Magellan User Manual

Version 0.1

October 16, 2015

1 Installing Magellan

Magellan depends on the following Python packages: pandas, numpy, scikit-learn, PyQT4 and jpype. Installing these packages one by one can be difficult for most lay users. Thus, if these packages have not been installed on your machine, we recommend that you install Anaconda.

Anaconda is a Python distribution for data analytics and scientific computing. It bundles a lot of packages required for data analytics, including the above packages, thus freeing the user from the burden of manually installing them. It works across platforms (such as Linux, Mac OS X, Windows).

Anaconda also comes with its own version of Python (so when you invoke python, you should invoke the version that comes with Anaconda, see more below). Currently Magellan supports Python 2.7. Support for Python 3.4 will be added at some point.

```
Magellan has been tested on these platforms:

1. Linux - Redhat enterprise linux with 2.6.32 kernel
2. Windows - version 10
3. Mac - OSX El Capitan

A user is free to install Magellan in any of the above platforms.
However if the user has an option to choose a platform,  we recommend
linux, as it was used as the primary platform for development and extensive
testing.
```

The installation instruction below is for Linux, but can be adapted for other platforms.

Step 1 - Installing Anaconda: This step can be skipped if the packages pandas, numpy, scikit-learn, PyQT4 and jpype are already installed.

- Download Anaconda installer for Python 2.7 from http://continuum.io/downloads into a directory in your home directory, say "HOME/anaconda", where "HOME" should be replaced by the path referring to your home directory.
- Follow the instructions at http://docs.continuumio/anaconda/install#linux-install to install Anaconda. During the installation process, you will be asked to choose a directory in which to install Anaconda. You can specify any directory. For now, we will assume that you specify "HOME/anaconda". Anaconda will install the above packages and many more additional packages. Then it will install its own version of Python. It will also ask if you want to add the installation directory ("HOME/anaconda" in this case) to the path variable. Say "yes".

- Close the current terminal and open a new one. This will have the effect of re-evaluating the path variable, thus making the installation directory active in the path variable. This means when you call python, you will be calling the one that comes with Anaconda. You can check for this by executing "which python" in the new terminal. It should list the path "HOME/anaconda/bin/python".

Step 2 - Installing Magellan:

- Download the Magellan package from http://pages.cs.wisc.edu/ pradap/magellan/Magellan-0.1.tar.gz into your home directory. You can download into any directory in your home directory. For now we assume that you will download into "HOME/", the top level.

- Unzip the package by executing "tar -xzvf Magellan-0.1.tar.gz". Magellan will be unpackaged into directory "HOME/Magellan-0.1".

- At the command prompt, execute "cd HOME/Magellan-0.1" to enter the above directory, then execute "python setup.py install" to install Magellan as a package for Python. This command will make changes to "HOME/Magellan-0.1" and "HOME/anaconda". Remember to use the python version that comes with anaconda. (If anaconda has not been installed, then use Python 2.7.)

At this point, Magellan has been installed on your machine.

Step 3 - Starting Up IPython: At this point, you can start the Python interactive environment, import Magellan as a package, then start using it.

However, instead of using the Python interactive environment, we strongly recommend using IPython. Simply put, IPython is also an interactive environment, but browser-based. It comes with a lot of very nifty features, and is used extensively by data scientists.

Anaconda contains the IPython package. If Anaconda has not been installed,then IPython package can be installed separately by following the instructions at http://ipython.org/install.html.

To start up IPython, do the following:

- First, decide on a directory where you will work (e.g., storing the two tables to be matched, saving your Python command histories, storing intermediate and final results, etc.). For now, we assume this directory is "HOME/magellan-workdir". Create this directory and "cd" into it.

- Next, execute "ipython notebook" inside this directory. It should open up a webpage and from there create a new notebook. The page http://ipython.org/notebook.html provides a good introductory information about ipython and its usage.

More information about using IPython: The IPython notebook dashboard will launch inside the user's default web browser. The page will resemble the screenshot as shown in figure 1.

The user can create a notebook by clicking on "new" button and then clicking on "python2" item as shown in figure 2 and the new notebook will resemble the screenshot in figure 3.

The title of the notebook can be changed to say "magellan-work" by left clicking on "Untitled0" at the top of the page as shown in the figure 4.

The IPython notebook is composed of different types of cells such as code-cell, markdown-cell, etc. By default each cell is a code-cell. A cell can be highlighted by clicking on it. Any valid python code can be written inside a code-cell, and it can be executed by pressing Shift-Enter. A simple example of executing a python code inside a code-cell is shown in figure 5.

For more elaborate tutorials on using IPython notebook please look at
https://ipython.org/ipython-doc/2/interactive/tutorial.html

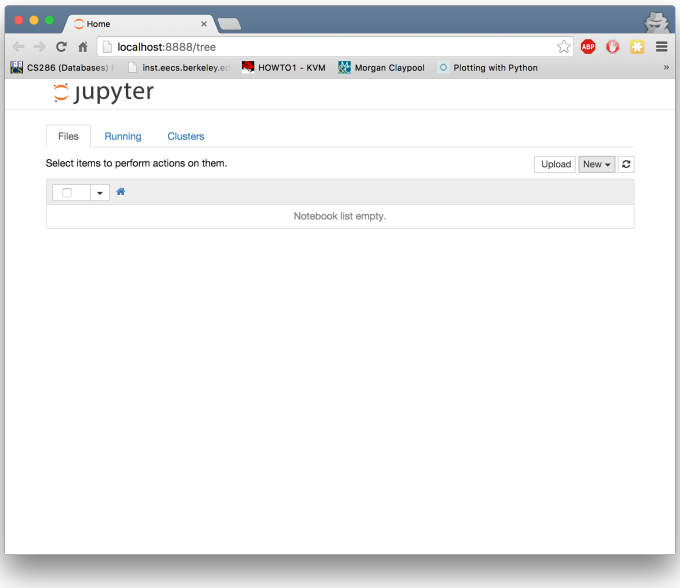

Figure 1: IPython notebook dashboard

The changes made to the notebook gets automatically saved to file title.ipynb where "title" is the title of the notebook. If the title was set to magellan-work, then the file will be stored as magellan-work.ipynb. The file will be stored in the directory from where ipython notebook was launched (i.e in magellan-workdir directory).

To exit IPython notebook, execute Ctrl + c in the terminal from where IPython notebook was launched.

Step 4 - Importing Magellan: Once IPython is in place, you should import Magellan:

```
import magellan as mg
```

You can now start using Magellan by executing its commands (which should be prefixed with "mg.").

Magellan Examples: You can find sample datasets and some Magellan examples at http://pages.cs.wisc.edu/ pradap/magella

2 Data Structures

A Note on Conventions: We will often use

- A and B to refer to the original two tables to be matched,
- C to refer to the candidate set table obtained from A and B after the blocking step,
- S to refer to a sample taken from C, and
- G to refer to a table that contains the tuple pairs in S plus a golden label for each pair (indicating whether the pair matches).

