Magellan: Toward Building Entity Matching Management Systems

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[3], Ganesh Krishnan[2], Rohit Deep[2], Vijay Raghavendra[2]
[1]University of Wisconsin-Madison, [2]@WalmartLabs, [3]Instacart

ABSTRACT

Entity matching (EM) has been a long-standing challenge in data management. Most current EM works 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 as a solution Magellan, a new kind of EM systems. Magellan is novel in four important aspects. (1) It provides how-to guides that tell 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 analysis and Big Data stacks in Python, allowing Magellan to borrow a rich set of capabilities in data cleaning, IE, visualization, learning, etc. (4) Magellan provides a powerful scripting environment to facilitate interactive experimentation and quick "patching" of the system. We describe research challenges raised by Magellan, then present extensive experiments with 44 students and users at several organizations that show the promise of the Magellan approach.

1. INTRODUCTION

Entity matching (EM) identifies data instances that refer to the same real-world entity, such as (David Smith, UW-Madison) and (D. M. Smith, UWM). This problem has been a long-standing challenge in data management [11, 16]. Most current EM works however has focused only on developing *matching algorithms* [11, 16].

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.

Proceedings of the VLDB Endowment, Vol. 9, No. 12

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 push 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 real-world impacts.

The question then is what kinds of EM systems we should build, and how? In this paper we begin by showing that current EM systems suffer from four limitations that prevent them from being used extensively in practice.

First, when performing EM users often must execute many steps, e.g., blocking, matching, exploring, cleaning, debugging, sampling, labeling, estimating accuracy, etc. Current systems however do not cover the entire EM pipeline, providing support for only a few steps (e.g., blocking, matching), while ignoring less well-known yet equally critical steps (e.g., debugging, sampling).

Second, EM steps often exploit many techniques, e.g., learning, mining, visualization, outlier detection, information extraction (IE), crowdsourcing, etc. Today however it is very difficult to exploit a wide range of such techniques. 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 standalone 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., extracting product weights) or to glue together 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. Suppose a user wants to perform EM with at least 95% precision and 80% recall. Should he or she use a learning-based EM approach, a rule-based approach, or both? If learning-based, then which technique to select among the many existing ones (e.g., decision tree, SVM, etc.)? How to debug the selected technique? What to do if after many tries the user still cannot reach 80% recall with a learning-based approach? Current EM systems provide no answers to such questions.

The Magellan Solution: To address these limitations, we describe Magellan, a new kind of EM systems currently being developed at UW-Madison, in collaboration with WalmartLabs. Magellan (named after Ferdinand Magellan, the first end-to-end explorer of the globe) 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, Magellan provides tools that help users do these steps. These tools seek to cover the entire EM pipeline (e.g., debugging, sampling), not just the matching and blocking steps.

Third, the tools are being built on top of the Python data analysis and Big Data stacks. 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 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 develop tools for the production stage on top of the Python Big Data stack (e.g., Pydoop, mrjob, PySpark, etc.).

Thus, Magellan is well integrated with the Python data eco-system, allowing users to easily exploit a wide range of techniques in learning, mining, visualization, IE, etc.

Finally, an added benefit of integration with Python is that Magellan is situated in a powerful interactive scripting environment that users can use to prototype code to "patch" the system.

Challenges: Realizing the above novelties raises major challenges. First, it turns out that developing effective how-to guides, even for very simple EM scenarios such as applying supervised learning to match, is already quite difficult and complex, as we will show in Section 4.

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 engineering. But here we show that many such steps (e.g., loading the data, sampling and labeling, debugging, 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., other Python packages). We distinguish this by saying that current EM systems are "closed-world systems" whereas Magellan is an "open-world system", because it relies on 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, as we discuss in Section 5.

In this paper we have taken the first steps in addressing the above challenges. We have also built and evaluated Magellan 0.1 in several real-world settings (e.g., at WalmartLabs, Johnson Control Inc., Marshfield Clinic) and in data science classes at UW-Madison. In summary, we make the following contributions:

- We argue that far more efforts should be devoted to building EM systems, to significantly advance the field.
- We discuss four limitations that prevent current EM systems from being used extensively in practice.
- We describe the Magellan system, which is novel in several important aspects: how-to guides, tools to support all steps of the EM pipeline, tight integration with the Python data eco-system, easy access to an interactive scripting environment, and open world vs. closed world systems.
- We describe significant challenges in realizing Magellan, including the novel challenge of designing open-world systems (that operate in an eco-system).
- We describe extensive experiments with 44 students and real users at various organizations that show the utility of Magellan, including improving the accuracy of an EM system in production.

This paper describes the most important aspects of Magellan, deferring details to [22]. Magellan will be released at sites.google.com/site/anhaidgroup/projects/magellan in Summer 2016, to serve research, development, and practical uses. Finally, the ideas underlying Magellan can potentially be applied to other types of DI problems (e.g., IE, schema matching, data cleaning, etc.), and an effort has been started to explore this direction and to foster an eco-system of open-source DI tools (see Magellan's website).

2. THE CASE FOR ENTITY MATCHING MANAGEMENT SYSTEMS

2.1 Entity Matching

This problem, also known as record linkage, data matching, etc., has received much attention in the past few decades [11, 16]. A common EM scenario finds all tuple pairs (a, b) that match, i.e., refer to the same real-world entity, between two tables A and B (see Figure 1). Other EM scenarios include matching tuples within a single table, matching into a knowledge base, matching XML data, etc. [11].

Most EM works have developed matching algorithms, exploiting rules, learning, clustering, crowdsourcing, among others [11, 16]. The focus is on improving the matching accuracy and reducing costs (e.g., run time). Trying to match all pairs in $A \times B$ often takes very long. So users often employ heuristics to remove obviously non-matched pairs (e.g., products with different colors), in a step called *blocking*, before matching the remaining pairs. Several works have studied this step, focusing on scaling it up to large amounts of data (see Section 7).

2.2 Current Entity Matching Systems

In contrast to the extensive effort on matching algorithms (e.g., 96 papers were published on this topic in 2009-2014 alone, in SIGMOD, VLDB, ICDE, KDD, and WWW), there has been relatively little work on building EM systems. As of 2016 we counted 18 major non-commercial systems (e.g., D-Dupe, DuDe, Febrl, Dedoop, Nadeef), and 15 major commercial ones (e.g., Tamr, Data Ladder, IBM InfoSphere) [11]. Our examination of these systems (see [22]) reveals the following four major problems:

1. Systems Do Not Cover the Entire EM Pipeline: When performing EM users often must execute many steps, e.g., blocking, matching, exploration, cleaning, extraction (IE), debugging, sampling, labeling, etc. Current systems provide support for only a few steps in this pipeline, while ignoring less well-known yet equally critical steps.

Table A

	Name	City	State
a_1	Dave Smith	Madison	WI
a_2	Joe Wilson	San Jose	CA
a_3	Dan Smith	Middleton	WI

Table B

	Name	City	State
b_1	David D. Smith	Madison	WI
b_2	Daniel W. Smith	Middleton	WI

Matches
(a_1, b_1)
(a_3, b_2)

Figure 1: An example of matching two tables.

For example, all 33 systems that we have examined provide support for blocking and matching. Twenty systems provide limited support for data exploration and cleaning. There is no meaningful support for any other steps (e.g., debugging, sampling, etc.). Even for blocking the systems merely provide a set of blockers that users can call; there is no support for selecting and debugging blockers, and for combining multiple blockers.

