Retrieving and Storing Data from Folksonomies

Hugo Alves  André Santanché

Technical Report - IC-12-15 - Relatório Técnico

May - 2012 - Maio

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.
Retrieving and Storing Data from Folksonomies

Hugo Alves∗ Andre Santanchê†

∗Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. Pesquisa desenvolvida com suporte financeiro parcial do CNPq e FAPESP
†Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.
Contents

1 Introduction

2 Formal Model for Folksonomies

3 Implementation
   3.1 Database Model
   3.2 Tool Model
   3.3 Source Code

4 Folksonomy Systems
   4.1 Flickr
      4.1.1 Authentication
      4.1.2 Definitions
      4.1.3 Access
      4.1.4 Request Protocols
      4.1.5 Response Formats
      4.1.6 API Methods
      4.1.7 API Examples
         i Getting Photo Information
         ii Photo Galleries Information
         iii Public List of User Contacts
         iv Public Photos of User Contacts
         v Latest Public Photos
         vi Most Popular Tags
   4.2 Delicious
      4.2.1 Authentication
         i OAth Python Library
      4.2.2 Feeds
         i Update Rate
         ii Feeds Available
         iii URL Patterns for Feeds

5 Conclusion
1 Introduction

The popularization of web-based systems offering services for content storage, indexing and sharing fostered a rapid growth of content available on-line. There are more than 5 billion images hosted on Flickr\(^1\) and more than 180 million URL addresses on Delicious\(^2\). These systems increasingly rely on tag-based metadata to organize and index all the amount of data. The tags are provided by users connected in social networks, who are free to use any word as tag; there is no central control. The term folksonomy – combining the words “folk” and “taxonomy” \([8]\) – has been used to characterize the product which emerges from this tagging in a social environment.

In order to analyze, index and classify their content, web systems compare tags attached to resources. Instead of considering the semantics of each tag in the comparison, tag-based systems usually rely on string matching approaches. While ontologies are increasingly adopted to enrich tags semantics, one common problem with the proposals to associate tags to formal ontologies concerns their unidirectionality, i.e., ontologies improve tags semantics, or the implicit/potential semantics of folksonomies is extracted to produce ontologies.

Differently from traditional techniques, we proposed a fusion approach, called *folksonomized ontology* (FO), which goes beyond this unidirectional perspective \([3]\). In one direction, the ontologies are “folksonomized”, i.e., the latent semantics from the folksonomic tissue is extracted and fused to ontologies. On the other direction, the knowledge systematically organized and formalized in ontologies gives structure to the folksonomic semantics, enhancing operations involving tags, e.g., content indexation and discovery. The folksonomic data fused to an ontology will tune it up to contextualize inferences over the repository.

In our fusion approach both ontologies and folksonomies are enriched in the process. This symbiosis is explored to:

**Tag disambiguation:** by finding groups of related tags and mapping them to ontology concepts, the FO can be applied to disambiguate tags and find the ones that are more related, going beyond statistical analyses by using semantic similarity metrics.

**Tag suggestion:** the current folksonomy systems consider only co-occurrence information to suggest related tags to users; a FO has a richer set of semantic relations among concepts, supporting suggestion of tags that were not used together before – folksonomies cannot do that.

**Semantic similarity:** a FO can support the computation of semantic similarity between concepts and, by extension, between tags; so, they can expand the usual techniques that focus only at syntactical similarity and co-occurrence of tags, achieving better results in discovery operations.

**Ontology evolvement:** a FO can be used to find missing relations in ontologies; the high co-occurrence between two groups of tags, and their corresponding concepts, can indicate a necessary relation in the ontology, if it does not exist yet.

\(^1\)http://blog.flickr.net/en/2010/09/19/5000000000/ - retrieved on November, 2011

In order to build a practical tool to validate our proposal, we have implemented a software module to access and collect data from folksonomy-based web systems. We confronted the model adopted by each system with models proposed in the literature, in order to propose a generic model to represent and store the collected data.

The goal of this technical report is to detail this work. In Section 2 we synthesize related work concerning formal models to represent folksonomies. In Section 3 we discuss implementation aspects of our module, which interacts with these systems to collect and store folksonomic data. In Section 4 we analyse the approach adopted by folksonomy-based systems to represent and store their folksonomies, including their Application Programming Interfaces (APIs). The systems that will be analysed here are Delicious [1] and Flickr [2].

2 Formal Model for Folksonomies

In folksonomy-based systems, users can attach a set of tags to resources. These tags are not tied to any central vocabulary, so the users are free to create and combine tags. Some strengths of folksonomies are their easiness of use and the fact that they reflect the vocabulary of their users [5]. In a first glimpse, tagging can transmit the wrong idea of a poor classification system. However, thanks to its simplicity, users are producing millions of correlated tags. It is a shift from classical approaches – in which a restricted group of people formalize a set of concepts and relations – into a social approach – in which the concepts and their relations emerge from the collective tagging [7]. In order to perform a systematic folksonomy analysis, to subsidize the extraction of its potential semantics, researchers are proposing models to represent its key aspects. Gruber [4] models a folksonomy departing from its basic “tagging” element, defined as the following relation:

\[
\text{Tagging}(\text{object}, \text{tag}, \text{tagger}, \text{source})
\]

In which \textit{object} is the described resource, \textit{tag} is the tag itself – a string containing a word or combined words –, \textit{tagger} is the tag’s author, and \textit{source} is the folksonomy system, which allows to record the tag provenience (e.g., Delicious, Flickr etc.).

In order to formalize a folksonomy Mika [6] departs from a tripartite graph with hyperedges. There are three disjoint sets representing the vertices:

\[
T = \{t_1, \ldots, t_k\}, \quad U = \{u_1, \ldots, u_l\}, \quad R = \{r_1, \ldots, r_m\}
\]

In which the sets \(T\), \(U\) and \(R\) correspond to tags, users and resources sets respectively.

A folksonomy system is a set of annotations \(A\) relating these three sets:

\[
A \subseteq T \times U \times R
\]

The folksonomy itself is a tripartite hypergraph:

\[
H(T) = \langle V, E \rangle
\]

In which \(V = T \cup U \cup R\), and \(E = \{\{t, u, r\} \mid (t, u, r) \in A\}\)
The folksonomy analysis can be simplified and directed by reducing this tripartite hypergraph into three bipartite graphs: \( TU \) relating tags to users, \( UR \) relating users to resources and \( TR \) relating tags to resources [6]. A graph \( TT \) is a relevant extension of this model for representing relations between tags. It allows to represent the co-occurrence of tags. The same approach can be applied to the user and resource sets.

3 Implementation

In this section we describe the tool we have implemented to retrieve data from Delicious and Flickr, as well as the database model. The implementation adopted the python language\(^3\). The data was stored by using the SQLite\(^4\) database manager.

The access of Flickr data required the implementation of the code to handle its protocol and to treat the results of the requests. The module to retrieve the data from Delicious, on the other hand, adopted a third-party library: DeliciousAPI\(^5\).

During the development, we faced an unexpected behavior of the library. The reason was a change in the structure of Delicious’ pages. This is still a challenge to be faced in web services research, mainly in public web services: whenever servers change their interfaces, the clients will break if there is not backwards compatibility. In order to fix it, we developed a patch to adjust the access, contributing to the community to fix the error\(^6\).

3.1 Database Model

As mentioned before, our database model results from a comparative analysis of related work and models adopted by folksonomy based systems. The logical modeling is depicted in the Figure 1. There are three main entities – User, Resource and Tag – following the model presented in Section 2. The Resource entity was specialized to better characterize its representation in the systems Flickr (Photo) and Delicious (URL). As the database was designed to simultaneously afford tags of many systems, the Source entity keeps track of the origin of the tag. This is an important information, since our algorithms were designed to work with a single folksonomy system each time.

