Entity Matching Meets Data Science: A Progress Report from the Magellan Project

Yash Govind[1], Pradap Konda[2], Paul Suganthan G. C.[3], Philip Martinkus[1*]
[1] *University of Wisconsin,* [2] *Facebook,* [3] *Google*

Haojun Zhang[1], Adel Ardalan[5], Sanjib Das[3], Derek Paulsen[1], Amanpreet Saini[1], Erik Paulson[6]
[1] *University of Wisconsin,* [3] *Google,* [5] *Columbia University,* [6] *Johnson Control*

Palaniappan Nagarajan[4], Han Li[1], Aravind Soundararajan[1], Sidharth Mudgal[4], Jeffrey R. Ballard[1]
[1] *University of Wisconsin,* [4] *Amazon*

Youngchoon Park[6], Marshall Carter[7], Mingju Sun[7], Glenn M. Fung[7], AnHai Doan[1]
[1] *University of Wisconsin,* [6] *Johnson Control,* [7] *American Family Insurance*

ABSTRACT

Entity matching (EM) finds data instances that refer to the same real-world entity. In 2015, we started the Magellan project at UW-Madison, joint with industrial partners, to build EM systems. Most current EM systems are stand-alone monoliths. In contrast, Magellan borrows ideas from the field of data science (DS), to build a new kind of EM systems, which is an ecosystem of interoperable tools. *This paper provides a progress report on the past 3.5 years of Magellan, focusing on the system aspects and on how ideas from the field of data science have been adapted to the EM context.* We argue why EM can be viewed as a special class of DS problems, and thus can benefit from system building ideas in DS. We discuss how these ideas have been adapted to build PyMatcher and CloudMatcher, EM tools for power users and lay users. These tools have been successfully used in 21 EM tasks at 12 companies and domain science groups, and have been pushed into production for many customers. We report on the lessons learned, and outline a new envisioned Magellan ecosystem, which consists of not just on-premise Python tools, but also interoperable microservices deployed, executed, and scaled out on the cloud, using tools such as Dockers and Kubernetes.

CCS CONCEPTS

• **Information systems** → **Information integration**; **Deduplication**; **Entity resolution**.

KEYWORDS

data integration, entity matching, entity resolution, machine learning, data science

ACM Reference Format:
Yash Govind, Pradap Konda, Paul Suganthan G. C., Philip Martinkus, Palaniappan Nagarajan, Han Li, Aravind Soundararajan, Sidharth Mudgal, Jeffrey R. Ballard, Haojun Zhang, Adel Ardalan, Sanjib Das, Derek Paulsen, Amanpreet Saini, Erik Paulson, Youngchoon Park, Marshall Carter, Mingju Sun, Glenn M. Fung, and AnHai Doan. 2019. Entity Matching Meets Data Science: A Progress Report from the Magellan Project. In *2019 International Conference on Management of Data (SIGMOD19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3299869.3314042

1 INTRODUCTION

Entity matching (EM) finds data instances that refer to the same real-world entity, such as (David Smith, UW-Madison) and (D. Smith, UWM). This problem has been a long-standing challenge in data management [1, 2], and will become even more important as data-driven applications proliferate.

*The first two authors contributed equally to this project.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3314042

As a result, in Summer 2015, in collaboration with several industrial partners, we started the Magellan project at the University of Wisconsin-Madison, with the goal of developing EM solutions [3]. Numerous works have studied EM, but most of them develop *EM algorithms*. In contrast, we seek to build *EM systems*, as we believe such systems are critical for advancing the EM field. Among others, they can help evaluate EM algorithms, integrate R&D efforts, and make practical impacts, the same way systems such as System R, Ingres, Apache Hadoop, and Apache Spark have helped advanced the fields of relational data management and Big Data.

Of course, Magellan is not the first project to build EM systems. Many such systems have been developed (we recently discussed 18 non-commercial systems and 15 commercial ones [3]). However, as far as we can tell, virtually all of these systems have been built as *stand-alone monolithic EM systems*, or parts of larger stand-alone monolithic systems that perform data cleaning and integration [1–3].

In contrast, *Magellan employs a radically different system building template for EM, by leveraging ideas from the field of data science (DS).* While DS is still "young", several common themes have emerged. First, for many DS tasks, there is a general consensus that it is not possible to fully automate the two stages of developing and productionizing DS workflows. As a result, many step-by-step guides that tell users how to execute the above two stages have been developed. Second, many "pain points" in these guides, i.e., steps that are time consuming for users, have been identified, and (semi)-automated tools have been developed to reduce user effort. Finally, tools have been designed to be highly interoperable, forming a growing ecosystem of DS tools. Popular examples of such ecosystems include PyData, the ecosystem of 138,000+ interoperable Python packages (as of May 2018), R, tidyverse, and many others [4].

We observed that EM bears strong similarities to many DS tasks [3] (see Section 3). As a result, we leveraged the above ideas to build a new kind of EM systems. Specifically, we develop how-to guides that tell a user how to execute an EM process step by step, identify the "pain points" in the guides, then develop tools to address these pain points. We design tools to interoperate and form a part of PyData.

As described, the notion of "system" in Magellan has changed. It is no longer a stand-alone monolithic system such as RDBMSs or most current EM systems. Instead, *this new "system" is actually an ecosystem of interoperable EM tools, situated in a larger ecosystem of DS tools, together with guides that tell users how to use these tools to perform EM.*

Since Summer 2015 we have pursued the above EM agenda, and developed a small ecosystem of EM tools and guides. This raised many research challenges, which we have tried to address [4]. We have applied these tools to solve 21 real-world EM problems in domain sciences and industry. Finally, we have proposed to apply the Magellan system building template to other data integration tasks, such as schema matching, data cleaning, and information extraction [4].

In this paper we describe the above progress, focusing on the system aspects and on how ideas from the field of data science have been adapted to the EM context. Specifically, we seek to answer the following questions.

First, is this new system template promising? Is it easy to build and maintain? Can we use it to solve practical EM problems? We show that this approach is indeed promising. With a relatively small team, we can build and maintain a growing ecosystem of EM tools, and used it to solve a broad range of EM problems.

Second, are the main components of this new system truly necessary? We argue that the answer is yes. We discuss why it is critically important to have how-to guides, an ecosystem of tools (instead of a few stand-alone monolithic ones), and tool interoperability.

Third, what is the "right" ecosystem of tools? We started out building into PyData. Over the past few years has our "vision" changed? We argue that starting with PyData was the correct decision, but that the "right" ecosystem will eventually have to be bigger than PyData. Among others, it will contain interoperable microservices, which are easily deployable on the cloud, using containers and execution orchestrators such as Dockers and Kubernetes. This is a new trend in the field of DS [5, 6] that we argue can be readily adapted to building EM tools.

Fourth, what are the system challenges for this new system template? We discuss challenges that we have observed in terms of building and maintaining such tools, of making tools interoperate, and of horizontal and vertical scaling, on and off the cloud.

Fifth, what is the role of machine learning (ML) in such systems? We argue that ML plays a key role in building EM tools, but that we must address significant challenges in order to use it effectively. Among others, we show that it must be coupled with hand-crafted rules, effective user interaction, and Big Data scaling, in order to achieve its full potentials.