2. Difficult to Exploit a Wide Range of Techniques: Practical EM often requires a wide range of techniques, e.g., learning, mining, visualization, data cleaning, IE, SQL querying, crowdsourcing, keyword search, etc. For example, to improve matching accuracy, a user may want to clean the values of attribute "Publisher" in a table, or extract brand names from "Product Title", or build a histogram for "Price". The user may also want to build a matcher that uses learning, crowdsourcing, or some statistical techniques.

Current EM systems do not provide enough support for these techniques, and there is no easy way to do so. Incorporating all such techniques into a single system is extremely difficult. But the alternate solution of just moving data among a current EM system and systems that do cleaning, IE, visualization, etc. is also difficult and time consuming. A fundamental reason is that most current EM systems are stand-alone monoliths that are not designed from the scratch to "play well" with other systems. For example, many current EM systems were written in C, C++, C#, and Java, using proprietary data structures. Since EM is often iterative, we need to repeatedly move data among these EM systems and cleaning/IE/etc systems. But this requires repeated reading/writing of data to disk followed by complicated data conversion.

3. Difficult to Write Code to "Patch" the System: In practice users often have to write code, either to implement a lacking functionality (e.g., to extract product weights, or to clean the dates), or to tie together system components. It is difficult to write such code correctly in "one shot". Thus ideally such coding should be done using an interactive scripting environment, to enable rapid prototyping and iteration. This code often needs access to the rest of the system, so ideally the system should be in such an environment too. Unfortunately only 5 out of 33 systems provide such settings (using Python and R).

4. Little Guidance for Users on How to Match: In our experience this is by far the most serious problem with using current EM systems in practice. In many EM scenarios users simply do not know what to do: how to start, what to do next? Interestingly, even the simple task of taking a sample and labeling it (to train a learning-based matcher) can be quite complicated in practice, as we show in Section 4.3. Thus, it is not enough to just build a system consisting of a set of tools. It is also critical to provide step-by-step guidance to users on how to use the tools to handle a particular EM scenario. No EM system that we have examined provides such guidance.

2.3 Entity Matching Management Systems

To address the above limitations, we propose to build a new kind of EM systems. In contrast to current EM systems, which mostly provide a set of implemented matchers/blockers, these new systems are far more advanced.

First and foremost, they seek to handle a wide variety of EM scenarios. These scenarios can use very different EM workflows. So it is difficult to build a single system to handle all EM scenarios. Instead, we should build a set of systems, each handling a well-defined set of similar EM scenarios. Each system should target the following goals:

1. **How-to Guide:** Users will have to be "in the loop". So it is critical that the system provides a how-to guide that tells users what to do and how to do it.
2. **User Burden:** The system should minimize the user burden. It should provide a rich set of tools to help users easily do each EM step, and do so for all steps of the EM pipeline, not just matching and blocking. Special attention should be paid to debugging, which is critical in practice.
3. **Runtime:** The system should minimize tool runtimes and scale tools up to large amounts of data.
4. **Expandability:** It should be easy to extend the system with any existing or future techniques that can be useful for EM (e.g., cleaning, IE, learning, crowdsourcing). Users should be able to easily "patch" the system using an interactive scripting environment.

Of these goals, "expandability" deserves more discussion. If we can build a single "super-system" for EM, do we need expandability? We believe it is very difficult to build such a system. First, it would be immensely complex to build just an initial system that incorporates all of the techniques mentioned in Goal 4. Indeed, despite decades of development, today no EM system comes close to achieving this.

Second, it would be very time consuming to maintain and keep this initial system up-to-date, especially with the latest advances (e.g., crowdsourcing, deep learning).

Third, and most importantly, a generic EM system is unlikely to perform equally well for multiple domains (e.g., biomedicine, social media, payroll). Hence we often need to extend and customize it to a particular target domain, e.g., adding a data cleaning package specifically designed for biomedical data (written by biomedical researchers). For the above three reasons, we believe that EM systems should be fundamentally expandable.

Clearly, systems that target the above goals seek to *manage* all aspects of the end-to-end EM process. So we refer to this kind of systems as *entity matching management systems (EMMSs)*. Building EMMSs is difficult, long-term, and will require a new kind of architecture compared to current EM systems. In the rest of this paper we describe Magellan, an attempt to build such an EMMS.

3. THE MAGELLAN APPROACH

Figure 2 shows the Magellan architecture. The system targets a set of EM scenarios. For each EM scenario it 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). The

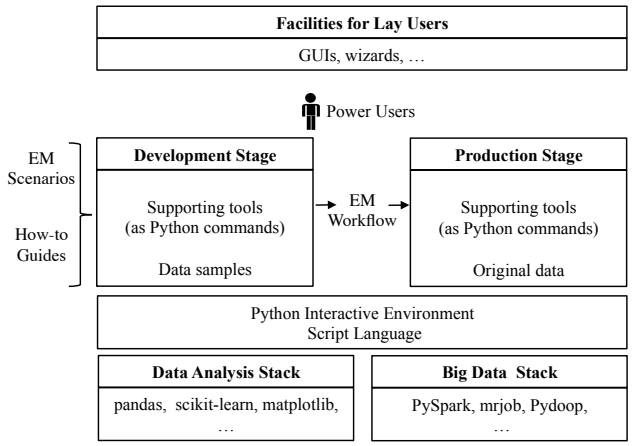

Figure 2: The Magellan architecture.

guide tells the user what to do, step by step. For each step the user can use a set of supporting tools, each of which is in turn a set of Python commands. This stage is typically done using data samples. In the production stage, the guide tells the user how to implement and execute the EM workflow on the entirety of data, again using a set of supporting tools.

Both stages have access to the Python script language and interactive environment (e.g., iPython). Further, tools for these stages are built on top of the Python data analysis stack and the Python Big Data stack, respectively. Thus, 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 2), and lay user actions can be translated into sequences of commands in the underlying Magellan.

In the rest of this section, we describe EM scenarios, workflows, and the development and production stages. Section 4 describes the how-to guides, and Section 5 describes the challenges of designing Magellan as an open-world system.

3.1 EM Scenarios and Workflows

We classify EM scenarios along four dimensions:

- **Problems:** Matching two tables; matching within a table; matching a table into a knowledge base; etc.
- **Solutions:** Using learning; using learning and rules; performing data cleaning, blocking, then matching; performing IE, then cleaning, blocking, and matching; etc.
- **Domains:** Matching two tables of biomedical data; matching e-commerce products given a large product taxonomy as background knowledge; etc.
- **Performance:** Precision must be at least 92%, while maximizing recall as much as possible; both precision and recall must be at least 80%, and run time under four hours; etc.

An EM scenario can constrain multiple dimensions, e.g., matching two tables of e-commerce products using a rule-based approach with desired precision of at least 95%.

Clearly there is a wide variety of EM scenarios. So we will build Magellan to handle a few common scenarios, and then extend it to more similar scenarios over time. Specifically, for now we will consider the three scenarios that match two given relational tables A and B using (1) supervised learning, (2) rules, and (3) learning plus rules, respectively. These scenarios are very common. In practice, users often try Scenario 1 or 2, and if neither works, then a combination of them (Scenario 3).