The physical modeling is based on the logical one. However, it includes control tables and flags to indicate: users already processed, resources already visited, and so on. Some auxiliary tables – prefixed by count – record the counts of occurrences or co-occurrences of analyzed items; they will support statistics produced by our system. It is composed of 13 tables, as we further detail:

**control**: records control data related to the execution of the process, like the timestamps of the last requisition to the systems. (name TEXT, value TEXT)

**count_resource**: records the count of each resource. (rid INTEGER, sid INTEGER, count INTEGER)

---

\(^3\)http://www.python.org/

\(^4\)http://www.sqlite.org/

\(^5\)https://github.com/quuxlabs/DeliciousAPI

\(^6\)https://github.com/quuxlabs/DeliciousAPI/commit/1cea76941797d6807ac8411b0e8437aa92a35a5
**count_rt**: records the count of each pair (resource, tag). \((\text{rid INTEGER}, \text{tid INTEGER}, \text{sid INTEGER}, \text{count INTEGER})\)

**count_tag**: records the count of each tag. \((\text{tid INTEGER}, \text{sid INTEGER}, \text{count INTEGER})\)

**count_tt**: records the count of each pair of tags. \((\text{t1 INTEGER}, \text{t2 INTEGER}, \text{sid INTEGER}, \text{count INTEGER})\)

**count tu**: records the count of each pair (tag, user). \((\text{tid INTEGER}, \text{uid INTEGER}, \text{sid INTEGER}, \text{count INTEGER})\)

**count ur**: records the count of each pair (user, resource). \((\text{uid INTEGER}, \text{rid INTEGER}, \text{sid INTEGER}, \text{count INTEGER})\)

**count user**: records the count of each user. \((\text{uid INTEGER}, \text{sid INTEGER}, \text{count INTEGER})\)

**resource**: records each resource, which has an internal id \text{rid} and a reference to the specific identification in the source system – e.g., a URL for Delicious or a Flickr internal identifier – in the \text{value} field. \((\text{rid INTEGER PRIMARY KEY, sid INTEGER, value TEXT, done NUMERIC})\)
source: records each source, which has an internal id \texttt{sid} and the specification of the source (e.g., delicious or flickr) in the \texttt{value} field. (\texttt{sid INTEGER}, \texttt{value TEXT})

tag: records each tag, assigning an internal id \texttt{tid} to each one. (\texttt{tid INTEGER}, \texttt{sid INTEGER}, \texttt{value TEXT})

tagging: records each tagging triple (resource, tag, user) in a given source. (\texttt{sid INTEGER}, \texttt{uid INTEGER}, \texttt{rid INTEGER}, \texttt{tid INTEGER}, \texttt{time TEXT})

user: records each user, assigning for each value (string of the username) an internal id. (\texttt{uid INTEGER}, \texttt{sid INTEGER}, \texttt{value TEXT}, \texttt{done NUMERIC})

The complete schema of the database is as follows:

```sql
CREATE TABLE source (sid INTEGER PRIMARY KEY, value TEXT);
CREATE TABLE control (name TEXT, value TEXT);
CREATE TABLE user (uid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE NOCASE, done NUMERIC, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid, value));
CREATE TABLE tag (tid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE NOCASE, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid, value));
CREATE TABLE resource (rid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE NOCASE, done NUMERIC, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid, value));
CREATE TABLE tagging (sid INTEGER, uid INTEGER, rid INTEGER, tid INTEGER, time TEXT, FOREIGN KEY(sid) REFERENCES source(sid), FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(tid) REFERENCES tag(tid), UNIQUE(sid, uid, rid, tid));
CREATE TABLE 'count_tu' (tid INTEGER, uid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(tid) REFERENCES tag(tid), FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(tid, uid, sid));
CREATE TABLE 'count_ur' (uid INTEGER, rid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(uid, rid, sid));
CREATE TABLE 'count_rt' (rid INTEGER, tid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(tid) REFERENCES tag(tid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(rid, tid, sid));
CREATE TABLE 'count_tt' (t1 INTEGER, t2 INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(t1) REFERENCES tag(tid), FOREIGN KEY(t2) REFERENCES tag(tid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(t1, t2, sid));
CREATE TABLE 'count_tag' (tid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(tid) REFERENCES tag(tid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(tid, sid));
CREATE TABLE 'count_user' (uid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(uid, sid));
CREATE TABLE 'count_resource' (rid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(rid, sid));
```
3.2 Tool Model

Figure 2 presents a UML diagram of the main classes of our tool. The five classes depicted in the figure are:

**Source**: represents a specific folksonomy system.
Database: abstracts and centralizes all database operations.

Crawler: abstract class whose instances represent a crawler a its operations.

DeliciousCrawler: implements the crawler operations for the Delicious context.

FlickrCrawler: implements the crawler operations for the Flickr context.

The Crawler abstract class makes simpler to extend the system, since it standardize the API to the crawler mechanism. Each new system will require only a new implementation extending Crawler.

3.3 Source Code

In this section we present the source of the tool, according the model presented in the previous section. Each block contains a class and comments explaining its functionality.

```python
import sys  # print without \n
import urllib  # http request and manipulation
import sqlite3  # database
import time, datetime  # time processing
from xml.etree.ElementTree import parse  # parser xml
import deliciousapi  # delicious data access
import random

class Source(object):
    """ Object Source - encapsulates the current folksonomy system """
    def __init__(self, source = 'all'):
        if source in ['all', 'flickr', 'delicious']:
            self.name = source
        else:
            self.name = 'all'

    def __str__(self):
        return self.name

    """ true if the source is delicious """
    def isDelicious(self):
        return self.name == 'delicious'

    """ true if the source is flickr """
    def isFlickr(self):
        return self.name == 'flickr'

""" Object Database - encapsulates the database operations """
```
class DataBase(object):
    
    """ constructor - try to create the tables if they don't exist
    dbfile - name of the database file
    """
    def __init__(self, dbfile = 'folk.sqlite'):
        self.conn = sqlite3.connect(dfile)
        self.conn.text_factory = str
        self.cursor = self.conn.cursor()
        self.cursor.execute('CREATE TABLE IF NOT EXISTS user (uid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE NOCASE, done NUMERIC, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid, value))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS tag (tid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE NOCASE, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid, value))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS source (sid INTEGER PRIMARY KEY, value TEXT COLLATE NOCASE)')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS resource (rid INTEGER PRIMARY KEY, sid INTEGER, value TEXT COLLATE NOCASE, done NUMERIC, FOREIGN KEY(sid) REFERENCES source(sid), UNIQUE(sid, value))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS tagging (sid INTEGER, uid INTEGER, rid INTEGER, tid INTEGER, time TEXT, FOREIGN KEY(sid) REFERENCES source(sid), FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(tid) REFERENCES tagging(tid))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS control (name TEXT, value TEXT)')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS count_tag (tid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(tid) REFERENCES tag(tid), FOREIGN KEY(sid) REFERENCES source(sid))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS count_user (uid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(sid) REFERENCES source(sid))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS count_resource (rid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(sid) REFERENCES source(sid))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS count_tu (tid INTEGER, uid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(tid) REFERENCES tag(tid), FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(sid) REFERENCES source(sid))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS count_ur (uid INTEGER, rid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(uid) REFERENCES user(uid), FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(sid) REFERENCES source(sid))')
        self.cursor.execute('CREATE TABLE IF NOT EXISTS count_rt (rid INTEGER, tid INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(rid) REFERENCES resource(rid), FOREIGN KEY(tid) REFERENCES tag(tid), FOREIGN KEY(sid) REFERENCES source(sid))')
self.cursor.execute('CREATE TABLE IF NOT EXISTS count_tt (t1 INTEGER, t2 INTEGER, sid INTEGER, count INTEGER, FOREIGN KEY(t1) REFERENCES tag(tid), FOREIGN KEY(t2) REFERENCES tag(tid), FOREIGN KEY(sid) REFERENCES source(sid))')
self.conn.commit()

""" destructor - close the resources """
def __del__(self):
    self.cursor.close()
    self.conn.close()

""" find the id of the given entity
if more than one entity is given, the order [tag; user; resource; source]
is important
return the id or 'None' if failed """
def getID(self, sid, tag = None, user = None, resource = None, source = None):
    res = None
    try:
        if tag is not None:
            self.cursor.execute('select tid from tag where value = ? and sid = ?', (tag.lower(), sid,))
        elif user is not None:
            self.cursor.execute('select uid from user where value = ? and sid = ?', (user.lower(), sid,))
        elif resource is not None:
            self.cursor.execute('select rid from resource where value = ? and sid = ?', (resource.lower(), sid,))
        elif source is not None:
            self.cursor.execute('select sid from source where value = ?', (source.lower(),))
        res = self.cursor.fetchone()
        if res is not None: res = res[0]
    except:
        pass
    return res

""" return the id of the given entity, if it isn’t in the database, insert
and return the id
if more than one entity is given, the order [tag; user; resource; source]
is important
return a pair <id, already>:
id - the id of the given entity - None if failed
already - [boolean] True if the entity was already in the db; False otherwise - None if failed"""
def getIDInsert(self, sid, tag = None, user = None, resource = None, source = None):
    # try to find the id in database
    id = self.getID(sid, tag = tag, user = user, resource = resource, source = source)
    if id is not None: return id, True

    # if it isn't, try to insert
    try:
        if tag is not None:
            self.cursor.execute('insert into tag values(NULL, ?, ?)', (sid, tag.lower(),))
            self.conn.commit()
            return self.getID(sid, tag = tag), False
        elif user is not None:
            self.cursor.execute('insert into user values(NULL, ?, ?, 0)', (sid, user.lower(),))
            self.conn.commit()
            return self.getID(sid, user = user), False
        elif resource is not None:
            self.cursor.execute('insert into resource values(NULL, ?, ?, 0)', (sid, resource.lower(),))
            self.conn.commit()
            return self.getID(sid, resource = resource), False
        elif source is not None:
            self.cursor.execute('insert into source values(NULL, ?)', (source.lower(),))
            self.conn.commit()
            return self.getID(sid, source = source), False
    except Exception,e:
        print '!', e
        return None, None

""" execute the given query and return the results of it
"""
def execute(self, query, params):
    try:
        self.cursor.execute(query, params)
        self.conn.commit()
        return self.cursor.fetchall()
    except Exception,e:
        print '!', e
        return None

""" execute the given select query and return the results of it
"""
def select(self, query, params):
    try:
        self.cursor.execute(query, params)
        return self.cursor.fetchall()
    except Exception, e:
        print '(!)', e
        return None

