Intro to Snorkel: Extracting Spouse Relations from the News

Part II: Generating *and modeling* noisy training labels

In this part of the tutorial, we will write **labeling functions** which express various heuristics, patterns, and *weak supervision* strategies to label our data.

In most real-world settings, hand-labeled training data is prohibitively expensive and slow to collect. A common scenario, though, is to have access to tons of *unlabeled* training data, and have some idea of how to label it programmatically. For example:

- We may be able to think of text patterns that would indicate two people mentioned in a sentence are married, such as seeing the word "spouse" between the mentions.
- We may have access to an external *knowledge base (KB)* that lists some known pairs of married people, and can use these to heuristically label some subset of our data.

Our labeling functions will capture these types of strategies. We know that these labeling functions will not be perfect, and some may be quite low-quality, so we will *model* their accuracies with a generative model, which Snorkel will help us easily apply.

This will ultimately produce a single set of **noise-aware training labels**, which we will then use to train an end extraction model in the next notebook. For more technical details of this overall approach, see our NIPS 2016 paper.

In [1]:

```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import os

# TO USE A DATABASE OTHER THAN SQLITE, USE THIS LINE
# Note that this is necessary for parallel execution amongst other things...
# os.environ['SNORKELDB'] = 'postgres:///snorkel-intro'

import numpy as np
from snorkel import SnorkelSession
session = SnorkelSession()
```

We repeat our definition of the `Spouse` `Candidate` subclass from Parts II and III.

In [2]:

```
from snorkel.models import candidate_subclass

Spouse = candidate_subclass('Spouse', ['person1', 'person2'])
```

Using a labeled *development set*

In our setting here, we will use the phrase "development set" to refer to a *small* 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.

In our case, we already loaded existing labels for a development set (`split` 1), so we can load them again now:

In [3]:

```
from snorkel.annotations import load_gold_labels

L_gold_dev = load_gold_labels(session, annotator_name='gold', split=1)
```

Creating and Modeling a Noisy Training Set

Our biggest step in the data programming pipeline is the creation - *and modeling* - of a noisy training set. We'll approach this in three main steps:

1. **Creating labeling functions (LFs):** This is where most of our development time would actually go into if this were a real application. Labeling functions encode our heuristics and weak supervision signals to generate (noisy) labels for our training candidates.

2. **Applying the LFs:** Here, we actually use them to label our candidates!

3. **Training a generative model of our training set:** Here we learn a model over our LFs, learning their respective accuracies automatically. This will allow us to combine them into a single, higher-quality label set.

We'll also add some detail on how to go about *developing labeling functions* and then *debugging our model* of them to improve performance.

1. Creating Labeling Functions

a LF: { 1 Positive
0 abstain
-1 Negative }

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 is just a Python function that accepts a `Candidate` and returns `1` to mark the `Candidate` as true, `-1` to mark the `Candidate` as false, and `0` to abstain from labeling the `Candidate` (note that the non-binary classification setting is covered in the advanced tutorials!).

In the next stages of the Snorkel pipeline, we'll train a model to learn the accuracies of the labeling functions and reweight them accordingly, and then use them to train a downstream model. It turns out by doing this, we can get high-quality models even with lower-quality labeling functions. So they don't need to be perfect! Now on to writing some:

In [4]:

```
import re
from snorkel.lf_helpers import (
    get_left_tokens, get_right_tokens, get_between_tokens,
    get_text_between, get_tagged_text,
)
```

Pattern-based LFs

These LFs express some common sense text patterns which indicate that a person pair might be married. For example, `LF_husband_wife` looks for words in `spouses` between the person mentions, and `LF_same_last_name` checks to see if the two people have the same last name (but aren't the same whole name).