EM Workflows: As discussed earlier, to handle an EM scenario, a user often has to execute many steps, such as cleaning, IE, blocking, matching, etc. The combination of these steps form an *EM workflow*. Figure 5 shows a sample workflow (which we explain in detail in Section 4.6).

3.2 Development Stage vs. Production Stage

From our experience with real-world users' doing EM, we propose that the how-to guide tell the user to solve the EM scenario in two stages: *development* and *production*. In the development stage the user tries to find a good EM workflow, e.g., one with high matching accuracy. This is typically done using data samples. In the production stage the user applies the workflow to the entirety of data. Since this data is often large, a major concern here is to scale up the workflow. Other concerns include quality monitoring, logging, crash recovery, etc. The following example illustrates these two stages.

EXAMPLE 1. *Consider matching two tables A and B each having 1M tuples. Working with such large tables will be very time consuming in the development stage, especially given the iterative nature of this stage. Thus, in the development stage the user U starts by sampling two smaller tables A' and B' from A and B, respectively. Next, U performs blocking on A' and B'. The goal is to remove as many obviously non-matched tuple pairs as possible, while minimizing the number of matching pairs accidentally removed. U may need to try various blocking strategies to come up with what he or she judges to be the best.*

The blocking step can be viewed as removing tuple pairs from $A' \times B'$. Let C be the set of remaining tuple pairs. Next, U may take a sample S from C, examine S, and manually write matching rules, e.g., "If titles match and the numbers of pages match then the two books match". U may need to try out these rules on S and adjust them as necessary. The goal is to develop matching rules that are as accurate as possible.

Once U has been satisfied with the accuracy of the matching rules, the production stage begins. In this stage, U executes the EM workflow that consists of the developed blocking strategy and matching rules on the original tables A and B. To scale, U may need to rewrite the code for blocking and matching to use Hadoop or Spark. □

As described, these two stages are very different in nature: one goes for accuracy and the other goes for scaling (among others). Consequently, they will require very different sets of tools. We now discuss developing tools for these stages.

Development Stage on a Data Analysis Stack: We observe that what users try to do in the development stage is very similar in nature to data analysis tasks, which analyze data to discover insights. Indeed, creating EM rules can be viewed as analyzing (or mining) the data to discover

accurate EM rules. Conversely, to create EM rules, users also often have to perform many data analysis tasks, e.g., cleaning, visualizing, finding outliers, IE, etc.

As a result, if we are to develop tools for the development stage in isolation, within a stand-alone monolithic 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 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. By Oct 2015, there were 490 packages available in the popular Anaconda distribution. There is a vibrant community of contributors to continuously improve this stack.

For Magellan, since our initial target audience is the IT community, where we believe Python is more familiar, we have been developing tools for the development stage on the Python data analysis stack.

Production Stage on a Big Data Stack: In a similar vein, we propose that tools for the production stage, where scaling is a major focus, be developed on top of a Big Data stack. Magellan uses 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).

In the rest of this paper we will focus on the development stage, leaving the production stage for subsequent papers.

4. HOW-TO GUIDES AND TOOLS

We now discuss developing how-to guides as well as tools to support these guides. Our goal is twofold:

- First, we show that even for relatively simple EM scenarios (e.g., matching using supervised learning), a good guide can already be quite complex. Thus developing how-to guides is a major challenge, but such guides are absolutely critical in order to successfully guide the user through the EM process.
- Second, we show that each step of the guide, including those that prior work may have viewed as trivial or engineering (e.g., sampling, labeling), can raise many interesting research challenges. We provide preliminary solutions to several such challenges in this paper. But much more remains to be done.

Recall that Magellan currently targets three EM scenarios: matching two tables A and B using (1) supervised learning, (2) rules, and (3) both learning and rules. For space reasons, we will focus on Scenario 1, briefly discussing Scenarios 2-3 in Section 4.7. For Scenario 1, we further focus on the development stage.

The Current Guide for Learning-Based EM: Figure 3 shows the current guide for Scenario 1: matching using supervised learning. The figure lists only the top six steps. While each step may sound like fairly informal advice (e.g., "create a set of features"), the full guide itself (available with Magellan 0.1) is considerably more complex and actually spells out in detail what to do (e.g., run a Magellan command to automatically create the features). We developed this guide based on observing how real-world users (e.g., at WalmartLabs and Johnson Control) as well as students in several UW-Madison classes handled this scenario.

1. Load tables A and B into Magellan. Downsample if necessary.
2. Perform blocking on the tables to obtain a set of candidate tuple pairs C.
3. Take a random sample S from C and label pairs in S as matched / non-matched.
4. Create a set of features then convert S into a set of feature vectors H. Split H into a development set I and an evaluation set J.
5. Repeat until out of debugging ideas or out of time:
   (a) Perform cross validation on I to select the best matcher. Let this matcher be X.
   (b) Debug X using I. This may change the matcher X, the data, labels, and the set of features, thus changing I and J.
6. Let Y be the best matcher obtained in Step 5. Train Y on I, then apply to J and report the matching accuracy on J.

Figure 3: The top-level steps of the guide for the EM scenario of matching using supervised learning.

The guide states that to match two tables A and B, the user should load the tables into Magellan (Step 1), do blocking (Step 2), label a sample of tuple pairs (Step 3), use the sample to iteratively find and debug a learning-based matcher (Steps 4-5), then return this matcher and its estimated matching accuracy (Step 6).

We now discuss these steps, possible tools to support them, and tools that we have actually developed. Our goal is to automate each step as much as possible, and where it is not possible, then to provide detailed guidance to the user. We focus on discussing problems with current solutions, the design alternatives, and opportunities for automation (referring the reader to [22] for the technical details). For ease of exposition, we will assume that tables A and B share the same schema.

4.1 Loading and Downsampling Tables

Downsampling Tables: We begin by loading the two tables A and B into memory. If these tables are large (e.g., each having 100K+ tuples), we should sample smaller tables A' and B' from A and B respectively, then do the development stage with these smaller tables. Since this stage is iterative by nature, working with large tables can be very time consuming and frustrating to the user.

Random sampling however does not work, because tables A' and B' may end up sharing very few matches, i.e., matching tuples (especially if the number of matches between A and B is small to begin with). Thus we need a tool that samples more intelligently, to ensure a reasonable number of matches between A' and B'.

We have developed such a tool, shown as the Magellan command c_1 in Figure 4. This command first randomly selects B_size tuples from table B to be table B'. For each tuple $x \in B'$, it finds a set P of $k/2$ tuples from A that may match x (using the heuristic that if a tuple in A shares many

```
c1: down_sample_tables (A, B, B_size, k)
c2: debug_blocker (A, B, C, output_size = 200)
c3: get_features_for_matching (A, B)
c4: select_matcher (matchers, table, exclude_attrs, target_attr, k = 5)
c5: vis_debug_dt (matcher, train, test, exclude_attrs, target_attr)
```

Figure 4: Sample commands discussed in Section 4. Magellan has 53 such commands.

tokens with x, then it is more likely to match x), and a set Q of $k/2$ tuples randomly selected from $A \setminus P$. Table A' will consist of all tuples in such Ps and Qs. The idea is for A' and B' to share some matches yet be as representative of A and B as possible.

More Sophisticated Downsampling Solutions: The above command was fast and quite effective in our experiments. However it has a limitation: it may not get all important matching categories into A' and B'. If so, the EM workflow created using A' and B' may not work well on the original tables A and B.