    """ execute the given update query
    if the parameter 'commit' is False, the commit is delayed
    ""
    def update(self, query, params, commit = True):
        try:
            self.cursor.execute(query, params)
            if commit: self.conn.commit()
        except Exception, e:
            print '(!)', e

    """ execute the given insert query
    if the parameter 'commit' is False, the commit is delayed
    ""
    def insert(self, query, params, commit = True):
        try:
            self.cursor.execute(query, params)
            if commit: self.conn.commit()
        except Exception, e:
            print '(!)', e

    """ execute the given delete query
    if the parameter 'commit' is False, the commit is delayed
    ""
    def delete(self, query, params, commit = True):
        try:
            self.cursor.execute(query, params)
            if commit: self.conn.commit()
        except Exception, e:
            print '(!)', e

    """ execute the commit in the database
    ""
    def commit(self):
        self.conn.commit()
""" constructor - sets the database
"""
def __init__(self, db = None):
    if db is None:
        self.db = DataBase()
    else:
        self.db = db

""" destructor - deletes the database
"""
def __del__(self):
    del self.db

""" calls the select of the current database
"""
def dbSelect(self, query, params):
    return self.db.select(query, params)

""" calls the update of the current database
"""
def dbUpdate(self, query, params):
    self.db.update(query, params)

""" calls the delete of the current database
"""
def dbDelete(self, query, params):
    self.db.delete(query, params)

""" get the id of the given entity.
the optional parameter 'insert' indicates if is necessary to insert the
entity in the database
"""
def dbGetId(self, sid, insert = False, tag = None, user = None, resource = None, source = None):
    if not insert:
        return self.db.getID(sid, tag, user, resource, source)
    else:
        return self.db.getIDInsert(sid, tag, user, resource, source)

""" insert a 'tagging object' (a triple user, resource, tag associated with a
source) in the database
""
def dbInsertTagging(self, sid, uid, rid, tid, time = None):
# no parameter (except 'time') can be 'None'
if sid is None or uid is None or rid is None or tid is None:
    print 'invalid values', sid, uid, rid, tid
    return

# transaction [begin] - delay commit until transaction ends
# update the counters -
# if there's no entity in db: counter = 1; else counter += 1.
    r = self.db.select('select count from count_tag where tid = ? and sid = ?', (tid, sid))
    if r == []: self.db.insert('insert into count_tag values (?, ?, 1)', (tid, sid), commit = False)
    else: self.db.update('update count_tag set count = ? where tid = ? and sid = ?', (int(r[0][0]) + 1, tid, sid), commit = False)
    r = self.db.select('select count from count_user where uid = ? and sid = ?', (uid, sid))
    if r == []: self.db.insert('insert into count_user values (?, ?, 1)', (uid, sid), commit = False)
    else: self.db.update('update count_user set count = ? where uid = ? and sid = ?', (int(r[0][0]) + 1, uid, sid), commit = False)
    r = self.db.select('select count from count_resource where rid = ? and sid = ?', (rid, sid))
    if r == []: self.db.insert('insert into count_resource values (?, ?, 1)', (rid, sid), commit = False)
    else: self.db.update('update count_resource set count = ? where rid = ? and sid = ?', (int(r[0][0]) + 1, rid, sid), commit = False)
    r = self.db.select('select count from count_tu where tid = ? and uid = ? and sid = ?', (tid, uid, sid))
    if r == []: self.db.insert('insert into count_tu values (?, ?, ?, 1)', (tid, uid, sid), commit = False)
    else: self.db.update('update count_tu set count = ? where tid = ? and uid = ? and sid = ?', (int(r[0][0]) + 1, tid, uid, sid), commit = False)
    r = self.db.select('select count from count_ur where uid = ? and rid = ? and sid = ?', (uid, rid, sid))
    if r == []: self.db.insert('insert into count_ur values (?, ?, ?, 1)', (uid, rid, sid), commit = False)
    else: self.db.update('update count_ur set count = ? where uid = ? and rid = ? and sid = ?', (int(r[0][0]) + 1, uid, rid, sid), commit = False)
    r = self.db.select('select count from count_rt where rid = ? and tid = ? and sid = ?', (rid, tid, sid))
    if r == []: self.db.insert('insert into count_rt values (?, ?, ?, 1)', (rid, tid, sid), commit = False)
else: self.db.update('update count_rt set count = ? where rid = ? and tid = ? and sid = ?', (int(r[0][0]) + 1, rid, tid, sid), commit = False)

# insert the 'tagging object'
self.db.insert('insert into tagging values(? , ?, ?, ?, ?)', (sid, uid, rid, tid, time), commit = False)

# get all tags in the same post
tags = self.db.select('select distinct tid from tagging where rid = ? and uid = ? and sid = ?', (rid, uid, sid))

for _t in tags:
    t = _t[0]
    # convention - id t1 is always less than id t2
    if tid < t:
        t1 = tid
        t2 = t
    else:
        t1 = t
        t2 = tid

    # update the counter of tag - tag relation
    r = self.db.select('select count from count_tt where t1 = ? and t2 = ? and sid = ?', (t1, t2, sid))
    if r == []:
        self.db.insert('insert into count_tt values (?, ?, ?, 1)', (t1, t2, sid), commit = False)
    else:
        self.db.update('update count_tt set count = ? where t1 = ? and t2 = ? and sid = ?', (int(r[0][0]) + 1, t1, t2, sid), commit = False)

# transaction [end]
self.db.commit()
fc = FlickrCrawler(self)
try: fc.crawl(proctime = proctime, trace = trace)
except Exception,e:
    print e
    pass
# all sources
else:
    start = time.time()
    dc = DeliciousCrawler(self)
    fc = FlickrCrawler(self)
    td = dc.initCrawlResult()
    tf = fc.initCrawlResult()

    # run until the timeout
    while True:
        try:
            # minimum processing time
            t = dc.crawl(proctime = 1, trace = trace)
            td = dc.sumCrawlResults(td, t)
        except Exception,e:
            print e
            pass
        try:
            # minimum processing time
            t = fc.crawl(proctime = 1, trace = trace)
            tf = fc.sumCrawlResults(tf, t)
        except Exception,e:
            print e
            pass

    # verify if the timeout has been reached
    end = time.time()
    delta = end - start
    if (delta / 60) > proctime: break

    # print the results
    if trace:
        print '$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$'
        print 'Total time:', datetime.timedelta(seconds = delta)
        dc.printCrawlResults(td)
        fc.printCrawlResults(tf)

    ...
```python
# api object
self.dapi = deliciousapi.DeliciousAPI(user_agent = "folklicious v0.1.5")
# controller - the main crawl object
self.contr = contr
# source id
self.sid, ondb = self.contr.dbGetId(None, insert = True, source = 'delicious')
# waiting time for each data category
self.waiting = {'hotlist': 1800, 'user': 300, 'url': 30}
# max elements in the 'not done' list
self.max_notdone = {'resources': 200}

""" the main method that get the data
the parameter 'proctime' is the minimum amount of time processing.
the actual time may be (and usually is) greater.
"""

def crawl(self, proctime = 10, trace = False):
    # start the processing
    start = time.time()

    # verify if more than x seconds have been elapsed since the last
    # requisition to hotlist - delicious restriction
    oldtime = self.contr.dbSelect("select value from control where name = 'del.hotlist.timestamp'", ())[0][0]
    old = int(oldtime[0][0])
    interval = time.time() - old
    if interval < self.waiting['hotlist']: # 30min
        if trace: print 'wait more', (self.waiting['hotlist'] - interval), 'seconds to call populateWithHotList'
        size_hot = 0
    else:
        # populate the hotlist
        size_hot = selfpopulateWithHotList(trace)
        # set the time of the requisition
        self.contr.dbUpdate("update control set value = ? where name = 'del.hotlist.timestamp'", (int(round(time.time())),))

    # initialize the variables
    waiting = 0
    size_url = 0
    size_usr = 0
    count_pop_user = 0

    # process until timeout
    while True:
        # 'progress bar'
```

for i in range(10, waiting):
    sys.stdout.write('.
    time.sleep(1)
    if i == waiting - 1: print ''
    count_pop_user += 1

# verify if more than x seconds have been elapsed since the last
# requisition to hotlist - delicious restriction
oldtime = self.contr.dbSelect("select value from control where name =
    'del.url.timestamp'", ()
old = int(oldtime[0][0])
interval = time.time() - old
if interval < self.waiting['url']: # 15min
    if trace: print 'wait more', (self.waiting['url'] - interval),
    'seconds to call populateWithUrl'
    waiting += 1
else:
    # populate with url as seed
    size_url += self.populateWithUrl(1, trace)

# only populate with user if tried to populate with url 20 times
if count_pop_user == 20:
    # verify if more than x seconds have been elapsed since the last
    # requisition to hotlist - delicious restriction
    oldtime = self.contr.dbSelect("select value from control where
        name = 'del.user.timestamp'", ()
    old = int(oldtime[0][0])
    interval = time.time() - old
    if interval < self.waiting['user']: # 15min
        if trace: print 'wait more', (self.waiting['user'] -
        interval), 'seconds to call populateWithUser'
        waiting += 1
    else:
        # populate with user as seed
        size_usr += self.populateWithUser(1, trace)

