Magellan: Toward Building Entity Matching Management Systems over Data Science Stacks

Pradap Konda[1], Sanjib Das[1], Paul Suganthan G.C.[1], AnHai Doan[1],
Adel Ardalan[1], Jeffrey R. Ballard[1], Han Li[1], Fatemah Panahi[1], Haojun Zhang[1],
Jeff Naughton[1], Shishir Prasad[2], Ganesh Krishnan[2], Rohit Deep[2], Vijay Raghavendra[2]
[1]University of Wisconsin-Madison, [2]@WalmartLabs

ABSTRACT

Entity matching (EM) has been a long-standing challenge in data management. Most current EM works, however, focus only on developing matching algorithms. We argue that far more efforts should be devoted to building EM systems. We discuss the limitations of current EM systems, then present Magellan, a new kind of EM systems that addresses these limitations. Magellan is novel in four important aspects. (1) It provides a how-to guide that tells users what to do in each EM scenario, step by step. (2) It provides tools to help users do these steps; the tools seek to cover the entire EM pipeline, not just matching and blocking as current EM systems do. (3) Tools are built on top of the data science stacks in Python, allowing Magellan to borrow a rich set of capabilities in data cleaning, IE, visualization, learning, etc. (4) Magellan provide a powerful scripting environment to facilitate interactive experimentation and allow users to quickly write code to "patch" the system. We have extensively evaluated Magellan with 44 students and users at various organizations. In this paper we propose demonstration scenarios that show the promise of the Magellan approach.

1. INTRODUCTION

Entity matching (EM) identifies data instances that refer to the same real-world entity (e.g., [David Smith, UW-Madison] and [D. M. Smith, UWM]). This problem has long been an important challenge in data management [2, 4]. Much of the current EM work however has focused only on developing *matching algorithms* [2, 4]. Relatively little work has examined building *EM systems* (that employ these algorithms).

Going forward, we believe that building EM systems is truly critical for advancing the field. EM is engineering by nature. We cannot just keep developing matching algorithms in a vacuum. This is akin to continuing to develop join algorithms without having the rest of the RDBMSs. At some point we must build end-to-end systems to evaluate matching algorithms, to integrate research and development efforts, and to make practical impacts.

In this aspect, EM can take inspiration from RDBMSs and Big Data systems. Pioneering systems such as System R, Ingres, and Hadoop have really helped drive these fields forward, by helping to evaluate research ideas, providing an architectural blueprint for the entire community to focus on, facilitating more advanced systems, and making widespread practical impacts.

The question then is what kinds of EM systems we should build, and how? As of early 2016 we counted 18 major non-commercial systems (e.g., D-Dupe, DuDe, Febrl, TAILOR, Dedoop, Nadeef) and 15 commercial ones (e.g., Tamr, Data Ladder, IBM InfoSphere BigMatch) [2]. Examining these systems reveals four major problems that we believe prevent them from being used widely in practice.

First, when performing EM users often must execute many steps, e.g., blocking, matching, exploration, cleaning, extraction (IE), debugging, sampling, labeling, estimating EM accuracy, etc. Current systems provide support for only a few steps in this pipeline (mostly blocking and matching), while ignoring less well-known yet equally critical steps (e.g., debugging, sampling).

Second, it is very difficult to add sufficient capabilities for wide-ranging EM needs to current EM systems. Practical EM often requires a wide range of techniques, e.g., learning, mining, visualization, data cleaning, IE, crowdsourcing, etc. Incorporating all such techniques into a single EM system is extremely difficult. EM is often an iterative process. So the alternate solution of moving data repeatedly among an EM system, a data cleaning system, an IE system, etc. does not work either, as it is tedious and time consuming. A major problem here is that most current EM systems are stand-alone monoliths that are not designed from the scratch to "play well" with other systems.

Third, users often have to write code to "patch" the system: either to implement a lacking functionality (e.g., to extract product weights) or to combine system components. Ideally such coding should be done using a script language in an interactive environment, to enable rapid prototyping and iteration. Most current EM systems however do not provide such facilities.

Finally, in many EM scenarios users often do not know what steps to take. For example, suppose a user wants to perform EM with at least 90% precision and as high recall as possible. How should the user start? Should he or she use

Proceedings of the VLDB Endowment, Vol. 9, No. 13

learning-based matchers first? If so, which matcher? Current EM systems provide no guides for such EM scenarios.