For example, consider matching companies. Tables A and B may contain two matching categories: (1) tuples with similar company names and addresses match because they refer to the same company, and (2) tuples with similar company names but different addresses may still match because they refer to different branches of the same company. Using the above command, tables A' and B' may contain many tuple pairs of Case 1, but no or very few pairs of Case 2.

To address this problem, we are working on a better "down-sampler". Our idea is to use clustering to create groups of matching tuples, then analyze these groups to infer matching categories, then sample from the categories. Major challenges here include how to effectively cluster tuples from the large tables A and B, and how to define and infer matching categories accurately.

4.2 Blocking to Create Candidate Tuple Pairs

In the next step, we apply blocking to the two tables A' and B' to remove obviously non-matched tuple pairs. Ideally, this step should be automated (as much as possible). Toward this goal, we distinguish three cases.

(1) We already know which matcher we want to use. Then it may be possible to analyze the matcher to infer a blocker, thereby completely automating the blocking step. For example, when matching two sets of strings (a special case of EM [11]), often we already know the matcher we want to use (e.g., $jaccard(x, y) > 0.8$, i.e., predicting two strings x and y matched if their Jaccard score exceeds 0.8). Prior work [11] has analyzed such matchers to infer efficient blockers that do not remove true matches. Thus, debugging the blocker is also not necessary.

(2) We do not know yet which matcher we want to use, but we have a set T of tuple pairs labeled matched / no-matched. Then it may be possible to partially automate the blocking step. Specifically, the system can use T to learn a blocker and propose it to the user (e.g., training a random forest then extracting the negative rules of the forest as blocker candidates [20]). The user still has to debug the blocker to check that it does not accidentally remove too many true matches.

(3) We do not know yet which matcher we want to use, and we have no labeled data. This is the case considered in this paper, since all we have so far are the two tables A' and B'. In this case the user often faces three problems (which have not been addressed by current work): (a) how to select the best blocker, (b) how to debug a given blocker, and (c) how to know when to stop? Among these, the first problem is open to partial automation.

Selecting the Best Blocker: A straightforward solution is to label a set of tuple pairs (e.g., selected using active learning [20]), then use it to automatically propose a blocker, as in Case 2. To propose good blockers, however, this solution may require labeling hundreds of tuple pairs [20], incurring a sizable burden on the user.

This solution may also be unnecessarily complex. In practice, a user often can use domain knowledge to quickly propose good blockers, e.g., "matching books must share the same ISBN", in a matter of minutes. Hence, our how-to guide tries to help the user identify these "low-hanging fruits" first.

Specifically, many blocking solutions have been developed, e.g., overlap, attribute equivalence (AE), sorted neighborhood (SNB), hash-based, rule-based, etc. [11]. From our experience, we recommend that the user try successively more complex blockers, and stop when the number of the tuple pairs surviving blocking is already sufficiently small. Specifically, the user can try overlap blocking first (e.g., "matching tuples must share at least k tokens in an attribute x"), then AE (e.g., "matching tuples must share the same value for an attribute y"). These blockers are very fast, and can significantly cut down on the number of candidate tuple pairs. Next, the user can try other well-known blocking methods (e.g., SNB, hash) if appropriate. This means the user can use multiple blockers and combine them in a flexible fashion (e.g., applying AE to the output of overlap blocking).

Finally, if the user still wants to reduce the number of candidate tuple pairs further, then he or she can try rule-based blocking. It is difficult to manually come up with good blocking rules. So we will develop a tool to automatically propose rules, as in Case 2, using the technique in [20], which uses active learning to select tuple pairs for the user to label.

Debugging Blockers: Given a blocker L, how do we know if it does not remove too many matches? We have developed a debugger to answer this question, shown as command c_2 in Figure 4. Suppose applying L to A' and B' produces a set C of tuple pairs ($a \in A', b \in B'$). Then $D = A' \times B' \setminus C$ is the set of all tuple pairs removed by L.

The debugger examines D to return a list of k tuple pairs in D that are most likely to match ($k = 200$ is the default). The user U examines this list. If U finds many matches in the list, then that means blocker L has removed too many matches. U would need to modify L to be less "aggressive", then apply the debugger again. Eventually if U finds no or very few matches in the list, U can assume that L has removed no or very few matches, and thus is good enough. See [22] for a detailed description of how we developed this debugger, including solving two challenges: how can the debugger judge that a tuple pair is likely to match, and how can it quickly search D to find such pairs?

Knowing When to Stop Modifying the Blockers: How do we know when to stop tuning a blocker L? Suppose applying L to A' and B' produces the set of tuple pairs $block(L, A', B')$. The conventional wisdom is to stop when $block(L, A', B')$ fits into memory or is already small enough so that the matching step can process it efficiently.

In practice, however, this often does not work. For example, since we work with A' and B', *samples* from the original tables, monitoring $|block(L, A', B')|$ does not make sense. Instead, we want to monitor $|block(L, A, B)|$. But applying L to the large tables A and B can be very time consuming, making the iterative process of tuning L impractical. Further, in many practical scenarios (e.g., e-commerce), the data to be matched can arrive in batches, over weeks, rendering moot the question of estimating $|block(L, A, B)|$.

As a result, in many practical settings users want blockers that have (1) high pruning power, i.e., maximizing $1 - |block(L, A', B')|/|A' \times B'|$, and (2) high recall, i.e., maximizing the ratio of the number of matches in $block(L, A', B')$ divided by the number of matches in $A' \times B'$.

Users can measure the pruning power, but so far they have had no way to estimate recall. This is where our debugger comes in. In our experiments (see Section 6) users reported they had used our debugger to find matches that the blocker L had removed, and when they found no or only a few matches, they concluded that L had achieved high recall and stopped tuning the blocker.

4.3 Sampling and Labeling Tuple Pairs

Let L be the blocker we have created. Suppose applying L to tables A' and B' produces a set of tuple pairs C. In the next step, user U should take a sample S from C, then label the pairs in S as matched / no-matched, to be used later for training matchers, among others.

At a first glance, this step seems very simple: why not just take a random sample and label it? Unfortunately in practice this is far more complicated.

For example, suppose C contains relatively few matches (either because there are few matches between A' and B', or because blocking was too liberal, resulting in a large C). Then a random sample S from C may contain no or few matches. But the user U often does not recognize this until U has labeled most of the pairs in S. This is a waste of U's time and can be quite serious in cases where labeling is time consuming or requires expensive domain experts (e.g., labeling drug pairs when we worked with Marshfield Clinic). Taking another random sample does not solve the problem because it is likely to also contain no or few matches.

To address this problem, our guide builds on [20] to propose that user U sample and label in iterations. Specifically, suppose U wants a sample S of size n. In the first iteration, U takes and labels a random sample S_1 of size k from C, where k is a small number. If there are enough matches in S_1, then U can conclude that the "density" of matches in C is high, and just randomly sample $n - k$ more pairs from C.

Otherwise, the "density" of matches in C is low. So U must re-do the blocking step, perhaps by creating new blocking rules that remove more non-matching tuple pairs in C, thereby increasing the density of matches in C. After blocking, U can take another random sample S_2 also of size k from C, then label S_2. If there are enough matches in S_2, then U can conclude that the density of matches in C has become high, and just randomly sample $n - 2k$ more pairs from C, and so on.