In [5]:

```
spouses = {'spouse', 'wife', 'husband', 'ex-wife', 'ex-husband'}
family = {'father', 'mother', 'sister', 'brother', 'son', 'daughter',
              'grandfather', 'grandmother', 'uncle', 'aunt', 'cousin'}
family = family | {f + '-in-law' for f in family}
other = {'boyfriend', 'girlfriend' 'boss', 'employee', 'secretary', 'co-worker'}

# Helper function to get last name
def last_name(s):
    name_parts = s.split(' ')
    return name_parts[-1] if len(name_parts) > 1 else None

def LF_husband_wife(c):
    return 1 if len(spouses.intersection(get_between_tokens(c))) > 0 else 0

def LF_husband_wife_left_window(c):
    if len(spouses.intersection(get_left_tokens(c[0], window=2))) > 0:
        return 1
    elif len(spouses.intersection(get_left_tokens(c[1], window=2))) > 0:
        return 1
    else:
        return 0

def LF_same_last_name(c):
    p1_last_name = last_name(c.person1.get_span())
    p2_last_name = last_name(c.person2.get_span())
    if p1_last_name and p2_last_name and p1_last_name == p2_last_name:
        if c.person1.get_span() != c.person2.get_span():
            return 1
    return 0
                                         why?
def LF_no_spouse_in_sentence(c):
    return -1 if np.random.rand() < 0.75 and len(spouses.intersection(c.get_parent().words)) == 0 else 0

def LF_and_married(c):
    return 1 if 'and' in get_between_tokens(c) and 'married' in get_right_tokens(c) else 0

def LF_familial_relationship(c):
    return -1 if len(family.intersection(get_between_tokens(c))) > 0 else 0

def LF_family_left_window(c):
    if len(family.intersection(get_left_tokens(c[0], window=2))) > 0:
        return -1
    elif len(family.intersection(get_left_tokens(c[1], window=2))) > 0:
        return -1
    else:
        return 0
```

```
def LF_other_relationship(c):
    return -1 if len(other.intersection(get_between_tokens(c))) > 0 else 0
```

Distant Supervision LFs

In addition to writing labeling functions that describe text pattern-based heuristics for labeling training examples, 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.

In [6]:

```
import bz2

# Function to remove special characters from text
def strip_special(s):
    return ''.join(c for c in s if ord(c) < 128)

# Read in known spouse pairs and save as set of tuples
with bz2.BZ2File('data/spouses_dbpedia.csv.bz2', 'rb') as f:
    known_spouses = set(
        tuple(strip_special(x.decode('utf-8')).strip().split(',')) for x in f.readlines()
    )
# Last name pairs for known spouses
last_names = set([(last_name(x), last_name(y)) for x, y in known_spouses if last_name(x) and last_name(y)

def LF_distant_supervision(c):
    p1, p2 = c.person1.get_span(), c.person2.get_span()
    return 1 if (p1, p2) in known_spouses or (p2, p1) in known_spouses else 0

def LF_distant_supervision_last_names(c):
    p1, p2 = c.person1.get_span(), c.person2.get_span()
    p1n, p2n = last_name(p1), last_name(p2)
    return 1 if (p1 != p2) and ((p1n, p2n) in last_names or (p2n, p1n) in last_names) else 0
```

For later convenience we group the labeling functions into a list.

In [7]:

```
LFs = [
    LF_distant_supervision, LF_distant_supervision_last_names,
    LF_husband_wife, LF_husband_wife_left_window, LF_same_last_name,
    LF_no_spouse_in_sentence, LF_and_married, LF_familial_relationship,
    LF_family_left_window, LF_other_relationship
]
```

Developing Labeling Functions

Above, we've written a bunch of labeling functions already, which should give you some sense about how to go about it. While writing them, we probably want to check to make sure that they at least work as intended before adding to our set. Suppose we're thinking about writing a simple LF:

In [8]:

```
def LF_wife_in_sentence(c):
    """A simple example of a labeling function"""
    return 1 if 'wife' in c.get_parent().words else 0
```

One simple thing we can do is quickly test it on our development set (or any other set), without saving it to the database. This is simple to do. For example, we can easily get every candidate that this LF labels as true:

In [9]:

```
labeled = []
for c in session.query(Spouse).filter(Spouse.split == 1).all():
    if LF_wife_in_sentence(c) != 0:
        labeled.append(c)
print("Number labeled:", len(labeled))
```

```
('Number labeled:', 278)
```

