MarcXimiL

Similarity analysis

- Near-duplicates detection
- Information monitoring
- Collection structure analysis
- Plagiarism detection
- Related records suggestion

http://marcximil.sourceforge.net
MarcXimiL philosophy

- Flexible at all levels
- Information Retrieval algorithms (adapted) and bibliographic specific algorithms
- Open source: GPL 3
- Central configuration (similarity strategy)
- Standard (MARCXML, OAI-PMH2, UNIX structure)
- Compatible (all systems, python 2.4.0 → 3.1.1)
- Easy to install (unpack)
MarcXimiL flexibility philosophy

- **Method:** each level of the similarity analysis is piloted by a function:
  - **Useful in config:** For each level a range of exchangeable functions that the end user may choose from.
  - **Useful in devel:** new functionalities or variations of existing ones take the form of new functions (and the existent remains usable).

- In the config file one sets up a combination of functions on a set of fields → **similarity strategy**

- NB: At one level, functions are exchangeable because they share a common programming interface.
MarcXimiL flexibility at all levels

- Parsing functions (3 type of MARC fields + concatenation)
- Pre-treatment functions (RAW, WC, INITIALS, SHINGLES) + normalisation (case, diacritics, punctuation...)
- Comparisons pairs and order (triangular, multiple colls ...)
- Global record similarity: 6 weighted averages and several specialized functions (maxsim, ubiquist, boundaries, ...)
- Field similarity functions: authors, date/year, vectorial[Dice,Salton,Jaccard], probabilistic, initials, shingles, doi/uid, Levenshtein based, % of items, ...
- Output functions: tabs, XML, devel. + global threshold
Configuration

INPUT_FILES = ['testdataCERNb.xml']
records_comp = records_comp_single
report = report_tab
globalvars.output_threshold = 0.5
record_rules = geometric_mean_breakout

record_structure = {
    '01recid': {
        'marc': '001',
        'weight': 0,
        'parse-func': parse_controlfield,
        'comp-func': fields_concat__raw
    },
    '02year': {
        'marc': '260 c',
        'weight': 1,
        'parse-func': parse_nonrep,
        'comp-func': years_comp__raw
    },
    '03authors': {
        'marc': ['100 a', '700 a'],
        'weight': 2,
        'parse-func': parse_multi,
        'comp-func': authors_comp__raw
    },
    '04title': {
        'marc': ['245 a', '245 b'],
        'weight': 3,
        'parse-func': parse_concat,
        'comp-func': okapibm25__wc
    },
    '05title': {
        'marc': '245 a',
        'weight': 3,
        'parse-func': parse_nonrep,
        'comp-func': levenshtein__raw
    }
}
MarcXimiL performance

Double blind study with only 1 rule:
a collection = 1990 records + 10 engineered near-duplicates

AB
5 strategies

4 collections
RERODOC et
ETH E-Collection
(article + theses),
• Frequent errors
• Cataloging variations

JK
5 strategies

4 collections
CERN (articles),
10 records with
Increasingly severe modifications
(d > c > b > a)
Results: and the winners are ... 

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Recall 10</th>
<th>Recall 20</th>
<th>Recall 50</th>
<th>Average</th>
<th>Timing [min]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2geom</td>
<td>5.6</td>
<td>6.8</td>
<td>8.5</td>
<td>7.0</td>
<td>8.3</td>
</tr>
<tr>
<td>2geombreak</td>
<td>5.0</td>
<td>6.4</td>
<td>7.5</td>
<td>6.3</td>
<td>7.1</td>
</tr>
<tr>
<td>abstract_fallback</td>
<td>5.4</td>
<td>6.9</td>
<td>9.1</td>
<td>7.1</td>
<td>5.3</td>
</tr>
<tr>
<td>boundaries_max</td>
<td>4.9</td>
<td>6.4</td>
<td>9.1</td>
<td>6.8</td>
<td>5.8</td>
</tr>
<tr>
<td>geometric_jk</td>
<td>4.1</td>
<td>7.3</td>
<td>8.6</td>
<td>6.7</td>
<td>6.4</td>
</tr>
<tr>
<td>initials</td>
<td>7.4</td>
<td>8.1</td>
<td>8.8</td>
<td>8.1</td>
<td>5.6</td>
</tr>
<tr>
<td>initialsbreak</td>
<td>7.3</td>
<td>7.3</td>
<td>8.9</td>
<td>7.8</td>
<td>5.3</td>
</tr>
<tr>
<td>maxsim</td>
<td>5.1</td>
<td>6.9</td>
<td>9.4</td>
<td>7.1</td>
<td>56.1</td>
</tr>
<tr>
<td>okapigeom</td>
<td>5.9</td>
<td>7.4</td>
<td>8.6</td>
<td>7.3</td>
<td>7.6</td>
</tr>
<tr>
<td>ubiquist</td>
<td>7.2</td>
<td>8.5</td>
<td>9.0</td>
<td>8.2</td>
<td>7.9</td>
</tr>
</tbody>
</table>
Results: complementarity

Best strategy = combine good and complementary strategies

At least one method was almost always able to place each engineered duplicate in the top 10.