4.4 Selecting a Matcher

Once user U has labeled a sample S, U uses S to select a good initial learning-based matcher. Today most EM systems supply the user with a set of such matchers, e.g., decision tree, Naive Bayes, SVM, etc., but do not tell the user how to select a good one.

Our guide addresses this problem. Specifically, user U first calls the command c_3 in Figure 4 to create a set of features $F = \{f_1, \ldots, f_m\}$, where each feature f_i is a function that maps a tuple pair (a, b) into a value. This command creates all possible features between the attributes of tables A' and B', using a set of heuristics. For example, if attribute *name* is textual, then the command creates feature $name\_3gram\_jac$ that returns the Jaccard score between the 3-gram sets of the two names (of tuples a and b) [22].

Next, U converts each tuple pair in the labeled set S into a feature vector (using features in F), thus converting S into a set H of feature vectors. Next, U splits H into a development set I and an evaluation set J.

Let M be the set of all learning-based matchers supplied by the EM system. Next, U uses command c_4 in Figure 4 to perform cross validation on I for all matchers in M, then examines the results to select a good matcher. Command c_4 highlights the matcher with the highest accuracy. However, if a matcher achieves just slightly lower accuracy (than the best one) but produces results that are easier to explain and debug (e.g., a decision tree), then c_4 highlights that matcher as well, for the user's consideration.

Thus, the entire process of selecting a matcher can be automated (if the user does not want to be involved), and in fact Magellan does provide a single command to execute the entire process.

4.5 Debugging a Matcher

Let the selected matcher be X. In the next step user U debugs X to improve its accuracy. Such debugging is critical in practice, yet has received very little attention in the research community.

Our guide suggests that user U debug in three steps: (1) identify and understand the matching mistakes made by X, (2) categorize these mistakes, and (3) take actions to fix common categories of mistakes.

Identifying and Understanding Matching Mistakes: U should split the development set I into two sets P and Q, train X on P then apply it to Q. Since U knows the labels of the pairs in Q, he or she knows the matching mistakes made by X in Q. These are *false positives* (non-matching pairs predicted matching) and *false negatives* (matching pairs predicted not). Addressing them helps improve precision and recall, respectively.

Next U should try to understand why X makes each mistake. For example, let $(a, b) \in Q$ be a pair labeled "matched" for which X has predicted "not matched". To understand why, U can start by using a debugger that explains how X comes to that prediction. For example, if X is a decision tree then the debugger (invoked using command c_5 in Figure 4) can show the path from the root of the tree to the leaf that (a, b) has traversed. Examining this path, as well as the pair (a, b) and its label, can reveal where things go wrong. In general things can go wrong in four ways:

- The data can be dirty, e.g., the price value is incorrect.
- The label can be wrong, e.g., (a, b) should have been labeled "not matched".
- The feature set is problematic. A feature is misleading, or a new feature is desired, e.g., we need a new feature that extracts and compares the publishers.

- The learning algorithm employed by X is problematic, e.g., a parameter such as "maximal depth to be searched" is set to be too small.

Currently Magellan has debuggers for a set of learning-based matchers, e.g., decision tree, random forest. We are working on improving these debuggers and developing debuggers for more learning algorithms.

Categorizing Matching Mistakes: After U has examined all or a large number of matching mistakes, he or she can categorize them, based on problems with data, label, feature, and the learning algorithm.

Examining all or most mistakes is very time consuming. Thus a consistent feedback we have received from real-world users is that they would love a tool that can automatically examine and give a preliminary categorization of the types of the matching mistakes. As far as we can tell, no such tool exists today.

Handling Common Categories of Mistakes: Next U should try to fix common categories of mistakes by modifying the data, labels, set of features, and the learning algorithm. This part often involves data cleaning and extraction (IE), e.g., normalizing all values of attribute "affiliation", or extracting publishers from attribute "desc" then creating a new feature comparing the publishers.

This part is often also very time consuming. Real-world users have consistently indicated needing support in at least two areas. First, they want to know exactly what kinds of data cleaning and IE operations they need to do to fix the mistakes. Naturally they want to do as minimally as possible. Second, re-executing the entire EM process after each tiny change to see if it "fixes" the mistakes is very time consuming. Hence, users want an "what-if" tool that can quickly show the effect of a hypothetical change.

Proxy Debugging: Suppose we need to debug a matcher X but there is no debugger for X, or the debugger exists but is not very informative. In this case X is effectively a "blackbox". To address this problem, in Magellan we have introduced a novel debugging method. In particular, we propose to train another matcher X' for which there is a debugger, then use that debugger to debug X', instead of X. This "proxy debugging" process cannot fix problems with the learning algorithm of X, but it can reveal problems with the data, labels, features, and fixing them can potentially improve the accuracy of X itself. Section 6.2 shows cases of proxy debugging working quite well in practice.

Selecting a Matcher Again: So far we have discussed selecting a good initial learning-based matcher X, then debugging X using the development set I. To debug, user U splits I into training set P and testing set Q, then identifies and fixes mistakes in Q. Note that this splitting of I into P and Q can be done multiple times. Subsequently, since the data, labels, and features may have changed, U would want to do cross validation again to select a new "best matcher", and so on (see Step 5 in Figure 3).

4.6 The Resulting EM Workflow

After executing the above steps, user U has in effect created an EM workflow, as shown in Figure 5. Since this workflow will be used in the production stage, it takes as input the two original tables A and B. Next, it performs a set of data cleaning, IE, and transformation operations on these tables. These operations are derived from the debugging step discussed in Section 4.5.

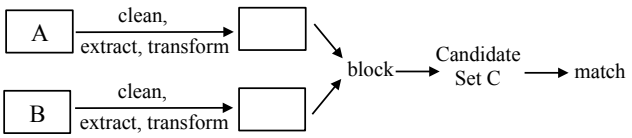

Figure 5: The EM workflow for the learning-based matching scenario.

Next, the workflow applies the blockers created in Section 4.2 to obtain a set of candidate tuple pairs C. Finally, the workflow applies the learning-based matcher created in Section 4.5 to the pairs in C.

Note that the steps of sampling and labeling a sample S do not appear in this workflow, because we need them only in the development stage, in order to create, debug, and train matchers. Once we have found a good learning-based matcher (and have trained it using S), we do not have to execute those steps again in the production stage.

4.7 How-to Guides for Scenarios with Rules

Recall that Magellan currently targets three EM scenarios. So far we have discussed a how-to guide and tools for Scenario 1: matching using supervised learning. We now briefly discuss Scenarios 2 and 3.

Scenario 2 uses only rules to match. This is desirable in practice for various reasons (e.g., when matching medicine it is often important that we can explain the matching decision). For this scenario, we have developed guides and tools to help users (a) create matching rules manually, (b) create rules using a set of labeled tuple pairs, or (c) create rules using active learning.

Scenario 3 uses both supervised learning and rules. Users often want this when using neither learning nor rules alone gives them the desired accuracy. For this scenario, we have also developed a guide and tools to help users. Our guide suggests that users do learning-based EM first, as described earlier for Scenario 1, then add matching rules "on top" of the learning-based matcher, to improve matching accuracy. We omit further details for space reasons.

5. DESIGNING FOR AN OPEN WORLD

So far we have discussed how-to guides and tools to support the guides. We now turn to the challenge of designing these tools as commands in Python.