MTables: We will need to store a lot of data as tables in Magellan. An obvious candidate for this is data frame in pandas. However, data frames do not allow us to store meta data, even though in Magellan we need to store a lot of meta data with a table. Here are a few examples:

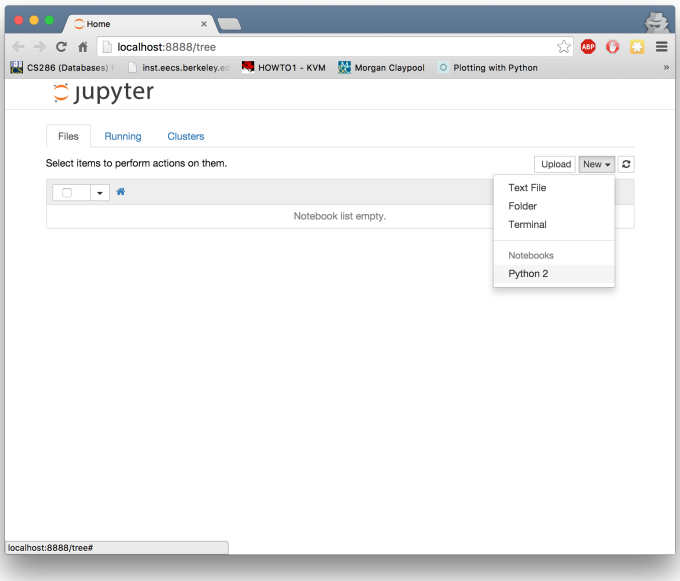

Figure 2: New notebook creation.

- Each table in Magellan should have a key, so that we can easily identify the tuples. Keys are also critical later for debugging, provenance, etc. Key is a piece of meta data that we want to store with a table.

- We may want to infer the types of attributes in a table (these are internal Magellan types, used for feature generation, not Python types). We would want to store these types in the meta data as well.

- The blocking step will create tuple pairs from two tables A and B. For example suppose we have A(aid,a,b) and B(bid,x,y), then the tuple pairs can be stored in a candidate set table C(cid,aid,bid,a,b,x,y). This table could be very big, taking up a lot of space. To save space, we may want to just store C as C(cid,aid,bid) and then have pointers back to tables A and B. In effect, C is a very simple view over base tables A and B. The two pointers back to A and B are meta data that we may want to store with table C. Another piece of meta data is "aid and bid in table C are foreign keys in tables A and B".

There are many other examples of meta data that we may want to store with a table. Consequently, data frame is not good enough for our purpose. We will need to extend it to store meta data as well.

MTable is our exension. Each MTable object is a data frame, but it also has an extra field called "properties". This is a dictionary that can have all kinds of keys that store meta data. Examples of such keys are:

```
+ key: the name of the key attribute of the table

+ ltable: the name of the left table (see below)

+ rtable: the name of the right table.
```

We assume that the key of a table is just a single attribute (i.e. composite keys are not allowed). This is a limitation of current MTable design, made to make key management easier. It may cause some issues later. So we need to watch for this and remove this limitation when it becomes too much.

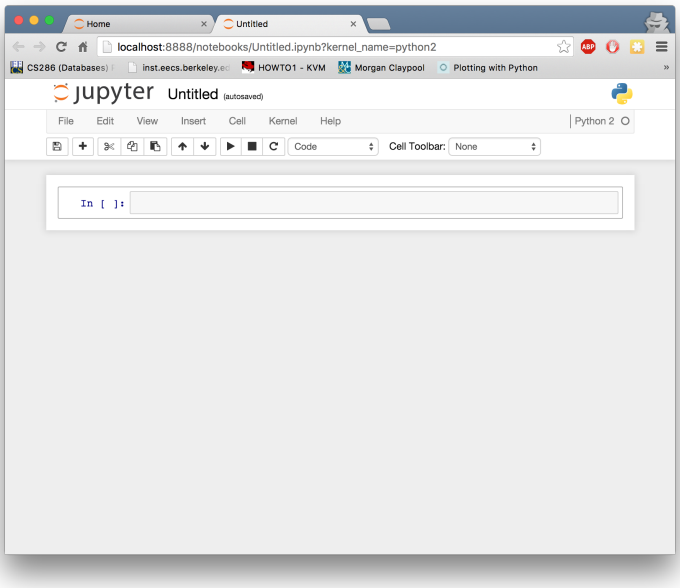

Figure 3: New IPython notebook

Virtual MTable: After performing blocking on two tables A and B, we typically obtain a table C of cand set. This table can be very large, so we typically represent it using a view over two tables A and B. This form a virtual MTable object. Such a table C will have

```
+ attributes _id, ltable.aid, rtable.bid, some attributes from A and B (user
  can specify which attributes from A and B should be included in C. Here
  aid and bid are the keys of A and B, respectively. We prefix them with
  ltable and rtable to handle cases where aid and bid are identical.

+ the properties dictionary (the meta data part) will have at least
  these fields:
  - ltable: points to A
  - rtable: points to B
  - foreign_key_ltable: ltable.aid (that is, ltable.aid is a foreign key of A)
  - foreign_key_rtable: rtable.bid
```

Thus, the important things to remember:

```
+ each regular MTable is just a panda data frame with a set of
key-value pairs describing the properties of the table. One key is
always ``key'', referring to the key column of the table.

+ each virtual MTable is a regular MTable, together with pointers to
tables A and B. The table does this using four keys: ltable, rtable,
foreign_key_ltable, and foreign_key_rtable.

+ so there are five reserved words for the properties: key, ltable,
rtable, foreign_key_ltable, and foreign_key_rtable.
```

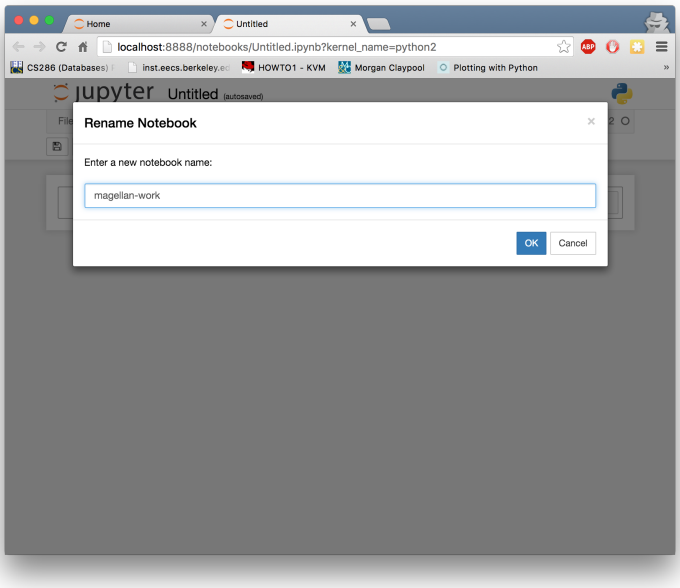

Figure 4: Renaming a notebook

Tuples: Each tuple from an MTable is a Panda series.

3 Reading the CSV Tables from Disk

The Minimal That You Should Do: The original tables A and B (to be matched) must be stored as CSV tables on disk.

Example 1. *Suppose we have stored the two tables to be matched as table_A.csv and table_B.csv on disk. Then table_A.csv may look like this:*

```
ID, name, birth_year, hourly_wage, zipcode
a1, Kevin Smith, 1989, 30, 94107
a2, Michael Franklin, 1988, 27.5, 94122
a3, William Bridge, 19886, 32
```

To read these tables into memory as MTable objects, use the read_csv command.

