Steps to use the `disambig` library for PATSTAT

Mark Huberty

October 16, 2012

1 Introduction

The `disambig` library provides a Bayesian disambiguation algorithm for author-patent instances in large patent databases like PATSTAT. Papers by Torvik and Fleming et al show that it attains high rates of accuracy.

At present, `disambig` solves a series of problems we have not yet addressed:

1. Clustering Our matching process presently generates pairwise or one-to-many matches. Additional logic is needed to aggregate those matches up. The usual method involves transitivity, in that if A:B match and B:C match, A:C are presumed to match.

   This process, of course, raises a second issue, *triplet correction.*

2. Triplet correction The initial matching step produces 1:1 or 1:many matches based on a distance threshold or similar quantity. However, we wish to cluster together all similar name instances. This leads to a transitivity issue: if A:B are a good match and B:C are a good match, should we consider A:C a good match?

   Theoretically, A:C should only be a good match if the A:C distance passes the threshold value test. But this requires cross-checking every possible triplet pair.

The `disambig` library, recently released, solves these problems already.

Furthermore, `disambig` points out the necessity of matching across a profile of data, rather than just the name match. For the USPTO case, they use first and last name, geographic location, and IPC code overlap to identify when authors of two different patents should be considered the same author.

We presently only consider name.

2 Potential issues

The original `disambig` code was built with the USPTO data in mind. This data has some advantages over PATSTAT:

- Explicit tracking of first versus last names
- Complete address information
- Standardized entry formats (b/c this is a single patent office, while PATSTAT aggregates across offices)

3 Requirements

If we were to pursue this route, we would need to do the following:

1. Generate a set of training data, derived from the PATSTAT database itself. This data doesn't have to be hand-labeled, but should have both matches and non-matches. The USPTO paper describes a good rubric for how to do this based on using relatively rare names.
2. Generate a set of tables for use in the `disambig` algorithm; it appears easiest to generate this once, so that it contains all the fields we require
3. Define the set of comparison fields we want
    - I'd recommend we use name, country, address, and the top 4 co-authors and ipc codes from each author:patent instance
4. Ensure the comparison functions in the existing `disambig` library map correctly to these comparison fields
    - This should be pretty straightforward, as PATSTAT provides less data than USPTO did
    - It's unclear whether

I'd then run the entire db process on a very small sample (10000 random rows) of the data to see what we get, before processing the entire thing.

Keep in mind that the USPTO data handled 8m records and took about a week. We likely have much more than that, and so need to alot more time to run the algorithm. Unfortunately, the `disambig` library is not yet built to permit parallelization, although some of its tasks are trivially parallelizable.

I would still use all of the cleaning code from our `psClean` library to prepare the data before running the disambiguation logic.

4 Steps

1. Preparing the software

   I've installed (as of 16 October) all the prerequisites listed in the disambig manual. Compiling the actual disambig algorithm will need to wait until we have the config files prepared.

2. Building the data set

   (a) ID all fields we want to match on. I would suggest name, address, country, first 4 ipc codes, first 4 co-authors. For name, we should split off known company IDs into a separate field.

   (b) Generate a table w/ one row per author-patent, holding this data This is very compute-intensive, owing to the number of records we will have to pull back. We'll probably need to do it sequentially by country or something similar. But I suspect it will be easier to do this once, generate a text file, and load it to the SQLLite tables, than to do it via a query at runtime.

   (c) Clean the consolidated data w/ our `psClean` algorithms

   (d) Assign block IDs based on what we want to block on. I would suggest blocking on the following:

      i. Block 1: One word in name and country match

      ii. Block 2: First N characters in name match

   This would implement a sequential blocking strategy along the lines of that presented in Fleming. We'll have to decide how many sequential blocks to use once we get a better handle on runtime.

3. Setting up the config files

   (a) The `data.desc` file We can configure this based on the final data file we prepare as documented in (1) above. The file format and description are pretty self-explanatory

   (b) The `sp.desc` file

   - We need to specify the result space for each comparison.
   - We should be able to reuse these functions already in `compfun.c`:
     - `jwcmp` (name comparison)
     - `classcmp` (ipc class comparison)

  - `coauthcmp` (coauthor comparison)
- We will want to write simple functions for comparing these:
  - country (returns MISSING, MATCH, NOMATCH)
  - corporate identifier (returns MISSING, MATCH, NOMATCH)

  Note that both comparisons can use the same function
- We need to figure out what to do about address compare. Fleming et al do a lat/long conversion first, but it's not clear how they do that. This would be the best way to go, but could be hard. See the end of this document.
- **NOTE** that the `sp.desc` file in the `disambig` repository doesn't exactly match `compfun.c` for some of the output formats. We'll need to look at that, I can't tell exactly why they changed it.

4. Training the algorithm

   From the Fleming paper, a good start on training the similarity profiles would be to generate the following datasets:

	Match Set
Condition on: name	Pairs of records w/ matching names whose names are rare
Condition on: other	Pairs of records in same block sharing > 1 coauthor
	Nonmatch Set
Condition on: name	Pairs of records w/ non-matching rare names
Condition on: other	Pairs of records in same block sharing no coauthors or ipc codes

5. Running the algorithm

   I'd suggest that we test on, say, 10% of the records (random subsample) to make sure the output looks sane. Then we can run the whole thing.

5 Next steps

I think the first steps are to:

1. Decide if we are going to handle address matching
2. Decide on the fields we will match on
3. Generate the data

Once we have these three in place, we can load the database and go from there.

6 Address matching

Assuming we use something like the NASA geo-coding files[1], address-to-lat/long would look something like this:

```
## Assumes that address_file is a dict of form country:region_list
## and region_list is a dict of form region_code:{region_attributes}
  latlong_list = []
  import re
  for country, address in zip(countries, addresses):
      address_dict = address_file[country]
      for master_code in address_dict:
          if re.search(master_code, address):
              lat = address_dict[master_code]['lat']
              longt = address_dict[master_code]['long']
              latlong = (lat, long)
          else:
              latlong = ('', '')
          latlong_list.append(latlong)
```

[1] See http://earth-info.nga.mil/gns/html/gis_countryfiles.html