Snorkel Workshop: Extracting Spouse Relations from the News

Part 5: Preprocessing & Building the Snorkel Database

In this tutorial, we will walk through the process of using `Snorkel` to identify mentions of spouses in a corpus of news articles.

In [ ]:
```
import sys; sys.version
```

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 [ ]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
import os

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

# Here, we just set how many documents we'll process for automatic testing- you can safely ignore this!
n_docs = 1000 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.

In [ ]:
```
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 [ ]:
```
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, parallelism=1)
```

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

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

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:

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

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

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 ngrams (up to trigrams) that were tagged as people. 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 3 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 CoreNLP as "PERSON"
- A `CandidateExtractor` combines this all together!

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

ngrams         = Ngrams(n_max=7)
person_matcher = PersonMatcher(longest_match_only=True)
cand_extractor = CandidateExtractor(Spouse,
                                    [ngrams, ngrams], [person_matcher, person_matcher],
                                    symmetric_relations=False)
```

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 [ ]:
```
from snorkel.models import Document
from lib.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)
```

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

In [ ]:
```
%%time
for i, sents in enumerate([train_sents, dev_sents, test_sents]):
    cand_extractor.apply(sents, split=i, parallelism=1)
    print("Number of candidates:", session.query(Spouse).filter(Spouse.split == i).count())
```

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 [ ]:
```
from lib.util import load_external_labels

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

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