Example 2. *The two tables mentioned in the above example can be read into Magellan MTable objects as follows:*

```
A = mg.read_csv('HOME/magellan-workdir/table_A.csv', key='ID')
B = mg.read_csv('HOME/magellan-workdir/table_B.csv', key='ID')
```

Here, "mg" refers to the Magellan package (as imported into the Python environment). Note that we do have to specify the keys of the tables. Here, we assume that both tables A and B have column ID as the keys.

If You Want to Understand and Play Around More: In general, Magellan assumes the format of a CSV file to be like this:

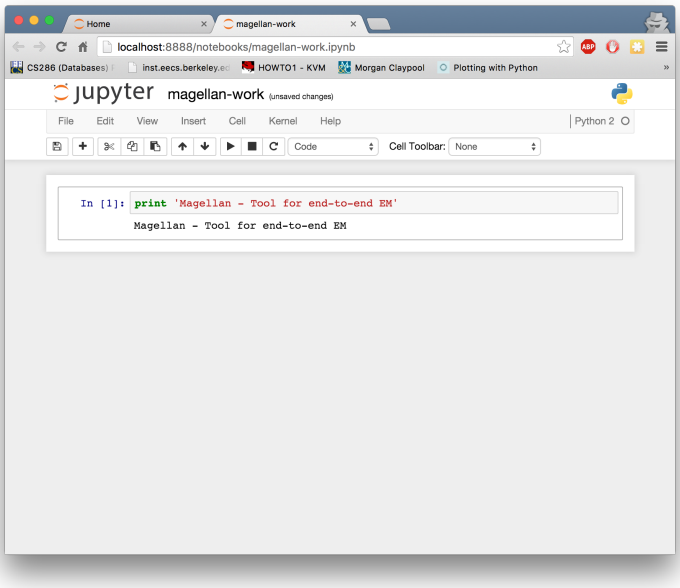

Figure 5: Sample execution in IPython notebook

```
# key=ssn
# ltable=POINTER
# rtable=POINTER
# foreign_key_ltable=ltable.ssn
# foreign_key_rtable=rtable.eid
ID, name, birth_year, hourly_wage, zipcode
a1, Kevin Smith, 1989, 30, 94107
a2, Michael Franklin, 1988, 27.5, 94122
a3, William Bridge, 19886, 32
```

where the lines starting with # are optional. If they exist, they list the key-value pairs to be inserted as meta data for the MTable object. See Section ... for more detail.

The formal format of the read_csv command is:

```
mg.read_csv(args_for_pandas_read_csv,
            key-value pairs for metadata seperated by commas)
```

Here, mg.read_csv will call panda's read_csv, so first we specify the arguments for panda's read_csv, if any. (See http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html for a list of such arguments). Then we supply key-value pairs seperated by commas (if any) for meta data. The command will return an MTable object.

Specifically, this command will first read the comma-separated values in the CSV file into a panda data frame. Next, it sets up the meta data of the table in two ways: (1) it first parses the comment lines of the csv file (if any) to extract key-value pairs and add those to the table as metadata. (2) Then it looks at the list of key-value pairs being supplied in read_csv argument line and add those to the table as metadata. So if a key was mentioned in the csv file and also in the argument list of read_csv, then the one in the argument list will override the one in the file.

As the default, Python will assume the csv file has the attribute names. If they are not there, the user will have to tell python read_csv command.

When a table X is created in Magellan, it must have a key (ie a column in the table serving as the key). When table X is created by reading from a file, the key can be obtained from the comment lines in the file, or can be supplied as an argument in read_csv.

If the specified column is not a key eg containing missing or duplicate data, then the read will fail and no table will be created.

Otherwise, the comment lines of the CSV file contain no key, and the user has also specified no key as an argument in read_csv. In this case a new column named "_id" (prefix of "_" to indicate that it is a column generated by Magellan) is automatically generated, then added to the input table as the first attribute and set as key.

Once the table has been created, the user can check to see which attribute of the table is a key, using the get_key command. For instance, in Example ... the user has obtained an MTable object A. Executing A.get_key() will return "ID" as the key of the table.

4 Browsing the Tables

If the user uses IPython he/she can view the tables (as IPython supports viewing pandas dataframes and MTable inherits from pandas dataframe). For example, if the variable "A" points to a MTable object then the table can be viewed (in an IPython session) by just typing the variable name and pressing Enter. IPython will display the MTable (in the same browser) that A points to, in a tabular format. If the table is large (like 3K tuples) then the table will be displayed with scroll bars (left - right and top - bottom) so that the user can browse it.

Magellan currently has no way yet for users to edit MTable objects. Thus, to edit the tables A and B, we recommend editing the CSV files in your favorite editor, then reloading them again into MTable objects.

5 Specifying Blockers and Performing Blocking

5.1 Types of Blockers and Blocker Hierarchy

```
After understanding and transforming the data, as discussed above,
users often must do blocking. Note that
+ by "blocking" we mean to block a tuple pair from going
through to the matching step. We don't mean dividing tuples
into blocks and match just within each block.

+ when applied to a tuple pair, a blocker returns true
if the pair should be blocked (this is the reverse
of what a matcher would return).

There are two types of blockers: tuple-level and global. A tuple-level
blocker can examine a tuple pair (x,y) in isolation and decide if it
should be admitted to the next stage (that is, matching). (For
example, an attribute equivalence based blocker is a tuple-level
blocker.) A global blocker can't make this decision in isolation.  It
would need to examine a bunch of other pairs as well.  For example,
consider a sorted neighborhood blocker that unions the two tables A
and B into a table D, say, sorts the tuples in D according to some
key, then considers matching only a tuple with the previous (w-1)
```

tuples (in the sort order). This blocker is global.

Blockers can be combined in complex ways, such as
+ apply b1 to the two tables
+ apply b2 to the two tables
+ apply b3 to the output of b2
+ combine the outputs of b1 and b3
So blockers can be applied sequentially.

Why we may want to apply blockers sequentially, such as b2 => b3.
Suppose a blocker applies a=b AND t=v. Then we may want to break
this blocker into two blockers b2 and b3, to be applied sequentially
so that it is easier to debug etc. Another scenario is that
b3 could be a set of rules.

This is tricky because if we apply a global blocker such
as sorted neighborhood to the output of another blocker, it is
not clear what is the semantics of this. A global blocker is
supposed to be applied globally to the two tables A and B.
It is not supposed to be applied to a candidate set.

Note that we may also want to apply a bocker to just a pair
of tuples, to see how the blocker works. Global blockers
also can't be applied to just a pair of tuples.

So in the current Magellan, we enforce the following:
+ a tuple-level blocker can be applied to two tables, to
a cand set, and to a pair of tuples.
+ a global blocker can be applied only to two tables.

Currently we have an inheritance hierarchy for blockers:

```
Blocker => AttrEquivBlocker
        => HashBlocker
        => SortedNBBlocker
        => CanopyBlocker
        => RuleBlocker
        ...
```

This is quite flat right now; but we can reorganize to add some
intermediate levels later.