The Magellan Solution: To address the above four limitations, we have developed Magellan[1], a new kind of EM systems. Magellan is described in depth in [7]. So in this paper we will describe it only briefly, focusing instead on the demonstration scenarios. Magellan is novel in several important aspects.

First, Magellan provides how-to guides that tell users what to do in each EM scenario, step by step. Second, it provides tools that help users do these steps. These tools seek to cover the entire EM pipeline (e.g., debugging), not just the matching and blocking steps.

Third, Magellan is built on top of the Python data science stacks, i.e., the data analysis and Big Data stacks for data science tasks. Specifically, we propose that users solve an EM scenario in two stages. In the development stage users find an accurate EM workflow using data samples. Then in the production stage users execute this workflow on the entirety of data. We observe that the development stage basically performs data analysis. So we propose to develop tools for this stage on top of the well-known Python data analysis stack, which provide a rich set of tools such as pandas, scikit-learn, matplotlib, etc. Similarly, we propose to develop tools for the production stage on top of the Python Big Data stack (e.g., Pydoop, mrjob, PySpark, etc.).

Finally, an added benefit of integration with Python is that Magellan has access to a well-known script language and an interactive environment that users can use to prototype code to "patch" the system.

The above novelties significantly advance the state of the art in developing EM systems. Realizing them however raises significant challenges, as we discuss in depth in [7]. First, it turns out that developing effective how-to guide, even for very simple well-known EM scenarios such as applying supervised learning to match, is already quite difficult and complex.

Second, developing tools to support these guides is equally difficult. In particular, current EM work may have dismissed many steps in the EM pipeline as trivial or engineering. But we have found that many such steps (e.g., loading the data, sampling and labeling, etc.) do raise difficult research challenges.

Finally, while most current EM systems are stand-alone monoliths, Magellan is designed to be placed within an "eco-system" and is expected to "play well" with others (e.g., Python packages). We distinguish this by saying that current EM systems are "closed-world systems" whereas Magellan is designed to be an "open-world system", because it needs many other systems in the eco-system in order to provide the fullest amount of support to the user doing EM. It turns out that building open-world systems raises non-trivial challenges, such as designing the right data structures and managing metadata [7].

We have taken the first steps in addressing the above challenges, and developed the first version of Magellan. We have deployed Magellan in several real-world applications (e.g., at WalmartLabs, Johnson Control Inc, Marshfield Clinic, and in biomedicine), as well as in data science classes at UW-Madison, for extensive evaluation (see [7]). We plan to open source and release Magellan 0.1 by June 2016, to serve research, development, and practical uses.

[1]The system is named after Ferdinand Magellan, who led the first end-to-end exploration of the globe.

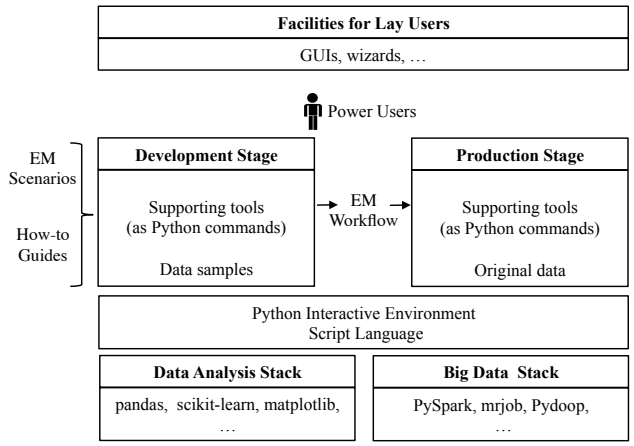


Figure 1: The Magellan architecture.

Related Work: Some works have discussed desirable properties for EM systems, e.g., being extensible and easy-to-deploy [3], being flexible and open source [1], and the ability to construct complex EM workflow consisting of distinct phases, each requiring a specific technique depending on the given application and data requirements [5]. These works however do not discuss the need for covering the entire EM pipeline, how-to guides, building on top of data analysis and Big Data stacks, and open-world systems, as we do in Magellan. The notion of "open world" has been discussed in [6], but in the context of crowd workers being able to manipulate data inside an RDBMS. Here we discuss a related but different notion of open-world systems, which refer to systems that often interact with and manipulate each other's data.

