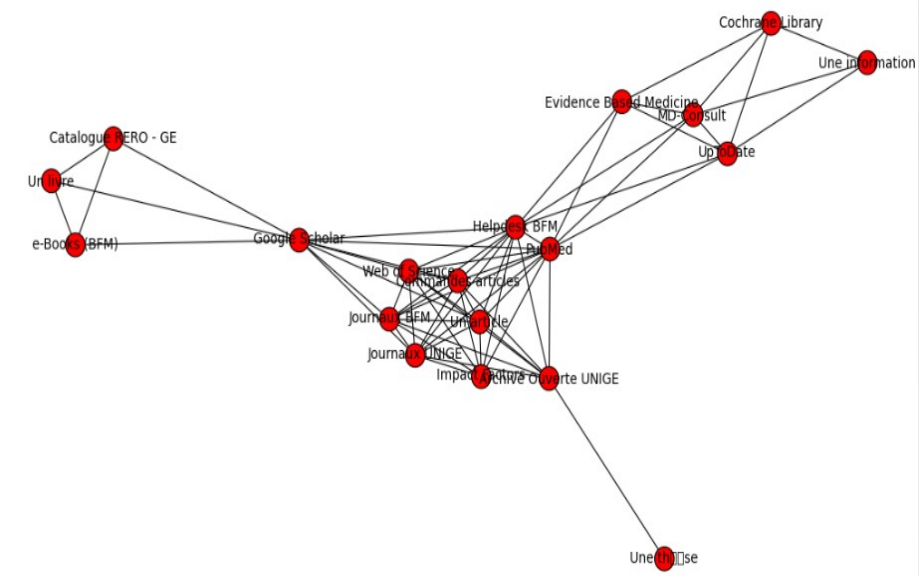
MarcXimiL – Mémoire du CESID



Développement d'un logiciel flexible pour la comparaison de notices bibliographiques

Et application à différentes collections

http://marcximil.sourceforge.net

Alain Borel & Jan Krause

Similarité bibliographique: les enjeux

- Dédoublonnage (foisonnement de l'information et des (quasi-)doublons)
- Classification automatique (p.ex.: FRBR)
- Gestion des contenus (politique documentaire)
- Suggestion de documents (p.ex: more like this)
- Filtrage de notices (veille documentaire)
- Détection de plagiat (une problématique actuelle)

Structure - MarcXimiL

Application	Description
Core – similarity.py	Analyse de similarité
Core – batch.py	Analyse de similarité en batch
Core – sort.py	Trie et tronque l'output
Core – colldescr.py	Description statistique d'une collection
Core – oai.py	Moissonnage de métadonnées par OAI-PMH2
Core – text2xmlmarc.py	Conversion: text MARC VTLS à MARCXML
enrich.py (prototype)	« More litke this » pour catalogues
monitor.py (prototype)	Veille documentaire basée sur Invenio
plagiarism.py (prototype)	Détection et gestion de la base de connaissance
visualize.py (prototype)	Représentation des résultats en Graph 2D
semantic.py (prototype)	Éditeur de relations sémantiques

MarcXimiL - philosophie

Philosophie de développement:

- **Flexible** (Configuration, possibilités de développement)**,**
- **Standard** (MARCXML, OAI-PMH2, structure: UNIX-like)**,**
- **Logiciel libre** (GNU GPLv3)**,**
- **Compatible** (Requiert uniquement Python de 2.4.0 à 3.1.1)**,**
- **Multiplateforme** (Linux, UNIX, MacOSX, Windows XP..7),
- **Simple à installer** (Suffit de désarchiver)**,**
- **Configuration claire et centralisée (stratégie),**
- **Techniques de Recherche d'Information** (éprouvées)

Exécution phase 1 : chargement

- **Programme principal : similarity.py**
- Plusieurs collections MARCXML peuvent être précisées dans le fichier de configuration et chargées.
- ∀ collections la fonction **records_load** (module load_records) est appelée.
- Les collections sont **stockées dans une variable globale** définie dans le module des variables globales (économie de mémoire).