We can then easily put this into the Viewer as usual (try it out!):

```
SentenceNgramViewer(labeled, session)
```

We also have a simple helper function for getting the empirical accuracy of a single LF with respect to the development set labels for example. This function also returns the evaluation buckets of the candidates (true positive, false positive, true negative, false negative):

Label function

In [10]:

```
from snorkel.lf_helpers import test_LF
tp, fp, tn, fn = test_LF(session, LF_wife_in_sentence, split=1, annotator_name='gold')
```

```
========================================
Scores (Un-adjusted)
========================================
Pos. class accuracy: 1.0
```

```
Neg. class accuracy: 0.0
Precision            0.235
Recall               1.0
F1                   0.38
----------------------------------------
TP: 62 | FP: 202 | TN: 0 | FN: 0
========================================
```

2. Applying the 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`. **Note that this will delete any existing `Labels` and `LabelKeys` for this candidate set.** We start by setting up the class:

In [11]:

```
from snorkel.annotations import LabelAnnotator
labeler = LabelAnnotator(lfs=LFs)
```

Finally, we run the `labeler`. Note that we set a random seed for reproducibility, since some of the LFs involve random number generators. Again, this can be run in parallel, given an appropriate database like Postgres is being used:

In [12]:

```
np.random.seed(1701)
%time L_train = labeler.apply(split=0)
L_train
```

```
In labeling, found 22276 cand IDs for split 0!
Clearing existing...
Running UDF...
100%|██████████| 22276/22276 [01:54<00:00, 194.20it/s]
CPU times: user 1min 49s, sys: 1.45 s, total: 1min 50s
Wall time: 1min 55s
```

Out[12]:

```
<22276x10 sparse matrix of type '<type 'numpy.int64'>'
	with 22267 stored elements in Compressed Sparse Row format>
```

If we've already created the labels (saved in the database), we can load them in as a sparse matrix here too:

In [13]:

```
%time L_train = labeler.load_matrix(session, split=0)
L_train
```

```
CPU times: user 234 ms, sys: 4.58 ms, total: 239 ms
Wall time: 240 ms
```

Out[13]:

```
<22276x10 sparse matrix of type '<type 'numpy.int64'>'
	with 22267 stored elements in Compressed Sparse Row format>
```

Note that the returned matrix is a special subclass of the `scipy.sparse.csr_matrix` class, with some special features which we demonstrate below:

In [14]:

```
L_train.get_candidate(session, 0)
```

Out[14]:

```
Spouse(Span("Anna Paquin", sentence=36995, chars=[313,323], words=[63,64]), Span("Nancy Holt", sentence=36
995, chars=[340,349], words=[69,70]))
```

In [15]:

```
L_train.get_key(session, 0)
```

Out[15]:

```
LabelKey (LF_distant_supervision)
```

We can also view statistics about the resulting label matrix.

- **Coverage** is the fraction of candidates that the labeling function emits a non-zero label for.
- **Overlap** is the fraction candidates that the labeling function emits a non-zero label for and that another labeling function emits a non-zero label for.
- **Conflict** is the fraction candidates that the labeling function emits a non-zero label for and that another labeling function emits a *conflicting* non-zero label for.

In [16]:

```
L_train.lf_stats(session)
```

Out[16]:

	j	Coverage	Overlaps	Conflicts
LF_distant_supervision	0	0.001481	0.001481	0.000628
LF_distant_supervision_last_names	1	0.008080	0.007856	0.004758
LF_husband_wife	2	0.104642	0.066798	0.017867
LF_husband_wife_left_window	3	0.078021	0.057910	0.010774
LF_same_last_name	4	0.016700	0.014994	0.010011
LF_no_spouse_in_sentence	5	0.603026	0.081657	0.009472

	j	Coverage	Overlaps	Conflicts
LF_and_married	6	0.000673	0.000539	0.000404
LF_familial_relationship	7	0.104283	0.091489	0.021413
LF_family_left_window	8	0.073352	0.067651	0.012076
LF_other_relationship	9	0.009337	0.006868	0.001122

3. Fitting the Generative Model

Now, we'll train a model of the LFs to estimate their accuracies. Once the model is trained, we can combine the outputs of the LFs into a single, noise-aware training label set for our extractor. Intuitively, we'll model the LFs by observing how they overlap and conflict with each other.