2. THE MAGELLAN ARCHITECTURE

Figure 1 shows the Magellan architecture. We build Magellan to handle a few common EM scenarios, and then extend it to more scenarios over time. The current Magellan considers the three scenarios that match two given relational tables A and B using (1) supervised learning, (2) rules, and (3) learning plus rules, respectively.

For each EM scenario Magellan provides a how-to guide. The guide proposes that the user solve the scenario in two stages: development and production.

In the development stage, the user seeks to develop a good EM workflow (e.g., one with high matching accuracy), using data samples. We observe that what users try to do in this stage is very similar in nature to data analysis tasks, which analyze data to discover insights. For example, creating EM rules can be viewed as analyzing the data to discover accurate EM rules. As a result, if we are to develop tools for this stage in isolation, within a stand-alone system, as current work has done, we would need to somehow provide a powerful data analysis environment, in order for these tools to be effective. This is clearly very difficult to do.

So instead, we propose that tools for the development stage be developed on top of an open-source data analysis stack, so that they can take full advantage of all the data analysis tools already (or will be) available in that stack. In particular, two major data analysis stacks have recently been developed, based on R and Python (new stacks such as

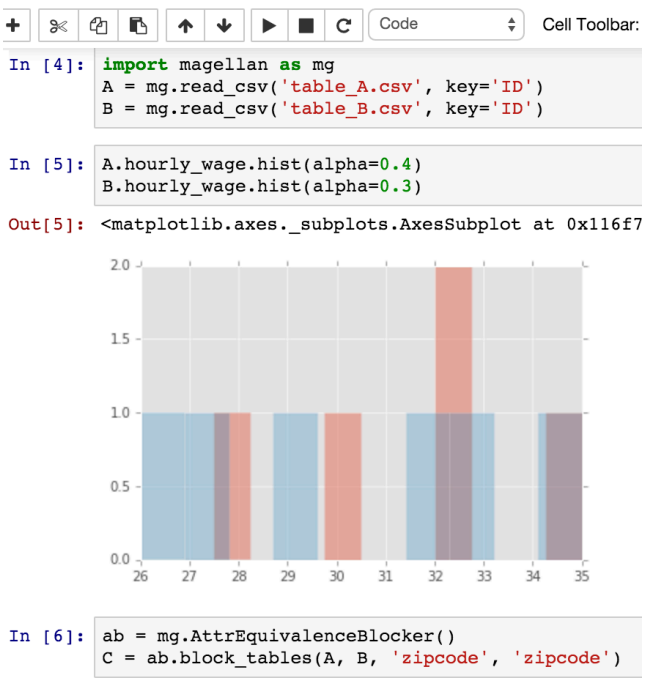


Figure 2: Magellan console in interactive IPython.

the Berkeley Data Analysis Stack are also being proposed). The Python stack for example includes the general-purpose Python language, numpy and scipy packages for numerical/array computing, pandas for relational data management, scikit-learn for machine learning, among others. More tools are being added all the time, in the form of Python packages. Magellan currently targets the Python data analysis stack.

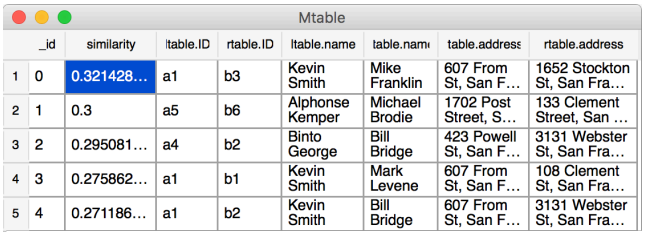
In the production stage the user applies the workflow created in the development stage to the entirety of data. Since this data is often large, a major concern here is to scale up the workflow. So we propose that tools for the production stage be developed on top of a Big Data stack. Magellan currently targets the Python Big Data stack, which consists of many software packages to run MapReduce (e.g., Pydoop, mrjob), Spark (e.g., PySpark), and parallel and distributed computing in general (e.g., pp, dispy).

The how-to guide tells the user what to do, step by step, in each stage. For each step the user can use a set of supporting tools, each of which is in turn a set of Python commands. Both stages have access to the Python script language and interactive environment (e.g., iPython). As described, Magellan is an "open-world" system, as it often has to borrow functionalities (e.g., cleaning, extraction, visualization) from other Python packages on these stacks.

Finally, the current Magellan is geared toward power users (who can program). We envision that in the future facilities for lay users (e.g., GUIs, wizards) can be laid on top (see Figure 1), and lay user actions can be translated into sequences of commands in the underlying Magellan.

