Distributional Thesaurus Part 1

April 28, 2018

1 JoBimText: Creation of Distributional Thesaurus

PART 1 : Preprocessing and Creating Initial Databases

1.0.1 Import Libraries required

```
In [1]: import sqlite3
```

1.0.2 Import the parsed sentences of the mouse corpus

```
In [2]: from parsed.output import sentence as parserOutput1
        from parsed.output2 import sentence as parserOutput2
        from parsed.output3 import sentence as parserOutput3
        from parsed.output4 import sentence as parserOutput4
        from parsed.output5 import sentence as parserOutput5
```

1.0.3 Concatenate all the parsed sentences

```
In [3]: parserOutput1.extend(parserOutput2)
        parserOutput1.extend(parserOutput3)
        parserOutput1.extend(parserOutput4)
        parserOutput1.extend(parserOutput5)

        print("Total sentences:", len(parserOutput1))
        print("Sample parse:", parserOutput1[0][0])
```

```
Total sentences: 20891
Sample parse: (('concludes', 'VBZ'), 'nsubj', ('device', 'NN'))
```

1.0.4 Initialize cursor for DB

```
In [7]: conn = sqlite3.connect('thesaurus.db')
        c = conn.cursor()
        c.execute('''SELECT name FROM sqlite_master WHERE type="table";''').fetchall()
```

```
Out[7]: [('sqlite_sequence',),
         ('le',),
```

```
('cf',),
('lecf',),
('sm',),
('psm',),
('simcount',)]
```

1.0.5 Create the first 3 tables

- Language Elements (LE)
- Context Features (CF)
- LE-CF

In [16]:
```
c.execute("drop table if exists le")

# table for language elements
c.execute('''CREATE TABLE le (
            id integer primary key autoincrement,
            name text not null,
            pos text not null,
            count int not null,

            CONSTRAINT uniqueConstraint UNIQUE(name, pos)
          );
''')


c.execute("insert into le (id, name, pos, count) values(1, '@', 'hole', 0)")
conn.commit()
```

In [17]:
```
c.execute("drop table if exists cf")

# table for context features
c.execute('''CREATE TABLE cf
            (id integer primary key autoincrement,
            le1 integer not null,
            le2 integer not null,
            rel text not null,
            count int not null,

            foreign key(le1) references le(id) on delete cascade,
            foreign key(le2) references le(id) on delete cascade

            constraint uniqueConstraint UNIQUE(le1, le2, rel)
            );
''')

conn.commit()
```

```
In [18]: c.execute("drop table if exists lecf")

         # table for le-cfs
         c.execute('''CREATE TABLE lecf (
                     id integer primary key autoincrement,
                     le integer not null,
                     cf integer not null,
                     count int not null,

                     foreign key(le) references le(id) on update cascade on delete cascade,
                     foreign key(cf) references cf(id) on update cascade on delete cascade,

                     constraint uniqueConstraint unique(le, cf)
                     )''')

         conn.commit()
```

1.1 Filling the DB with the parsed data

All of the update functions have the same structure

- If row with parameters passed already exists in the DB, update it with `count = count + 1`
- Else create row with `count = 1`

```
In [19]: def update_le(name, pos):
             if not name.isalpha():
                 name = spl
             row = c.execute('''select * from le where name='{}' and pos='{}' '''.format(name,
             if row is None:
                 c.execute('''insert into le (name, pos, count) values('{}', '{}', '{}')'''.fo
             else:
                 c.execute('''update le set count={} where name='{}' and pos='{}' '''.format(r
                 return row[0]

             conn.commit()
             return c.execute('''select id from le where name='{}' and pos='{}' '''.format(nam

In [20]: def update_cf(le1, le2, rel):
             row = c.execute('''select id, count from cf where le1='{}' and le2='{}' '''.forma
             if row is None:
                 c.execute('''insert into cf (le1, le2, rel, count) values('{}', '{}', '{}', '
             else:
                 c.execute('''update cf set count={} where le1='{}' and le2='{}' and rel='{}'
                 return row[0]

             conn.commit()
             return c.execute('''select id from cf where le1='{}' and le2='{}' and rel='{}' ''
```

```
In [21]: def update_le_cf(le, cf):
             row = c.execute('''select count from lecf where le='{}' and cf='{}' '''.format(le
             if row is None:
                 c.execute('''insert into lecf (le, cf, count) values('{}', '{}', '{}')'''.forr
             else:
                 c.execute('''update lecf set count={} where le='{}' and cf='{}' '''.format(rov
             conn.commit()
```

Filling in values for tables LE, CF AND LE-CF from the parsed data

For each relation

- Replace words that have non alphabetic character with special token `@spl@`
- Update LE table
- Apply Holing Operation on dependencies and create Context Features
- Update CF table
- Update LE-CF table.

```
In [22]: spl = "@spl@"
         hole = ["@", "hole"]
         hole_id = c.execute("select id from le where name='@' and pos='hole'").fetchone()[0]

         for parse in parserOutput1:
             for dependency in parse:
                 w1 = dependency[0]
                 w2 = dependency[2]
                 rel = dependency[1]

                 le1 = w1[0].replace("'", r"''")
                 le2 = w2[0].replace("'", r"''")

                 # update le table:
                 w1_id = update_le(le1, w1[1])
                 w2_id = update_le(le2, w2[1])

                 # update cf table with holing operation:
                 cf1_id = update_cf(hole_id, w2_id, rel)
                 cf2_id = update_cf(w1_id, hole_id, rel)

                 # update le-cf table:
                 update_le_cf(w1_id, cf1_id)
                 update_le_cf(w2_id, cf2_id)

             conn.commit()
```