Congress Lobbying Database: Documentation and Usage*

In Song Kim†

December 13, 2015

1 Introduction

This document concerns the code in the `/lobbydata/code/database` directory of our repository, which sets up and provides access to a system of databases (running on `SQLite` and the `Whoosh` text indexing library) that store relationships between bills, their lobbiers, and various other related pieces of data.

1.1 Dependencies

The table below is a summary of the packages on which the database depends, along with a short summary of the functionalities that they provide (further information and documentation can be easily found on their PyPI (Python Package Index) pages). The packages are roughly grouped by their function (database, parsing, etc). For the purposes of actual deployment, the file that manages these packages is `database/REQUIREMENTS`.

Package	Description
`SQLAlchemy`	Basic Python bindings and model representations for SQL-type databases.
`Elixir`	Higher-level abstractions for dealing with SQL-type databases, extending the functionality that `SQLAlchemy` makes available.
`Whoosh`	A library for creating full-text index databases that are searchable with reasonable efficiency (if in the future better efficiency here is needed, there are non-Python packages that can do a better job). SQL is not good, generally speaking, for full-text search, hence the necessity of this for the bill CRS summaries and lobbying report specific issue texts.
`BeautifulSoup`	Convenient library for parsing XML and HTML data into Python objects, although when speed becomes an issue there are faster but less convenient (and more code-verbose) alternatives, such as `xml.etree` in the core Python library.
`nltk`	The "natural language toolkit" for Python, providing tools for tokenizing and statistically analyzing English-language texts.
`path.py`	Convenience tools to make dealing with the filesystem easier from Python.
`python-dateutil`	Convenience tools for dealing with datetimes and time ranges.

*Financial support from the National Science Foundation Doctoral Dissertation Research Improvement Grant SES-1264090 is acknowledged. I thank Tim Kunisky, Feng Zhu for their excellent research assistance.

†Assistant Professor, Department of Political Science, Massachusetts Institute of Technology, Cambridge MA 02142. Phone: 617–253–3138, Email: insong@mit.edu, URL: http://www.princeton.edu/∼insong

Before going any further, you will need to install all of these packages, which can be conveniently done through the REQUIREMENTS file, by running the following command in the shell (you will need the pip utility for Python package management):

```
> pip install -r REQUIREMENTS
```

Also, there are a few subpackages to install for the nltk package. In a Python interpreter, run the following:

```
>>> import nltk
>>> nltk.download()
```

That will open up an interface for downloading extras/packages for nltk. Then, you should go into the Corpora tab and install the packages stopwords and wordnet. If no errors arise during this installation procedure, it is safe to proceed to the next steps.

2 Getting Started

2.1 Configuration

All of the necessary configuration for the database system can be done by modifying the variables in general.py. The variables are listed below along with the effect that they have on the database creation process. Realistically speaking, to get things working locally you should just change DATA_DIR to something that is not Della-specific. Everything else should be (more or less) good to go, assuming no large repository reorganizations have happened.

Variable	Description
TEST_MODE	Intended as a testing mode for bill detection and related algorithms, but this is not yet complete, so avoid setting this to True before taking a look at the testing code.
CONGRESSES	The range of congresses that all processes will be concerned with (note that in Python, the range(x, y) syntax gives the numbers $x, x+1, \ldots, y-1$, not including y).
OUTPUT_DIR	The directory for generic outputs of analytics scripts.
ROOT_DIR	The database directory (these directories are given relative to the trade/code directory).
DATA_DIR	The directory used for storing databases, which can be totally separate from the code directories. For instance, on Della it is useful to put this under /tigress since it requires lots of storage space.
LOBBY_REPO_DIR	The location of the lobby repository (parallel to the trade repository in the current setup).
CLIENT_NAME_MATCHES_FILE	The file containing the client name filtering matches (i.e. the output of Josh's script).

2.2 Initialization

Installing the databases is (or should be) very easy, just go to the trade/code directory and run the command sudo python -m database.setup. This will probably take a very long time to run from scratch—possibly up to several days.

3 Working with the Database

3.1 Starting and Ending Sessions

In order to get started with a database session, navigate to the trade/code directory, and run the following sequence of commands in the Python interpreter:

```
>>> import database.general
>>> from database.bills.models import *
>>> from database.lda.models import *
>>> from database.firms.models import *
>>> database.general.init_db()
```

When finished, the following command will safely close the database without accidentally permanently writing any changes that may have been made to the data:

```
>>> database.general.close_db(write=False)
```

If you are in fact correcting errors in the database or otherwise performing operations that cause changes that you would like permanently registered, then just change the above to instead pass the argument `write=True`.

