Snorkel Workshop: Extracting Spouse Relations from the News

Part 2: Writing Labeling Functions

In Snorkel, our primary interface through which we provide training signal to the end extraction model we are training is by writing **labeling functions (LFs)** (as opposed to hand-labeling massive training sets). We'll go through some examples for our spouse extraction task below.

A labeling function isn't anything special. It's just a Python function that accepts a `Candidate` as the input argument and returns `1` if it says the `Candidate` should be marked as true, `-1` if it says the `Candidate` should be marked as false, and `0` if it doesn't know how to vote and abstains. In practice, many labeling functions are unipolar: it labels only `1` s and `0` s, or it labels only `-1` s and `0` s.

Recall that our goal is to ultimately train a high-performance classification model that predicts which of our `Candidate` s are true mentions of spouse relations. It turns out that we can do this by writing potentially low-quality labeling functions!

In [ ]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline

import re
import sys
import numpy as np

# Connect to the database backend and initalize a Snorkel session
from lib.init import *
from lib.scoring import *
from lib.lf_factories import *

from snorkel.lf_helpers import test_LF
from snorkel.annotations import load_gold_labels
from snorkel.lf_helpers import (
    get_left_tokens, get_right_tokens, get_between_tokens,
    get_text_between, get_tagged_text,
)

# initialize our candidate type definition
Spouse = candidate_subclass('Spouse', ['person1', 'person2'])
```

I. Background

A. Preprocessing the Database

In a real application, there is a lot of data preparation, parsing, and database loading that needs to be completed before we dive into writing labeling functions. Here we've pre-generated a database instance for you. All *candidates* and *gold labels* (i.e., human-generated labels) are queried from this database for use in the the tutorial.

See our preprocessing tutorial Workshop 5 Advanced Preprocessing for more details on how this database is built.

B. Using a *Development Set* of Human-labeled Data

In our setting, we will use the phrase *development set* to refer to a set of examples (here, a subset of our training set) which we label by hand and use to help us develop and refine labeling functions. Unlike the *test set*, which we do not look at and use for final evaluation, we can inspect the development set while writing labeling functions. This is a list of `{-1,1}` labels.

In [ ]:
```
L_gold_dev = load_gold_labels(session, annotator_name='gold', split=1)
```

C. Data Exploration

How do we come up with good keywords and patterns to encode as labeling functions? One way is to manually explore our training data. Here we load a subset of our training candidates into a `SentenceNgramViewer` object to examine candidates in their parent context. Our goal is to build an intuition for patterns and keywords that are predictive of a candidate's true label.

In [ ]:
```
from snorkel.viewer import SentenceNgramViewer

# load our list of training & development candidates
train_cands = session.query(Candidate).filter(Candidate.split == 0).all()
dev_cands   = session.query(Candidate).filter(Candidate.split == 1).all()

SentenceNgramViewer(train_cands[0:500], session, n_per_page=1)
```

D. Labeling Function Metrics

1. Coverage

One simple metric we can compute quickly is our *coverage*, the number of candidates labeled by our LF, on our training set (or any other set).

2. Precision / Recall / F1

If we have gold labeled data, we can also compute standard precision, recall, and F1 metrics for the output of a single labeling function. These metrics are computed over 4 *error buckets*: *True Positives* (tp), *False Positives* (fp), *True Negatives* (tn), and *False Negatives* (fn).

$$precision = \frac{tp}{(tp + fp)}$$

$$recall = \frac{tp}{(tp + fn)}$$

$$F1 = 2 \cdot \frac{(precision \cdot recall)}{(precision + recall)}$$

II. Labeling Functions

A. Pattern Matching Labeling Functions

One powerful form of labeling function design is defining sets of keywords or regular expressions that, as a human labeler, you know are correlated with the true label. In the terminology of Bayesian inference, this can be thought of as defining a *prior* over your word features.

For example, we could define a dictionary of terms that occur between person names in a candidate. One simple dictionary of terms indicating a true relation could be:

```
marriage = {'husband', 'wife'}
```

We can then write a labeling function that checks for a match with these terms in the text that occurs between person names.