Each tuple-level blocker will implement at least the following
three methods:
+ block_tables (so it can be applied directly to tables A and B)
+ block_candset (so it can be applied to the output of another blocker)
+ block_tuples (so if we want to see if a tuple pair survives the blocker

```
                we can try it)

Each global blocker will implement only block_tables. It can't
implement block_candset and block_tuples because the meaning
of those is unclear.

So far we have discussed tuple-level vs global blockers. Another
way to classify blockers into built-in, blackbox, and
rule-based blockers. We discuss these three types next.
```

5.2 Built-In Blockers

Built-in blockers are those that have been built into Magellan and the user can simply just call them. Magellan currently offers two built-in blockers: attribute equivalence and overlap.

Attribute Equivalence Blocker

```
block_tables(ltable, rtable, l_block_attr, r_block_attr,
                    l_output_attrs=none, r_output_attrs=none)

This blocker will do l_block_attr = r_block_attr blocking on tables
ltable and rtable. It creates an MTable object with attributes _id,
key attribute from ltable, key attribute from rtable, followed by
lists l_output_attrs and r_output_attrs if they are specified. Each of
these output and key attrs will be prefixed with "ltable." or
"rtable.".

For meta data, key-value pairs for key, ltable, rtable, foreign_key_ltable,
foreign_key_rtable (which are key attributes from ltable and rtable)
will be added.

Examples of calling this blocker:

ab = AttrEquivBlocker()
c = ab.block_tables(A, B, 'zipcode', 'zipcode')

Another example:
c = ab.block_tables(A, B, l_block_attr='zipcode', r_block_attr='zipcode',
            l_output_attrs=['name', 'birth_year', 'hourly_wage', 'zipcode'],
            r_output_attrs=['name', 'birth_year', 'hourly_wage', 'zipcode']
                                )

Blocking tuples:

ab.block_tuples(ltuple, rtuple, lattr, rattr)

this returns true if the tuple pair should be blocked.
```

Overlap Blocker:

This blocker takes an attribute x of table A, an attribute y of table B, and returns true (that is, drop the tuple pair) if x and y do not share any token (where the token can be a word or a q-gram). For example, if two titles do not share any token, then drop the pair.

The signature of the blocker function is:

block_tables(A, B, x, y, rem_stop_words = False, qgram = None, word_level = True, overlap_size=s, l_output_attrs=None, r_output_attrs=None)

Here, qgram, word_level, l_output_attrs, r_output_attrs are optional arguments. It cant be the case that we have both word_level and qgram parameter set. If rem_stop_words is set to True, then common stopwords are removed before tokenization. If qgram = k then we tokenize x and y attributes into sets of k grams, then check if there are at least ``s'' overlap tokens (overlap-size should also be optional, with default being 1). If word_level = true then we tokenize x and y at the word level.

This blocker is implemented using an inverted index on one of the tables.

Examples of calling this blocker:

```
ob = OverlapBlocker()
C = ob.block_tables(A, B, l_block_attr=title, r_block_attr=title,
overlap_size=3)
```

Another example

```
C = ob.block_tables(A, B, l_block_attr=title, r_block_attr=title,
word_level=False, qgram=3, overlap_size=10)
```

Similar to other tuple-level blockers, this blocker supports block_candset and block_tuples command.

5.3 Blackbox Blockers

By ``blackbox blockers'', we basically mean the user supplies a python function that Magellan can call. Currently we only consider tuple-level blackbox blockers. There are many possible ways to write a blackbox blocker. Magellan currently supports the following way. First, define a function foo:

```
def foo(ltuple, rtuple):
  xxx
  this function must return true or false
```

Then define

```
bb = BlackBoxBlocker()
bb.add_blackbox_fn(foo)
bb.block_tables(...)
```

5.4 Rule-Based Blockers

We should consider how user can write a new blocker quickly (and perhaps declaratively, without writing much code). For now, we limit this to user's writing rule-based blockers.

If user wants to write rule-based blockers, then he/she must start by defining a set of features. Each feature will be a function that when applied to a tuple pair will return a single value. We will discuss how to define a set of features in Section ... Once defined, Magellan stores this set of features in a feature table. We often refer to this feature table as block_f.

Then the user may be able to write a rule-based blocker like this:

```
rb = RuleBlocker()
rb.add_rule(rule1, block_f)
rb.add_rule(rule2, block_f)
rb.block_tables(ltable, rtable, l_output_attrs, r_output_attrs)
```

Here block_f is a set of features for blocking, in the form of a data frame (see Section ...). Each rule is a string that specifies a conjunction of predicates. Each predicate has three parts: an expression, an operation, and a value. The expression can be evaluated over a tuple pair, producing a single value. For now, we limit such expression to just a single feature (being applied to a tuple pair). So a predicate may look like this:

```
title_title_jac_3gram_3gram(ltuple, rtuple) > 0.8
```

Eventually we can relax this so that any expression that can be evaluated can be plugged in there by the user.

The blocker is then a disjunction of rules.

IMPORTANT: We said above that each rule is a string that specifies a conjunction of predicates. In practice, Magellan requires each rule to be a LIST of strings, each string specifies a predicate. The examples below illustrate this point.

EXAMPLE:

```
rb.add_rule([’name_name_lev(ltuple, rtuple) < 0.6’,
             ’zipcode_zipcode_lev(ltuple, rtuple) < 0.8’] , block_f)

Here [’name_name_lev(ltuple, rtuple) < 0.6’,
      ’zipcode_zipcode_lev(ltuple, rtuple) < 0.8’] is
a list of predicates (each predicate is a string, and two predicates are
separated by comma). The user does not explicitly write a conjunction
operator, magellan assumes that conjunction should be applied to the
predicates mentioned in the list. Internally this rule will be converted
to a Python function as shown below

def rule_1(ltuple, rtuple):
    return name_name_lev(ltuple, rtuple) < 0.6
           and zipcode_zipcode_lev(ltuple, rtuple) < 0.8

If the user adds another rule
rb.add_rule([’city_name_lev(ltuple, rtuple) < 0.7’], block_f)

Then, magellan will create a Python function for that rule as shown below

def rule_2(ltuple, rtuple):
  return city_name_lev(ltuple, rtuple) < 0.7

The blocker will execute rules in the order added and will return
true if at least one of the rule returns True.
```

Summary:

```
To add a rule, we specify a list that lists all
conjuncts of the rule (eg., [’name_name_lev(ltuple, rtuple) < 0.6’,
’zipcode_zipcode_lev(ltuple, rtuple) < 0.8’]). Each conjunct is a
string that can only refer to the name of a feature (as listed in
block_f), ‘‘ltuple’’, and ‘‘rtuple’’.  Anything else will cause a
run-time error.

Thus, as of now, we cannot have an arbitrary function inside
a rule of a rule-based blocker. In other words, we can’t define

def foo(ltuple, rtuple):
 xxx

then write

rb.add_rule([’foo(ltuple, rtuple) < 0.7’], block_f)

To do this, we need to first add foo as a feature to block_f.
```

5.5 Combining Multiple Blockers

If the user uses multiple blockers, he/she often has to combine them to get a consolidated candidate set. There are many different ways to combine the candidate sets such as doing union, intersection, majority vote, weighted vote, etc. Currently magellan supports union-based combining and all the other approaches are left for future work.

In magellan, combine_block_outputs_via_union command can be used to do union-based combining.

```
mg.combine_block_outputs_via_union(blocker_output_list)
```

Here, blocker_output_list is a list of MTable objects. The command produces an MTable object which is the combined candidate set. (See more below).