Finally, has our work deepened our understanding of EM? Do we think this new system template is the way to go for EM? Our answer is a clear yes. We argue that

EM is fundamentally very different from relational data management, and thus necessitates a very different kind of system architecture. In particular, EM "in the wild" appears very messy, with users wanting to try all kinds of EM workflows, and changing what they want to do on the fly, depending on what they have just learned. Put differently, many real-world EM projects are really a "conversation" between the EM team and the domain expert team, which moves forward as new results are produced and discussed. The new system template is well suited for facilitating this conversation.

The closest work to this paper is [4]. But in that work we did not discuss the system aspects of Magellan. Instead, we argued that the same system template (i.e., ecosystem and how-to guides) may also apply to other data integration tasks, such as cleaning, extraction, etc.

During the 70s to 90s our community extensively studied the system aspects of RDBMSs, and unquestionably such works helped push the field forward. In contrast, most existing EM works have focused on algorithmic solutions. We have argued for devoting more attention to the system aspects of EM, so that we can make faster progress. This includes building EM systems, discussing what kinds of system templates are appropriate, evaluating such systems in practice, and sharing the lessons learned. *This paper contributes to, and significantly advances this line of system work for EM,* by sharing our experience and lessons learned in building and evaluating a new kind of EM system in the past few years. More information about Magellan can be found at sites.google.com/site/anhaidgroup/projects/magellan.

2 PRELIMINARIES

In this section we briefly introduce the EM problem, then discuss related work.

The Entity Matching Problem: This problem, also known as entity resolution, record linkage, data matching, etc., has received enormous attention (see [1, 2, 7, 8] for recent books, tutorials, and surveys). 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. [1].

When matching two tables A and B, considering all pairs in $A \times B$ often takes very long. So users often execute a blocking step followed by a matching step [1]. *The blocking step* employs heuristics to quickly remove obviously non-matched tuple pairs (e.g., persons residing in different states, see Figure 1). *The matching step* applies a matcher to the remaining pairs to predict matches.

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.

Related Work in Entity Matching: The vast body of work in EM falls roughly into three groups: algorithmic, human-centric, and system. Most EM works develop *algorithmic solutions* for blocking and matching, exploiting rules, learning, clustering, crowdsourcing, external data, etc. [1, 2, 7]. The focus is on improving accuracy, minimizing runtime, and minimizing cost (e.g., crowdsourcing fee), among others [2, 8].

A smaller but growing body of EM work (e.g., those at HILDA workshops) studies *human-centric* challenges, such as crowdsourcing, effective user interaction, and user behavior during the EM process [9].

The third group of EM work develops *EM systems*. In 2016 we found 18 non-commercial systems (e.g., D-Dupe, DuDe, Febrl, Dedoop, Nadeef) and 15 commercial ones (e.g., Tamr, Informatica, Data Ladder, IBM InfoSphere) [3, 10]. Most of them are stand-alone monoliths. Our recent work [3] discusses their limitations and introduces Magellan, which builds an ecosystem of EM tools. Few works if any have discussed the system aspects of EM (e.g., what is the "right" system architecture? what are the design principles?). Both the book [1] and a recent work of ours [3] have touched upon some of these issues.

Related Work in Data Integration: The field of data integration (DI), which subsumes EM, has had a long history [7, 11]. It developed a range of DI system architectures in the 90s and 00s: global-as-view (GAV), local-as-view (LAV), GLAV, peer-to-peer, best-effort, data space, etc. [7]. Prominent DI system projects at that time include Garlic, Tsimmis, and Information Manifold [12–14], and prominent recent commercial efforts include Tamr and Trifacta [15, 16].

As far as we can tell, most of these works adopt a stand-alone system architecture, not an ecosystem of tools as in Magellan. Further, EM was either not discussed or viewed as just a small set of operators. Thus, our Magellan work is complementary to these works, and our ecosystem-of-tool approach can potentially be applied to other DI problems (e.g., schema matching, data cleaning), as we discuss in Section 6.

Related Work in Data Science: Magellan borrows ideas from the field of data science (DS). This field has been growing rapidly, and system related challenges have been discussed in a range of communities, from academia

to those of practitioners. There are four DS trends that that are most relevant to our work in Magellan. First, there are efforts to build ecosystems of on-premise interoperable DS tools. Such ecosystems can be general, e.g., R, PyData, tidyverse, or domain-specific (e.g., Bioconductor) [4]. Magellan was inspired by these works.

Second, many DS projects started to adopt a microservice software architecture, where the code is decomposed into a set of self-contained but interoperable services, each doing just one task [5]. Tools have been developed to package such services for easy deployment (e.g., Docker) and to coordinate their execution (e.g., Kubernetes) [6]. All of these make it ever easier for many DS projects to move to the cloud. In Section 6 we argue that an ecosystem of EM tools should also move in this direction.

Third, many DS projects use machine learning, and system aspects of ML deployment have been widely discussed (e.g., at SysML, NIPS, ICML, SIGMOD, VLDB). Magellan studies how to use ML for EM. Thus it can benefit from DS work in this direction and in return can contribute to it.

The final interesting trend is the growing interest in academia, especially in the software engineering community, in studying ecosystems of software tools (e.g., [17–27]). So far however there has been very little work in the database community that studies how to build ecosystems of data tools. Our hope is that the work in Magellan will help grow this interest.

3 THE MAGELLAN AGENDA

In this section we first argue that EM can be viewed as a special class of DS problems. So it can be highly promising to apply DS system building templates to EM systems. Next, we describe the system building template of PyData. Finally, we build on it to suggest a system building agenda for Magellan.

Viewing EM as a Special Class of DS Problems: Most current EM works view EM as a problem of two steps: blocking and matching [1]. (Some recent works did consider the pre-processing of the data, e.g., data cleaning, and post-processing, e.g., clustering and merging matches [1, 2, 15].)

Using this view, EM systems have typically been built in a way similar to RDBMSs. Such a system has a set of logical operations (e.g., blocking, matching) with multiple physical implementations. Given an EM workflow (composing of these operations) specified by the user using a GUI or a declarative language, the system translates the workflow into an execution plan, then optimizes and executes this plan.

In Magellan we view EM as a special class of DS problems, for the following reasons.

They share the same two stages: Many DS problems focus on developing an accurate DS workflow then productionizing it. Many EM problems can also be viewed as developing an accurate EM workflow (using data samples) then executing it in production on the entirety of the data (see Section 4 for an example).

The two stages raise many similar challenges: The development stage needs to effectively engage the user to develop an accurate workflow. This often raises challenges in data exploration, profiling, understanding, cleaning, model fitting and evaluation, etc. The production stage needs to execute the workflow on a large amount of data, raising challenges in scaling, logging, crash recovery, monitoring, etc. These challenges are remarkably similar for many DS and EM problems.

The two stages use many similar techniques: To address the above challenges, many DS and EM problems use the same set of techniques, e.g., ML, visualization, effective user interaction, and Big Data scaling such as Hadoop and Spark.

Thus, we believe EM can be viewed as a special class of DS problems, which focuses on finding the semantic matches, e.g., (David Smith, UWM) and (D. Smith, UW-Madison). If so, perhaps certain system building templates in the DS world can be effectively applied to build EM systems. We focus in particular on the system templates of PyData, which we briefly describe next.

The System Building Template of PyData: While PyData has been rapidly growing, we are not aware of any explicit description of its "system template". But our examination reveals three important underlying ideas:

How-to guides for users: First, PyData developers do not assume that DS problems can be solved automatically end to end. Users often must be "in the loop" and often do not know what to do, how to start, etc. As a result, developers provide detailed how-to guides that tell users how to solve a DS problem, step by step. Numerous guides have been developed, described in books, articles, Jupyter notebooks, training camps, blogs, tutorials, etc.