- Initials | ubiquist => 93% of precision
- Initials & ubiquist => 46% of precision
- Initials | ubiquist | okapigeom => 96% of precision
- Initials & ubiquist & okapigeom => 29% of precision
Results: recall & noise analysis

The best strategies withstand well up to 2 altered fields:

1/3 of authors **OK**
1/3 of title **OK**
Several sentences from the abstract **OK**
2 years of difference **OK**
Word permutations **OK**
Author format modification **OK**

Most difficult:

Typos in title (except using the Levenshtein algorithm that is slow)

The best approach (withstands 3 or 4 altered fields)
Combine the results of the 3 best unrelated strategies (withstands 3 or 4 alterations)

Ameliorations:

Ubiquist : use subtitles as well (if cataloguing rules are compatible).
Initials : use digrams, soundex, or initials with shingling (to avoid collisions).
Results: speed

- Python 3.1.x is faster than Python 2.x (~25%)
- Duration increases rapidly with the number of records:
  - Dell Latitude D620 (Intel 1.83GHz) / Geometric mean on 5 fields.
    - 1'000 records in less than a minute
    - 2'000 records in less than 5 min
    - 5'000 records in less than half an hour
    - 10'000 records in less than 3 hours

- Adjustments must be made for large collections.
  - NB: If a catalogue receives 100 records weekly, 5000 record yearly: check the new records against the last two years will take:
    \[ 100 \times (2 \times 5000) = 1'000'000 \text{ comparisons} = \text{much less than 5 minutes} \]
Speed optimisations

In total possibly more than 1'000x or 10'000x:

- **Multiprocessing module (Python>=2.6)**: use all CPUs. Just replace the loop on record pairs by multiprocessing execution. ~ 3x (quadcore)
- **Python code optimisation (profiling)**: > 2x
- **Compile comparison and indexation functions**: C++, ... ~ 2-20x
- **Optimise record global similarity functions**: stop field comparisons as soon as possible ~ 1.2x
- **Intelligent records_comp functions**: e.g. pre-grouping based on subject indexation, then full comparisons on subsets only 100-1000x
- **More efficient caching**: use a relational database ~ 1-20x
### MarcXimiL : other functionalities

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core – similarity.py</td>
<td>Similarity analysis</td>
</tr>
<tr>
<td>Core – batch.py</td>
<td>Batch similarity analysis</td>
</tr>
<tr>
<td>Core – sort.py</td>
<td>Sort and truncates output</td>
</tr>
<tr>
<td>Core – colldescr.py</td>
<td>Collection statistical description</td>
</tr>
<tr>
<td>Core – oai.py</td>
<td>OAI-PMH2 harvesting</td>
</tr>
<tr>
<td>Core – text2xmlmarc.py</td>
<td>Conversion: text MARC VTLS to MARCXML</td>
</tr>
<tr>
<td>enrich.py (prototype)</td>
<td>« More like this » enhancement for catalogues</td>
</tr>
<tr>
<td>monitor.py (prototype)</td>
<td>Invenio based information monitoring</td>
</tr>
<tr>
<td>plagiarism.py (prototype)</td>
<td>Plagiarism detection and KB management</td>
</tr>
<tr>
<td>visualize.py (prototype)</td>
<td>2D Graph representation</td>
</tr>
<tr>
<td>semantic.py (prototype)</td>
<td>Semantic MARCXML collection editor</td>
</tr>
</tbody>
</table>
CDS Invenio: a way of integration

- MarcXimiL as python package (import in Invenio, e.g. bibsim, that would be bibsched compatible).
- Put Invenio search equations in config file instead of file names
- New level of flexibility: load_records_invenio function and associated field parsing functions
- Create a new output function that is able to feed back similarity data to Invenio. Or eventually use enrich.py to do so, then: bibupload -c collection.xml
Conclusion

- MarcXimL = good near-duplicate detection tool
- Offers many other opportunities: information monitoring, plagiarism detection, and so on.
- Main strength: flexibility in setup as well as in future developments.
- For large collections, speed is presently an issue. But this can be improved in many ways.
- Integration with Indico should be quite straightforward (depending on the requirements).