3. DEMONSTRATION OVERVIEW

We now describe the proposed demonstration. We use real-world data sets to show the following features of Magellan: (1) how users can use the how-to guide to develop an end-to-end EM workflow in an interactive fashion, (2) how users can seamlessly use other data analysis packages in the



	_id	similarity	ltable.ID	rtable.ID	ltable.name	table.nam	table.address	rtable.address
1	0	0.321428...	a1	b3	Kevin Smith	Mike Franklin	607 From St, San F...	1652 Stockton St, San Fra...
2	1	0.3	a5	b6	Alphonse Kemper	Michael Brodie	1702 Post Street, S...	133 Clement Street, San ...
3	2	0.295081...	a4	b2	Binto George	Bill Bridge	423 Powell St, San F...	3131 Webster St, San Fra...
4	3	0.275862...	a1	b1	Kevin Smith	Mark Levene	607 From St, San F...	108 Clement St, San Fra...
5	4	0.271186...	a1	b2	Kevin Smith	Bill Bridge	607 From St, San F...	3131 Webster St, San Fra...

Figure 3: The GUI of the blocking debugger.

Python eco-system in developing the workflow, and (3) how users can easily extend Magellan with code to incorporate additional functionalities. We will focus on the EM scenario of matching two relational tables using supervised learning and rules.

In practice, developing an end-to-end EM workflow often takes hours or days. So a large part of our demonstration (e.g., labeling a few hundred tuple pairs) will be "canned" scenarios. But we will provide opportunities for the audience to interact "live" with the system, and to take the demo "off the rails".

3.1 Using the Guide to Develop a Workflow

Here we show that a user can use Magellan to interactively develop an end-to-end EM workflow, using the IPython console shown in Figure 2. Specifically, the user will perform the following steps.

Loading and Downsampling the Tables: First, the user will load the two tables into memory using Magellan. These tables are large, so the user will downsample the tables to a reasonable size, then use the smaller tables for the rest of the EM workflow development.

Exploring and Cleaning the Tables: Next, the user will explore the tables, plot statistics, identify outliers, impute missing values, and clean the attribute values. The user will use visualization and data analysis packages in Python such as matplotlib, plotly, and pandas.

Blocking to Create Candidate Tuple Pairs: Next, the user will iteratively create a blocking pipeline and reduce the number of tuple pairs considered for matching. This involves three steps: (1) selecting the best blocker, (2) debugging blockers, and (3) knowing when to stop modifying the blockers. To select the best blocker, the user will try overlap blocking first, then attribute equivalence blocking, then other well-known blocking methods (e.g., hash, canopy clustering) if appropriate. Finally, the user can try rule-based blocking. Note that this means the user can use multiple blockers and combine them in a flexible fashion.

Next, the user will use the blocking debugger to check whether the current sequence of blockers is removing too many matches. The debugger console is shown in Figure 3, which displays a list of tuple pairs that are potential matches but missed by the current blocker pipeline. The user will use the debugging output to decide when to stop tuning the blockers. Let the candidate set of tuple pairs obtained by blocking in this step be C.

Sampling and Labeling Tuple Pairs: Next, the user will sample and label from C, for supervised learning purposes. If there are few true matches in C, then random sampling does not work, as the sample may end up containing very few matches. To address this problem, Magel-

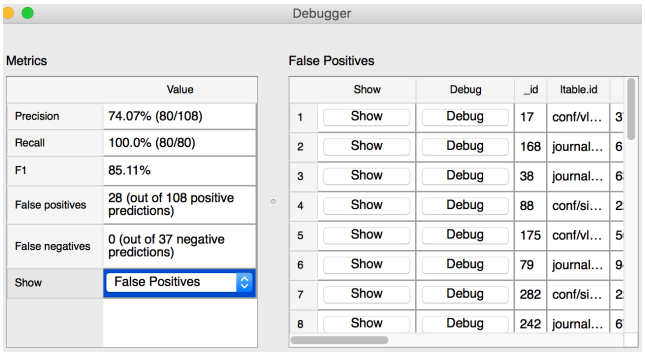


Figure 4: The GUI of the matching debugger.

lan's how-to guide proposes the following. Suppose the user wants to sample and label a set S of size n, first the user will take a random sample S_1 of size k from C, where k is a small number (e.g., 50) and label S_1. If there are enough matches in S_1 then the user will conclude that the "density" of matches is high, and just randomly sample $n - k$ more pairs from C. Otherwise, the user will re-do the blocking step, thereby increasing the density of matches in C. After blocking, the user will take another random sample S_2, also of size k from C and label S_2. If there are enough matches in S_2 then the user will conclude that the "density" of matches is high, and just randomly sample $n - 2k$ more pairs from C. Otherwise, the user will repeat the process until there are enough matches in the labeled set.

