Magellan: Toward Building Entity Matching Management Systems

Pradap Konda[1], Sanjib Das[1], Paul Suganthan G.C.[1], Philip Martinkus[1], AnHai Doan[1], Adel Ardalan[1], Jeffrey R. Ballard[1], Yash Govind[1], Han Li[1], Fatemah Panahi[2], Haojun Zhang[1], Jeff Naughton[2], Shishir Prasad[3], Ganesh Krishnan[3], Rohit Deep[3], Vijay Raghavendra[3]

[1]University of Wisconsin-Madison, [2]Google, [3]@WalmartLabs

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 describe Magellan, a new kind of EM system. 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 execute these steps; the tools seek to cover the entire EM pipeline, not just blocking and matching as current EM systems do. (3) Tools are built into the Python open-source data science ecosystem, 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 and present extensive experiments 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 [13, 22]. Most current EM works however have focused only on developing *matching algorithms* [13, 22].

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 ever-more-complex join algorithms without having the rest of the RDBMS. At some point we must build end-to-end systems to evaluate matching algorithms, to integrate R&D efforts, to educate our students in EM, 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 drastically 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. 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, cleaning).

Second, EM steps often must exploit many techniques, e.g., learning, mining, visualization, outlier detection, information extraction (IE), crowdsourcing, etc. Current EM systems (most of which are stand-alone monoliths that are not designed from scratch to "play well" with other systems) do not provide enough support for these techniques.

Third, users often have to write code to "patch" the system (e.g., to implement a lacking functionality or combine system components), ideally using a scripting 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 how to proceed end to end. Suppose a user wants to perform EM with at least 95% precision and 80% recall. Should he or she start out using a learning or rule-based EM approach? If learning-based, then which technique to select among the many existing ones? How to debug? What to do if after many tries the user still cannot reach 80% recall? 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 several industrial partners. Magellan (named after Ferdinand Magellan, the first end-to-end explorer of the globe) is novel in several important aspects.

First, Magellan focuses on helping power users (those who know how to code) execute a set of *EM scenarios* (e.g., using supervised learning to match two tables with a target accuracy). For each EM scenario, Magellan provides a comprehensive *how-to guide* that tells users what to do, step by step, end to end.

Second, Magellan identifies "pain points" in each guide, i.e., steps that require a lot of user effort, then provides tools to address those pain points. As we will see, these tools cover

©VLDB Endowment 2016. This is a minor revision of the paper entitled Magellan: Toward Building Entity Matching Management Systems, published in the Proceedings of the VLDB Endowment, Vol. 9, No. 12, 2150-8097/16/08.
DOI: https://doi.org/10.14778/2994509.2994535.
.

the entire EM pipeline (e.g., debugging, sampling), not just the blocking and matching steps.

Third, the tools are being built within the Python data science ecosystem, allowing users to easily exploit a wide range of techniques in learning, visualization, cleaning, etc. (as captured in numerous Python packages in this ecosystem, such as pandas, scikit-learn, matplotlib, pytorch, pyspark, etc.).

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

As described, Magellan assumes that the EM process cannot be automated. Instead it must involve the human user. So Magellan provides a detailed how-to guide that spells out where the human user must be involved and how, and where a tool can be used to reduce the user effort. Thus, Magellan is an example of *"human-in-the-loop" data management systems*, which have received significant recent attention [19].

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, as we will show in Section 3.3.

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., sampling, 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 "ecosystem" and is expected to "play well" with others (e.g., other Python packages). We say that Magellan is an "open-world system", because it relies on many other systems in the ecosystem 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.

Current Status: In the past three years we have started to address the above challenges. Specifically, we have open sourced Magellan [3]. As far as we can tell, Magellan is the most comprehensive open-source EM system today, in terms of the number of features it supports.

Magellan has been successfully used in five domain science projects at UW-Madison (in economics, biomedicine, environmental science [32, 33, 37, 9]), and at several companies (e.g., Johnson Control, Marshfield Clinic, Recruit Holdings [1], WalmartLabs). For example, at WalmartLabs it improved the recall of a deployed EM solution by 34%, while reducing precision slightly by 0.65%. It has also been used by 400+ students to match real-world data in five data science classes at UW-Madison (e.g., [2]).