EXAMPLE FOR COMBINING BLOCKERS

```
--------------------------------------------------------------------------------
ab = AttrEquivalenceBlocker()
C = ab.block_tables(A, B, 'zipcode', 'zipcode', l_output_attrs=['zipcode',
'name', 'birth_year'], r_output_attrs=['zipcode', 'name', 'birth_year'])

feature_table = mg.get_features_for_blocking(A, B)
rb = mg.RuleBasedBlocker()
# Add rule : block tuples if name_name_mel(ltuple, rtuple) < 0.4
rb.add_rule(['name_name_mel(ltuple, rtuple) < 0.4'], feature_table)
D = rb.block_tables(A, B, l_output_attrs=['name'], r_output_attrs=['name'])

def my_function(x, y):
# The blocker function will drop tuples whose last name do not match
# The function has to do the following steps
#  1) Get name attributes from each of the tuples
#  2) Split name attribute to get last name
#  3) if last names donot match then return True

    # get name attribute
    x_name = x['name']
    y_name = y['name']
    # get last names
    x_name = x_name.split(' ')[1]
    y_name = y_name.split(' ')[1]
    # check if last names match
    if x_name != y_name:
        return True
    else:
        return False
```

```
bb.set_black_box_function(my_function)
F = bb.block_tables(A, B, l_output_attrs=['name'], r_output_attrs=['name'])

# combine blocker outputs
G = mg.combine_block_outputs_via_union([D, E, F])
```

Input/Ouput

The command takes in a list of MTable objects (candidate sets, typically output from blockers) and returns a consolidated MTable object. The output MTable object contains the union of tuple pairs ids and other attributes from the input list.

Assumptions

The command assumes the following about the input MTables. Each input MTable is expected to contain valid ltable and rtable properties (i.e they are expected to refer to valid MTables). Each input MTable (in the input list), must be a result of blocking from same underlying tables. Concretely the ltable, rtable properties must refer to same object across all input MTables. All input MTables must have foreign_key_ltable property set to ltable.aid (where aid is key attribute name of ltable) and foreign_key_rtable set to rtable.bid(where bid is key attribute name of rtable). In other words, each input MTable is expected to contain two columns ltable.aid and rtable.bid. Further, ltable.aid and rtable.bid must be set as foreign keys to ltable and rtable. In each input MTable, for the attributes included from ltable or rtable, the attribute names must be prefixed with ltable. and rtable.

As an example, the schema for an input MTable may look like this: _id, ltable.ID, rtable.ID, ltable.name, ltable.zipcode, rtable.name, rtable.hourly_wage

Handling different attribute lists

The input MTables may contain different attribute lists and it begs the question how to combine them. Currently magellan takes a union of attribute names that has prefix ltable. or rtable. across input tables. After taking the union, for each tuple id pair included in output, the attribute values (for union-ed attribute names) are probed from ltable/rtable and included in the output.

NOTE: A subtle point: If an input candidate set has a column added by user (say label for some reason), then that column will not be present in the output. The reason is, the same column may not be present in other candidate sets so it is not clear about how to

```
combine them. One possibility is include label in output for all
tuple id pairs, but set as NaN for the values not present. Currently
magellan does not include such columns and addressing it will be part
of future work.

Handling ids from input MTables

Each input MTable can have different key attribute and combining them
can be difficult. Currently the command adds a new key attribute (with
name set to _id) to output MTable.

Steps involved

To be more precise, the command performs the following seven steps:

Get a union of ltable.aid, rtable.bid pairs from all the input
candidate sets (MTables). To be concrete let us name the set as U.

Get a union of attribute names (that are included from ltable and
rtable) from all the input MTables. Let L be the set for ltable and R
be the set for rtable.

Get the underlying ltable, rtable from which all the input MTables
were derived.

Create an empty list M
For each ltable.aid, rtable.bid in U:
   Probe ltable to get values in L and rtable to get values in R
   Create a tuple with the values. Attribute names from L are
   prefixed with ltable. and attribute name from R are prefixed
   with rtable. Add the tuple to M
Create a MTable from M, and add a key attribute _id
Return M
```

6 Creating Features for Blocking

6.1 Introduction

```
Recall that when doing blocking, the user can use built-in blockers,
blackbox blockers, or rule-based blockers. For rule-based blockers,
the user often has to create a set of features.

In creating features, user will have to refer to tokenizers, sim
functions, and attributes of the tables. There will be two ways
to create features:
+ system can automatically generate a set of features,
  then the user can remove or add some more.
+ user can skip the automatic process and generate features of their owns.
```

Note that features will also be used in the matching process, as we will discuss later. The set of features for blocking and the set of features for matching can be quite different however. For example, for blocking we may only want to have features that are inexpensive to compute.

Lay users may just want to ask the system to automatically generate a set of features. To do this, please see Section ... For users who want to generate features of their own, please read below.

6.2 Tokenizers and Sim Funtions Currently Available in Magellan

A tokenizer is a function that takes a string and optionally a number of other arguments, then tokenize the string and return an output value which is often a set of tokens. eg, qgram_tokenizer(str, q), delimiter_tokenizer(str, de_char).

We list all tokenizers currently available in Magellan in the global variable _global_tokenizers.

A sim function takes two arguments, which are typically two attribute values such as two book titles, then return an output value which is typically a sim score between the two attribute values. We list all sim functions currently available in Magellan in the global variable _global_sim_funs.

Note that _global_tokenizers and _global_sim_funs will be used for both blocking and matching purposes.

NOTE THAT EACH TOKENIZER AND SIM FUNCTION HAS A LONG NAME AND A SHORT NAME. The long name is the informative name of the tokenizer/sim fun. The short name is used in the concatenation to create the feature names. These names are currently stored in the Python source-code file that contains the code for tokenizers and sim functions, and will be read into memory when Magellan is imported into Python.

The sim functions in current Magellan:
['jaccard', 'lev', 'cosine', 'monge_elkan', 'needleman_wunsch',
'smith_waterman', 'smith_waterman_gotoh', 'jaro', 'jaro_winkler',
'soundex', 'exact_match', 'abs_norm', 'rel_diff']

The tokenizers in current Magellan: ['tok_delim', 'tok_qgram']

6.3 Obtaining Tokenizers and Sim Functions for Blocking

To start, we should execute the command

```
block_t = get_tokenizers_for_blocking(q = [3,5], de_char = ' ')
```

```
or something similar to obtain a set of tokenizers that we will use for
blocking. block_t will be a dict where keys are tokenizer names
and values are tokenizer codes. User can inspect t and delete/add tokenizers
as appropriate.

The above command will do two things. First, it returns
single-argument tokenizers, i.e., those that take a string then
produce a set of tokens. Second, it examines the set
_global_tokenizers then selects only those deemed appropriate for
blocking, e.g., those that are quite cheap to execute.

The reason we want to obtain single-argument tokenizers is because
eventually we want to write expressions such as
jacccard_sim(tok_qgram(x.title), tok_qgram(y.title)). Here the
tokenizer will have just a single argument.  So if we have tokenizers
that take multiple arguments, then we want to generate all templates
from them that have just a single argument, to faciliate writing
expressions such as above.

Similarly, we should do

block_s = get_sim_funs_for_blocking()

to obtain a set of sim funs that we can use for blocking. Note that
not all sim functions in _global_sim_funs would be appropriate for
blocking, as some may be very expensive. This command tries to select
only the cheap sim functions. Here, block_s is a dict where keys are
sim fun names and values are sim fun codes. (pointers to code, that
is.)

block_t and block_s are the two dicts that user can add/delete
tokenizers/sim funs as appropriate. (User can't add stuff directly
to _global_tokenizers and _global_sim_funs.)
```