It is important to note that such a guide is *not* a user manual on how to use a tool. Rather, it is a step-by-step instruction to the user on how to start, when to use which tools, and when to do what manually. Put differently, it is an (often complex) algorithm for the user to follow. (See Section 4 for an example.)

Tools for pain points of the guides: Even without tools, users should be able to follow a guide and manually execute all the steps to solve a DS problem. But some of the steps can be very time consuming. PyData developers have identified such "pain point" steps and developed (semi)automatic tools to reduce the human effort.

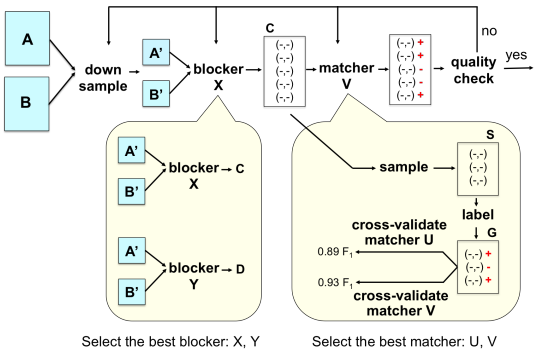

Figure 2: The steps of the guide for PyMatcher.

Tools are designed to be atomic and interoperable: PyData developers put a lot of effort into developing tools that are atomic (i.e., each tool does just one thing) and interoperable. This creates a growing ecosystem of tools over time.

Our Agenda: Using the system template of PyData, we developed the following agenda for Magellan. First, we identify common EM scenarios. Next, we develop how-to guides to solve these scenarios end to end.

Then we identify the pain points in the guides and develop (semi)automatic tools for the pain points. We will design tools to be atomic and interoperable, and design them as a part of the PyData ecosystem. We will also use ML where appropriate. Developing these tools will raise research challenges, which we will address.

Finally, we will work with users (e.g., domain scientists, companies, students) to evaluate our EM systems.

In the past 3.5 years we have tried to realize the above agenda in two major thrusts: PyMatcher and CloudMatcher, which we describe next.

4 PYMATCHER

We now describe PyMatcher, an EM system developed for power users (those who know programming, ML, and EM). We discuss system development, real-world applications, and lessons learned.

4.1 System Development

Problem Scenarios: In this first thrust of Magellan, we consider an EM scenario that commonly occurs in practice. In this scenario, a user U wants to match two tables (e.g., see Figure 1), with as high matching accuracy as possible, or with accuracy exceeding a threshold (e.g., at least 90% precision and recall). U is a "power user" who knows programming, EM, and ML (we consider "lay users" later in Section 5).

Developing How-to Guide: We developed an initial guide based on our experience, then kept refining it based on user feedback and on watching how real users do EM. As of Nov 2018, we have developed a guide for the above EM scenario, which consists of two smaller guides for the development and production stages, respectively. We now focus on the guide for the development stage (discussing the guide for the production stage later in this section).

This guide (which is illustrated in Figure 2) heavily uses ML. To explain it, suppose user U wants to match two tables A and B, each having 1 million tuples. Trying to find an accurate workflow using these two tables would be too time consuming, because they are too big. Hence, U will first "down sample" the two tables to obtain two smaller tables A' and B', each having 100K tuples, say (see the figure).

Next, suppose the EM system provides two blockers X and Y. Then U experiments with these blockers (e.g., executing both on Tables A' and B' and examining their output) to select the blocker judged the best (according to some criterion). Suppose U selects blocker X. Then next, U executes X on Tables A' and B' to obtain a set of candidate tuple pairs C.

Next, U takes a sample S from C, and labels the pairs in S as "match"/"no-match" (see the figure). Let the labeled set be G, and suppose the EM system provides two learning-based matchers U and V (e.g., decision trees, logistic regression). Then U uses the labeled set G to perform cross validation for U and V. Suppose V produces higher matching accuracy (such as F_1 score of 0.93, see the figure). Then U selects V as the matcher, and applies V to the set C to predict "match"/"no-match", shown as "+" or "-" in the figure. Finally, U may perform quality check (by examining a sample of the predictions and computing the resulting accuracy), then go back and debug and modify the previous steps as appropriate. This continues until U is satisfied with the accuracy of the EM workflow.

Developing Tools for the Steps of the Guide: Over the past 3.5 years 13 developers have developed tools for the steps of the above guide (see Appendix A for details). As of June 2018, PyMatcher consists of 6 Python packages with 37K lines of code and 104 commands (and is open sourced [3]). As far as we can tell, PyMatcher is the most comprehensive open-source EM system today, in terms of the number of features it supports.

Principles for Developing Tools & Packages: Recall that each tool is roughly equivalent to a Python command, and tools are organized into Python packages. We adopted five principles for developing tools and packages:

(1) They should *interoperate* with one another, and with existing PyData packages.

(2) They should be *atomic*, i.e., does only one thing.
(3) They should be *self-contained*, i.e., they can be used by themselves, not relying on anything outside.
(4) They should be *customizable*.
(5) They should be *efficient* for both human and machine.

We now illustrate these principles. As an example of facilitating *interoperability* among the commands of different packages, we use only generic well-known data structures such as Pandas dataframe to hold tables (e.g., the two tables A and B to match, the output table after blocking, etc.).

Designing each command, i.e., tool, to be *"atomic"* is somewhat straightforward. Designing each package to be so is more difficult. Initially, we designed just one package for all tools of all steps of the guide. Then as soon as it was obvious that a set of tools form a coherent stand-alone group, we extracted it as a new package. However, this extraction is not always easy to do, as we will discuss soon.

Ignoring self-containment for now, to make tools and packages highly *customizable*, we expose all possible "knobs" for the user to tweak, and provide easy ways for him/her to do so. For example, given two tables A and B to match, PyMatcher can automatically define a set of features (e.g., $jaccard(3gram(A.name), 3gram(B.name))$). We store this set of features in a global variable F. We give users ways to delete features from F, and to declaratively define more features then add them to F.

As an example of making a tool, i.e., a command, X *efficient for a user*, we can make X easy to remember and specify (i.e., it does not require the user to enter many arguments). Often, this also means that we provide multiple variations for X, because each user may best remember a particular variation.

Command X is *efficient for machine* if it minimizes runtime and space. For instance, let A and B be two tables with schema (id,name,age). Suppose X is a blocker command that when applied to A and B produces a set of tuple pairs C. Then to save space, X should not use (A.id, A.name, A.age, B.id, B.name, B.age), but only (A.id, B.id) as the schema of C.

If so, we need to store the "metadata information" that there is a key-foreign key (FK) relationship between tables A, B and C. Storing this metadata in the tables themselves is not an option if we have already elected to store the tables using Pandas dataframe (which cannot store such metadata, unless we redefine the dataframe class). So we can use a stand-alone catalog Q to store such metadata for the tables.

But this raises a problem. If we use a command Y of some other package to remove a tuple from table A, Y is not even aware of catalog Q and so will not modify the metadata stored in Q. As a result, the metadata is now incorrect: Q still claims that a FK relationship exists between tables A and C. But this is no longer true.

To address this problem, we can design the tools to be *self-contained*. For example, if a tool Z is about to operate on table C and needs the metadata "there is a FK constraint between A and C" to be true, it will first check that constraint. If the constraint is still true, then Z will proceed normally. Otherwise, Z outputs a warning that the FK constraint is no longer correct, then stops or proceeds (depending on the nature of the command). Thus, Z is self-contained in that it does not rely on anything outside to ensure the correctness of the metadata that it needs.