Applying Magellan to the above real-world applications raised many research challenges. Examples include helping users finalize their matching definition [32, 19], debugging blocking [34], debugging rule-based EM [36], human-in-the-loop EM [19], applying deep learning to match textual data [35], hands-off string matching, data cleaning, and more. We have started to address some of these research challenges [34, 35, 36, 19], describe case studies [32], and summarize the lessons learned [19, 32]. Magellan and the data generated in this project have also been used by other research groups (e.g., [21, 25]).

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.

CloudMatcher and BigGorilla: In terms of broader impacts, Magellan is an on-premise EM solution for power users. In a related project, we have been developing CloudMatcher, a hands-off cloud/crowd EM service for lay users (to be deployed soon at *cloudmatcher.io*) [27, 17, 26]. Our Magellan work has significantly influenced the development of CloudMatcher, by suggesting desired functionalities and pointing out possible limitations of such EM services [27].

The ideas underlying Magellan can potentially be applied to other types of data integration problems (e.g., schema matching, information extraction, data cleaning, etc.). We have started to flesh out a similar system-building agenda for data integration [20, 18]. We have also been partnering with Recruit Institute of Technology to encourage a community around BigGorilla, a repository of data preparation and integration tools [41]. The goal of BigGorilla is to foster an ecosystem of such tools, as a part of the Python data science ecosystem, for research, education, and practical purposes.

The rest of this paper motivates Magellan then discusses the solution architecture, empirical evaluation, lessons learned, and ongoing research directions. This paper is a condensed version of [29]. More details can be found in that paper and in [30, 31], and on the Magellan project's homepage [3].

2. LIMITATIONS OF CURRENT ENTITY MATCHING SYSTEMS

Entity matching (EM) has received much attention [13, 22]. A common EM scenario finds all tuple pairs 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. [13].

Most EM works have developed matching algorithms that exploit rules, learning, clustering, crowdsourcing, among others [13, 22]. 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 5).

In contrast to the extensive effort on matching algorithms, 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 [13]), and 15 major commercial ones (e.g., Tamr, Data Ladder, IBM InfoSphere, IBM Midas [28, 38]). Our examination of these systems (see [30]) 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.

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 current EM systems. 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 3.3. 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 and what to do when no tool is available. No EM system that we have examined provides such guidance.

3. THE MAGELLAN SOLUTION

We now describe Magellan and discuss how it addresses the above limitations. 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.

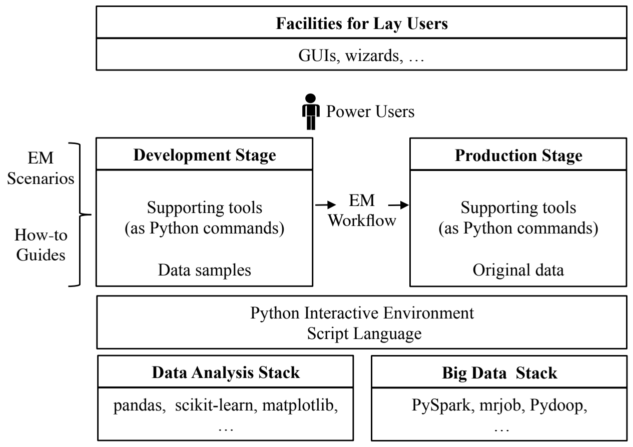

Figure 2: The Magellan architecture.

In the development stage, the user develops a good EM workflow (e.g., one with high matching accuracy). The guide tells the user what to do, step by step. For each step which is a "pain point", the user can use a set of supporting tools (each of which is 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 tools.

Both stages have access to the Python interactive scripting environment (e.g., Jupyter Notebook). Further, tools are built into the Python data science ecosystem. Thus, Magellan is an "open-world" system, as it often has to borrow functionalities (e.g., cleaning, extraction, visualization) from other Python packages in the ecosystem.