Exécution phase 1.1 : parsing

- La fonction records_load appelle micro_dom
- **micro_dom** est spécialisée dans l'**extraction des notices** de la collection soit les éléments <record> (plus **performant** qu'un parseur DOM).
- Ensuite, la fonction records_load appelle les **fonctions de parsing des champs** sur chaque notice (module parse_records utilise **minidom**).
- Seul les **champs sélectionnés** dans le **fichier de configuration** sont extraits (une **fonction de parsing** par champ doit y être précisée).

Exécution phase 1.1 : parsing

- Il y a 4 fonctions de parsing de champs.
- Elles partagent une interface de programmation commune (~ même arguments, 1 output).
  - Une pour chaque type de champ MARC:
    - Champs de contrôle (ex: id notice : 001)
    - Champs non répétables (ex.: titre : 245$a)
    - Champs répétables (ex.: auteurs sec. : 700$a)
  - Une pour concaténer des champs MARC, ce qui a pour application:
    - Fusionner des champs apparentés pour les analyser ensemble (ex: titre + sous-titre)

Exécution phase 1.2: pré-traitement

- **Un pré-traitement adéquat est automatiquement sélectionné** lors du chargement (fonction represent_field appelée par records_load).
  - dépend de la famille de la fonction de similarité associée
- Il y a actuellement 4 fonctions de pré-traitement:
  - RAW : le champ est chargé tel-quel
  - WC : prépare un dictionnaire des TF pour chaque champ et un dictionnaire des IDF sur la collection pour chaque terme
  - INITIALS : idem, mais basé sur les initiales
  - SHINGLES : idem, mais basé sur des sacs de mots
- En général, un prétraitement déclenche la normalisation des champs: casse, diacritiques, ponctuation et espaces.

Exécution 1.3: caching

- **Limitation de l'usage de la mémoire vive** pour les grandes collections.
- Le cache est directement alimenté par load_records.
- La collection est **divisée en segments**
  - dont une fraction est gardée dans la mémoire vive (dans une variable globale)
  - le reste est stocké sur le disque dur sous forme d'objets Python sérialisés (pickle)
- Gestion du cache: **first in, first out**.

Exécution phase 2: comparaisons

- Le programme principal (similarity.py) appelle une des fonctions **records_comp** (module compare_records).
- Celles-ci déterminent les **paires de notices à comparer** et leur ordre. Elles partagent une interface de programmation commune.
- Trois de ces fonctions sont intéressantes:
  - records_comp_single (une collection, triangulaire) -> doublons
  - records_comp_2collections_caching (2 collections, comparaisons entre les collections uniquement) -> veille, détection de plagiat
  - records_comp_multiple_caching (multiples collections, compare toutes notices, inter- et intracollections) -> cas général

Exécution phase 2.1: sim globale

- La fonction records_comp appelle une des **fonctions de similarité globale** sur chaque paire de notices (module global_similarity). Une telle fonction gère:
  - l'exécution de fonctions de **comparaisons entre les champs**
  - ainsi que l'**intégration des valeurs de similarité** de chaque paire de champs en une valeur globale entre notices.
- Des exemples de fonctions de similarité globale suivent. Elles partagent une interface de programmation commune.

Exécution phase 2.1: moyennes

- **Similarité globale** en utilisant des moyennes pondérées: 6 fonctions
- Trois types de **moyennes classiques** sont implémentés: arithmétique, harmonique et géométrique.
  - impliquent le calcul systématique de similarité sur chaque paire de champs (prends du temps).
- Les trois **variantes « breakout »**
  - stoppent le traitement si le résultat de la comparaison d'un champ est au dessous d'un seuil (< 0.8)
  - les champs sont traités par ordre alphabétique, p.ex.: « 01year », « 02title », etc..