Trade-Offs Among the Principles: It should be clear by now that the above principles often interact and conflict with one another. For example, as discussed, to make commands interoperate, we may use Pandas data frames to hold the tables, and to make commands efficient, we may need to store metadata such as FK constraints. But this means the constraints should be stored in a global catalog. This makes extracting a set of commands to create a new package is difficult, because the commands need access to this global catalog.

There are many examples like this, which together suggest that designing an "ecosystem" of tools and packages that follow the above principles require making trade-offs. We have made several such trade-offs in designing PyMatcher. But obtaining a clear understanding of these trade-offs and using it to design a better ecosystem is still ongoing work.

The Production Stage: So far we have focused on the development stage for PyMatcher, and have developed only a basic solution for the production stage. Specifically, we assume that after the development stage, the user has obtained an accurate EM workflow W, which is captured as a Python script (of a sequence of commands). We have developed tools that can execute these commands on a multi-core single machine, using customized code or Dask (which is a Python package developed by Anaconda that can be used to quickly modify a Python command to run on multiple cores, among others). We have also developed a how-to guide that tells the user how to scale using these tools.

4.2 Real-World Applications

From 2016 to date PyMatcher has been successfully applied to multiple real-world EM applications in both industry and domain sciences. It has been pushed into production in most of these applications, and has attracted

Problem Owner	Problem Type	Notable Result	In Production?	Team	Other
Walmart	Debug a system in production that matches products	Improved recall by 34%, reduced precision by 0.65%	Yes	1 student, 1 employee	Funding
Recruit Holdings	Matching names of stores, companies, properties	Reported 98.9% accuracy on matching 10K store names	Yes	Multiple employees	Press release
Johnson Controls	Matching suppliers	Precision and recall in 96-100%	Unknown	1 student	Funding
Marshfield Clinic	Matching drugs	99.2% precision and 95.3% recall	Yes	1 student, 1 employee	Paper
Economics (UW)	Matching grants. Build a better EM pipeline	Precision in [96.7%, 98.8%], recall in [94.2%, 97.1%] A system in production achieves 100% precision, recall in [65.1%, 71.8%]	Not yet	2 students	Paper published, funding from UW
Land Use (UW)	Matching cattle ranches. Build a better EM pipeline	Precision in [89.7%, 99.0%], recall in [79.2%, 92.2%] A system in production achieves precision in [94.9%, 100%], recall in [29.4%, 46.6%]	Yes	1 student, 1 programmer, 2 staff persons	Paper planned, funding from UW
Biomedicine (UW)	Matching ontology terms	metasra.biostat.wisc.edu	Yes	1 student	Paper published
Limnology (UW)	Matching table attributes	High-value datasets created from multiple datasets	Yes	2 students	Funding from UW

Table 1: Real-world deployment of PyMatcher.

significant funding (e.g., $950K from UW-Madison, $1.1M from NSF, and $480K from industry). It has also been used by 400+ students in 5 data science classes at UW-Madison. Finally, it has resulted in multiple publications, both in the database field and in domain sciences [4].

Table 1 summarizes the real-world applications. The 1st column shows that PyMatcher has been used in a variety of companies and domain sciences. The 2nd column shows that PyMatcher has been used for 3 purposes: debugging an EM pipeline in production (Walmart), building a better EM pipeline than an existing one (economics, land use), and integrating disparate datasets (e.g., Recruit, Marshfield Clinic, limnology).

The 3rd column shows the main results. *This column shows that* PyMatcher *found EM workflows that were significantly better than the EM workflows in production in three cases: Walmart, Economics (UW), and Land Use (UW).* The 4th column indicates that, based on those results, PyMatcher has been put into production in 6 out of 8 applications. This is defined as either (a) PyMatcher is used in a part of an EM pipeline in production, or (b) the data resulted from using PyMatcher has been pushed into production, i.e., being sent to and consumed by real-world customers.

The 5th column shows that in all cases that we know of, PyMatcher does not require a large team to work on it (and the teams are only part-time). The final column lists additional notable results. (Note that funding from UW came from highly selective internal competitions.)

To illustrate the above applications, Appendix B describes in more details "Land Use", the latest application in which PyMatcher has been put into production.

4.3 Lessons Learned

We now discuss the lessons learned from working with the above applications.

How-to Guide: Recall that these guides are captured in documents and Jupyter notebooks that tell the user how to execute the EM process step by step. *We find these guides indispensible.* They provide assurance to our customers that we can help them do EM end to end. They provide a common vocabulary and roadmap for everyone on the team to follow, regardless of their background. Even for the EM steps where we currently do not have tools, the guide still helps enormously, because it tells the customers what to do, and they can do it manually or find some external tools to help with it. *We simply cannot emphasize enough how important it is to have such guides.*

Surprisingly, our initial how-to guides (e.g., [3]) were woefully inadequate. Users kept trying new things, and there are so many aspects of EM that have been underexplored (see below). As a result, we had to constantly revise and expand our guides.

Messy EM: A related point is that EM is so much messier than we thought. Fundamentally it was a "trial and error" process, where users kept experimenting until they find a satisfactory EM workflow. As a result, users tried all kinds of workflows, customization, data processing, etc.

In all of the applications that we have worked with, it was impossible to tell what users wanted to try next, or what complications would come up next. For example, in the "Land Use (UW)" project described earlier, after blocking and matching two tables A and B, users realized they should have added more data to these tables. They also scared us when they said, after we had done a lot of matching, that despite already doing EM for 3 years, they were still unsure about how the match definition applies to certain tuple pairs. A recent paper of ours

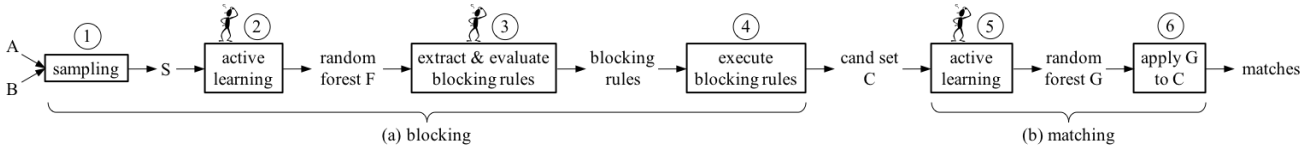

Figure 3: The workflow of Falcon, where a lay user labels tuple pairs as match/no-match in Steps ②, ③, and ⑤.

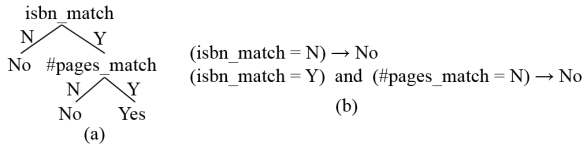

Figure 4: (a) A decision tree learned by Falcon and (b) blocking rules extracted from the tree.

describes similar and many more "horror stories" in the "Economics (UW)" project [28].

Monolithic Systems vs. Ecosystems of Tools: Because EM is so messy and users want to try so many different things, we found that an ecosystem of tools is ideal. For every new scenario that users want to try, we can quickly put together a set of tools and a mini how-to guide that they can use. This gives us a lot of flexibility.