    # reset the counter
    count_pop_user = 0

# verify if the minimum processing time has been reached
end = time.time()
delta = end - start
if (delta / 60) > proctime:
    if trace:
        print 'del :: time reached', datetime.timedelta(seconds=delta)
        print 'del ::', size_hot, 'new resources from hotlist'
        print 'del ::', size_url, 'new tuples'
        print 'del ::', size_usr, 'new urls'
break

        # return the elapsed time, and the amount of elements
        return (delta, size_hot, size_url, size_usr)

        """ initialize the results
        """
        def initCrawlResult(self):
            return (0, 0, 0, 0)

        """ sum two sets of results
        """
        def sumCrawlResults(self, t1, t2):
            return (t1[0] + t2[0], t1[1] + t2[1], t1[2] + t2[2], t1[3] + t2[3])

        """ print the results
        """
        def printCrawlResults(self, t):
            print '$del :: total time', datetime.timedelta(seconds=t[0])
            print '$del ::', t[1], 'new resources from hotlist'
            print '$del ::', t[2], 'new tuples'
            items = self.contr.dbSelect('select count(*) from resource where sid = ?
                and done = 0', (self.sid,))
            print '$del ::', t[3], 'new urls,', items[0][0], 'to be processed'

        """ populate the db using the hotlist
        """
        def populateWithHotList(self, trace = False):
            """ get the resources not processed yet
            """
            notdone = self.contr.dbSelect('select count(*) from resource where sid = ?
                and done = 00', (self.sid,))[0][0]
            if notdone > self.max_notdone['resources']:
                if trace: print 'There are %d (%d) resources notdone.' % (notdone,
                    self.max_notdone['resources'])
                return 0
            """ get the URLs in hotlist
            """
            urls = self.dapi.get_urls()
            if trace: print len(urls), 'urls retrieved'
            count = 0

            """ for each URL
            """
            for u in urls:
                """ try to insert in db
                """
                rid, ondb = self.contr.dbGetId(self.sid, insert = True, resource = u)
                if not ondb:
                    count += 1
                    if trace: print 'inserted', u
elif trace: print 'already inserted', u
    if trace: print count, 'new urls'

    # return the number of new objects stored
    return count

""" populate the db using the URL
"""
def populateWithUrl(self, max_urls = 10, trace = False):
    if trace: print 'del :: populateWithUrl - max_urls:', max_urls

    # get the resources not processed yet
    res = self.contr.dbSelect('select value from resource where done = 0 and
             sid = ? limit ?', (self.sid, max_urls,))
    total = len(res)
    if total > max_urls: total = max_urls
    if trace: print total, 'urls'
    curr = 1
    count = 0

    # process each resource ...
    for _r in res:
        # ... until reach the maximum
        if curr > max_urls: break

        r = _r[0]
        if trace: print 'resource %03d/%03d' %(curr, total)

        # get the bookmarks associated with that URL
        # with 'max_bookmarks=0' all of them are returned, but it consumes
        # more time
        meta = self.dapi.get_url(r, max_bookmarks=0)

        # store the requisition time
        self.contr.dbUpdate('update control set value = ? where name =
                             "del.url.timestamp"', (int(round(time.time())),))

        # process the bookmarks
        total_bookmarks = len(meta.bookmarks)
        if trace: print 'resource %s - %d bookmarks' %(r, total_bookmarks)
        if total_bookmarks > 0 :
            curr_bookmarks = 0
            for b in meta.bookmarks:
                curr_bookmarks += 1
                # [user, taglist, comment, time]
                user = b[0]
                taglist = b[1]
                timestamp = b[3]

                # user id
uid, ondb = self.contr.dbGetId(self.sid, insert = True, user = user)
# resource id
rid, ondb = self.contr.dbGetId(self.sid, insert = True, resource = r)

# failed to insert
if uid is None or rid is None:
    break

for t in taglist:
    count += 1
    # tag id
tid, ondb = self.contr.dbGetId(self.sid, insert = True, tag = t)
    # insert the tagging object
    self.contr.dbInsertTagging(self.sid, uid, rid, tid, str(timestamp))
    if trace:
        print 'ids:', uid, rid, tid
        print 'inserted', (user, r[:60], t, str(timestamp))
        print '[%03d/%03d resources][%03d/%03d bookmarks][%4d inserts]' % (curr, total, curr_bookmarks, total_bookmarks, count)

    # set the resource as done
    self.contr.dbUpdate('update resource set done = 1 where value = ? and sid = ?', (r, self.sid,))
else:
    # set the resource as done - no tags
    self.contr.dbUpdate('update resource set done = 1 where value = ? and sid = ?', (r, self.sid,))
    if trace: print 'no tags -> done', r[:75]
    curr += 1

# return the number of new objects stored
return count

""" populate the db using the URL
"""
def populateWithUser(self, max_users = 10, trace = False):
    if trace: print 'del :: populateWithUser - max_users:', max_users

    # get the resources not processed yet
    notdone = self.contr.dbSelect('select count(*) from resource where sid = ? and done = 0', (self.sid,))[0][0]
    # if there are more resources than the maximum, don’t try to populate with user
    if notdone > self.max_notdone['resources']:
if trace: print 'There are %d (%d) resources not done.' % (notdone, 
    self.max_notdone['resources'])
return 0

# get the users not processed yet
users = self.contr.dbSelect('select value from user where done = 0 and 
    sid = ? limit ?', (self.sid, max_users,))
total = len(users)
if total > max_users: total = max_users
if trace: print total, 'users'
curr = 1
count = 0

# process each user ...
for _u in users:
    # ... until reach the maximum
    if curr > max_users: break

    u = _u[0]
    if trace: print 'user %03d/%03d' %(curr, total)
    try:
        # get the bookmarks associated with that user
        meta = self.dapi.get_user(u, max_bookmarks = 10)
        # store the requisition time
        self.contr.dbUpdate('update control set value = ? where name = 
            "del.user.timestamp"', (int(round(time.time())),))
    except:
        # if there was an error, mark that resource as already processed 
        # and continue to the next one
        if trace: print 'Delicious error: user', u, 'marked as done'
        self.contr.dbUpdate('update user set done = 1 where value = ? and 
            sid = ?', (u, self.sid,))
        continue

    if len(meta.bookmarks) > 0 :
        for b in meta.bookmarks:
            # [url, taglist, title, comment, time]
            r = b[0]
            # resource id
            rid, ondb = self.contr.dbGetId(self.sid, insert = True, 
                resource = r)
            if not ondb:
                count += 1
                if trace: print 'inserted', r, '%03d' %rid
            else:
                if trace: print 'already inserted', r, '%03d' %rid
            if trace: print '
            [%03d/%03d] %d inserts' %(curr, total, count)
# set the user as done
self.contr.dbUpdate('update user set done = 1 where value = ? and sid = ?', (u, self.sid))
curr += 1

# return the number of new objects stored
return count

""" Object FlickrCrawler - encapsulates the flickr crawler operations
"""
class FlickrCrawler(object):
    """ constructor - initialize the variables
    """
def __init__(self, contr):
    # controller - the main crawl object
    self.contr = contr
    # source id
    self.sid, ondb = self.contr.dbGetId(None, insert = True, source = 'flickr')
    # base URL
    self.url = 'http://api.flickr.com/services/rest/'
    # api key - REPLACE WITH YOUR OWN API KEY
    self.api_key = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
    # api methods
    self.methods = {
        'echo':   'flickr.test.echo',
        'pInfo':  'flickr.photos.getInfo',
        'pList':  'flickr.people.getPublicPhotos',
        'uList':  'flickr.contacts.getPublicList',
        'uPhotos': 'flickr.photos.getContactsPublicPhotos',
        'pRecent': 'flickr.photos.getRecent',
        'tHot':   'flickr.tags.getHotList',
        'tPhoto': 'flickr.tags.getListPhoto',
        'panda':  'flickr.panda.getPhotos',
    }

    """ the main method that get the data
    the parameter 'proctime' is the minimum amount of time processing.
    the actual time may be (and usually is) greater.
    """
def crawl(self, proctime = 10, trace = False):
    # start the processing
    start = time.time()
    # initialize the variables
    size_tag = 0
    size_res = 0
count_pop_res = 0

# process until timeout
while True:
    count_pop_res += 1

    # populate the db with tags
    size_tag += self.populateTags(1, trace = trace)

    # only populate with user if tried to populate with tags 20 times
    if count_pop_res == 20:
        size_res += self.populateResources(1, trace = trace)
        count_pop_user = 0

    # verify if the minimum processing time has been reached
    end = time.time()
    delta = end - start
    if (delta / 60) > proctime:
        if trace:
            print 'fli :: time reached', datetime.timedelta(seconds=delta)
            print 'fli ::', size_tag, 'new tuples'
            print 'fli ::', size_res, 'new resources'
        break

    # return the elapsed time, and the amount of elements
return (delta, size_tag, size_res)

""" initialize the results
"""
def initCrawlResult(self):
    return (0, 0, 0)