Exécution phase 2.1: boundaries_max

- Cette fonction, comme maxsim, et est basée sur la similarité maximale. Elle comporte deux différences:
  - ∀ champ, un paramètre optionnel « threshold »
    - permet de ne pas prendre en compte la similarité du champ dans le calcul de la similarité globale (si inférieure au seuil).
  - ∀ champ, un param. optionnel « global-threshold »
    - permet d'attribuer une similarité par défaut très basse à la paire de notices (si inférieure au seuil).
- La fonction boundraries est similaire mais basée sur une moyenne.

Exécution phase 2.1: ubiquist

Cette fonction nécessite les champs suivants: recids, title_dice, authors, year, doi, abstract.

1. Une faible valeur est attribuée à similarité globale.

2. Si les DOI sont identiques, 1.0 est renvoyé et le traitement s'arrête.

3. Sinon, si la similarité des auteur est >= 0.85 et celle de l'année est >=0.9, la similarité globale devient 0.85 * la similarité des auteurs.

4. Puis la similarité des l'abstracts et des titres est calculée. La similarité globale devient le maximum de: la similarité globale actuelle, la similarité des titres, et la similarité des abstracts.

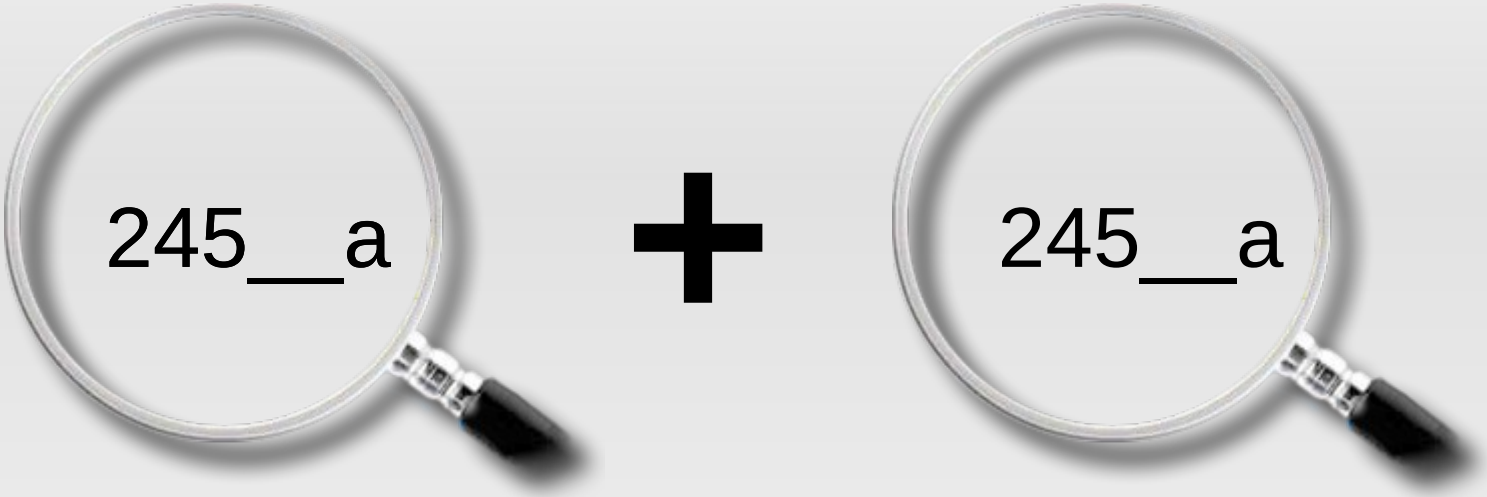
Exécution phase 2.2: comparaisons de champs

```
def functionname__ENDING(fieldtype, field1, field2, options = None)
```