Finally, the current Magellan is geared toward power users (who can program). 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 what follows we elaborate on the Magellan architecture.

3.1 EM Scenarios and Workflows

We classify EM scenarios along four dimensions: *(1) Problems:* Matching two tables; matching within a table; matching a table into a knowledge base; etc. *(2) Solutions:* Using learning; using learning and rules; performing data cleaning, blocking, then matching; performing IE, then cleaning, blocking, and matching; etc. *(3) Domains:* Matching two tables of biomedical data; matching e-commerce products given a large product taxonomy as background knowledge; etc. *(4) 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 build Magellan to handle a few common scenarios, and then extend it to more 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).

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 4 shows a sample workflow (which we explain in Section 3.3).

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 finds a good EM workflow, typically 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. The Python stack for example includes the 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. As of March 2018, there were 536 Python 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.

3.3 How-to Guides and Tools

We now discuss developing how-to guides and tools to support these guides. 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 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 (e.g., sampling, labeling), can raise many interesting research challenges.

Recall that Magellan currently targets three EM scenarios (Section 3.1). For space reasons, we will focus on the scenario of matching using supervised learning, and on developing a guide for the development stage of this scenario. Figure 3 shows the current version of this guide, listing only the top six steps. While each step may sound fairly informal (e.g., "create a set of features"), the full guide (available with Magellan's release) is far more complex and 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.

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 briefly discuss these steps (see [30] for more details). For ease of exposition, we will assume that tables A and B share the same schema.

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 develop-

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.

ment stage with these smaller tables. Since this stage is iterative by nature, working with large tables can be very time consuming. Random sampling however does not work, because tables A' and B' may end up sharing very few matches. 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, which proved quite effective in our experiments (see [30]).

This tool however 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 current tool, 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.

Blocking to Create Candidate Tuple Pairs: Next, we apply blocking to the tables A' and B' to generate a set C of tuple pairs $(a \in A', b \in B')$. Many blocking solutions have been developed [13]. In practice, however, users often have three questions which current work has not addressed: (1) how to select the best blocker, (2) how to debug a given blocker, and (3) how to know when to stop?

Selecting the Best Blocker: There is no satisfactory solution yet to this problem. For now, based on our experience, we recommend that the user try successively more complex blockers. Specifically, the user can try overlap blocking first (e.g., "matching tuples must share at least k tokens in an attribute x"), then attribute equivalence blocking (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., sorted neighborhood, hash) if appropriate. Finally, the user can try rule-based blocking. 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).

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 [34]. 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. If the user 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.

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.

As a result, users often 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 4) 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.

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 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). We have developed a solution to address this problem, building on the work in [26] (see [30]).

Selecting a Matcher: Once user U has labeled a sample S, U uses S to select a good initial learning-based matcher. Our guide provides a tool to address this problem. The tool first automatically generates a set of features, uses them to convert each pair in S into a feature vector, then performs cross validation over a subset of the feature vectors to select the matcher with the highest estimated accuracy from among those supplied by Magellan.

Debugging a Matcher: Let the selected matcher be X. Next, user U debugs X to improve its accuracy. Such debugging is critical in practice, yet has received little attention. 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: Given a labeled set I for debugging purpose, U should split I into two sets P and Q, train X on P then apply it to Q to identify the matching mistakes made by X in Q (this process can be repeated many times, using different P and 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, using a match debugger where available. There are four major categories of mistakes. (1) The data can be dirty, e.g., the price value is incorrect. (2) The label can be wrong, e.g., a pair should have been labeled "not matched". (3) 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. (4) 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.