""" sum two sets of results
"""
def sumCrawlResults(self, t1, t2):
    return (t1[0] + t2[0], t1[1] + t2[1], t1[2] + t2[2])

""" print the results
"""
def printCrawlResults(self, t):
    print '$fli :: total time', datetime.timedelta(seconds=t[0])
    print '$fli ::', t[1], 'new tuples'
    items = self.contr.dbSelect('select count(*) from resource where sid = ?
and done = 0', (self.sid,))
    print '$fli ::', t[2], 'new resources,', items[0][0], 'to be processed'

""" execute a post request
"""
def post(self, params):
return urllib.urlopen(self.url, params)

""" create the params object """

def createParams(self, method, params, trace = False):
    params['method'] = method
    params['api_key'] = self.api_key
    if trace: print params
    return urllib.urlencode(params)

""" get random photos - this service is called panda in flickr """

def getPhotosPanda(self, per_page = None, panda = None):
    # choose the panda - 'ling ling', 'hsing hsing', and 'wang wang'
    if panda is None: panda = random.randint(1, 3)
    if panda == 1:
        lst = {'panda_name': 'ling ling'}
    elif panda == 2:
        lst = {'panda_name': 'hsing hsing'}
    else:
        lst = {'panda_name': 'wang wang'}
    if per_page is not None:
        lst['per_page'] = per_page

    # return the result
    return self.post(self.createParams(self.methods['panda'], lst))

""" populate the db using resources """

def populateResources(self, num_photos, pandaid = None, trace = False):
    # the limit is 500 photos
    if num_photos <= 500:
    # xml with photos from pandas
    pxml = self.getPhotosPanda(num_photos, pandaid)
    # get the information
    root = parse(pxml).getroot().find('photos')
    interval = root.get('interval')
    lastupdate = root.get('lastupdate')
    photos = root.findall('photo')
    total = len(photos)
    curr = 1
    count = 0

    # for each photo of the xml
    for p in photos:
# insert in the db
pid = p.get('id')
rid, ondb = self.contr.dbGetId(self.sid, insert = True, resource = pid)
if not ondb:
    count += 1
    if trace: print '%s: %03d/%03d' % (rid, curr, total)
curr += 1

# flickr gives the lastupdate time and the waiting interval
wait = time.time() - (int(lastupdate) + int(interval))
# so, verify if need to wait
if wait > 0:
    if trace: print 'waiting', wait, 'before next requisition'
    time.sleep(wait)

# more than 500 photos - multiple requests
else:
    n = 0
    r = 0
    while n < num_photos:
        r += self.populateResources(500)
        n += 500
    return r

# return the number of new objects stored
return count

""" populate the db using tags
""

def populateTags(self, num_resources = 10, trace = False):
    if trace: print 'fli :: populateTags - num_resources:', num_resources

    # get the resources not processed yet
    res = self.contr.dbSelect('select rid from resource where done = 0 and
                               sid = ? limit ?', (self.sid, num_resources,))
curr = 1
    if len(res) > num_resources:
        total = num_resources
    else:
        total = len(res)
count = 0

    # process each resource ...
    for _r in res:
        rid = _r[0]
        # ... until reach the maximum
        if curr > total:
break

if trace: print 'resources: %03d/%03d' % (curr, total)

# get the data from photo
data = self.getDataFromPhoto(rid, trace = trace)
notags = True

for d in data:
    # user id
    uid, ondb = self.contr.dbGetId(self.sid, insert = True, user = d['user'])
    # tag id
tid, ondb = self.contr.dbGetId(self.sid, insert = True, tag = d['tag'])
    # insert the tagging object
    self.contr.dbInsertTagging(self.sid, uid, rid, tid)
    count += 1
    if trace: print '%d new tuple: %s' % (count, (self.sid, uid, rid, tid,))
    notags = False

    # set the resource as done
    self.contr.dbUpdate('update resource set done = 1 where rid = ? and sid = ?', (rid, self.sid,))
    if trace: print 'rid:', rid, 'set done - notags:', notags
curr += 1

    # return the number of new objects stored
return count

""" return the xml file with the tags of a given photo
"""
def getTagsFromPhoto(self, photo_id):
    return self.post(self.createParams(selfMethods['tPhoto'], {'photo_id': photo_id}))

""" return the data from a given photo
"""
def getDataFromPhoto(self, photo_id, trace = False):
    # get the tags
tags = self.getTagsFromPhoto(photo_id)
root = parse(tags).getroot()
stat = root.get('stat')
if stat == 'fail' and trace: print '!!] Flickr error:', root.find('err').get('msg'), 'photo_id', photo_id
# prepare the result with all pairs {user; tag}
result = []
for t in root.findall('photo/tags/tag'):
    result.append({'user': t.get('author'), 'tag': t.text})

# return the result data
return result

# the main execution
if __name__ == '__main__':
    # create the crawler object
    c = Crawler()
    # execute a crawl operation in all sources and with minimum 60 minutes
    c.crawl(Source(), proctime=60, trace=True)

    # example of crawl in delicious with minimum 45 minutes
    # c.crawl(Source('delicious'), proctime=45, trace=True)

    # example of crawl in flickr with minimum 15 minutes
    # c.crawl(Source('flickr'), proctime=15, trace=True)

    # delete the object and release all resources
    del c

4 Folksonomy Systems

4.1 Flickr

Flickr is an online community and an image and video hosting website. It has a large user community and huge amount of resources (mainly images). Whenever authors uploads an image they insert tags to describe it. The main search mechanism of Flickr is based on tags.

Flickr offers a web service API to access its data by non-commercial applications. One important aspect of the API is that all data should be codified by using the UTF-8 standard. If the API receives a sequence in any codification but UTF-8, it assumes that the codification is ISO-8859-1 and then transforms it to UTF-8. Any other codification could result in incorrect data. In the following subsections we detail aspects of the Flickr API.

4.1.1 Authentication

The services can be accessed in non-authenticated or authenticated modes. There are API methods that are only available to the authenticated users. The authentication process is described in the Flickr OAuthAPI webpage\(^7\). In our study we focused on public data. Therefore, we did not use the authenticate methods.

\(^7\)http://www.flickr.com/services/api/auth.oauth.html
4.1.2 Definitions

Services' protocols and data objects exchanged by Flickr and clients follow a set of basic standard definitions further detailed.

**Dates**

There are two types of dates: *taken* – the date when the photo was taken; *posted* – the date when the photo was posted on the system. *Taken* dates must follow the MySQL ‘datetime’ format (e.g., 2004-11-29 16:01:26) and they have 4 levels of accuracy:

- 0 Y-m-d H:i:s
- 4 Y-m
- 6 Y
- 8 circa...

On the other hand, *posted* dates are always in the unix timestamp format, i.e., an unsigned integer specifying the number of seconds since Jan 1st 1970 GMT.

**Buddyicons**

Buddyicon is a 48x48 pixel image that represent the user – an “avatar”. It is necessary to inform the user’s NSID (user id), icon server and icon farm to access the buddyicon of a user.

If the icon server parameter is a number greater than zero. A URL to request the icon takes the format:

```
http://farm{icon-farm}.static.flickr.com/{icon-server}/
buddyicons/{nsid}.jpg
```

There is also a URL to request the default buddyicon:

```
```

**URLs**

The photo URL is built by using the following parameters: photo ID, server ID, farm ID, and secret. The URL takes the format:

```
http://farm{farm-id}.static.flickr.com/{server-id}/{id}_{secret}.jpg
```

or

```
http://farm{farm-id}.static.flickr.com/{server-id}/{id}_{secret}_{mstzb}.jpg
```

or

```
http://farm{farm-id}.static.flickr.com/{server-id}/
{id}_{o-secret}_o.(jpg|gif|png)
```

The second URL affords one of the following size suffixes (indicated in the URL by brackets):

- s – small square 75x75
- t – thumbnail, 100 on longest side

---

8 http://www.flickr.com/services/api/misc.dates.html
9 http://www.flickr.com/services/api/misc.buddyicons.html
10 http://www.flickr.com/services/api/misc.urls.html
m – small, 240 on longest side
- – medium, 500 on longest side
z – medium 640, 640 on longest side
b – large, 1024 on longest side
The last URL is specific for images in the original size:
o – original image, either a jpg, gif or png, depending on source format

**Tags** When a photo has tags, the format of the tag field in XML is as follows:

```xml
<tag id="1234" author="12037949754@N01" raw="woo yay">wooyay</tag>
```

The parameters are:
id – The photo id.
author – The NSID of the user who added the tag.
raw – The “raw” version of the tag - as entered by the user. This version can contain spaces and punctuation.
tag-body – The “clean” version of the tag – as processed by Flickr.

### 4.1.3 Access

API keys are necessary to access and use the flickr data. The process to obtain those keys is the following:

1. Register to Flickr (as a user).
2. Go to the API request page (http://www.flickr.com/services/apps/create/apply/).
3. Choose the appropriate option (in this work we used “Non-Commercial”).
4. Fill in the form.

In the end of this process the API keys are generated.