- `ENDING:` famille de fonctions
- `fieldtype:` identificateur du ou des champs examinés
- `field1`, `field2:` contenus des champs dans les 2 notices
- `options:` réservé pour d'éventuels futurs développements

Valeur de sortie: None ou [0.0;1.0]

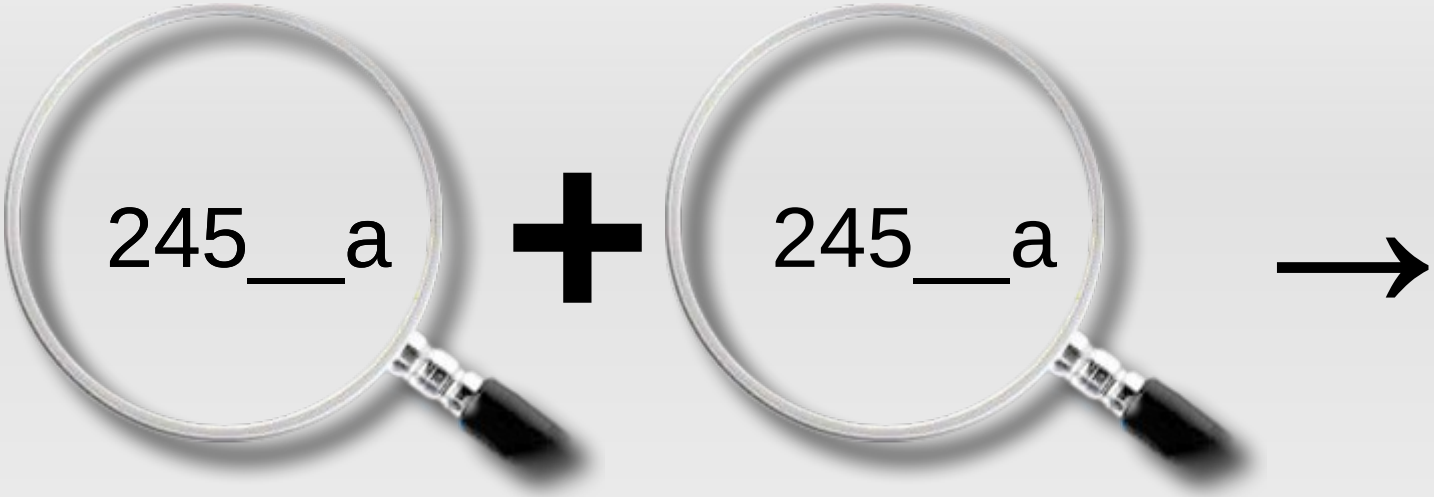
Exécution phase 2.2: comparaisons de champs - RAW



Méthodes de comparaison directe:
- identificateurs (DOI...)
- année
- ensembles d'éléments (ex. auteurs)
- Levenshtein

etc.

Exécution phase 2.2: comparaisons de champs - WC



Méthodes utilisant les fréquences des termes,
y.c. dans l'ensemble de la collection (dict. global):
- modèle vectoriel de recherche d'information (ntf, nidf; cosinus/Dice/Jaccard)
- méthode probabiliste Okapi BM25

Exécution phase 2.2: comparaisons de champs - autres

INITIALS: les champs sont représentés par la liste des 1ers caractères des mots (utile pour les titres de journaux, souvent abrégés)

SHINGLES: similaire à la famille WC, mais à la place des mots on observe les suites de *n* mots (typiquement 4) => plus de sensibilité à la forme des phrases, pas seulement au vocabulaire