This challenge turned out to be highly non-trivial, as we will see. It raises a fundamental question: what do we mean by "building on top of a data analysis stack"? To answer, we introduce the notion of closed-world vs. open-world systems for EM contexts. We show that Magellan should be built as an open-world system, but building such systems raises difficult problems such as designing appropriate data structures and managing metadata. Finally, we discuss how Magellan addresses these problems.

5.1 Closed-World vs. Open-World Systems

A closed-world system controls its own data. This data can only be manipulated by its own commands. For this system, its own world is the only world. There is nothing else out there and thus it does not have a notion of having to "play well" with other systems. It is often said that

RDBMSs are such closed-world systems. Virtually all current EM systems can also be viewed as closed-world systems.

In contrast, an open-world system K is aware that there is a whole world "out there", teeming with other systems, and that it will have to interact with them. The system therefore possesses the following characteristics:

- K expects other systems to be able to manipulate K's own data.
- K may also be called upon by other systems to manipulate their own data.
- K is designed in a way that facilitates such interaction.

Thus, by building Magellan on the Python data analysis stack we mean building an open-world system as described above (where "other systems" are current and future Python packages in the stack). This is necessary because, as discussed earlier, in order to do successful EM, Magellan will need to rely on a wide range of external systems to supply tools in learning, mining, visualization, cleaning, IE, etc. Building an open-world system however raises difficult problems. In what follows we discuss problems with data structures and metadata. (We have also encountered several other problems, such as missing values, data type mismatch, package version incompatabilities, etc., but will not discuss them in this paper.)

5.2 Designing Data Structures

At the heart of Magellan is a set of tables. The tuples to be matched are stored in two tables A and B. The intermediate and final results can also be stored in tables. Thus, an important question is how to implement the tables.

A popular Python package called pandas has been developed to store and process tables, using a data structure called "data frame". Thus, the simplest solution is to implement Magellan's tables as data frames. A problem is that data frames cannot store metadata, e.g., a constraint that an attribute is a key of a table.

A second choice is to define a new Python class called MTable, say, where each MTable object has multiple fields, one field points to a data frame holding the tuples, another field points to the key attributes, and so on.

Yet a third choice is to subclass the data frame class to define a new Python class called MDataFrame, say, which have fields such as "keys", "creation-date", etc. besides the inherited data frame holding the tuples.

From the perspective of building open-world systems, as discussed in Section 5.1, the last two choices are bad because they make it difficult for external systems to operate on Magellan's data. Specifically, MTable is a completely unfamiliar class to existing Python packages. So commands in these packages cannot operate on MTable objects directly. We would need to redefine these commands, a time-consuming and brittle process.

MDataFrame is somewhat better. Since it is a subclass of data frame, any existing command (external to Magellan) that knows data frames can operate on MDataFrame objects. Unfortunately the commands may return inappropriate types of objects. For example, a command deleting a row in an MDataFrame object would return a data frame object, because being an external command it is not aware of the MDataFrame class. This can be quite confusing to users, who want external commands to work smoothly on Magellan's objects.

For these reasons, we take the first choice: storing Magellan's tables as data frames. Since virtually any existing Python package that manipulates tables can manipulate data frames, this maximizes the chance that commands from these packages can work seamlessly on Magellan's tables.

In general, we propose that an open-world system K use the data structures that are most common to other systems to store its data. This brings two important benefits: it is easier for other systems to operate on K's data, and there will be far more tools available to help K manipulate its own data. If it is not possible to use common data structures, K should provide procedures that convert between its own data structures and the ones commonly used by other open-world systems.

5.3 Managing Metadata

We have discussed storing Magellan's tables as data frames. Data frames however cannot hold metadata (e.g., key and foreign key constraints, date last modified, ownership). Thus we will store such metadata in a central catalog.

Regardless of where we store the metadata, however, letting external commands directly manipulate Magellan's data leads to a problem: the metadata can become inconsistent. For example, suppose we have created a table A and stored in the central catalog that "sid" is a key for A. There is nothing to prevent a user U from invoking an external command (of a non-Magellan package) on A to remove "sid". This command however is not aware of the central catalog (which is internal to Magellan). So after its execution, the catalog still claims that "sid" is a key for A, even though A no longer contains "sid". As another example, an external command may delete a tuple from a table participating in a key-foreign key relationship, rendering this relationship invalid, while the catalog still claims that it is valid.

In principle we can rewrite the external commands to be metadata aware. But given the large number of external commands that Magellan users may want to use, and the rapid changes for these commands, rewriting all or most of them in one shot is impractical. In particular, if a user U discovers a new package that he or she wants to use, we do not want to force U to wait until Magellan's developers have had a chance to rewrite the commands in the package to be metadata aware. But allowing U to use the commands immediately, "as is", can lead to inconsistent metadata, as discussed above.

To address this problem, we design each Magellan's command c from the scratch to be metadata aware. Specifically, we write c such that at the start, it checks for all constraints that it requires to be true, in order for it to function properly. For example, c may know that in order to operate on table A, it needs a key attribute. So it looks up the central catalog to obtain the constraint that "sid" is a key for A. Command c then checks this constraint to the extent possible. If it finds this constraint invalid, then it alerts the user and asks him or her to fix this constraint.

Command c will not proceed until all required constraints have been verified. During its execution, it will try to manage metadata properly. In addition, if it encounters an invalid constraint it will alert the user, but will continue its execution, as this constraint is not critical for its correct execution (those constraints have been checked at the start of

the command). For example, if it finds a dangling tuple due to a violation of a foreign key constraint, it may just alert the user, ignore the tuple, and then continue.

6. EMPIRICAL EVALUATION

We now empirically evaluate Magellan. It is difficult to evaluate such a system in large-scale experiments with real-world data and users. To address this challenge, we evaluated Magellan in two ways. First, we asked 44 UW-Madison students to apply Magellan to many real-world EM scenarios on the Web. Second, we provided Magellan to real users at several organizations (WalmartLabs, Johnson Control, and Marshield Clinic) and reported on their experience. We now elaborate on these two sets of experiments.

6.1 Large-Scale Experiments on Web Data

Our largest experiment was with 24 teams of CS students (a total of 44 students) at UW-Madison in a Fall 2015 data science class. These students can be considered the equivalents of power users at organizations. They know Python but are not experts in EM.

We asked each team to find two data-rich Web sites, extract and convert data from them into two relational tables, then apply Magellan to match tuples across the tables. The first four columns of Table 1 show the teams, domains, and the sizes of the two tables, respectively. Note that two teams may cover the same domain, e.g., "Movies", but extract from different sites. Overall, there are 12 domains, and the tables have 7,313 tuples on average, with 5-17 attributes.

We asked each team to do the EM scenario of supervised learning followed by rules, and aim for precision of at least 90% with recall as high as possible. This is a very common scenario in practice.

The Baseline Performance: The columns under "Initial Learning-Based Matcher (A)" show the matching accuracies achieved by the best learning-based matcher (after cross validation, see Section 4.4): $P = 56 - 100\%, R = 37.5 - 100\%, F_1 = 56 - 99.5\%$. These results show that many of these tables are not easy to match, as the best learning-based matcher selected after cross validation does not achieve high accuracy. In what follows we will see how Magellan was able to significantly improve these accuracies.