Many "trial" scenarios require only a part of the entire PyMatcher ecosystem. Having an ecosystem allows us to very quickly pull out the needed part, and popular parts end up being used everywhere. For example, the packages py_stringmatching and py_stringsimjoin (described in Appendix A) are so useful in many projects (not just in EM) that they ended up being installed on Kaggle, a popular data science platform.

Extensibility is also much easier with an ecosystem. For example, recently we have developed a new matcher that uses deep learning to match textual data [29]. We used PyTorch, a new Python library, to develop it, released it as a new Python package in the PyMatcher ecosystem, then extended our guide to show how to use it. This smoothly extended PyMatcher with relatively little effort.

Clearly we can try to achieve the above three desirable traits (flexibility/customizability, partial reuse, and extensibility) with monolithic stand-alone systems for EM, but our experience suggests it would be significantly harder to do so than with an ecosystem of tools.

Challenges: Our experience identifies the following major challenges. First, it is difficult to design the ecosystem of tools to satisfy the five principles describe earlier (which state that tools and packages must be interoperable, atomic, self-contained, customizable, and efficient). We have only just started to understand these issues, and have developed only preliminary solutions.

Second, PyData, the ecosystem of on-premise Python packages, is a great starting point, but is not the eventual ecosystem for PyMatcher. For example, EM often requires a set of users to label a set of tuple pairs (as match/no-match). Such labeling is best done via a cloud-based tool. As another example, the production stage needs tools that can process data fast on a machine cluster. Such tools do not fit the mold of on-prem Python packages. We discuss our view of the eventual ecosystem for PyMatcher in Section 6.

Third, machine learning "in the wild" is surprisingly difficult, even though so far we have used only mainstream, well-known ML techniques. We definitely need more support, such as how to minimize labeling effort, how to label accurately with multiple people, how many training examples to sample, how to debug ML algorithms, etc.

The final (related) point is that training users in ML/EM (so that they can use PyMatcher) is difficult. We learned this in multiple projects, where training domain scientists took forever. As a result, we believe that if the EM need is one-shot, short-term, or exploratory, then self-service EM systems that require no ML/EM knowledge from users would be the best way to proceed. The CloudMatcher thrust of Magellan, which we describe next, builds such self-service systems.

5 CLOUDMATCHER

We discuss CloudMatcher development, real-world applications, and lessons learned.

5.1 System Development

Problem Scenarios: We use the term "lay user" to refer to a user who does not know programming, ML, or EM, but understands what it means to be match (and thus can label tuple pairs as match/no-match). Our goal is to build a system that such lay users can use to match two tables A and B. We call such systems self-service EM systems.

Developing an EM System for a Single User: In a recent work [30] we have developed Falcon, a self-service EM system that can serve a single user (in [30] we call this system "hands-off" instead of "self-service").

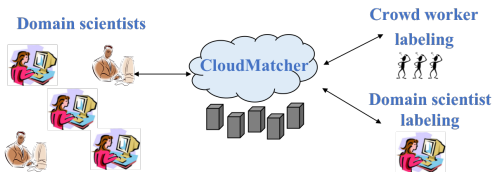

Figure 5: Self-service EM with CloudMatcher.

Since CloudMatcher builds on Falcon, we begin by briefly describing Falcon.

To match two tables A and B, *like most current EM solutions,* Falcon *performs blocking and matching, but it makes both stages self-service* (see Figure 3). In the blocking stage (Figure 3.a), it takes a sample S of tuple pairs (Step ①), then performs active learning with the lay user on S (in which the user labels tuple pairs as match/no-match) to learn a random forest F (Step ②), which is a set of n decision trees. The forest F declares a tuple pair p a match if at least αn trees in F declare p a match (where α is pre-specified).

In Step ③, Falcon extracts all tree branches from the root of a tree (in random forest F) to a "No" leaf as candidate blocking rules. For example, the tree in Figure 4.a predicts that two book tuples match only if their ISBNs match and the number of pages match. Figure 4.b shows two blocking rules extracted from this tree. Falcon enlists the lay user to evaluate the extracted blocking rules, and retains only the precise rules. In Step ④, Falcon executes these rules on tables A and B to obtain a set of candidate tuple pairs C. This completes the blocking stage (Figure 3.a). In the matching stage (Figure 3.b), Falcon performs active learning with the lay user on C to obtain another random forest G, then applies G to C to predict matches (Steps ⑤ and ⑥).

As described, Falcon is well suited for lay users, who only have to label tuple pairs as match/no-match. We implemented Falcon as CloudMatcher 0.1 and deployed as shown in Figure 5, with the goal of providing self-service EM to domain scientists at UW. Any scientist wanting to match two tables A and B can go the homepage of CloudMatcher, upload the tables, then label a set of tuple pairs (or ask crowd workers say on Mechanical Turk to do so). CloudMatcher uses the labeled pairs to block and match, as described earlier, then returns the set of matches between A and B.

Developing an EM System for Multiple Users: We soon recognized however that CloudMatcher 0.1 does not scale, because it can execute only one EM workflow at a time. So we designed CloudMatcher 1.0, which can efficiently execute multiple concurrent EM workflows (e.g., submitted by multiple scientists at the same time). Developing CloudMatcher 1.0 was highly challenging [31]. Our solution was to break each submitted EM workflow into multiple DAG fragments, where each fragment performs only one kind of task, e.g., interaction with the user, batch processing of data, crowdsourcing, etc. Next, we execute each fragment on an appropriate execution engines. We developed 3 execution engines: user interaction engine, crowd engine, and batch engine. To scale, we interleave the execution of DAG fragments coming from different EM workflows, and coordinate all of the activities using a "metamanager". See a recent workshop paper [31] for more details.

Providing Multiple Basic Services: CloudMatcher 1.0 implemented only the above rigid Falcon EM workflow. As we interacted with real users, however, we observed that many users want to flexibly customize and experiment with different EM workflows. For example, a user may already know a blocking rule, so he or she wants to skip the step of learning such rules. Yet another user may want to use CloudMatcher just to label tuple pairs (e.g., to be used in PyMatcher).

So we developed CloudMatcher 2.0, which extracts a set of basic services from the Falcon EM workflow and makes them available on CloudMatcher, then allows users to flexibly combine them to form different EM workflows (including the original Falcon one). Appendix C shows the list of services that we currently provide. *Basic services* include uploading a dataset, profiling a dataset, edit the metadata of a dataset, sampling, generating features, training a classifier, etc. We have combined these basic services to provide *composite services*, such as active learning, obtaining blocking rules, and Falcon (see the bottom of the table). For example, the user can invoke the "Get blocking rules" service to ask CloudMatcher to suggest a set of blocking rules that he/she can use. As another example, the user can invoke the "Falcon" service to execute the end-to-end Falcon EM workflow.

5.2 Real-World Applications

From 2017 to date CloudMatcher has been successfully applied to multiple EM applications, and has attracted commercial interest. It has been in production at American Family Insurance since Summer 2018, and is being considered for production at two other major companies.

Table 2 summarizes CloudMatcher's performance on 13 real-world EM tasks. The first two columns show that CloudMatcher has been used in 5 companies, 1 non-profit, and 1 domain science group, for a variety of EM tasks. The next two columns show that CloudMatcher was used to match tables of varying sizes, from 300 to 4.9M tuples.

