0.16  11611

                  PP: precision:  93.70%; recall:  94.30%; FB1:  9
4.00  4842
```

```
                 PRT: precision:  66.67%; recall:  69.81%; FB1:  6
8.20  111
                SBAR: precision:  83.37%; recall:  74.95%; FB1:  7
8.94  481
                  VP: precision:  90.34%; recall:  90.34%; FB1:  9
0.34  4658
```