A user can get "help" about a similarity function or a tokenizer by executing the following command in IPython session

```
help(command_name).
```

For example, if a user wants to quickly get information about "tok_qgram" command, he/she can execute help(tok_qgram) and he/she will be displayed with information about the function, parameters and the return value. In the above example, he/she will be displayed with

```
tok_qgram(s, q)
    q-gram tokenizer; splits the input string into a list of q-grams

    Parameters
    ----------
```

```
    s : string
        source string to be converted into qgrams
    q : integer
        q-value

    Returns
    -------
    qgram_list : list,
         q-gram list of source string
```

6.4 Obtaining Attribute Types and Correspondences for Blocking

```
In the next step, we need to obtain some information about A and B
so that we can generate features.

To start, we want to obtain the types of attributes in A and B,
so that we can apply the right tokenizers/sim functions to each of them.

atypes1 = get_attr_types(A)
atypes2 = get_attr_types(B)

atypes1 should return a dictionary D: a key is an attribute name, a value
is the type of that attribute. D should also specify that this
information is for table A. So D should have a key ``_table'', and the
value of that should be a pointer to A. atypes2 behaves similarly.

Next, we need to obtain correspondences between the attributes of A and B,
so that we can generate features based on those.

block_c = get_attr_corres(A,B)
This should return a dict with three keys:
+ corres: points to a list of correspondences: (a,a), (b,b), ... right now
this list contains only pairs of attributes with exact same names in A and B.
+ ltable: points to A
+ rtable: points to B.

Discussion:

+ It is important that atypes1, atypes2, and block_c specify that the
information they capture refers to A and B. Otherwise, this information is
underspecified.

+ It is very important that the list of correspondences (a,a), (b,b), ...
should be arranged in the order of attributes appearing in Tables A and B.
Eg, if a appears before b in Table A, then the correspondences for a should
appear before those for b. In case of conflicting appearance order, try to
make sure the correspondences appear in the order of the attributes in A.
This is to make sure that when user examines the list of attr corres,
```

he can easily comprehend and check to see if anything is missing.

6.5 Getting a Set of Features

Recall that so far we have obtained
+ block_t, the set of tokenizers,
+ block_s, the set of sim functions
+ atypes1 and atypes2: types of attributes in A and B
+ block_c: correspondences of attributes in A and B

To obtain a set of features, we execute command
block_f = get_features(A, B, atypes1, atypes2, block_c, block_t, block_s)

Briefly, this function will go through the correspondences. For each correspondence m, it examines the types of the involved attributes, then apply the appropriate tokenizers and sim functions to generate all appropriate features for this correspondence.

The command should return the features, but how these features should be organized? Three desirable things:
+ there should be an order among the features, in the order the attributes appear in table A. this is to help user understandability.
+ it should be easy to delete/insert features
+ to help understandabilty even more, we should capture even more information about each feature so that we can display that information to the user.

Accordingly, the function will return block_f, which is a data frame. block_f has attributes such as: feature_name, left attribute, right attribute, tokenizer, sim function, pointer to feature code, etc.
Each feature name has the form: title_title_jac_qgm_3_qgm_3.

Note that we can then display this table of features to the user. This whole thing assumes that data frame preserves the order of the tuples.

Feature code is a function that takes exactly two arguments ltuple and rtuple (ie two tuples), then return a single value. In this case, for example, we have the feature code look something like this:

```
function(ltuple, rtuple) {
p = get the title from ltuple
q = get the title from rtuple
return jaccard(qgm_3(p), qgm_3(q))
}
```

Discussion:

+ User can mistakenly enter block_f = get_features(B, A, atypes1, atypes2, block_c, block_t, block_s).

To prevent this, get_features should check to make sure that the table pointed to in atypes1 is the first table, the table pointed to in atypes2 is the second table, the two tables ltable and rtable in block_c are in the correct order.

+ A feature thus conceptually is a function that takes two tuples and return a score. It does not depend on any two tables. So technically a feature can be applied to any two tables, as long as the attributes referred to in the feature exist in those tables.

+ It is possible for a feature to relate a set of attributes in one tuple x to another set of attributes in the other tuple y. For example, a feature can compute the sim score between concat(ltuple.address_line1, ltuple.address_line2) with rtuple.address. Currently, Magellan can't automatically generate such features. But user can manually add them (see below).

6.6 Adding/Removing Features

Given the set of features block_f, user can delete certain features, add new features.

One way to add features is to write blackbox features (see Section ...).

Another way to add features is to write a feature expression in a ``declarative'' way. Magellan will then compile it into a feature. For example, user can write something like this:

```
r = get_feature_fn(``jac_sim(3gram_tokenizer(ltuple.publisher),
3gram_tokenizer(rtuple.issuer))'', block_t, block_s)

add_feature(block_f, ``publisher_issuer_jac_3gram_3gram'', r)
```

Here block_t and block_s refer to the set of tokenizers and sim functions for blocking, respectively.

The first command creates a feature which is a function that will take two tuples ltuple and rtuple, get the attribute publisher from ltuple, issuer from rtuple, tokenize them, then compute jaccard score.

The second command creates a feature with a particular name, supplying the above function as the feature code.

As described, the feature we have just created is *independent* of any table (eg A and B). Instead, it expects as the input two tuples ltuple and rtuple.

User can create more complex features. For example,

```
r = get_feature_fn(``jac_sim(3gram_tokenizer(ltuple.address_line1 +
 ltuple.address_line2), 3gram_tokenizer(rtuple.location))'', block_t, block_s).

This is why we define a feature to be a function taking two tuples ltuple
and rtuple. That allows us to define features like this. Arbitrary
complex expression involving function names from block_t and block_s,
and attribute names, and ltuple and rtuple, are allowed.

Discussion:

+ the nice thing about having features *not* depending on
tables is that later we can apply these features to any pair of table,
not just A and B, as long as the feature functions can get their
arguments (that is, the attributes).  This is a bit reminiscent of
duck typing.

+ notice that we can refer to the tuples from the first and second table
in feature definition as ltuple and rtuple without having to give names
to the tables.
```

6.7 Summary of the Manual Feature Generation Process

```
Here we summarize the entire manual feature generation process.

We start with _global_tokenizers and _global_sim_funs.

To generate the features, we execute the following commands
block_t = get_tokenizers_for_blocking(q = [3,5], de_char = ' ')
block_s = get_sim_funs_for_blocking()
atypes1 = get_attr_types(A)
atypes2 = get_attr_types(B)
block_c = get_attr_corres(A,B)
block_f = get_features(A, B, atypes1, atypes2, block_c, block_t, block_s)

We can add a feature like this:
r = get_feature_fn(``jac_sim(3gram_tokenizer(x.publisher),
3gram_tokenizer(y.issuer))'', block_t, block_s)

add_feature(block_f, ``publisher_issuer_jac_3gram_3gram'', r)

NOTE 1: atypes1 and atypes2 are not prefixed with ``block'' because
these two variables store the types of the attributes and this
information is not specific to blocking (when we do matching, we will
use the same information).

NOTE 2: the variable names block_t, block_s, ..., block_f are just
recommended names. The user can name these variables in any way he/she
likes.
```

NOTE 3: User of course can modify these variables in any way he/she likes, to add/remove tokenizers, sim_funs, correspondences, etc.