Using the How-to Guide: The columns under "Final Learning+Rule Matcher (D)" show the final matching accuracies that the teams obtained: $P = 91.3 - 100\%, R = 64.7 - 100\%, F_1 = 78.6 - 100\%$. All 24 teams achieved precision exceeding 90%, and 20 teams also achieved recall exceeding 90%. (Four teams had recall below 90% because their data were quite dirty, with many missing values.) All teams reported being able to follow the how-to guide. Together with qualitative feedback from the teams, this suggests that users can follow Magellan's how-to guide to achieve high matching accuracy on diverse data sets. We elaborate on these results below, broken down by blocking and matching.

Blocking and Debugging Blockers: All teams used 1-5 blockers (e.g., attribute equivalence, overlap, rule-based), for an average of 3. On average 3 different types of blockers were used per team. This suggests that it is relatively easy to create a blocking pipeline with diverse blocker types.

All teams debugged their blockers, in 1-10 iterations, for an average of 5. 18 out of 24 teams used our debugger (see Section 4.2), and reported that it helped in four ways.

(a) Cleaning data: By examining tuple pairs (returned by the debugger) that are matches accidentally removed by blocking, 12 teams discovered data that should be cleaned. For example, one team removed the edition information from book titles, and another team normalized the date formats in the input tables.

(b) Finding the correct blocker types and attributes: 12 teams were able to use the debugger for these purposes. For example, one team found that using attribute equivalence (AE) blocker over "phone" removed many matches, because the phone numbers were not updated. So they decided to use "zipcode" instead. Another team started with AE over "name" then realized that the blocker did not work well because many names were misspelled. So they decided to use a rule-based blocker instead.

(c) Tuning blocker parameters: 18 teams used the debugger for this purpose, e.g., to change the overlap size for "address" in an overlap blocker, or to use a different threshold for a Jaccard measure in a rule-based blocker.

(d) Knowing when to stop: 12 teams explicitly mentioned in their reports that when the debugger returned no or very few matches, they concluded that the blocking pipeline had done well, and stopped tuning this pipeline.

Teams reported spending 4-32 hours on blocking (including reading documentations). Overall, 21 out of 24 teams were able to prune away more than 95% of $|A \times B|$, with an average reduction of 97.3%, suggesting that they were able to construct blocking with high pruning rate.

Feedback-wise, teams reported liking (a) the ability to create rich and flexible blocking sequences with different types of blockers, (b) the diverse range of blocker types provided by Magellan, and (c) the debugger. They complained that certain types of blockers (e.g., rule-based ones) were still slow (an issue that we are currently addressing).

Matching and Debugging Matchers: Recall from Section 4.5 that after cross validation on labeled data to select the best learning-based matcher X, user U iteratively debugged X to improve its accuracy. Teams performed 1-5 debugging iterations, for an average of 3 (see Column "Num of Iterations (C)" in Table 1). The actions they took were:

(a) Feature selection: 21 teams added and deleted features, e.g., adding more phone related features, removing style related features.

(b) Data cleaning: 12 teams cleaned data based on the debugging result, e.g., normalizing colors using a dictionary, detecting that the tables have different date formats. 16 teams found and fixed incorrect labels during debugging.

(c) Parameter tuning: 3 teams tuned the parameters of the learning algorithm, e.g., modifying the maximum depth of decision tree based on debugging results.

These debugging actions helped improve accuracies significantly, from 56-100% to 73.3-100% precision, and 37.5-100% to 61-100% recall (compare columns under "A" with those under "B" in Table 1).

Adding rules further improves accuracy. 19 teams added 1-5 rules, found in 1-5 iterations (see column "E"). This improved precision from 73.3-100% to 91.3-100% and recall from 61-100% to 64.7-100% (compare columns under

Team	Domain	Size of Table A	Size of Table B	Cand. Set Size	Initial Learning-Based Matcher (A)			Final Learning-Based Matcher (B)			Num. of Iterations (C)	Final Learning + Rules Matcher (D)			Num. of Iterations (E)	Diff. in F_1 between (D) and (A) in %
					P	R	F1	P	R	F1		P	R	F1		
1	Vehicles	4786	9003	8009	71.2	71.2	71.2	91.43	94.12	92.75	4	100	100	100	2	30.27
2	Movies	7391	6408	78079	99.28	95.13	97.04	98.21	100	99.1	2	100	100	100	1	2.12
3	Movies	3000	3000	1000000	98.9	99.44	99.5	98.63	98.63	98.63	1	98.63	98.63	98.63	0	-0.87
4	Movies	3000	3000	36000	68.2	69.16	68.6	98	100	98.99	3	98	100	98.99	1	44.3
5	Movies	6225	6392	54028	100	95.23	97.44	100	100	100	3	100	100	100	1	2.63
6	Restaurants	6960	3897	10630	100	37.5	54.55	100	88.89	94.12	3	100	88.89	94.12	1	72.54
7	Electronic Products	4559	5001	823832	73	51	59	73.3	64.71	68.75	2	100	64.71	78.57	1	33.17
8	Music	6907	55923	58692	92	79.31	85.19	90.48	82.61	86.36	2	100	92.16	95.92	2	1.37
9	Restaurants	9947	28787	400000	100	78.5	87.6	94.44	97.14	95.77	4	94.44	97.14	95.77	0	9.33
10	Cosmetic	11026	6445	36026	56	56	56	96.67	87.88	92.06	3	96.43	87.1	91.53	4	64.39
11	E-Books	6482	14110	13652	96.67	96.67	96.67	100	95.65	97.78	4	100	98.33	99.13	1	1.15
12	Beer	4346	3000	4334961	84.5	59.6	65.7	100	60.87	75.68	4	91.3	91.3	91.3	4	15.19
13	Books	3506	3508	2016	93.46	100	96.67	91.6	100	95.65	2	91.6	100	95.65	0	-1.06
14	Books	3967	3701	4029	74.17	82.2	82.5	100	84.85	91.8	3	100	84.85	91.8	5	11.27
15	Anime	4000	4000	138344	95.9	88.9	92.2	100	100	100	2	100	100	100	1	8.46
16	Books	3021	3098	931	74.2	100	85.2	96.34	84.95	90.29	2	94.51	92.47	93.48	1	5.97
17	Movies	3556	6913	504	94.2	99.33	96.6	95.04	94.26	94.65	2	95.04	94.26	94.65	1	-2.02
18	Books	8600	9000	492	91.6	100	84.8	94.8	100	90.2	3	100	92.31	96	1	6.37
19	Restaurants	11840	5223	5278	98.6	93.8	96.1	95.6	94.02	95.57	2	100	94.12	96.97	1	-0.55
20	Books	3000	3000	257183	94.24	72.88	81.71	90.91	83.33	86.96	2	92.31	100	96	1	6.43
21	Literature	3885	3123	1590633	84.4	86.9	85.5	100	95.65	97.83	3	100	95.65	97.83	0	14.42
22	Restaurants	3014	5883	78190	100	93.59	96.55	100	100	100	5	100	100	100	0	3.57
23	E-Books	6501	14110	18381	94.6	92.5	93.4	94.6	97.22	95.89	2	100	100	100	1	2.67
24	Baby Products	10000	5000	11000	78.6	44.8	57.7	96.43	72.97	83.08	5	100	72.97	84.37	2	43.99

Table 1: Large-scale experiments with Magellan on Web data.

"D" with those under "B"). Overall, Magellan improved the baseline accuracy in columns "A" significantly, by as much as 72.5% F_1, for an average of 18.8% F_1. For 3 teams, however, accuracy dropped by 0.87-2.02% F_1. This is because the baseline F_1s already exceeded 94%, and when teams tried to add rules to increase F_1 further, they overfit the development set.