Ignoring the next two columns on accuracy, let us zoom in on the three columns under "Cost" in Table

Problem Owner	Problem Type	Table A	Table B	Precision (%)	Recall (%)	Cost			Time		
						Questions	Crowd	Compute	User/Crowd	Machine	Total
Fortune 500 Company	Phoenix customers	300	300	96.4	99.03	160	-	$2.33	9m	5m	14m
	Commercial insurance policy holders	1,049	17,572	96.15	97.22	321	-	$2.33	18m	25m	43m
	Commercial farm/ranch policy members	109,974	4,922,505	99.5	95	780	-	$13.96	50m	4h 58m	5h 48m
	Vehicles	18,938	72,898	66.02 – 80.02	81.65 – 93.15	851	-	$7.00	2h	46m	2h 46m
	Drivers	790	634	99.86	94.89	250	-	$2.33	10m	8m	18m
Johnson Controls International	Addresses	90,673	231,081	93.22 – 95.72	76.93 – 81.01	1200	$72	-	36h 48m	38m	37h 26m
	Vendors	50,295	50,292	29.95 – 38.04	91.89 – 98.10	1160	$69.60	-	30h 31m	58m	31h 29m
	Vendors (no Brazil)	28,152	28,149	95.44 – 97.75	88.82 – 92.41	1200	$72	-	22h 19m	22m	22h 41m
UW Health	Doctors & staff	1,786	1,786	99.66	98.18	1200	-	$4.66	50m	15m	1h 5m
Large Data Integration Company	Persons	48,119	48,119	100 – 100	98.42 – 100	462	-	$7.00	36m	1h 35m	2h 11m
Marshfield Clinic	Drugs	446,048	440,048	99.14 – 99.63	98.45 – 99.14	1162	-	-	1h 10m	8h 40m	9h 50m
Non-profit Org	Elected officials	9,751	706,878	93.75 – 96.32	95.50 – 97.76	960	$57.60	-	23h 14m	23m	23h 37m
Domain Science	UMetrics economics	2,616	21,530	94.5 – 96.5	98.12 – 99.21	680	$61.20	-	23h 12m	12m	23h 24m

Table 2: Real-world deployment of CloudMatcher.

2. The first column ("Questions") lists the number of questions CloudMatcher had to ask, i.e., the number of tuple pairs to be labeled. This number ranges from 160 to 1200 (the upper limit for the current CloudMatcher).

In the next column ("Crowd"), a cell such as "$72" indicates that for the corresponding EM task, CloudMatcher used crowd workers on Mechanical Turk to label tuple pairs, and it cost $72. A cell "-" indicates that the task did not use crowdsourcing. It used a single user instead, typically the person who submitted the EM task, to label, and thus incurred no monetary cost.

In the third column ("Compute"), a cell such as "$2.33" indicates that the corresponding EM task used AWS, which charged $2.33. A cell such as "-" indicates that the EM task used a local machine owned by us, and thus incurred no monetary cost.

Turning our attention to the last three columns under "Time", the first column ("User/Crowd") lists the total labeling time, either by a single user or by the Mechanical Turk crowd. We can see that when a single user labeled, it was typically quite fast, with time from 9m to 2h. When a crowd labeled, time was from 22h to 36h (this does not mean crowd workers labeled non-stop and took that long, it just meant Mechanical Turk took that long to finish the labeling task). These results suggest that CloudMatcher can execute a broad range of EM tasks with very reasonable labeling time from both users and crowd workers. The next two columns under "Time" show the machine time and the total time.

We now zoom in on the accuracy. The columns "Precision" and "Recall" show that in all cases except three, CloudMatcher achieves high accuracy, often in the 90 percentage. The three cases of limited accuracy are "Vehicles", "Addresses", and "Vendors". A domain expert at American Family Insurance (AmFam) labeled tuple pairs for "Vehicles". But the data was so incomplete that even he was uncertain in many cases on whether the tuple pair match. At some point he realized that he had incorrectly labeled a set of tuple pairs, but CloudMatcher provided no way for him to "undo" the labeling, hence the low accuracy. This EM task is currently being re-executed at AmFam.

For "Vendors", it turned out that the portion of data that consists of Brazilian vendors is simply incorrect: the vendors entered some generic addresses instead of their real addresses. As a result, even users cannot match such vendors. Once, we removed such vendors from the data, the accuracy significantly improved (see the row for "Vendors (no Brazil)"). It turned out that "Addresses" had similar dirty data problems, which explained the low recall of 76-81%.

5.3 Lessons Learned

We now discuss the lessons learned from working with the above applications.

The Promise of Self-Service EM: The idea of just having to label a set of tuple pairs sounded very appealing, and our customers were enthusiastic about using CloudMatcher. CloudMatcher also seemed to perform well on a broad range of EM tasks. We concluded that self-service EM is a great way to start the "EM journey". If a user has only one-shot or short-term EM needs, then perhaps it is best to start out trying CloudMatcher. If CloudMatcher already achieves the desired accuracy, then

great. Otherwise, the user can look into using PyMatcher, the more powerful system.

Challenges in Data Cleaning and Labeling: Our experience also made clear that data cleaning is critical for EM (e.g., see the "Vendors" and "Addresses" cases). It is important that we can detect dirty data, isolate it, and then clean it, to maximize EM accuracy. But how to do so in a self-service fashion is still unclear.

We also want to reduce the number of pairs that users must label. In the case of matching two sets of strings (a special case of EM), we have developed Smurf, which removes the need to label to learn blocking rules. It only needs labeling to learn a random forest-based matcher [32]. This drastically reduces the labeling effort by 43-76%, yet achieving the same accuracy. We are currently extending Smurf to match tuples, and incorporating it into CloudMatcher.

Challenges in Developing a Monolithic System: Our biggest challenge, however, is that CloudMatcher *has been slowly growing into a big stand-alone monolithic EM system, exactly the kind of system we would like to avoid building.* Its code base is large (47K LOC, as discussed in Appendix D), and its many modules are highly interdependent. We found that it is increasingly hard to understand, maintain, and extend CloudMatcher. This is exacerbated by developing it in academia, where most students do not stay for long.

Another problem with CloudMatcher being a monolithic system is that we cannot easily use "pieces" of it. For example, most basic services of CloudMatcher (e.g., learning blocking rules, executing blocking rules, labeling tuple pairs, etc.) would be very helpful for both the development and production stages of PyMatcher. But to use any one of them, a PyMatcher user would need to install the entire CloudMatcher, too much overhead. It is also cumbersome to quickly get data into and out of CloudMatcher for some of these services (e.g., to move data between CloudMatcher and PyMatcher).

The above problems are not new. They commonly arise in stand-alone monolithic systems. But as we built and worked with CloudMatcher, we experienced them first hand. This experience further motivated the Magellan approach of developing ecosystems of interoperable tools.

Toward Cloud-Based Interoperable Microservices: To address the above "monolithic system" challenges, we are leveraging ideas from a recent trend in building data science systems. Specifically, many DS projects have started to adopt a microservice software approach, where the code is decomposed into a set of *microservices*, which are self-contained but interoperable services, each doing just one task [5]. Tools have been developed to easily deploy such services (e.g., Docker) and to coordinate their execution on the cloud (e.g., Kubernetes) [6].

In short, many DS projects are moving to the cloud, and a support infrastructure is emerging to help build "cloud native" applications, which are composed of interoperable microservices that can be smoothly deployed, executed, and scaled out on the cloud.

We are currently exploring how to redesign CloudMatcher *in this direction.* Specifically, we want to extract each basic service of CloudMatcher (e.g., data profiling, learning blocking rules, labeling, etc. see Table 4) as a microservice, then develop ways to combine their execution on the cloud (e.g., AWS) to provide the end-to-end EM service to users. *The new* CloudMatcher *will be a set of interoperable microservice tools, and thus will follow a system architecture similar to that of* PyMatcher.