6.8 Ways for User to Edit the Manual Feature Generation Process

_global_tokenizers and _global_sim_funs are data frames (each with two columns). They can only be viewed, not editable by the user.

block_t and block_s are dictionaries (of shortname-code). User can add/remove tokenizers and sim functions to/from these.

atypes1 and atypes2 are dictionaries (of aname-type). Again the user can easily edit these.

block_c is a dictionary of correspondences. Again the user can easily edit these.

Editing block_f is significantly more involved. block_f is a data frame. Removing a feature means removing a tuple from this table.

To add a feature, we have two ways: blackbox and declarative. To add a blackbox feature, we first define it:

```
def my_feature(ltuple, rtuple):
  xxx
  return a single value v
```

Then we add it to the table block_f using add_blackbox_feature command like this:

```
add_blackbox_feature(block_f, my_feature, my_feature)
```

The second way to add a feature is to do it declaratively. We have described earlier how to do this:

```
r = get_feature_fn(str, block_t, block_s)
add_feature(block_f, feature_name, r)
```

Here str is a string that declaratively describes the feature. What is allowed in str: the names of tokenizers and sim functions as mentioned in block_t and block_s, ``ltuple'', ``rtuple'', and attribute names. Arbitrary expression involvind the aboves is okay, because we simply pass this string to later functions to evaluate.

Anything else in str will cause a run-time error. In particular, we can't mention function names that are not in block_t and block_s. To do this, we can first add such function names to block_t and block_s,

then we can safely refer to them in str.

6.9 How to Do the Whole Thing Automatically for Lay Users

Recall that to get the features for blocking, eventually user
must execute the following:

block_f = get_features(A, B, atypes1, atypes2, block_c, block_t, block_s)

where atypes1/atypes2 are the attribute types of A and B, block_c is
the correspondences between their attributes, block_t is the set of tokenizers,
and block_s is the set of sim functions.

If a user doesn't want to go through the hassle of creating
these intermediate variables, then the user can execute the following:

block_f = get_features_for_blocking(A,B)

The system will automatically generate a set of features
(stored in block_f) that user can then use for blocking purposes.

IMPORTANT: the command get_features_for_blocking will automatically
create the following variables:

_block_t, _block_s, _atypes1, _atypes2, _block_c.

The user should be able to examine these Magellan-created variables,
modify them as appropriate, and then perhaps re-generate the set
of features.

7 Running the JVM to Execute the Features

Magellan provides a set of built-in similarity functions and
tokenizers. A user can use them to create new features or complex
similarity functions and tokenizers.

All the built-in similarity functions except Levenshtein and Jaccard
are written in Java. So the user should start the JVM before he/she
can execute a command that involves built-in similarity functions
written in Java.

For example, if the user wants to apply a rule-based blocker over two
input tables, then he/she would do the following:

1. Create rule-based blocker object.

2. Add some rules that involve features written using built-in

similarity functions (and tokenizers).

3. Execute the rules over input tables.

In this case, if the features involve similarity functions written in Java then the user should start the JVM before executing the rules (i.e., before the third step). If not, the built-in similarity functions would fail to execute and dump core.

It is enough to start the JVM once per IPython session (i.e. The user need not start the JVM again if there are subsequent commands in the same session that involve built-in similarity functions written in Java.)

In Magellan, 'init_jvm' command should be used to start the JVM and its function signature is given below:

```
Magellan.init_jvm(jvm_file_path=None)
```

The command would return True if starting the JVM was successful, else it would raise an error.

Here jvm_file_path is the path to JVM file. If it is not specified (or set to None), then Magellan would make the best effort to find the JVM file path and start the JVM. If it is specified, then the given path would be used to start the JVM.

Examples of using the init_jvm command:

```
mg.init_jvm()
```

```
mg.init_jvm(
'/scratch/pradap/local/share/jdk1.7.0_25/jre/lib/amd64/server/libjvm.so'
)
```

Starting the JVM can be tricky. We recommend the following steps to start the JVM:

1. If the user does not know the JVM file path, then do not pass any argument to the init_jvm command. If the command returns True, then move on with the other steps in EM workflow, else proceed to step 3.

2. If the user knows the JVM file path, then give it as an argument to the init_jvm command. If the command returns True, then move on with the other steps in EM workflow, else proceed to step 3.

3. Go to the Java install directory, and search for the JVM file. The JVM file name would vary depending on the platform. In Linux, it is

libjvm.so, in Mac OSX, it is libjvm.dylib and in Windows, it is jvm.dll. Once the file is found, copy the whole file path and give it as an argument to the init_jvm command.

8 Debugging Blocking

In a typical entity matching workflow, a user loads in two tables to match and often uses a blocker to remove obvious non-matches. But it is often not clear whether the blocker drops only non-matches or it also removes a lot of potential matches.

In such cases, it is important to debug the output of blocker. In Magellan, debug_blocker command can be used for that purpose. debug_blocker command takes in two input tables A, B, blocker output C and returns a table D containing a set of tuple pairs that are potential matches and yet are not present in the blocker output C. Table D also contains similarity measure computed for each reported tuple pair (as its second column).

The user can examine these potential matches in table D. If the user finds that many of them are indeed true matches, then that means the blocker may have removed too many true matches. In this case the user may want to ''relax'' the blocker by modifying its parameters, or choose a different blocker. On the other hand, if the user does not find many true matches in table D, then it could be the case that the blocker has done a good job and preserve all the matches (or most of the matches) in the blocker output C.

The function signature for debug_blocker is:

```
debug_blocker(A, B, C, attr_corres=None, output_size=200)
```

Here 'attr_corres' is a list of attribute correspondences between A and B. If it is not specified (or set to None), then attribute correspondences will be a list of attribute pairs with the exact same names in A and B. 'output_size' is the number of tuple pairs to be included in output table, by default this value is set to 200.

The debugger will use only the attributes mentioned in these attribute correspondences to try to find potentially matching pairs and place those pairs into D. Thus, our recommendation is that (a) if the tables have idential schemas or share a lot of attributes with the same names, then set attr_corres=None, in this case the debugger will use all the attributes with the same name between the two schemas, (b) otherwise think about what attribute pairs you want to see the debugger use, then specify those in attr_corres.

```
Examples of calling debug_blocker:

ob = OverlapBlocker()
C = ob.block_tables(A, B, l_block_attr=title, r_block_attr=title,
                    overlap_size=3)
corres = [('ID','ssn'), ('name', 'ename'), ('address', 'location'),
          ('zipcode', 'zipcode')]
D = debug_blocker(A, B, C, attr_corres=corres, output_size=100)
```

9 Suggestions on the Methodology

We recommend that you first load the two tables A and B. Next, you should browse the tables to obtain some understanding about the data. You can edit the tables using an outside editor, then load the tables again into Magellan.