Selecting the Best Matcher: Next, the user will select the best matcher in two steps: (a) selecting the best learning-based matcher X, and (b) adding rules on top of X. The user will first split the labeled data set into a development set I and an evaluation set J. The user will use I to select the best matcher and finally estimate the matcher's accuracy using J .

(a) Selecting the Best Learning-Based Matcher: First, the user will automatically create a set F of all possible features between the sampled input tables. Next, the user uses F to convert the development set I into a set of feature vectors H. The user then uses H to cross-validate all the learning-based matchers available in Magellan. The user will examine the results to select the matcher X with the highest accuracy.

Next the user will debug X in three steps: (1) identify and understand the mistakes made by X, (2) categorize these mistakes, and (3) take actions to fix common categories of mistakes. To identify and understand the mistakes made by X, the user will first split the development set H into two sets P and Q. The user will use the matching debugger to debug X, using P and Q (internally the debugger will use P to train X and apply it over Q). The user will see a debugging GUI, as shown in Figure 4. He or she will examine the false positives and false negatives, and categorize the errors into four types: (1) errors in the attribute values, (2) labeling errors, (3) issues with the feature set, and (4) issues with the parameters of X. The user will then fix these problems using Magellan and select the best learning-based matcher again, until he or she is out of ideas on what else to do to improve the matcher.

(b) Adding Rules to the Learning-Based Matcher: Next, the user adds rules to "patch" X. He or she will use the matching debugger to understand the mistakes made by X, and use Magellan to write a set of positive and negative rules to handle these mistakes. Now the new matcher Y will be the learning-based matcher followed by an ordered set of rules. The user will evaluate the accuracy of Y using cross-validation and will iterate on modifying the rules, until he or she is out of time or ideas to improve Y.

Estimating the Accuracy of the Best Matcher: Finally, the user will use the evaluation set to compute and return an estimation of the matching accuracy of the learning-based matcher augmented with rules.

3.2 "Playing Well" with Other Systems

In this part of the demo, we will show that Magellan can readily interoperate with other data analysis packages in Python. Specifically, we will demonstrate that Magellan can use scientific, visualization, and string matching packages in Python to build an EM workflow. The user will use popular visualization packages such as plotly and matplotlib in Python to explore the data set, and use string matching packages such as fuzzywuzzy to create custom features to train and select learning-based matchers.

3.3 "Patching" the System with Code

In this part of the demo, we will show that Magellan can be extended easily to incorporate additional functionalities. Specifically, we will show that the user will be able to easily add a new blocker such as sorted-neighborhood blocker (for which the user has written the code) and add it seamlessly to the blocking pipeline.

4. CONCLUSIONS

We argue that far more efforts should be devoted to building EM systems, to significantly advance the field. Toward this goal, in this demonstration we will present Magellan, a new kind of EM systems, which is novel in several important aspects: how-to guides, tools to support the entire EM pipeline, tight integration with the PyData eco-system, open world vs. closed world systems, and easy access to an interactive script environment. We focus on showing (1) how users can use the how-to guide to develop an end-to-end EM workflow, (2) how Magellan can seamlessly use other data analysis packages in the Python eco-system, and (3) how users can easily extend Magellan with code to incorporate additional functionalities.

5. REFERENCES

[1] P. Christen. Febrl: A freely available record linkage system with a graphical user interface. HDKM, 2008.
[2] P. Christen. *Data Matching*. Springer, 2012.
[3] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: A commodity data cleaning system. SIGMOD, 2013.
[4] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.
[5] M. Fortini, M. Scannapieco, L. Tosco, and T. Tuoto. Towards an open source toolkit for building record linkage workflows. In *In Proc. of the SIGMOD Workshop on Information Quality in Information Systems*, 2006.
[6] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, 2011.
[7] P. Konda et al. Magellan: Toward building entity matching management systems. In *UW-Madison Technical Report*, 2016.