Exécution phase 2.3: output

report_toto()

quickndirty: sortie directe de la comparaison de notices (dictionnaire Python), sans formattage

tab: valeurs des comparaisons de champs + sim. globale => tri et analyse des résultats simples

xml: format XML simple, évt. utile pour de futures applications

Configuration

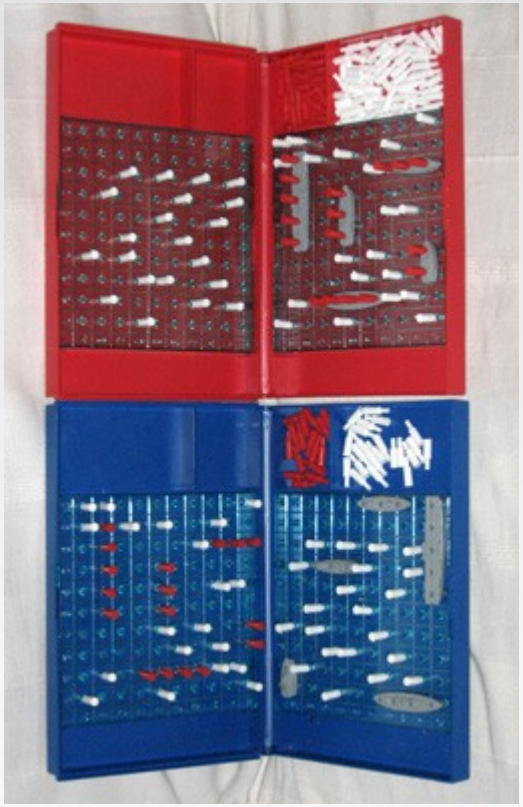
etc/similarity_config.py

```
INPUT_FILES = ['testdataCERNb.xml']

records_comp = records_comp_single

report = report_tab
globalvars.output_threshold = -1 # use -1 to output everthing

record_structure = { \
    '01recid'        : {'marc'      : '001',
                        'weight'    : 0,
                        'parse-func': parse_controlfield,
                        'comp-func' : fields_concat__raw },

    '01recid2'       : {'marc'      : ['0247 a'],
                        'weight'    : 0,
                        'parse-func': parse_concat,
                        'comp-func' : fields_concat__raw },

    '02year'         : {'marc'      : '260  c',
                        'weight'    : 1,
                        'parse-func': parse_nonrep,
                        'comp-func' : years_comp__raw },
    '03year2'        : {'marc'      : '269  c',
                        'weight'    : 1,
                        'parse-func': parse_nonrep,
                        'comp-func' : years_comp__raw },

    '04title'        : {'marc'      : ['245  a', '245  b'],
                        'weight'    : 2,
                        'parse-func': parse_concat,
                        'comp-func' : okapibm25__wc }}

record_rules = geometric_mean_breakout
```

Tests: méthodologie

1990 notices + 10 quasi-doublons artificiels
Recherches de doublons en double aveugle

AB
- RERODOC et ETH E-Collection (articles + thèses), 4 colls. initiales
- erreurs de saisie courantes, style de cataloguage



JK
- CERN (articles + preprints), 1 collection
- 10 notices avec des modifications de + en + sévères (d > c > b > a)

Tests: stratégies JK

Structures de notices liées aux stratégies globales déjà présentées:
- abstract_fallback
- boundaries_max
- maxsim
- ubiquist

- geometric_jk (année, auteurs, titre, abstract)

Tests: stratrégies AB

Structure de notice +/- commune:

- L'année
- Le titre (+ sous-titre)
- Les auteurs (+ directeurs de thèse)
- La source (journal ou autre)

Moy. géométrique (variante avec interruption)

1) Traitement vectoriel du titre (2geom)