Next you can move to the blocking step. Here's a suggestion of the things to try:

- First try a built-in blocker: either attribute equivalence or overlap blocker, or both. These blockers have been optimized. They run relatively fast, and they can significantly reduce the number of tuple pairs to be considered.

- Next, try a rule-based blocker (or several rule-based blockers one after another). Right now, we have no guidance yet for how to write the rules. But you can experiment around. For example, if you think that for two book tuples to match, the titles should be fairly similar, then you can write a rule where you compute the jaccard similarity score, say, over the 3gram tokenizations of the titles. The rule can then state that if this similarity score is less then 0.7 threshold, then return true (thus discarding the tuple pair). You can experiment with different thresholds.

  Note that to do this, you have to first generate a set of features. You can do this automatically or manually, as described earlier. (Do not forget to start the JVM before trying to execute the rule-based blockers; see Section 7.)

- If you have exhausted trying out rule-based blocking, and still want to try something else, you can try blackbox blockers.

- In the worst case scenario, you can try to do blocking outside Magellan, and then import the output table into Magellan. If you decide to do this, you have to justify why none of the above methods works for you.

- Magellan currently has limited ability to do debugging. It comes with a relatively simple debugger, as described in Section 8. You can try to use it to debug your blocking methods.

  If your blocker takes as input tables A and B, and produces a table C of tuple pairs, then when calling the debugger on A, B, and C, it will produce a table D that lists the tuple pairs that the debugger thinks are likely to match BUT are not in C. In other words, the debugger list what it thinks are true positives that have been killed by your blocker. If these are indeed true positives, you can get a hint on how to relax your blocker to accommodate these.

10 Misc Issues

10.1 CSV Format

We select CSV format because it's well known and can be read by numerous external programs. Further, it can be easily inspected and edited by humans. Currently we handle two CSV formats: one with attribute names in the first line, and one without. An example of a CSV file with attribute names is shown below:

```
ID, name, birth_year, hourly_wage, zipcode
a1, Kevin Smith, 1989, 30, 94107
a2, Michael Franklin, 1988, 27.5, 94122
a3, William Bridge, 19886, 32
```

An example of a CSV file with out attribute names is shown below:

```
a1, Kevin Smith, 1989, 30, 94107
a2, Michael Franklin, 1988, 27.5, 94122
a3, William Bridge, 19886, 32
```

Adding Comments to CSV Files: When writing an MTable into a CSV file, we will want to store its properties (that is, key-value pairs) somewhere. For now, we will store these in comment fields of a CSV file. We assume it will look like this:

```
# key=ssn
# ltable=POINTER
# rtable=POINTER
# foreign_key_ltable=ltable.ssn
# foreign_key_rtable=rtable.eid
```

Here, "#" starts a comment line. Each comment line stores a key-value pair. A pair such as "ltable=POINTER" states that the key "ltable" is a pointer pointing to an external object (this external object is not saved).

10.2 Writing a Magellan MTable to Disk

For now we consider writing a regular MTable or a virtual MTable to disk. Later we will consider how to write other objects to disk (such as blocker, matcher).

To write an MTable to disk (regular or virtual), we first write the list of its attributes into the first line of the CSV file. Next, we write the list of key-value pairs (representing its properties) into comment lines. Finally, we write the data frame of the table to file in comma separated format. An example:

```
# key=ssn
# ltable=POINTER
# rtable=POINTER
# foreign_key_ltable=ltable.ssn
# foreign_key_rtable=rtable.eid
ID, name, birth_year, hourly_wage, zipcode
a1, Kevin Smith, 1989, 30, 94107
a2, Michael Franklin, 1988, 27.5, 94122
a3, William Bridge, 19886, 32
```

We do this using to_csv command. Its function signature:

```
MTable.to_csv(args_to_pandas_dataframe_to_csv)

Parameters:
 args_to_pandas_dataframe_to_csv: same as to\_csv command in pandas dataframe
  ref:http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to\_csv.html
```

Example 3. *The user might want to save the table_A back to disk as "table_updated.csv", he can execute the following command for that purpose*

```
A.to_csv('table_updated.csv')
```

10.3 Reading a Magellan MTable from Disk

Again, for now we will consider reading MTables. We consider other types of objects (eg blocker, matcher) later. Suppose an MTable X has been saved on disk in csv format, then we can use read_csv to read it into Magellan. The format of this command is:

```
magellan.read_csv(args_for_pandas_read_csv, list of key-value pairs for metadata)

Paramaters:
 key : string, defaults to None. If present, contains key attribute name
 args_for_pandas_read_csv :
  ref: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html
Returns:
 result : MTable
```

This command will first read the comma-separated values into a panda data frame. Next, it sets up the meta data of the table in two ways: (1) it first parses the comment lines of the csv file (if any) to extract key-value pairs and add those to the table as metadata. (2) Then it looks at the list of key-value pairs being supplied in read_csv argument line and add those to the table as metadata. So if a key was mentioned in the csv file and also in the argument list of read_csv, then the one in the argument list will override the one in the file.

As the default, python will assume the csv file has the attribute names. If they are not there, we will have to tell python read_csv command.

Each Table Must Have a Key: When a table X is created in Magellan, it must have a key (ie a column in the table serving as the key). When table X is created by reading from a file, the key can be obtained from the comment lines in the file, or can be supplied as an argument in read_csv.

If the specified column is not a key eg containing missing or duplicate data, then the read will fail and no table will be created.

Otherwise, the table X has no key. In this case a new column named "_id" (prefix of "_" to indicate that it is a generated column) is automatically generated, added to the input table as the first attribute and set as key.

Example 4. *Suppose the user wants to match two tables as in earlier example their corresponding datasets can be immported into Magellan as shown below*

```
A = mg.read_csv('../magellan/data/table_A.csv', key='ID')
B = mg.read_csv('../magellan/data/table_B.csv', key='ID')
```

10.4 Writing/Reading Other Types of Magellan Objects

After creating a blocker or feature table, it is desirable to have a way to persist the objects to disk for future use. Magellan provides two commands for that purpose: ''save_object'' and ''load_object''.

''save_object'' persists a python object to disk. The python object can be (but not limited to) blocker or feature table.

The function signature of ‘‘save_object’’ is

save_object(object, file_path)

Parameters

object : Python object. It can be Magellan objects such as rule-based blocker or feature table.

file_path : String, file path to persist the object

Returns

status : boolean, returns True if the command executes successfully

Note

save_object should not be used to persist MTable, as the command will not persist metadata. It is recommended to use ‘‘to_csv’’ in MTable for persistence purposes.

Example for save_object

```
feature_table = get_features_for_blocking(A, B)
rb = RuleBasedBlocker()
rb.add_rule([name_name_lev(ltuple, rtuple) < 0.4], feature_table)
save_object(rb, rule_based_blocker.pkl)
```

‘‘load_object’’ loads a python object from disk.

The function signature of ‘‘load_object’’ is

load_object(file_path)

Parameters
file_path : String, file path to load an object from

Returns
result : Python object

Example for load_object
rb_copy = load_object(rule_based_blocker.pkl)

10.5 Managing Missing Values

+ missing values cause problems because each function manages them in a different way.

```
+ for now, we are doing nothing. User needs to be aware of this.
```