```
def LF_marriage_terms_between(c):
    return 1 if len(marriage.intersection(get_between_tokens(c))) > 0 else 0
```

The idea is that we can easily create dictionaries that encode themes or categories descibing all kinds of relationships between 2 people and then use these objects to *weakly supervise* our classification task.

```
other_relationship = {'boyfriend', 'girlfriend'}
```

IMPORTANT Good labeling functions manage a trade-off between high coverage and high precision. When constructing your dictionaries, think about building larger, noiser sets of terms instead of relying on 1 or 2 keywords. Sometimes a single word can be very predictive (e.g., `ex-wife`) but it's almost always better to define something more general, such as a regular expression pattern capturing *any* string with the `ex-` prefix.

1. Labeling Function Factories

The above is a reasonable way to write labeling functions. However, this type of design pattern is so common that we rely on another abstraction to help us build LFs more quickly: *labeling function factories*. Factories accept simple inputs, like dictionaries or a set of regular expressions, and automatically builds labeling functions for you.

The `MatchTerms` and `MatchRegex` factories require a few parameter definitions to setup:

```
name:    a string that describes the category of terms/regular expressons
label:   patterns correlate with a True or False label (1 or -1)
search:  search a specific part of the sentence ('left'|'right'|'between'|'sentence')
window:  the length of tokens to match against for ('left'|'right') search spaces
```

2. Term Matching Factory

We illustrate below how you can use the `MatchTerms` factory to create and test an LF on training candidates. When examining candidates in the `SentenceNgramViewer`, notice that husband or wife always occurs between person names. That is the supervision signal encoded by this LF!

In [ ]:
```
marriage  = {'husband', 'wife'}

# we'll initialize our LFG and test its coverage on training candidates
LF_marriage = MatchTerms(name='marriage', terms=marriage, label=1, search='between').lf()

# what candidates are covered by this LF?
labeled = coverage(session, LF_marriage, split=0)

# now let's view what this LF labeled
SentenceNgramViewer(labeled, session, n_per_page=1)
```

Viewing Error Buckets

If we have gold labeled data, we can evaluate formal metrics. It's useful to view specific errors for a given LF input in the `SentenceNgramViewer`.

Below, we'll compute our empirical scores using human-labeled development set data and then look at any false positive matches by our `LF_marriage` LF. We can see below from our scores that this LF isn't very accurate -- only 36% precision!

In [ ]:
```
tp, fp, tn, fn = error_analysis(session, LF_marriage, split=1, gold=L_gold_dev)

# now let's view what this LF labeled
SentenceNgramViewer(fp, session, n_per_page=1)
```

Other Search Contexts

We can also search other sentence contexts, such as a window of text to the left or right of our candidate spans.

In [ ]:
```
other_relationship  = {'boyfriend', 'girlfriend'}

LF_other_relationship = MatchTerms(name='other_relationship', terms=other_relationship,
                                   label=-1, search='left', window=1).lf()
labeled = coverage(session, LF_other_relationship, split=1)

# now let's view what this LF labeled
SentenceNgramViewer(labeled, session, n_per_page=1)
```

4. Regular Expression Factory

Sometimes we want to express more generic textual patterns to match against candidates. Perhaps we want to match a specific phrase like 'power couple' or look for modifier prefixes like 'ex' wife, husband, etc.

We can generate this supervision in the same way as above using sets of regular expressions -- a formal language for string matching.

In [ ]:
```
exes_rgxs = {' ex[- ](husband|wife)'}

LF_exes = MatchRegex(name='exes', rgxs=exes_rgxs, label=-1, search='between').lf()
labeled = coverage(session, LF_exes, split=1)