### 4.1.4 Request Protocols

There are three request protocols: REST\(^\text{12}\), XML-RPC\(^\text{13}\), and SOAP\(^\text{14}\). We adopted REST since it is the simplest option and it meets our needs.

The following pair requisition/response illustrates a REST requisition on Flickr. In this case we are using a fictitious API key.

```
http://www.flickr.com/services/rest/?method=flickr.test.echo&format=rest
&foo=bar&api_key=cc4094c55264c02ec2a83001b95a0837

<rsp stat="ok">
```

---

\(^\text{11}\)http://www.flickr.com/services/api/misc.tags.html
\(^\text{12}\)http://www.flickr.com/services/api/request.rest.html
\(^\text{13}\)http://www.flickr.com/services/api/request.xmlrpc.html
\(^\text{14}\)http://www.flickr.com/services/api/request.soap.html
4.1.5 Response Formats

There are five response formats: REST\textsuperscript{15}, XML-RPC\textsuperscript{16}, SOAP\textsuperscript{17}, JSON\textsuperscript{18}, and PHP\textsuperscript{19}. Again, we chose the REST format.

4.1.6 API Methods

In this section we briefly describe the API methods that are of most relevant to this work. The complete list of methods can be viewed in the API page (http://www.flickr.com/services/api/).

auth: methods to authenticate the app.

- flickr.auth.checkToken\textsuperscript{20}: returns the credentials attached to a token
- flickr.auth.getFrob\textsuperscript{21}: returns the frob to be used in authentication
- flickr.auth.getFullToken\textsuperscript{22}: returns the full token from a mini-token
- flickr.auth.getToken\textsuperscript{23}: returns the token from a frob

contacts

- flickr.contacts.getList\textsuperscript{24}: returns the contact list from a user
- flickr.contacts.getPublicList\textsuperscript{25}: returns the public contact list from a user (doesn’t need authentication)

galleries

- flickr.galleries.getInfo\textsuperscript{26}: returns the information of a gallery
- flickr.galleries.getList\textsuperscript{27}: returns the list of galleries

\textsuperscript{15}http://www.flickr.com/services/api/response.rest.html
\textsuperscript{16}http://www.flickr.com/services/api/response.xmlrpc.html
\textsuperscript{17}http://www.flickr.com/services/api/response.soap.html
\textsuperscript{18}http://www.flickr.com/services/api/response.json.html
\textsuperscript{19}http://www.flickr.com/services/api/response.php.html
\textsuperscript{20}http://www.flickr.com/services/api/flickr.auth.checkToken.html
\textsuperscript{21}http://www.flickr.com/services/api/flickr.auth.getFrob.html
\textsuperscript{22}http://www.flickr.com/services/api/flickr.auth.getFullToken.html
\textsuperscript{23}http://www.flickr.com/services/api/flickr.auth.getToken.html
\textsuperscript{24}http://www.flickr.com/services/api/flickr.contacts.getList.html
\textsuperscript{25}http://www.flickr.com/services/api/flickr.contacts.getPublicList.html
\textsuperscript{26}http://www.flickr.com/services/api/flickr.galleries.getInfo.html
\textsuperscript{27}http://www.flickr.com/services/api/flickr.galleries.getList.html
Retrieving and Storing Data from Folksonomies

- `flickr.galleries.getListForPhoto`\(^{28}\): returns the list of galleries that contains a given photo
- `flickr.galleries.getPhotos`\(^{29}\): returns the photos of a given gallery

**interestingness**

- `flickr.interestingness.getList`\(^{30}\): returns the most interesting photos of a specific date

**machinetags**

- `flickr.machinetags.getNamespaces`\(^{31}\): returns a list of unique namespaces
- `flickr.machinetags.getPairs`\(^{32}\): returns a list of unique namespace and predicate pairs
- `flickr.machinetags.getPredicates`\(^{33}\): returns a list of unique predicates
- `flickr.machinetags.getRecentValues`\(^{34}\): returns the most recent machinetags
- `flickr.machinetags.getValues`\(^{35}\): returns a list of unique values for a namespace and predicate

**panda**

- `flickr.panda.getList`\(^{36}\): returns a list of pandas (photo services)
- `flickr.panda.getPhotos`\(^{37}\): returns a list of photos of the given panda

**people**

- `flickr.people.findByEmail`\(^{38}\): returns a user’s NSID, given his/her email address
- `flickr.people.findByUsername`\(^{39}\): returns a user’s NSID, given his/her username
- `flickr.people.getInfo`\(^{40}\): gets information about a user
- `flickr.people.getPhotos`\(^{41}\): returns photos from the given user’s photostream
- `flickr.people.getPhotosOf`\(^{42}\): returns a list of photos containing a particular Flickr member

\(^{28}\)http://www.flickr.com/services/api/flickr.galleries.getListForPhoto.html
\(^{29}\)http://www.flickr.com/services/api/flickr.auth.getToken.html
\(^{30}\)http://www.flickr.com/services/api/flickr.interestingness.getList.html
\(^{31}\)http://www.flickr.com/services/api/flickr.machinetags.getNamespaces.html
\(^{32}\)http://www.flickr.com/services/api/flickr.machinetags.getPairs.html
\(^{33}\)http://www.flickr.com/services/api/flickr.machinetags.getPredicates.html
\(^{34}\)http://www.flickr.com/services/api/flickr.machinetags.getRecentValues.html
\(^{35}\)http://www.flickr.com/services/api/flickr.machinetags.getValues.html
\(^{36}\)http://www.flickr.com/services/api/flickr.panda.getList.html
\(^{37}\)http://www.flickr.com/services/api/flickr.panda.getPhotos.html
\(^{38}\)http://www.flickr.com/services/api/flickr.people.findByEmail.html
\(^{39}\)http://www.flickr.com/services/api/flickr.people.findByUsername.html
\(^{40}\)http://www.flickr.com/services/api/flickr.people.getInfo.html
\(^{41}\)http://www.flickr.com/services/api/flickr.people.getPhotos.html
\(^{42}\)http://www.flickr.com/services/api/flickr.people.getPhotosOf.html
• `flickr.people.getPublicPhotos`: gets a list of public photos for the given user photos

• `flickr.photos.getAllContexts`: returns all visible sets and pools the photo belongs to
• `flickr.photos.getContactsPhotos`: fetches a list of recent photos from the calling users’ contacts
• `flickr.photos.getContactsPublicPhotos`: fetches a list of recent public photos from a users’ contacts
• `flickr.photos.getContext`: returns next and previous photos for a photo in a photostream
• `flickr.photos.getCounts`: gets a list of photo counts for the given date ranges for the calling user
• `flickr.photos.getInfo`: gets information about a photo. The calling user must have permission to view the photo
• `flickr.photos.getPerms`: gets permissions for a photo
• `flickr.photos.getRecent`: returns a list of the latest public photos uploaded to flickr
• `flickr.photos.getSizes`: returns the available sizes for a photo. The calling user must have permission to view the photo
• `flickr.photos.getUntagged`: returns a list of your photos with no tags
• `flickr.photos.getWithGeoData`: returns a list of your geo-tagged photos
• `flickr.photos.getWithoutGeoData`: returns a list of your photos which haven’t been geo-tagged
• `flickr.photos.recentlyUpdated`: returns a list of your photos that have been recently created or which have been recently modified
• `flickr.photos.search`: returns a list of photos matching some criteria

---

43 http://www.flickr.com/services/api/flickr.people.getPublicPhotos.html
44 http://www.flickr.com/services/api/flickr.photos.getAllContexts.html
45 http://www.flickr.com/services/api/flickr.photos.getContactsPhotos.html
46 http://www.flickr.com/services/api/flickr.photos.getContactsPublicPhotos.html
47 http://www.flickr.com/services/api/flickr.photos.getContext.html
48 http://www.flickr.com/services/api/flickr.photos.getCounts.html
49 http://www.flickr.com/services/api/flickr.photos.getInfo.html
50 http://www.flickr.com/services/api/flickr.photos.getPerms.html
51 http://www.flickr.com/services/api/flickr.photos.getRecent.html
52 http://www.flickr.com/services/api/flickr.photos.getSizes.html
53 http://www.flickr.com/services/api/flickr.photos.getUntagged.html
54 http://www.flickr.com/services/api/flickr.photos.getWithGeoData.html
55 http://www.flickr.com/services/api/flickr.photos.getWithoutGeoData.html
56 http://www.flickr.com/services/api/flickr.photos.recentlyUpdated.html
57 http://www.flickr.com/services/api/flickr.photos.search.html
Retrieving and Storing Data from Folksonomies

photos.licenses

- `flickr.photos.licenses.getInfo`: fetches a list of available photo licenses for Flickr

photosets

- `flickr.photosets.getContext`: returns next and previous photos for a photo in a set
- `flickr.photosets.getInfo`: gets information about a photoset
- `flickr.photosets.getList`: returns the photosets belonging to the specified user
- `flickr.photosets.getPhotos`: gets the list of photos in a set

reflection

- `flickr.reflection.getMethodInfo`: returns information for a given flickr API method
- `flickr.reflection.getMethods`: returns a list of available flickr API methods

tags

- `flickr.tags.getClusterPhotos`: returns the first 24 photos for a given tag cluster
- `flickr.tags.getClusters`: returns a list of tag clusters for the given tag
- `flickr.tags.getHotList`: returns a list of hot tags for the given period
- `flickr.tags.getListPhoto`: gets the tag list for a given photo
- `flickr.tags.getListUser`: gets the tag list for a given user (or for the user currently logged)
- `flickr.tags.getListUserPopular`: gets the popular tags for a given user (or for the user currently logged)
- `flickr.tags.getListUserRaw`: gets the raw versions of a given tag (or all tags) for the user currently logged
- `flickr.tags.getRelated`: returns a list of tags ‘related’ to a given tag, based on clustered usage analysis