6 DISCUSSION

Building on our discussion of PyMatcher and CloudMatcher, we now discuss what we have learned regarding a number of questions raised in the introduction.

Promise of the New Approach: Our experience has been that it is relatively easy to build and maintain such systems. PyMatcher was built mostly by 2 Ph.D. students, with the help from several hourly paid students and 1-2 other Ph.D. students. CloudMatcher was built mostly by 2 Ph.D. students, with the help from several hourly students.

PyMatcher was relatively easy to manage because its tools are self-contained Python packages, which can be developed, modified, released, and maintained in a way that is fairly independent of other packages. Initially, CloudMatcher was also relatively easy to manage, as its code base was highly modular and small. But in the past one year it has become increasingly harder to manage, as its monolithic code base has grown. Our hope is that by rebuilding CloudMatcher as a set of much smaller microservices, we will significantly reduce the complexity of managing it.

Our real-world deployment of PyMatcher and CloudMatcher has clearly demonstrated that they can be used to solve a range of practical EM problems.

The New Envisioned Magellan Ecosystem: We started out building into PyData. Over the past few years, however, our vision has changed. Figure 6 illustrates the new envisioned Magellan ecosystem. In this new ecosystem, we still have PyData tools and PyMatcher tools such as py_stringmatching, py_entitymatching, etc. These tools are on-premise. They can be downloaded and used on a local machine, the way numerous DS projects are being done today.

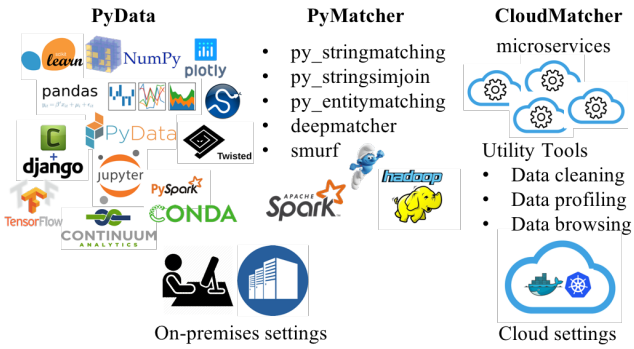

Figure 6: The new envisioned ecosystem of tools for Magellan.

But we will also have microservice tools that interoperate and cloud native. The new CloudMatcher will compose of multiple such services. We will build many other services, to perform data cleaning, profiling, browsing, etc. for EM. Most of these services can be easily deployed, executed, and scaled out on the cloud. Some of the tools and services will be self-service, i.e., very easy for lay users to use.

System Challenges: As described, this is a very different kind of "data systems" than what we have used to, such as RDBMSs. Such a data system is in fact an ecosystem of tools and microservices that is "at home" on the cloud. Building such ecosystem will raise numerous challenges, some of which we have discussed in this paper. Examples include satisfying and doing trade-offs among the five principles described in Section 4, horizontal and vertical scaling of services on the cloud, and understanding and managing interoperability challenges (e.g., data structures, metadata, missing values, data type mismatch, package version incompatabilities, etc.).

Machine Learning & Data Science: Our work makes clear that ML can be very beneficial to EM, mainly because it provides an effective way to capture complex matching patterns in the data, and to capture domain expert's knowledge about such patterns. ML is clearly at the heart of EM workflows supported by PyMatcher and CloudMatcher. In many real-world applications we have worked with, ML helps significantly improve recall while retaining high precision, compared to rule-based EM solutions.

Yet to our surprise, deploying even traditional ML techniques to solve EM problems already raises many challenges, such as labeling, coping with new data, etc. Our experience using PyMatcher also suggests that the most accurate EM workflows are likely to involve a combination of ML and rules. More generally, we believe ML must be used effectively in conjunction with handcrafted rules, visualization, and good user interaction, in order to realize its full potential.

In this paper we have shown how powerful ideas in DS can be used to solve EM. In the reverse direction, EM tools as developed in this paper can also be naturally integrated into the DS ecosystem of tools, and thus help solve DS problems (e.g., in data wrangling).

Understanding of EM & the New System Template: Our experience with Magellan (especially in working with a broad range of real-world users) has significantly deepened our understanding of EM "in the wild" and the new system template.

We now understand that EM in the wild is very messy, with users wanting to try all kinds of EM workflows and scenarios, and changing what they want to do on the fly, depending on what they have just learned. Put differently, *many EM projects in the wild are really a "conversation" between the EM team and the domain expert team, and this conversation moves forward as new results are produced and discussed.*

If this is the case, then it follows that it is critical to have how-to guides that tell both teams how to conduct this conversation, what to do first, what to do second, and so on.

Having multiple interoperable tools would allow users to quickly assemble new EM workflows and try out new things. It is also easier to develop, modify, and release independent self-contained tools than a large monolithic system. Fundamentally, the benefits of an ecosystem of EM tools (vs. monolithic systems) are the same as those that have been articulated in the past few years for microservice software architectures.

7 CONCLUSIONS

We have described Magellan, a project to build EM systems. We have discussed how Magellan borrowed ideas from the field of data science, to build a novel kind of EM systems. We described PyMatcher and CloudMatcher, EM tools for power users and lay users, and real-world applications of these tools. We reported on the lessons learned, and outline a new more powerful Magellan ecosystem, with on-premise Python packages and cloud-native microservices.

Going forward, we are working on extending PyMatcher and re-designing CloudMatcher as a set of interoperable cloud-native microservices. We are also looking for more real-world applications to "test drive" Magellan. Finally, we plan to apply the Magellan system building template to other data integration problems, such as schema matching, data exploration, and information extraction.

REFERENCES

[1] Peter Christen. *Data Matching*. Springer, 2012.

[2] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.

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

[4] AnHai Doan et al. Toward a system building agenda for Data Integration (and Data Science). *IEEE Data Eng. Bull.*, 41(2):35–46, 2018.

[5] Sam Newman. *Building microservices : designing fine-grained systems*. O'Reilly Media, Sebastopol, CA, 2015.

[6] Kelsey Hightower. *Kubernetes : up and running: dive into the future of infrastructure*. O'Reilly Media, Sebastopol, CA, 2017.

[7] AnHai Doan, Alon Y. Halevy, and Zachary G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[8] George Papadakis et al. Web-scale, Schema-Agnostic, End-to-End Entity Resolution. In *The Web Conference (WWW), Lyon, France, April*, 2018.

[9] Workshop on Human-In-the-Loop Data Analytics, http://hilda.io/.

[10] George Papadakis et al. The return of JedAI: End-to-End entity resolution for structured and semi-structured data. *PVLDB*, 11(12):1950–1953, 2018.

[11] IEEE Data Engineering Bulletin, Special Issue on Large-Scale Data Integration, 2018, http://sites.computer.org/debull/A18june/issue1.htm.

[12] Mary Tork Roth et al. The Garlic Project. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, page 557, 1996.

[13] Sudarshan S. Chawathe et al. The TSIMMIS Project: Integration of heterogeneous information sources. In *IPSJ*, pages 7–18, 1994.

[14] Alon Y. Levy et al. The world wide web as a collection of views: Query processing in the information manifold. In *VIEWS*, pages 43–55, 1996.

[15] Michael Stonebraker et al. Data Integration: The current status and the way forward. *IEEE Data Eng. Bull.*, 41(2):3–9, 2018.