# now let's view what this LF labeled
SentenceNgramViewer(labeled, session, n_per_page=1)
```

B. Distant Supervision Labeling Functions

In addition to using factories that encode pattern matching heuristics, we can also write labeling functions that *distantly supervise* examples. Here, we'll load in a list of known spouse pairs and check to see if the candidate pair matches one of these.

DBpedia http://wiki.dbpedia.org/ Out database of known spouses comes from DBpedia, which is a community-driven resource similar to Wikipedia but for curating structured data. We'll use a preprocessed snapshot as our knowledge base for all labeling function development.

We can look at some of the example entries from DBPedia and use them in a simple distant supervision labeling function.

In [ ]:
```
from lib.dbpedia import known_spouses

list(known_spouses)[0:5]
```

In [ ]:
```
LF_distant_supervision = DistantSupervision("dbpedia", kb=known_spouses).lf()
labeled = coverage(session, LF_distant_supervision, split=1)

# score out LF against dev set labels
score(session, LF_distant_supervision, split=1, gold=L_gold_dev)

SentenceNgramViewer(labeled, session, n_per_page=1)
```

C. Writing Custom Labeling Functions

The strength of LFs is that you can write any arbitrary function and use it to supervise a classification task. This approach can combine many of the same strategies discussed above or encode other information.

For example, we observe that when mentions of person names occur far apart in a sentence, this is a good indicator that the candidate's label is False.

In [ ]:
```
def LF_too_far_apart(c):
    """Person mentions occur at a distance > 50 words"""
    return -1 if len(list(get_between_tokens(c))) > 50 else 0
```

labeled = coverage(session, LF_too_far_apart, split=1) score(session, LF_too_far_apart, split=1, gold=L_gold_dev)

D. Composing Labeling Functions

Another useful technique for writing LFs is composing multiple, weaker LFs together. For example, our `LF_marriage` example above has low precision. Instead of modifying `LF_marriage`, we'll compose it with our `LF_too_far_apart` from above.

```
LF_marriage                              TP: 63 | FP: 114
LF_marriage AND NOT LF_too_far_apart     TP: 60 | FP: 86
```

We missed 3 true candidates, but we cut our false positive rate by 28 candidates!

In [ ]:
```
def LF_marriage_and_too_far_apart(c):
    return 1 if LF_too_far_apart(c) != -1 and LF_marriage(c) == 1 else 0

LF_marriage_and_not_same_person = lambda c: LF_too_far_apart(c) != -1 and LF_marriage(c)

score(session, LF_marriage_and_too_far_apart, split=1, gold=L_gold_dev)
```

VI. Development Sandbox

A. Writing Your Own Labeling Functions

Using the information above, write your own labeling functions for this task.

In [ ]:
```
#
# PLACE YOUR LFs HERE
#
```

B. Applying Labeling Functions

Next, we need to actually run the LFs over **all** of our training candidates, producing a set of `Labels` and `LabelKeys` (just the names of the LFs) in the database. We'll do this using the `LabelAnnotator` class, a UDF which we will again run with `UDFRunner`.

1. Preparing your Labeling Functions

First we put all our labeling functions into list:

In [ ]:
```
LFs = [

    # Place your lf function variable names here
    # NOTE: Below are the demo LFs- for demonstrating the syntax only, and will result in a poor score!
    # Add your LFs to increase the performace!!
    LF_marriage,
    LF_other_relationship,
    LF_exes,
    LF_distant_supervision,
    LF_too_far_apart,
    LF_marriage_and_too_far_apart
]
```

Then we setup the label annotator class:

In [ ]:
```
from snorkel.annotations import LabelAnnotator
labeler = LabelAnnotator(lfs=LFs)
```

2. Generating the Label Matrix

In [ ]:
```
np.random.seed(1701)

%time L_train = labeler.apply(split=0, parallelism=1)
print(L_train.shape)

%time L_dev = labeler.apply_existing(split=1, parallelism=1)
print(L_dev.shape)
```

In [ ]:
```
L_train.lf_stats(session)
```

3. Label Matrix Empirical Accuracies

If we have a small set of human-labeled data

In [ ]:
```
L_dev.lf_stats(session, labels=L_gold_dev.toarray().ravel())
```

3. Iterating on Labeling Function Design

When writing labeling functions, you will want to iterate on the process outlined above several times. You should focus on tuning individual LFs, based on emprical accuracy metrics, and adding new LFs to improve coverage.