References:

58 http://www.flickr.com/services/api/flickr.photos.licenses.getInfo.html
60 http://www.flickr.com/services/api/flickr.photosets.getInfo.html
61 http://www.flickr.com/services/api/flickr.photosets.getList.html
63 http://www.flickr.com/services/api/flickr.reflection.getMethodInfo.html
64 http://www.flickr.com/services/api/flickr.reflection.getMethods.html
65 http://www.flickr.com/services/api/flickr.tags.getClusterPhotos.html
66 http://www.flickr.com/services/api/flickr.tags.getClusters.html
67 http://www.flickr.com/services/api/flickr.tags.getHotList.html
68 http://www.flickr.com/services/api/flickr.tags.getListPhoto.html
69 http://www.flickr.com/services/api/flickr.tags.getListUser.html
70 http://www.flickr.com/services/api/flickr.tags.getListUserPopular.html
71 http://www.flickr.com/services/api/flickr.tags.getListUserRaw.html
72 http://www.flickr.com/services/api/flickr.tags.getRelated.html
4.1.7 API Examples

In this section we show some examples of the API. In each example there are two blocks, one containing the request and other with the response.

i Getting Photo Information

http://api.flickr.com/services/rest/?method=flickr.photos.getInfo &api_key=f629fbcf316fbea8611ca0b2d33f2ea7&photo_id=120292580

/**
api_key (required): api key
photo_id (required): photo id
secret (optional): if correct the permission check is not performed
**/
Retrieving and Storing Data from Folksonomies

[ii] Photo Galleries Information

http://api.flickr.com/services/rest/?method=flickr.galleries.getListForPhoto
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&photo_id=2080242123&per_page=5

/**
 * api_key (required): api key
 * photo_id (required): photo id
 * per_page (optional): number of galleries in result. default 100 - max 500
 * page (optional): the page of the result. default 1
 */
<galleries <total="19" page="1" pages="4" per_page="5" photo_id="2080242123">

<gallery id="25845796-72157624218638653"
    url="http://www.flickr.com/photos/25891118@N06/galleries/72157624218638653"
    owner="25891118@N06" primary_photo_id="2080242123"
    date_create="1277335620" date_update="1278257567" count_photos="4"
    count_videos="0" primary_photo_server="2209" primary_photo_farm="3"
    primary_photo_secret="55c93c007d"> <title>Nature</title> <description /></gallery>

<gallery id="51165959-72157624161029199"
    url="http://www.flickr.com/photos/fer10/galleries/72157624161029199"
    owner="51198098@N04" primary_photo_id="4691319257"
    date_create="1276662136" date_update="1276906396" count_photos="14"
    count_videos="0" primary_photo_server="4007" primary_photo_farm="5"
    primary_photo_secret="f411f6ba4e"> <title>Bellezas</title> <description>Expectacular</description></gallery>

<gallery id="1344252-72157623919289749"
    url="http://www.flickr.com/photos/68196577@N00/galleries/72157623919289749"
    owner="68196577@N00" primary_photo_id="4536144000"
    date_create="1273628141" date_update="1278099389" count_photos="15"
    count_videos="0" primary_photo_server="4032" primary_photo_farm="5"
    primary_photo_secret="49a59c20ff"> <title>WHAT?!</title> <description /></gallery>

<gallery id="15624814-72157623792903801"
    url="http://www.flickr.com/photos/15646144@N07/galleries/72157623792903801"
    owner="15646144@N07" primary_photo_id="2080242123"
    date_create="1272049888" date_update="1272052268" count_photos="1"
    count_videos="0" primary_photo_server="2209" primary_photo_farm="3"
    primary_photo_secret="55c93c007d"> <title>For Mobile</title> <description /></gallery>

<gallery id="20945644-72157623529610741"
    url="http://www.flickr.com/photos/20966974@N07/galleries/72157623529610741"
    owner="20966974@N07" primary_photo_id="2080242123"
    date_create="1269051584" date_update="1269051616" count_photos="1"
    count_videos="0" primary_photo_server="2209" primary_photo_farm="3"
    primary_photo_secret="55c93c007d"> <title>Fall</title> <description /></gallery>
</galleries>
iii Public List of User Contacts

http://api.flickr.com/services/rest/?method=flickr.contacts.getPublicList
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&user_id=67526850@N00

/**
   api_key (required): api key
   user_id (required): photo id
   per_page (optional): number of result items. default 1000 - max 1000
   page (optional): the page of the result. default 1
 **/

<rsp stat="ok">
<contacts page="1" per_page="1000" perpage="1000" total="19">
   <contact nsid="28404674@N00" username="(^_^) wellwin" iconserver="41"
      iconfarm="1" ignored="0" />
   <contact nsid="38075047@N00" username="dgray_xplane" iconserver="32"
      iconfarm="1" ignored="0" />
   <contact nsid="27009262@N00" username="Dion Hinchcliffe" iconserver="7"
      iconfarm="1" ignored="0" />
   <contact nsid="80095026@N00" username="fanstone" iconserver="53" iconfarm="1"
      ignored="0" />
   <contact nsid="80739942@N00" username="keystone1111" iconserver="3128"
      iconfarm="4" ignored="0" />
   <contact nsid="46752978@N00" username="kisco" iconserver="34" iconfarm="1"
      ignored="0" />
   <contact nsid="92455005@N00" username="laurentaug" iconserver="4"
      iconfarm="1" ignored="0" />
   <contact nsid="89529267@N00" username="LynetteRadio" iconserver="40"
      iconfarm="1" ignored="0" />
   <contact nsid="92518516@N00" username="modahome" iconserver="120"
      iconfarm="1" ignored="0" />
   <contact nsid="73314839@N00" username="Naaaif" iconserver="110" iconfarm="1"
      ignored="0" />
   <contact nsid="20056291@N00" username="nicolasnova" iconserver="4059"
      iconfarm="5" ignored="0" />
   <contact nsid="21296916@N03" username="Paul Hughes: Design Thinking"
      iconserver="2186" iconfarm="3" ignored="0" />
   <contact nsid="54412022@N00" username="publicmind" iconserver="17"
      iconfarm="1" ignored="0" />
   <contact nsid="46557603@N00" username="Ralf Beuker" iconserver="2386"
      iconfarm="3" ignored="0" />
   <contact nsid="49147885@N00" username="squidish" iconserver="21" iconfarm="1"
      ignored="0" />
   <contact nsid="36112663@N00" username="tangyg" iconserver="0" iconfarm="0"
      ignored="0" />
   <contact nsid="34862120@N08" username="think.smith" iconserver="3126"
      iconfarm="4" ignored="0" />
</contacts>
</rsp>
iv Public Photos of User Contacts

http://api.flickr.com/services/rest/?method=flickr.photos.getContactsPublicPhotos
&api_key=f629fbcf316fbea8611ca0b2d33f2ea7&user_id=67526850@N00

/**
 * api_key (required): api key
 * user_id (required): user id
 * count (optional): number of photos. default 10 - max 50. only used if without
 * parameter 'single_photo'
 * just_friends (optional): if 1 returns only photos of family and friends
 * single_photo (optional): only returns the last photo of each contact
 * include_self (optional): if 1 includes photos of the user (specified in
 * 'user_id')
 * extras (optional): extra information (license, date_upload, date_taken,
 * owner_name, icon_server, original_format, last_update)
 **/