The Resulting EM Workflow: After executing the above steps, user U has in effect created an EM workflow,

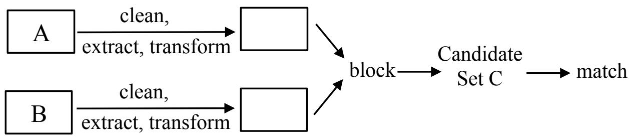

Figure 4: The EM workflow for the learning-based matching scenario.

as shown in Figure 4. 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 3.3. Next, the workflow applies the blockers created in Section 3.3 to obtain a set of candidate tuple pairs C. Finally, the workflow applies the learning-based matcher created in Section 3.3 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. EMPIRICAL EVALUATION

As of March 2018, Magellan [3] consists of 6 Python packages, 37K lines of code, and 231 commands. It has been developed over 3 years by 13 developers. We now describe using Magellan in data science classes, with domain scientists at UW-Madison, and at companies.

4.1 Using Magellan in Data Science Classes

So far 400+ students (including 90+ undergraduates) have used Magellan in 5 data science classes at UW-Madison. These students can be considered the equivalents of power users at organizations. They know Python but are not experts in EM. We asked them to form team of 2-3 students, then 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 [16]. We typically 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 (a very common scenario in practice).

We now describe in more details our experience with a Fall 2015 class, which consisted of 44 students divided into 24 teams (see [30] for details). These teams extracted tables in 12 domains (e.g., Vehicles, Movies, Restaurants, Music, etc.). The tables have 7,313 tuples on average, with 5-17 attributes. On these tables, the best learning-based matcher (after cross validation) achieved accuracy P = 56-100%, R = 37.5-100%, F_1 = 56-99.5%, suggesting that many of these tables are not easy to match. Using Magellan, however, the teams were able to significantly improve these accuracies, achieving 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.

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 [34], and reported that it helped in four ways: cleaning data, finding the correct blocker types and attributes, tuning blocker parameters, and knowing when to stop (see [30]). 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.

Recall from Section 3.3 that after cross validation 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. They added and deleted features, cleaned data based on the debugging result, and tuned the parameters of the learning algorithm. These actions helped improve accuracies from 56-100% to 73.3-100% precision, and 37.5-100% to 61-100% recall. Adding rules further improves accuracy: precision from 73.3-100% to 91.3-100% and recall from 61-100% to 64.7-100%.

4.2 Domain Sciences and Companies

So far Magellan has been applied to five projects in three domain sciences at UW-Madison. First, a team of applied economists used Magellan to match two tables of 1,832 and 1,916 grant descriptions, respectively [32]. Magellan achieved significantly better accuracy, improving recall by 23% while achieving comparable precision, compared to a rule-based EM solution deployed at [32]. The same team also used Magellan to match two tables of 1,851 and 13.5M organization descriptions, respectively.

A team in biomedicine used Magellan to match two tables of 453K and 451K of drug descriptions, achieving 99.1% precision and 95.2% recall [33, 37]. Another team in biomedicine used Magellan to match attribute names within a community data repository [9]. Finally, a team in environmental sciences also used Magellan to match attribute names within a community data repository. These last two examples show how Magellan can also be used to match schema elements, not just data instances.

Magellan has also been used for EM at several companies, including WalmartLabs, Johnson Control, Marshfield Clinic, and Recruit Holdings. At WalmartLabs, Magellan was able to help improve the recall of a deployed EM solution by 34% while reducing precision slightly by 0.65%. Johnson Control has used Magellan to match addresses (between tables of size 90K vs. 231K) and vendors (within a single table of size 50K). Marshfield Clinic was involved in the drug matching project described earlier [33, 37]. Recruit Holdings used Magellan to match stores, companies, and properties (e.g., de-duplicating 10K store names with 98.9% accuracy) [1].

4.3 Discussion

Our experience with Magellan suggest that users can successfully follow the how-to guide to achieve high EM accuracy on diverse data sets. In fact, we consider the how-to guide to be the single most important component of the system. Without it, users are lost: they do not even know where to start, when to use what tools, and how.

Our experience further suggests that the various tools developed for Magellan (e.g., debuggers) can be highly effective in helping the users. It also clearly shows that practical EM requires a wide range of capabilities, e.g., cleaning, extraction, visualization, underscoring the importance of placing Magellan in an ecosystem that provides such capabilities. (In fact, Magellan currently uses 11 packages in the Python ecosystem to provide such capabilities.)