In [17]:

```
from snorkel.learning import GenerativeModel

gen_model = GenerativeModel()
gen_model.train(L_train, epochs=100, decay=0.95, step_size=0.1 / L_train.shape[0], reg_param=1e-6)
```

```
Inferred cardinality: 2
```

In [18]:

```
gen_model.weights.lf_accuracy
```

Out[18]:

```
array([0.07592901, 0.07395425, 0.11954169, 0.11397737, 0.07065144,
       0.6901572 , 0.07358515, 0.15698341, 0.13658573, 0.08221857])
```

We now apply the generative model to the training candidates to get the noise-aware training label set. We'll refer to these as the training marginals:

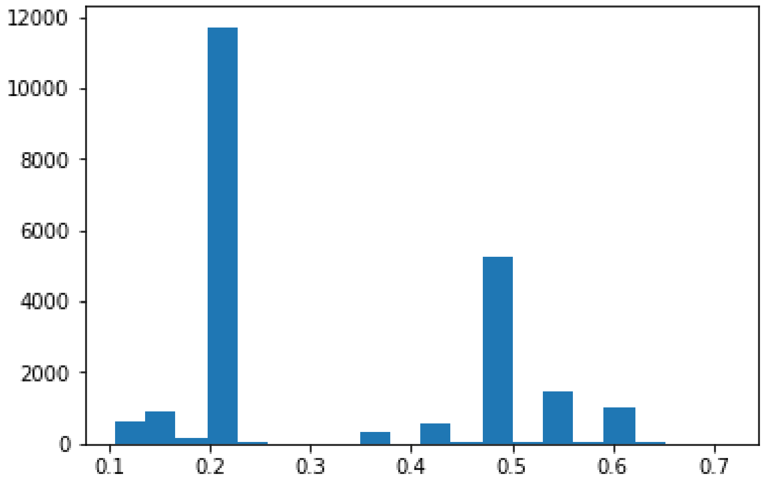
In [19]:

```
train_marginals = gen_model.marginals(L_train)
```

We'll look at the distribution of the training marginals:

In [20]:

```
import matplotlib.pyplot as plt
plt.hist(train_marginals, bins=20)
plt.show()
```



We can view the learned accuracy parameters, and other statistics about the LFs learned by the generative model:

In [21]:

```
gen_model.learned_lf_stats()
```

Out[21]:

	Accuracy	Coverage	Precision	Recall
0	0.534134	0.6665	0.541630	0.360980
1	0.532118	0.6694	0.539711	0.358431
2	0.565538	0.6645	0.574173	0.380980
3	0.565055	0.6702	0.574157	0.377255
4	0.543329	0.6716	0.553972	0.358235
5	0.796530	0.7146	0.804610	0.574902
6	0.526747	0.6711	0.535660	0.343137
7	0.577701	0.6673	0.588448	0.383529
8	0.568233	0.6661	0.572989	0.370196
9	0.540609	0.6698	0.551551	0.366078

10 LFs

Using the Model to Iterate on Labeling Functions

Now that we have learned the generative model, we can stop here and use this to potentially debug and/or improve our labeling function set. First, we apply the LFs to our development set:

In [22]:
```
L_dev = labeler.apply_existing(split=1)
```

```
In labeling, found 2814 cand IDs for split 1!
Clearing existing...
Running UDF...
100%|██████████| 2814/2814 [00:17<00:00, 160.33it/s]
```

And finally, we get the score of the generative model:

In [23]:
```
tp, fp, tn, fn = gen_model.error_analysis(session, L_dev, L_gold_dev)
```

```
========================================
Scores (Un-adjusted)
========================================
Pos. class accuracy: 0.558
Neg. class accuracy: 0.928
Precision            0.361
Recall               0.558
F1                   0.438
----------------------------------------
TP: 106 | FP: 188 | TN: 2436 | FN: 84
========================================
```

Interpreting Generative Model Performance

At this point, we should be getting an F1 score of around 0.4 to 0.5 on the development set, which is pretty good! However, we should be very careful in interpreting this. Since we developed our labeling functions using this development set as a guide, and our generative model is composed of these labeling functions, we expect it to score very well here!