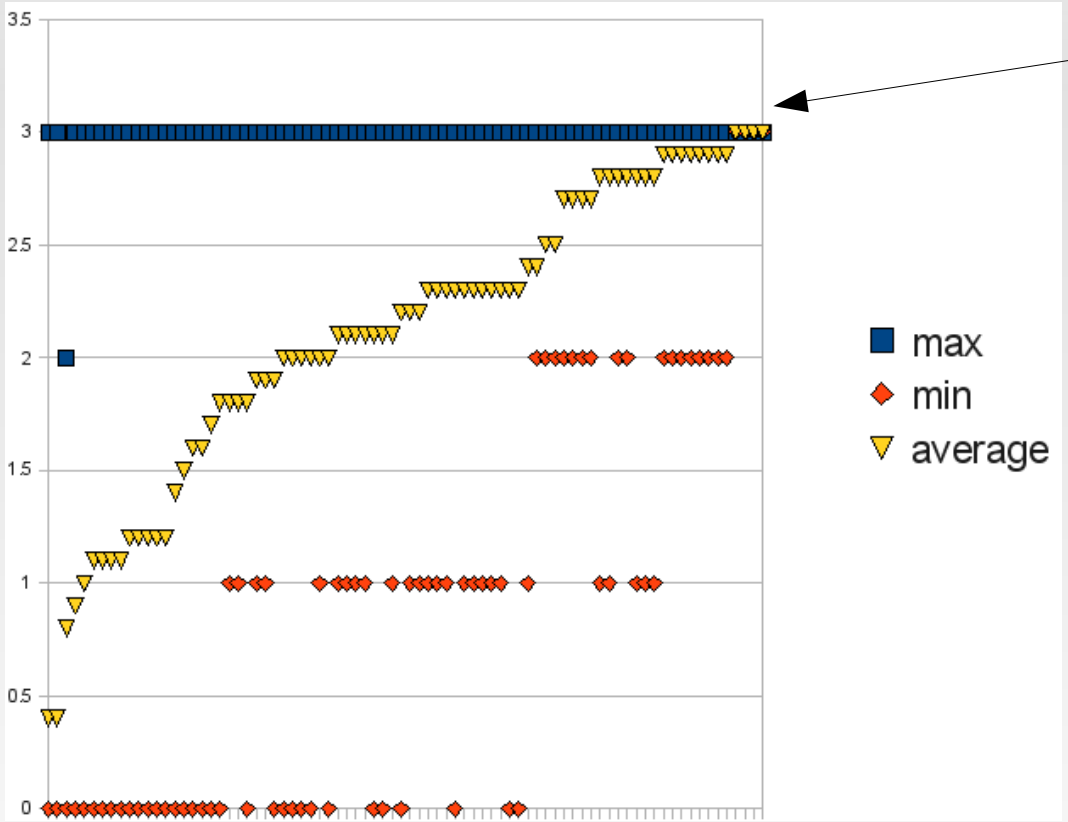
2) Okapi BM25 sur le titre (okapigeom)

3) Ensemble des initiales du titre et des auteurs

Résultats: and the winners are...

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

Complémentarité



Au moins une des méthodes
a placé tous les doublons parmi
les 10 1ers résultats

Initials | ubiquist
=> 93% de précision
Initials & ubiquist
=> 46% de précision

Initials | ubiquist | okapigeom
=> 96% de précision
Initials & ubiquist & okapigeom
=> 29% de précision

Analyse du rappel

Les meilleures stratégies résistent à plusieurs changements simultanés parmi les suivants:

~~1/3 des auteurs~~ **OK**

~~1/3 du titre~~ **OK**

~~Quelques phrases de l'abstract~~ **OK**

2 ans de différence **OK**

Permutations de mots **OK**

Modification du format des auteurs **OK**

Plus problématiques:

Mots du titre modifiés

Analyse du bruit

Quelques tendances notables:

- des faux positifs dûs à des champs non pris en compte (ex. sous-titre)

Outils de management : Management de l'information vs. Outils de management : Management des processus

- des faux positifs inhérents à la méthode (ex. collisions pour initials, sous-ensemble de termes pour okapigeom)

RF System Design for the EMMA FFAG vs. Radiation Damage Studies for the Slow Extraction from SIS100

Water Cherenkov Detectors response to a Gamma Ray Burst in the Large Aperture GRB Observatory

Discussion : suggestions d'améliorations

- La fonction de similarité **« initials » est très efficace**, mais n'est pas adaptée pour les grandes collections (**collisions** dès 2000 notices). Ceci peut être amélioré par l'utilisation de **digrammes** ou de l'algorithme **soundex**.
- Les stratégies basés sur les moyennes géométriques peuvent être améliorées par l'inclusion de l'analyse de l'abstract.