At the same time, our experience also raises many interesting challenges. First, it turns out that at the start of an EM project, users often do not know what it means to match, i.e., there are often many alternative match definitions, and users often are not even aware of these, let alone selecting the right one [19, 32]. This can significantly complicate the EM process. Thus, it is highly desirable to have a step in the how-to guide (together with some tools) to help users explore and finalize the match definition. Second, some users want to play around with multiple match definitions, just to see how sensitive to these definitions the inferences based on the matches are. Third, a portion of the data may turn out to be so dirty for EM that it should be removed before continuing with the EM process, but how can we detect such portions? Fourth, an EM team is often geographically distributed. How can they use Magellan in such settings. Finally, Magellan is an "open-world system", in that it relies on many other packages in the Python ecosystem 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 [30]. There are many other challenges (e.g., how to debug and serve learning models, how to visualize the matches, etc.). In recent papers we have tried to summarize some of these case studies, lessons learned, and challenges [32, 19, 29, 27, 35]. We have also started to address some of these challenges [34, 35, 36, 19]. But much more remains to be done.

5. RELATED WORK

Numerous EM algorithms have been proposed [13, 22]. But far fewer EM systems have been developed. We discussed these systems in Section 2 (see also [13]). For matching using supervised learning (Section 3.3), 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 [15], being flexible and open source [12], 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 [23]. The IBM Midas project has also proposed a language for helping users tackle the different stages of the EM pipeline [28, 38]. These works however 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, learning blockers, and using crowdsourcing for blocking (see [14] for a survey). As far as we know, there has been no work on debugging blocking, as we do in Magellan (see [34]).

On sampling and labeling, several works have studied active sampling [39, 6, 8]. 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 [26], 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 [5], Naive Bayes [7], SVM [11], ensemble model [40]). But they do not allow users to examine errors and inspect raw data. In this aspect, the work closest to ours is [4], which addresses iterative building and debugging of supervised learning models. The system proposed in [4] can potentially be implemented as a Magellan's tool for debugging learning-based matchers.

Finally, the notion of "open world" has been discussed in [24], 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 [10] 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.

6. CONCLUSIONS & ONGOING WORK

We have argued that significantly more attention should be paid to building EM systems. We 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 ecosystem, open world vs. closed world systems, and easy access to an interactive script environment.

We are conducting more real-world evaluation of Magellan, further examining the research challenges raised in this paper, and extending Magellan with more capabilities (e.g., crowdsourcing). Building on Magellan, we have also been working on two other projects. CloudMatcher is a cloud/crowd EM service for lay users [27, 17, 26]. The how-to guide of Magellan helps us determine which capabilities to add to CloudMatcher, to make it useful in performing EM end to end [27]. BigGorilla is a joint effort led by UW-Madison and Recruit Institute of Technology to encourage a community around an open-source ecosystem of data preparation and integration tools [41]. Currently, BigGorilla curates tools for schema matching, information extraction, and entity matching (including Magellan), among others.

Acknowledgment: We thank the SIGMOD Record's associate editors for shepherding this paper. This work is generously supported by WalmartLabs, Google, Johnson Control, American Family Insurance, UW-Madison UW2020 grant, NIH BD2K grant U54 AI117924, and NSF Medium grant IIS-1564282.

7. REFERENCES

[1] BigGorilla: An Open-source Data Integration and Data Preparation Ecosystem: `https://recruit-holdings.com/news_data/release/2017/0630_7890.html`.

[2] CS 838: Data Science: Principles, Algorithms, and Applications `https://sites.google.com/site/anhaidgroup/courses/cs-838-spring-2017/project-description/stage-3`.

[3] Magellan home page `https://sites.google.com/site/anhaidgroup/projects/magellan`.

[4] S. Amershi et al. Modeltracker: Redesigning performance analysis tools for machine learning. CHI, 2015.

[5] M. Ankerst et al. Visual classification: An interactive approach to decision tree construction. KDD, 1999.

[6] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. SIGMOD, 2010.