In fact, it is probably somewhat *overfit* to this set. However this is fine, since in the next tutorial, we'll train a more powerful end extraction model which will generalize beyond the development set, and which we will evaluate on a *blind* test set (i.e. one we never looked at during development).

Doing Some Error Analysis

At this point, we might want to look at some examples in one of the error buckets. For example, one of the false negatives that we did not correctly label as true mentions. To do this, we can again just use the `Viewer`:

In [24]:
```
from snorkel.viewer import SentenceNgramViewer

# NOTE: This if-then statement is only to avoid opening the viewer during automated testing of this noteb
# You should ignore this!
import os
if 'CI' not in os.environ:
    sv = SentenceNgramViewer(fn, session)
else:
    sv = None
```

In [25]:
```
sv
```

In [26]:
```
c = sv.get_selected() if sv else list(fp.union(fn))[0]
c
```

Out[26]:
```
Spouse(Span("Frank", sentence=14960, chars=[0,4], words=[0,0]), Span("Kathie Lee Gifford's", sentence=1496
0, chars=[10,29], words=[2,5]))
```

We can easily see the labels that the LFs gave to this candidate using simple ORM-enabled syntax:

In [27]:
```
c.labels
```

Out[27]:
```
[Label (LF_no_spouse_in_sentence = -1)]
```

We can also now explore some of the additional functionalities of the `lf_stats` method for our dev set LF labels, `L_dev`: we can plug in the gold labels that we have, and the accuracies that our generative model has learned:

In [28]:
```
L_dev.lf_stats(session, L_gold_dev, gen_model.learned_lf_stats()['Accuracy'])
```

Out[28]:

	j	Coverage	Overlaps	Conflicts	TP	FP	FN	TN	Empirical Acc.	Learned Acc.
LF_distant_supervision	0	0.001066	0.001066	0.000355	2	1	0	0	0.666667	0.540976
LF_distant_supervision_last_names	1	0.006397	0.006397	0.003198	7	11	0	0	0.388889	0.523514

	j	Coverage	Overlaps	Conflicts	TP	FP	FN	TN	Empirical Acc.	Learned Acc.
LF_husband_wife	2	0.092751	0.062189	0.019545	94	151	0	0	0.383673	0.556170
LF_husband_wife_left_window	3	0.069652	0.055082	0.013859	82	107	0	0	0.433862	0.557460
LF_same_last_name	4	0.024165	0.018834	0.013859	19	49	0	0	0.279412	0.538473
LF_no_spouse_in_sentence	5	0.624023	0.098081	0.015636	0	0	55	1630	0.967359	0.793239
LF_and_married	6	0.002843	0.001777	0.001777	1	7	0	0	0.125000	0.551017
LF_familial_relationship	7	0.117982	0.103767	0.029495	0	0	15	306	0.953271	0.583724
LF_family_left_window	8	0.074982	0.068941	0.012438	0	0	1	203	0.995098	0.575739
LF_other_relationship	9	0.008884	0.008529	0.002843	0	0	4	17	0.809524	0.541944

Note that for labeling functions with low coverage, our learned accuracies are closer to our prior of 70% accuracy.

Saving our training labels

Finally, we'll save the `training_marginals`, which are our **probabilistic training labels**, so that we can use them in the next tutorial to train our end extraction model:

In [29]:

```
from snorkel.annotations import save_marginals
%time save_marginals(session, L_train, train_marginals)
```

```
Saved 22276 marginals
CPU times: user 12.3 s, sys: 116 ms, total: 12.5 s
Wall time: 13 s
```

Next, in Part III, we'll use these probabilistic training labels to train a deep neural network.