Teams reported spending 5-50 hours, for an average of 12 hours (including reading documentation and labeling samples) on matching. They reported liking debugger support, ease of creating custom features for matchers, and support for rules to improve learning-based matching. They would like to have more debugger support, including better ordering and visualization of matching mistakes.

6.2 Experience with Organizational Data

We now describe our experience with Magellan at WalmartLabs, Marshfield Clinic, and Johnson Control. These are newer and still ongoing evaluations.

WalmartLabs deploy multiple EM systems for various purposes. As a first project, the EM team tried to debug a system that matches product descriptions. Since it is a complicated "blackbox" in production, they tried proxy debugging (Section 4.5). Specifically, they debugged a random forest based matcher and used the debugging result to clean the data, fix labels, and add new features. This significantly improved the system in production: increasing recall by 34% while reducing precision slightly by 0.65%. This indicates the promise of proxy debugging. In fact, 3 teams out of the 24 teams discussed in the previous subsection also used proxy debugging.

For Marshfield Clinic, we are currently helping to develop an EM workflow that uses learning and rules to match drug descriptions. Here labeling drug descriptions is very expensive, requiring domain experts who have limited time. They are also concerned about skewed data, i.e., too few matches in the current data. Taken together, this suggests that the sampling and labeling solution we discussed in Section 4.3 is well motivated, and we have been using a variant of that solution to help them label data. Yet another problem is that the Marshfield team is geographically distributed, so they would really like to have a cloud-based version of Magellan.

Finally, we are currently also working with Johnson Control to match data related to heating and cooling in buildings. The data that we have seen so far is very dirty. So the JCI team wants to extend Magellan with many more cleaning capabilities, in terms of Python packages that can immediately be made to work with Magellan's data.

6.3 Summary

Our experiments show that (a) current users can successfully follow the how-to guide to achieve high matching accuracy on diverse data sets, (b) the various tools developed for Magellan (e.g., debuggers) can be highly effective in helping the users, (c) practical EM requires a wide range of capabilities, e.g., cleaning, extraction, visualization, underscoring the importance of placing Magellan in an eco-system that provides such capabilities, and (d) there are many more EM challenges (e.g., cloud services) raised by observing Magellan "in the wild".

7. RELATED WORK

Numerous EM algorithms have been proposed [11, 16]. But far fewer EM systems have been developed. We discussed these systems in Section 2.2 (see also [11]). For

matching using supervised learning (Section 4), some of these systems provide only a set of matchers. None provides support for sampling, labeling, selecting and debugging blockers and matchers, as Magellan does.

Some recent works have discussed desirable properties for EM systems, e.g., being extensible and easy-to-deploy [14], being flexible and open source [10], 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 [17]. These works do not discuss 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 this paper.

Several works have addressed scaling up blocking (e.g., [13, 21, 25, 1]), learning blockers [7, 15], and using crowdsourcing for blocking [20] (see [12] for a survey). As far as we know, there has been no work on debugging blocking, as we do in Magellan.

On sampling and labeling, several works have studied active sampling [23, 4, 6]. These methods however are not directly applicable in our context, where we need a representative sample in order to estimate the matching accuracy (see Step 6 in Figure 3). For this purpose our work is closest to [20], which uses crowdsourcing to sample and label.

Debugging learning models has received relatively little attention, even though it is critical in practice, as this paper has demonstrated. Prior works help users build, inspect and visualize specific ML models (e.g., decision trees [3], Naive Bayes [5], SVM [9], ensemble model [24]). But they do not allow users to examine errors and inspect raw data. In this aspect, the work closest to ours is [2], which addresses iterative building and debugging of supervised learning models. The system proposed in [2] can potentially be implemented as a Magellan's tool for debugging learning-based matchers.

Finally, the notion of "open world" has been discussed in [18], but in the context of crowd workers' manipulating data inside an RDBMS. Here we discuss a related but different notion of open-world systems that often interact with and manipulate each other's data. In this vein, the work [8] is related in that it discusses the API design of the scikit-learn package and its design choices to seamlessly tie in with other packages in Python.

8. CONCLUSIONS & FUTURE WORK

In this paper we have argued that significantly more attention should be paid to building EM systems. We then described 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 plan to conduct more evaluation of Magellan, to further examine the research problems raised in this paper, to extend Magellan with more capabilities (e.g., crowdsourcing), and to deploy it on the cloud as a service. We will also explore managing more EM scenarios. In particular, we plan to extend Magellan to handle string matching, which uses workflows similar to those of matching using supervised learning. Other interesting EM scenarios include linking a table into a knowledge base (e.g., [19]) and matching using iterative blocking [26]. The former can potentially be incorporated into the current Magellan, but the latter will likely require a new EM management system (as it uses a very different kind of EM workflows). See [22] for a detailed discussion of these EM scenarios.

Acknowledgment: We thank the reviewers for invaluable comments. This work is supported by gifts from Walmart-Labs, Google, Johnson Control, and by NIH BD2K grant U54 AI117924.

9. REFERENCES

[1] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman. Fuzzy joins using MapReduce. ICDE, 2012.
[2] S. Amershi, M. Chickering, S. M. Drucker, B. Lee, P. Simard, and J. Suh. Modeltracker: Redesigning performance analysis tools for machine learning. CHI, 2015.
[3] M. Ankerst, C. Elsen, M. Ester, and H.-P. Kriegel. Visual classification: An interactive approach to decision tree construction. KDD, 1999.
[4] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. SIGMOD, 2010.
[5] B. Becker, R. Kohavi, and D. Sommerfield. Visualizing the simple Bayesian classifier. In *Information Visualization in Data Mining and Knowledge Discovery*, 2002.
[6] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. KDD, 2012.
[7] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. ICDM, 2006.
[8] L. Buitinck et al. API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.
[9] D. Caragea, D. Cook, and V. Honavar. Gaining insights into support vector machine pattern classifiers using projection-based tour methods. KDD, 2001.
[10] P. Christen. Febrl: A freely available record linkage system with a graphical user interface. HDKM, 2008.
[11] P. Christen. *Data Matching*. Springer, 2012.
[12] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE TKDE*, 24(9):1537–1555, 2012.
[13] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. *PVLDB*, 9(11):864–875, 2016.
[14] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: A commodity data cleaning system. SIGMOD, 2013.
[15] A. Das Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. CIKM, 2012.
[16] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.
[17] 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.
[18] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. SIGMOD, 2011.
[19] A. Gattani et al. Entity extraction, linking, classification, and tagging for social media: A Wikipedia-based approach. *PVLDB*, 6(11):1126–1137, 2013.
[20] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. SIGMOD, 2014.
[21] L. Kolb, A. Thor, and E. Rahm. Dedoop: efficient deduplication with Hadoop. *PVLDB*, 5(12):1878–1881, 2012.
[22] P. Konda et al. Magellan: Toward building entity matching management systems. 2016. Technical Report, http://www.cs.wisc.edu/~anhai/papers/magellan-tr.pdf.
[23] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. KDD, 2002.
[24] J. Talbot, B. Lee, A. Kapoor, and D. Tan. Ensemblematrix: Interactive visualization to support machine learning with multiple classifiers. CHI, 2009.
[25] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. SIGMOD, 2010.
[26] S. E. Whang et al. Entity resolution with iterative blocking. SIGMOD, 2009.