3.2 Basic Database Objects and Relationships

To see what objects are stored in the various databases, look at the `models.py` files in the directories `bills`, `lda`, and `firms` (the imports described above are what give you access to all of these classes). Each class has some fields, where are available to access on any object of the class. The system is best clarified with an example: consider the case of bill objects, which are represented by the `Bill` class. This class, like any other, has an `id` field that is its *primary key*, i.e. the value of this field uniquely identifies a `Bill` object. For bills, the `id` is a string of the form `110_HR7311`, where `110` is the number of the congress to which this bill belongs, and `HR7311` is the bill number. To retrieve a particular bill by its `id`, use the following snippet:

```
>>> b = Bill.query.get('110_HR7311')
```

Once this command completes, the object `b` will have all of the fields listed under `Bill` in the file `bills/models.py`. So, for instance, we can get the date that the bill was introduced with `b.introduced`, get its CRS summary text with `b.summary`, and so forth. Any field inside `Bill` that is initialized as `Field(ABC)` where `ABC` is some text (example possible values are `Integer` for integer fields, `Unicode(L)` for a string field of maximum length `L`, or `DateTime` for a date/time field) is accessed in this straightforward way.

Other fields are registered as `ManyToMany(ABC)`, `ManyToOne(ABC)`, or `OneToMany(ABC)`, where `ABC` is now the name of some other model. These fields contain references to one or more instances of some other model. For instance, in `Bill`, the field definition

```
titles = OneToMany('BillTitle')
```

indicates that each bill has one or more (hence `Many`) associated objects of class `BillTitle`, which are its titles. In `BillTitle`, we see the field giving the reverse relation,

```
bill = ManyToOne('Bill')
```

which indicates that many `BillTitle` objects can share the same `Bill` object (for some intuition, think of a `OneToMany` field as a "my children" relation, and of a `ManyToOne` field as a "my parent" relation).

Thus, if `b` is a `Bill` object, then `b.titles` will give an iterable (effectively a Python list, for all basic purposes) containing all the titles of `b`. Conversely, if `t` is a `BillTitle` object, then `t.bill` is the `Bill` object to which the title belongs.

The last possibility of these more complex relationships is a `ManyToMany` field, which as its name suggests creates a generic relation between two object types (where neither object plays the "child" or "parent" role). For example, we see in `Bill`,

```
terms = ManyToMany('Term')
```

and in `Term`,

```
bills = ManyToMany('Bill')
```

which means that for a bill `b`, looking at `b.terms` gives all of the terms that that bill is classified under, and for a term `t`, looking at `t.bills` gives all bills under that term.

3.3 Filtering Operations

The more sophisticated and interesting sorts of queries that are possible are those that involve not just fetching particular bills or other objects and examining their relationships, but also involve filtering sets of objects by useful criteria.

Example: filter by columns of each Model

```
>>> reports = LobbyingReport.query.\
                  filter_by(year=2011)
```

returns all lobbying reports filed in 2011

Example: filter by membership in at least one ManyToMany related table

```
>>> trade = LobbyingReport.query.\
                filter(LobbyingReport.issues.any(LobbyingIssue.code.\
                in_(['TRADE (DOMESTIC/FOREIGN)'])))
>>> trade.count()
52418
```

returns all lobbying reports that at least has 'TRADE (DOMESTIC/FOREIGN)' as one of issues lobbied.

3.4 Full-Text Indices

Two types of data items are duplicated in a separate full-text index database to facilitate more efficient searching: the CRS summary text of each bill, and the text of each lobbying report specific issue. The code concerning the creation and access of these indices is found in the files `bills/ix_utils.py` and `lda/ix_utils.py`, respectively.

The primary useful methods, in turn, for accessing these indices are `summary_search` and `issue_search`, in the above two files respectively. These both take one required argument, called `queries_list`, which is a list of the queries (as strings) to make to the full-text index. They also have two optional boolean arguments, `return_objects` and `make_phrase`, which default to `False`. Setting `return_objects` to `True` will return a collection of `Bill` or `LobbyingSpecificIssue` Python objects rather than just their `id` values. Setting `make_phrase` to `True` will make each query into a phrase that is searched for a single unit, rather than separately searching for each word (as in the difference when searching Google for *red cat running* versus *"red cat running"* in quotes).

In `lda/ix_utils.py`, there is an additional method exposed for using the index that is called `get_bill_specific_issues_by_titles`, which is a simple special case of `issue_search` that searches for all of the titles of a particular bill in the specific issues, used in the database construction process to find the bills mentioned by title in specific issues.

