Intro. to Snorkel: Extracting Spouse Relations from the News

In this tutorial, we will walk through the process of using `Snorkel` to identify mentions of spouses in a corpus of news articles. The tutorial is broken up into 3 notebooks, each covering a step in the pipeline:

1. Preprocessing
2. Training
3. Evaluation

Part I: Preprocessing

In this notebook, we preprocess several documents using `Snorkel` utilities, parsing them into a simple hierarchy of component parts of our input data, which we refer to as *contexts*. We'll also create *candidates* out of these contexts, which are the objects we want to classify, in this case, possible mentions of spouses. Finally, we'll load some gold labels for evaluation.

All of this preprocessed input data is saved to a database. (Connection strings can be specified by setting the `SNORKELDB` environment variable. In Snorkel, if no database is specified, then a SQLite database at `./snorkel.db` is created by default--so no setup is needed here!

Initializing a `SnorkelSession`

First, we initialize a `SnorkelSession`, which manages a connection to a database automatically for us, and will enable us to save intermediate results. If we don't specify any particular database (see commented-out code below), then it will automatically create a SQLite database in the background for us:

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'

from snorkel import SnorkelSession
session = SnorkelSession()

# Here, we just set how many documents we'll process for automatic testing- you can safely
 ignore this!
n_docs = 500 if 'CI' in os.environ else 2591
```

Loading the Corpus

Next, we load and pre-process the corpus of documents.

Configuring a `DocPreprocessor`

We'll start by defining a `TSVDocPreprocessor` class to read in the documents, which are stored in a tab-seperated value format as pairs of document names and text.

docs in snorkel
? = sentense

In [2]:
```
from snorkel.parser import TSVDocPreprocessor

doc_preprocessor = TSVDocPreprocessor('data/articles.tsv', max_docs=n_docs)
```

Running a `CorpusParser`

We'll use Spacy, an NLP preprocessing tool, to split our documents into sentences and tokens, and provide named entity annotations.

In [3]:
```
from snorkel.parser.spacy_parser import Spacy
from snorkel.parser import CorpusParser

corpus_parser = CorpusParser(parser=Spacy())
%time corpus_parser.apply(doc_preprocessor, count=n_docs)
```

```
Clearing existing...
Running UDF...
```

```
100%|██████████| 2591/2591 [02:20<00:00, 27.89it/s]
```

```
CPU times: user 2min 36s, sys: 3.06 s, total: 2min 39s
Wall time: 2min 42s
```

We can then use simple database queries (written in the syntax of [SQLAlchemy](), which Snorkel uses) to check how many documents and sentences were parsed:

In [4]:
```
from snorkel.models import Document, Sentence

print("Documents:", session.query(Document).count())
print("Sentences:", session.query(Sentence).count())
```

```
('Documents:', 2591)
('Sentences:', 67778)
```

Generating Candidates

The next step is to extract *candidates* from our corpus. A `Candidate` in Snorkel is an object for which we want to make a prediction. In this case, the candidates are pairs of people mentioned in sentences, and our task is to predict which pairs are described as married in the associated text.

Defining a `Candidate` schema

We now define the schema of the relation mention we want to extract (which is also the schema of the candidates). This must be a subclass of `Candidate`, and we define it using a helper function. Here we'll define a binary *spouse relation mention* which connects two `Span` objects of text. Note that this function will create the table in the database backend if it does not exist:

helper function

In [5]:
```
from snorkel.models import candidate_subclass

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

Kinda like namedtuple

Writing a basic `CandidateExtractor`

Next, we'll write a basic function to extract **candidate spouse relation mentions** from the corpus. The [Spacy]() parser we used performs *named entity recognition* for us.

We will extract `Candidate` objects of the `Spouse` type by identifying, for each `Sentence`, all pairs of n-grams (up to 7-grams) that were tagged as people. (An n-gram is a span of text made up of n tokens.) We do this with three objects:

- A `ContextSpace` defines the "space" of all candidates we even potentially consider; in this case we use the `Ngrams` subclass, and look for all n-grams up to 7 words long
- A `Matcher` heuristically filters the candidates we use. In this case, we just use a pre-defined matcher which looks for all n-grams tagged by Spacy as "PERSON". The keyword argument `longest_match_only` means that we'll skip n-grams contained in other n-grams.
- A `CandidateExtractor` combines this all together!

In [6]:
```
from snorkel.candidates import Ngrams, CandidateExtractor
from snorkel.matchers import PersonMatcher

ngrams         = Ngrams(n_max=7)   7 words long
person_matcher = PersonMatcher(longest_match_only=True)
cand_extractor = CandidateExtractor(Spouse, [ngrams, ngrams], [person_matcher, person_match
er])
```

Next, we'll split up the documents into train, development, and test splits; and collect the associated sentences.

Note that we'll filter out a few sentences that mention more than five people. These lists are unlikely to contain spouses.

In [7]:
```
from snorkel.models import Document
from util import number_of_people

docs = session.query(Document).order_by(Document.name).all()

train_sents = set()
dev_sents   = set()
test_sents  = set()

for i, doc in enumerate(docs):
    for s in doc.sentences:
        if number_of_people(s) <= 5:
            if i % 10 == 8:
                dev_sents.add(s)
            elif i % 10 == 9:
                test_sents.add(s)
            else:
                train_sents.add(s)
```

0 – 7 train   8 dev   9 test

Finally, we'll apply the candidate extractor to the three sets of sentences. The results will be persisted in the database backend.

In [8]:
```
%%time
for i, sents in enumerate([train_sents, dev_sents, test_sents]):
    cand_extractor.apply(sents, split=i)
    print("Number of candidates:", session.query(Spouse).filter(Spouse.split == i).count())
```

```
Clearing existing...
Running UDF...
```

```
100%|██████████| 53991/53991 [05:08<00:00, 174.77it/s]
```

```
('Number of candidates:', 22276)
Clearing existing...
Running UDF...
```

```
100%|██████████| 6789/6789 [00:34<00:00, 199.34it/s]
```

```
('Number of candidates:', 2814)
Clearing existing...
Running UDF...
```

```
100%|██████████| 6194/6194 [00:32<00:00, 187.88it/s]
```

```
('Number of candidates:', 2702)
CPU times: user 6min 1s, sys: 6.11 s, total: 6min 7s
Wall time: 6min 15s
```

Loading Gold Labels

Finally, we'll load gold labels for development and evaluation. Even though Snorkel is designed to create labels for data, we still use gold labels to evaluate the quality of our models. Fortunately, we need far less labeled data to *evaluate* a model than to *train* it.

In [9]:
```
from util import load_external_labels

%time missed = load_external_labels(session, Spouse, annotator_name='gold')
```

```
AnnotatorLabels created: 2698
AnnotatorLabels created: 2608
CPU times: user 1min 26s, sys: 746 ms, total: 1min 27s
Wall time: 1min 28s
```

Next, in Part II, we will work towards building a model to predict these labels with high accuracy using data programming