[7] B. Becker, R. Kohavi, and D. Sommerfield. Visualizing the simple Bayesian classifier. In *Information Visualization in Data Mining and Knowledge Discovery*, 2002.

[8] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. KDD, 2012.

[9] M. Bernstein et al. MetaSRA: normalized human sample-specific metadata for the sequence read archive. *Bioinformatics*, 33(18):2914–2923, 2017.

[10] L. Buitinck et al. API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

[11] D. Caragea, D. Cook, and V. Honavar. Gaining insights into support vector machine pattern classifiers using projection-based tour methods. KDD, 2001.

[12] P. Christen. Febrl: A freely available record linkage system with a graphical user interface. HDKM, 2008.

[13] P. Christen. *Data Matching*. Springer, 2012.

[14] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *TKDE*, 24(9):1537–1555, 2012.

[15] M. Dallachiesa et al. Nadeef: A commodity data cleaning system. SIGMOD, 2013.

[16] S. Das et al. The Magellan data repository. `https://sites.google.com/site/anhaidgroup/projects/data`.

[17] S. Das et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, 2017.

[18] A. Doan. What is our agenda for data science? In *CIDR*, 2017.

[19] A. Doan et al. Human-in-the-loop challenges for entity matching: A midterm report. In *HILDA*, 2017.

[20] A. Doan et al. Toward a system building agenda for data integration and cleaning. In *IEEE Data Engineering Bulletin, Special Issue on Data Integration (to appear)*, 2018.

[21] M. Ebraheem et al. DeepER–deep entity resolution. *arXiv preprint arXiv:1710.00597*, 2017.

[22] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.

[23] M. Fortini et al. Towards an open source toolkit for building record linkage workflows. In *In SIGMOD Workshop on Information Quality in Information Systems*, 2006.

[24] M. J. Franklin et al. CrowdDB: answering queries with crowdsourcing. SIGMOD, 2011.

[25] C. Ge et al. Private exploration primitives for data cleaning. *arXiv preprint arXiv:1712.10266*, 2017.

[26] C. Gokhale et al. Corleone: Hands-off crowdsourcing for entity matching. SIGMOD, 2014.

[27] Y. Govind et al. Cloudmatcher: A cloud/crowd service for entity matching. In *BIGDAS*, 2017.

[28] M. A. Hernández et al. HIL: a high-level scripting language for entity integration. In *EDBT*, 2013.

[29] P. Konda et al. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[30] P. Konda et al. Magellan: Toward building entity matching management systems. 2016. Technical Report, `http://www.cs.wisc.edu/~anhai/papers/magellan-tr.pdf`.

[31] P. Konda et al. Magellan: Toward building entity matching management systems over data science stacks. *PVLDB*, 9(13):1581–1584, 2016.

[32] P. Konda et al. Performing entity matching end to end: A case study. 2016. Technical Report, `http://www.cs.wisc.edu/~anhai/papers/umetrics-tr.pdf`.

[33] E. LaRose et al. Entity matching using Magellan: Mapping drug reference tables. In *AIMA Joint Summit*, 2017.

[34] H. Li et al. Matchcatcher: A debugger for blocking in entity matching. In *EDBT*, 2018.

[35] S. Mudgal et al. Deep learning for entity matching: A design space exploration. In *SIGMOD*, 2018.

[36] F. Panahi et al. Towards interactive debugging of rule-based entity matching. In *EDBT*, 2017.

[37] P. Pessig. Entity matching using Magellan - Matching drug reference tables. In CPCP Retreat 2017. `http://cpcp.wisc.edu/resources/cpcp-2017-retreat-entity-matching`.

[38] K. Qian et al. Active learning for large-scale entity resolution. In *CIKM*, 2017.

[39] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. KDD, 2002.

[40] J. Talbot et al. Ensemblematrix: Interactive visualization to support machine learning with multiple classifiers. CHI, 2009.

[41] W.-C. Tan et al. Big gorilla: an open-source ecosystem for data preparation and integration. In *IEEE Data Engineering Bulletin, Special Issue on Data Integration (to appear)*, 2018.