A simple example of using these indices to find bills pertaining to a particular textually-distinguished subject (trade-related bills in our case) can be found in `analytics/lobbied_bills_data.py`. We define a list of queries on our bills in the following way:

```
from database.analytics.bill_utils import *
    bill_queries = [
        u'trade barrier',
        u'tariff barrier',
        ...
        u'uruguay round',
        u'harmonized tariff schedule'
    ]
```

Then, to get the `id`'s of the bills that contain one of these phrases, we do this:

```
    query_bill_ids = database.bills.ix_utils.summary_search(
        bill_queries,
        make_phrase=True,
        return_objects=False
    )
```

This returns a list of `id`'s as strings. If we wanted the corresponding `Bill` objects instead, we could instead pass the argument `return_objects=True`. Note that here it is important that we use `make_phrase=True`, since otherwise the query `'uruguay round'` would match all bills that contain both the word `uruguay` and the word `round`, not necessarily together, which is not what we want.

A simple example of using these indices to find lobbying reports that contain a particular phrase,

```
from database.analytics.lda_utils import *
reports =  lda_issue_search(
    ['Free trade agreements with South Korea']
)
```

3.5 Calculating Herfindahl Indices for Industry Clients Belong to

We provide a tool to measure the size of each firm in relation to the industry. Herfindahl index measures the levels of competition among firms (clients) within the same industry.

3.5.1 herfindahl.py (in lobby/code/hfcc)

Given lobbying database containing firm, LDA, and bill information,

1. Pulls firm-level financial and LDA report information from lobbying database
2. Computes Herfindahl indices
3. Outputs rows with firm information (sorted by industry as identified by NAICS2), industry Herfindahl index, and lobbying information for firm.

To run: From lobby/code, type "python -m hfcc.herfindahl" No additional parameters needed.

3.5.2 herfindadd.py (in lobby/code/hfcc)

Given output (csv) files generated by herfindahl.py,

1. Adds indicator showing whether firm lobbied on at least one trade issue
2. Adds firm-level compustat financial data
3. Generates (in addition) new csv files with industry-level information in rows

To run: Ensure output files from herfindahl.py are in lobby/code From lobby/code, type "python -m hfcc.herfindadd" On-screen documentation will detail additional parameters that are needed.

Example

```
python -m hfcc.herfindadd namerica naics -s 1996 -e 2011
```

runs the script for the North America files, using NAICS (rather than SIC), starting from 1996 and ending in 2011 (herfindahl.py generates one file per year per classification system (NAICS / SIC).)

4 Collected Implementation Details

4.1 Identifying Congress From Bill Number

This section concerns the procedure by which, given a bill number found in a specific issue text, we attempt to identify the most likely congress to which that bill would belong. The relevant code is in `lda/db_utils.py`, particularly in the method `find_top_match_bill`. This method takes an argument `bill_number` that is the number of the bill in question, an argument `context` that is the section of the specific issue text in which this bill number was found (or more generally any text against which we might want to test bill similarity), an argument `start_congress` that contains the latest congress that we believe this bill could belong to, and lastly an argument `n` that indicates how many congresses to consider (defaulting to three).

Then, the candidate congresses are the `n` congresses preceding `start_congress` (and including `start_congress` itself). We then look for bills having number `bill_number` in each of these congresses, and obtain their texts. Our operating hypothesis is that the bill text that is most statistically similar to the context (i.e. the specific issue text) will be the bill that we are interested in, since presumably the context mentioning the bill would be similar to the bill text itself.

The actual similarity computation is performed by the method `find_top_match_index`, which only takes in the list of bill texts and the context text, and returns the index in the list of bill texts of the text having the greatest similarity to the context. This method uses a vectorizer on the texts to convert strings to frequency vectors of words (there is a sequence of tokenizing operations involved, which clean the text, remove stopwords, and so forth), and then computes the maximum cosine similarity between a bill-text vector and the context vector. That is, if the frequency vectors of the bill texts are b_i for $1 \leq i \leq N$, and c is the frequency vector of the context, then the method will return the value

$$i = \operatorname*{argmax}_{1 \leq i \leq N} \frac{b_i \cdot c}{\|b_i\| \|c\|} = \operatorname*{argmax}_{1 \leq i \leq N} \frac{b_i \cdot c}{\|b_i\|}$$

where $\cdot$ is the dot product and $\| \bullet \|$ is the L^2-norm, both defined over frequency vectors (we build the total vocabulary of all words occuring in any of the b_i and c and make the frequency vectors over this vocabulary, so that the dimensions of all of these vectors are the same).