<rsp stat="ok">
<photo id="120292580" secret="fca8637ab6" server="47" farm="1"
  dateuploaded="1143731314" isfavorite="0" license="5" rotation="0"
  originalsecret="fca8637ab6" originalformat="png" views="10163" media="photo">
  <owner nsid="67526850@N00" username="Alex Osterwalder" realname="Alexander
  Osterwalder" location="Genvea, Switzerland" />
  <title>Web2.0 Business Model Characteristics</title>
  <description>The outcome of a short late-night brainstorming session on the
  characteristics of a Web2.0 business model. The reflections are based on
  what I write at my &la<br href="http://business-model-design.blogspot.com&qu<br;business
  model design blog&la;&gt;/description>
  <visibility ispublic="1" isfriend="0" isfamily="0" />
  <dates posted="1143731314" taken="2006-03-30 22:08:34" takengranularity="0"
  lastupdate="1202967678" />
  <editability cancomment="0" canaddmeta="0" />
  <usage candownload="1" canblog="0" canprint="0" canshare="0" />
  <comments>2</comments>
  <notes />
  <tag id="2017715-120292580-380852" author="67526850@N00" raw="business
  model" machine_tag="0">businessmodel</tag>
  <tag id="2017715-120292580-11227" author="67526850@N00" raw="web2.0"
  machine_tag="0">web20</tag>
  <tag id="2017715-120292580-2956157" author="67526850@N00" raw="business
  model innovation" machine_tag="0">businessmodelinnovation</tag>
  <tag id="2017715-120292580-2956158" author="67526850@N00" raw="business
  model ontology" machine_tag="0">businessmodelontology</tag>
</photo>
</rsp>
Retrieving and Storing Data from Folksonomies

```xml
<tag id="2017715-120292580-2109580" author="67526850@N00" raw="osterwalder" machine_tag="0">osterwalder</tag>
</tags>
$url type="photopage">
http://www.flickr.com/photos/osterwalder/120292580/
</url>
</photo>
</rsp>

v Latest Public Photos

http://api.flickr.com/services/rest/?method=flickr.photos.getRecent
&api_key=f629fbcf316fba8611ca0b2d33f2ea7&per_page=10

/**
  api_key (required): api key
  extras (optional): extra information (description, license, date_upload,
  date_taken, owner_name, icon_server, original_format, las_update, geo,
  tags, machine_tags, o_dims, views, media, path_alias, url_sq, url_t, url_s,
  url_m, url_o)
  per_page (optional): number of result items. default 100 - max 500
  page (optional): page of the result. default 1
**/

<rsp stat="ok">
<photos page="1" pages="100" perpage="10" total="1000">
  <photo id="4771876711" owner="30428372@N05" secret="94ef60dcfa" server="4098" farm="5" title="Picture0136" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="4771876775" owner="10047346@N00" secret="5cd4161426" server="4122" farm="5" title="Aberdeenshire" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="4771876795" owner="29221546@N07" secret="4b06f6eb86" server="4080" farm="5" title="P1030381" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="4771876809" owner="89235411@N00" secret="30b600cd9" server="4123" farm="5" title="P1000277" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="4771876827" owner="51617540@N07" secret="985439dc86" server="4095" farm="5" title="01winery1" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="4772515510" owner="58562067@N00" secret="d1e84f605b" server="4134" farm="5" title="IMG_5164" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="47725155890" owner="50585245@N06" secret="db914e1f92" server="4079" farm="5" title="DSC00558" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="4772515614" owner="10887912@N03" secret="cc143872a3" server="4116" farm="5" title="IMG_2268" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="4772515634" owner="32128624@N05" secret="f17de864e" server="4093" farm="5" title="DSC00216" ispublic="1" isfriend="0" isfamily="0" />
  <photo id="4772515640" owner="73657575@N00" secret="71e526d11e" server="4118" farm="5" title="Therion @ GMM 2010" ispublic="1" isfriend="0" isfamily="0" />
</photos>
</rsp>
```
vi  Most Popular Tags

http://api.flickr.com/services/rest/?method=flickr.tags.getHotList &api_key=f629fbcf316fbea8611ca0b2d33f2ea7&period=week

/**
 * api_key (required): api key
 * period (optional): time period of result. 'day' (default) or 'week'
 * count (optional): number of result items. default 20 - max 200
 **/

<rsp stat="ok">
<hottags period="week" count="20">
  <tag score="100">me2mobileme2photo</tag>
  <tag score="100">canadaday2010</tag>
  <tag score="100">tdf10</tag>
  <tag score="100">japanexpo</tag>
  <tag score="100">happybirthdayamerica</tag>
  <tag score="100">animeexpo2010</tag>
  <tag score="100">zurifascht</tag>
  <tag score="100">cosfest</tag>
  <tag score="100">glasto2010</tag>
  <tag score="100">huracanalex</tag>
  <tag score="100">macysfireworks</tag>
  <tag score="100">canadadayfireworks</tag>
  <tag score="100">happy4th</tag>
  <tag score="100">peachtreeroadrace</tag>
  <tag score="100">jul10</tag>
  <tag score="100">redwhiteandboom</tag>
  <tag score="100">goodwoodfestivalofspeed2010</tag>
  <tag score="100">rondevanfrankrijk</tag>
  <tag score="100">marincountyfair</tag>
  <tag score="100">stpaulscarnival</tag>
</hottags>
</rsp>

4.2 Delicious

Delicious is a social bookmarking service on the web launched in 2003, i.e., a service where its users can save and share bookmarks (URL addresses). It is important to stress that our study was done before the acquisition of delicious by AVOS Systems. Thus, the following content might be outdated.
4.2.1 Authentication

It is possible to access public data from Delicious in an anonymously way, by using the web feeds (a data format to publish content) service of the system. On the other hand, in order to access private data, the requests must be authenticated by using the OAuth – an open protocol to enable an application to access end user information from a Web service. The process of authentication is described in OAuth Authorization Flow web page\(^82\).

i OAUTH Python Library There is a python library that supports Oauth authentication: oauth2\(^83\).

4.2.2 Feeds

To access public data from Delicious, there are read-only data feeds\(^84\), which adopted in our study, since our work is focused on public data. The response of feed requests comes in two possible formats – RSS\(^85\) and JSON\(^86\).

i Update Rate

Due to practical reasons, the Delicious system does accept update requests of the feeds very often. RSS feeds, for instance, may not be updated more than twice an hour. Requesting data more often than allowed by the system may result in HTTP 503 errors, indicating either that the requests were blocked or throttled by the servers.

ii Feeds Available

All feeds follow this base URL prefix:

```
http://feeds.delicious.com/v2/{format}/
```

Where the placeholder \{format\} is the feed format: rss or json.

The following parameters are accepted:

?count = \{1..1000\} limit the results – default (15).

?plain or ?fancy disable or enable HTML content.

?callback=js call allows the inclusion of a wrapper call. Only JSON data.

Additional placeholders used in URLs further described are:

\{format\} rss or json.

\{username\} user’s login name on delicious

\(^82\)http://developer.yahoo.com/oauth/guide/oauth-auth-flow.html
\(^83\)http://github.com/simplegeo/python-oauth2
\(^84\)http://delicious.com/help/feeds
\(^85\)http://en.wikipedia.org/wiki/RSS_(protocol)
\(^86\)http://json.org/
\{\text{tag+}[\text{tag+}...+\text{tag}]\} \text{ tag or intersection of tags.}

\{\text{url md5}\} \text{ MD5 hash of a URL.}

\{\text{key}\} \text{ security key that allows view private data.}

iii URL Patterns for Feeds

- Recent bookmarks:
  
  \url{http://feeds.delicious.com/v2/{format}/recent}

- Recent bookmarks by tag:
  
  \url{http://feeds.delicious.com/v2/{format}/tag/{tag+tag+...+tag}}

- Bookmarks for a specific user:
  
  \url{http://feeds.delicious.com/v2/{format}/{username}}

- Private bookmarks for a specific user:
  
  \url{http://feeds.delicious.com/v2/{format}/{username}?private={key}}

- Bookmarks for a specific user by tag(s):
  
  \url{http://feeds.delicious.com/v2/{format}/{username}/{tag+tag+...+tag}}

- Private bookmarks for a specific user by tag(s):
  
  \url{http://feeds.delicious.com/v2/{format}/{username}/{tag+tag+...+tag}?private={key}}

- Public summary information about a user:
  
  \url{http://feeds.delicious.com/v2/{format}/userinfo/{username}}

- A list of all public tags for a user:
  
  \url{http://feeds.delicious.com/v2/{format}/tags/{username}}

- A list of related public tags for a user/tag combination:
  
  \url{http://feeds.delicious.com/v2/{format}/tags/{username}/{tag+tag+...+tag}}

- Bookmarks from subscriptions of a given user:
  
  \url{http://feeds.delicious.com/v2/{format}/subscriptions/{username}}
• Private feed for of third-party suggested bookmarks for a given user:
  \[
  \text{http://feeds.delicious.com/v2/\{format\}/inbox/\{username\}?private=\{key\}}
  \]

• Bookmarks from network members of a given user:
  \[
  \text{http://feeds.delicious.com/v2/\{format\}/network/\{username\}}
  \]

• Bookmarks from network members of a given user by tag:
  \[
  \text{http://feeds.delicious.com/v2/\{format\}/network/\{username\}/\{tag\}+\{tag\}+\ldots+\{tag\}}
  \]

• A list of network members of a given user:
  \[
  \text{http://feeds.delicious.com/v2/\{format\}/networkmembers/\{username\}}
  \]

• Recent bookmarks for a URL:
  \[
  \text{http://feeds.delicious.com/v2/\{format\}/url/\{url md5\}}
  \]

• Summary information about a URL:
  \[
  \text{http://feeds.delicious.com/v2/json/urlinfo/\{url md5\}}
  \]

5 Conclusion

Folksonomies maintained by web systems are an important source of information. In order to access and manage them these web systems usually provide web APIs. As a partial result of a research we are conducting concerning folksonomies, we have developed the tool presented here, which can access, retrieve and store data from folksonomies.

In this technical report we showed the tool we developed, detailing the strategy to access folksonomy based systems. We also showed our work of a unified tag database, derived from the comparison of related work and models adopted by web systems.

References