Discussion : suggestions d'améliorations

- La stratégie « **ubiquist** » est la plus efficace. Elle exploite efficacement l'abstract lorsqu'il est présent. Pour l'améliorer, lorsque les règles de catalogage le permettent, il faut **fusionner le sous-titre** avec le titre.
- A propos des articles, l'usage des **informations concernant le périodique** (titre, volume, numéro, et pages) peuvent améliorer l'analyse de similarité. Mais pour en tirer avantage il faut mettre au point des algorithmes spécifiques à chaque institution.
- **Exploiter la complémentarité** des stratégies en combinant des stratégies complémentaires.

Discussion: vitesse

- La différence entre systèmes d'exploitations (XP/Ubuntu) est à la base de l'ordre de 10%.
- Python 3.x est plus rapide que Python 2.x (~15% avec Psyco, ~25% sans Psyco).
- La durée augmente rapidement avec les notices:

  Dell Latitude D620 (Intel 1.83GHz) / Geometric mean on 5 fields.

  - 1'000 notices en moins d'une minute
  - 2'000 notices en moins de 5 minutes
  - 5'000 notices en moins d'une ½ heure
  - 10'000 notices en moins de 3h
- Des ajustements doivent êtres faits pour pouvoir traiter de grandes collections.

Perspectives : vitesse

Pour améliorer la vitesse (peut-être plus de 100x):

- Module multiprocessing (Python>=2.6) : permet d'utiliser tous les CPUs en remplaçant la boucle sur les paires de notices. ~ 3x (quadcore)
- Optimisation du code (en Python) : profiler et améliorer. ~ 2x
- Compiler les fonctions de comparaison : C++, fortran, ... ~ 2-10x
- Optimiser les fonctions de similarité globale : faire en sorte que les comparaisons de champs s'arrêtent le plus vite possible si il y a peu de chances qu'il s'agisse d'un doublon. ~ 1.2x
- Fonctions records_comp intelligentes : tri ou pré-groupement rapide puis comparaison complète sur des groupes de taille réduite >>10x
- Meilleur caching : utiliser un SGBR (grandes collections) ~ 1-20x

Perspectives : MarcXimiL package

- Un paquet python est un ensemble de modules, souvent distribué dans le format .egg et répertorié dans PyPI (et easy_install).
- Quelques ajustements permettent de transformer MarcXimiL pour qu'ils soit importable sous cette forme par d'autres logiciels Python. Applications probables:
  - Utilisation dans CDS Invenio
  - Interface graphique (GUI) ... à ce propos une bare de progression pourrait être introduite via une nouvelle variable globale.

Perspectives : CDS Invenio

CDS Invenio est un excellent logiciel intégré de gestion de bibliothèque et peut bénéficier des fonctionnalités de MarcXimiL, à condition que:

- MarcXimiL soit importable en paquet.
- Des équations de recherche à la place du nom de fhicher des collections.xml dans le fichier de config.
- Accès direct aux notices d'Invenio:

```
>>> from invenio.search_engine import perform_request_search
>>> coll = perform_request_search(cc="ARTICLES")   # get a collection
```

- Et faire des ajustements (records_load_invenio)

Conclusion

- Développement d'un logiciel de comparaison

  de notices MARC XML en Python, utilisant entre autres des modèles de recherche d'information **OK**
- Test pour la détection de doublons en utilisant entre autres des méthodes de recherche d'information **Bonne précision en combinant différentes approches Vitesse à améliorer**
- Nombreuses autres utilisations possibles **Plusieurs applications prototypées**