[16] Joseph M. Hellerstein et al. Self-Service Data Preparation: Research to practice. *IEEE Data Eng. Bull.*, 41(2):23–34, 2018.

[17] Cláudia Maria Lima Werner. Building software ecosystems from a reuse perspective. In *International Workshop on Software Ecosystems*, 2009.

[18] Tom Mens et al. Analysing the evolution of social aspects of open source software ecosystems. In *International Workshop on Software Ecosystems*, volume 746, pages 1–14, 2011.

[19] Rodrigo dos Santos et al. A proposal for software ecosystems engineering. *CEUR Workshop Proceedings*, 746, 2011.

[20] Eric Yu et al. Understanding software ecosystems: A strategic modeling approach. *CEUR Workshop Proceedings*, 746, 01 2011.

[21] F.W. Santana et al. Towards the analysis of software projects dependencies: An exploratory visual study of software ecosystems. *CEUR Workshop Proceedings*, 987:1–12, 01 2013.

[22] J. Yates Monteith et al. Hadoop and its evolving ecosystem. *CEUR Workshop Proceedings*, 987, 06 2013.

[23] Spauwen Ruvar et al. Towards the roles and motives of open source software developers. *CEUR Workshop Proceedings*, 987, 06 2013.

[24] J. Yates Monteith et al. Scientific research software ecosystems. *ACM International Conference Proceeding Series*, 08 2014.

[25] International Workshop on Software Ecosystems, http://ceur-ws.org.

[26] Workshop on Software Ecosystem Architectures, http://wea.github.io/.

[27] ACM International Workshop on Software-Defined Ecosystems, https://dl.acm.org/citation.cfm?id=2756594.

[28] Pradap Konda et al. Performing entity matching end to end: A case study. In *EDBT*, 2019.

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

[30] Sanjib Das, Paul Suganthan G.C., AnHai Doan, Jeffrey F. Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1431–1446, New York, NY, USA, 2017. ACM.

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

[32] Paul Suganthan G. C. et al. Smurf: Self-Service String Matching Using Random Forests. *PVLDB*, 12(3), 2019.

A DEVELOPING TOOLS FOR PYMATCHER

We developed the tools for PyMatcher as follows. For each main step of the how-to guide (see Column A of Table 3), we first examined existing packages to find tools, i.e, commands, that can be used to support the step. Column B shows the major existing packages that we use. For example, for the step "read/write data", we find that the popular Pandas package provides many read/write commands that users can use. For "data exploration", we recommend commands in Pandas, pandas-profiling, and OpenRefine. For "blocking", we use Dask to scale up blocking commands. For "selecting a matcher", we use scikit-learn and pytorch, etc.

Next, we wrote our own Python code to develop more tools to support the steps of the guide (see Column C). For example, for "blocking", we developed multiple blockers: attribute equivalence, hash-based, rule-based, etc.

While developing the blockers, we found that they heavily use a large set of string tokenizers and similarity measures. So we wrote a Python package called py_stringmatching that implements these. Matching two sets of strings is also heavily used by the blockers, so we wrote another package called py_stringsimjoin that implements multiple methods to match two large sets of strings fast [11] (see Column C).

Finally, we identified "pain points" of the steps and developed tools to address them (see Column D). For example, intelligently down sampling two tables (e.g.,

Step of the How-to Guide (A)	Use Existing Packages (B)	Write Our Own Code (C)	Develop Tools for Pain Points (D)	Number of Commands (E)
Read/Write Data	pandas			6
Down Sample			Down sampler	1
Data Exploration	pandas pandas-profiling pandas-table OpenRefine			2
Blocking	Dask joblib	Multiple blockers py_stringmatching py_stringsimjoin	Blocking debugger	21
Sampling	pandas			1
Labeling	PyQt5		GUI labeler	2
Creating Feature Vectors	joblib	py_stringmatching	Automatic feature creation Manual feature creation	12
Matching	scikit-learn PyTorch XGBoost		Matching debuggers Deep learning-based matcher	20
Computing Accuracy	pandas			4
Adding Rules			Rule specification and execution	9
Managing Metadata			Catalog management	22

Main Packages: py_stringmatching, py_stringsimjoin, py_entitymatching, py_labeler, DeepMatcher

Table 3: Developing tools for the steps of the guide.

A and B to A' and B', see Figure 2) is tricky. So we developed a tool for it [3]. Debugging a blocker to assess its accuracy is very challenging. So we developed such a tool for the "blocking" step [3]. Column D lists the main "pain points" tools that we have developed. Working with users (see below) has helped us identify many more pain points, and we are still in the process of developing tools for them.

The last column, Column E, lists the number of commands (where each command can be roughly viewed as a tool that users can use) we have provided for each step of the PyMatcher how-to guide. The bottom of the table lists the five major packages that collectively make up the PyMatcher "ecosystem".

B SAVING THE AMAZON FOREST

We now describe "Land Use", the latest application in which PyMatcher has been put into production.

The Amazon forest in Brazil is shrinking rapidly, because cattle ranchers often clear the forest on their properties to obtain more grazing lands. Brazil has many laws against this but it is very difficult to enforce them. To address this problem, many companies (e.g., McDonald's) have promised to not buy beef from slaughterhouses that obtain cows from "bad" ranches (where deforestation happens). While promising, it has been very difficult to track this. A bad ranch X often sells cows to a ranch Y, which later sells them to a ranch Z, which sells to a slaughterhouse U. U can check that ranch Z is not "bad", but cannot check the entire supply chain, all the way to the origin ranch X.

To address this problem, since 2008 Professor Holly Gibbs and her team at UW-Madison has pioneered a Big Data solution. Briefly, they can regularly obtain data about the sell/buy transactions among the ranches, and about ranches with deforestation (via analyzing satellite images). Combining these two kinds of data helps determine if a bad ranch is in a particular supply chain. This approach can revolutionize research in the land use community and make major real-world impacts.

Services	Description
Upload dataset	Uploads dataset into the system for matching, labeling, etc.
Profiling	Profiles the input dataset to provide more insights on the data.
Edit metadata	Verify/add primary keys, attribute types, etc.
Sample data	Get sample pairs or down-sample two tables for matching purposes.
Find potential matches	Finds the top-k most similar pairs across A and B using Jaccard similarity to be used to select seed pairs for active learning (AL).
Get seed pairs for active learning	Recommends matching/non-matching pairs to the user to select seed for AL.
Generate features	Generates a set of features based on the types and characteristics of the attributes of the two table
Generate feature vectors	Take a set S of tuple pairs and a set F of features, then converts each pair into a set of feature vectors.
Create a classifier	Allows the user to select scikit-learn based machine learning model and its parameters.
Train classifier	Train a classifier by selecting a training dataset and a missing value strategy.
Identify informative ex.	Applies the trained model on the unlabeled dataset to identify informative (controversial) examples.
Labeling service	This service provides two modes of labeling. One is where the user labels the example pairs and other is using crowdsourcing platforms.
Extract blocking	Extracts the blocking rules from the learned matcher M. These rules can be evaluated later or used as it is for blocking purposes.
Evaluating blocking rules	This service takes in a set of blocking rules, compute their precision & coverage, then retains those with high precision & coverage.
Selecting optimal sequence	This service returns a rule sequence $R'*$ from R that when applied to $A \times B$ would minimize run time while maximizing precision and selectivity.
Apply blocking rules	This service applies a sequence of blocking rules R' to two tables A and B, producing a set of tuple pairs $C \subseteq A \times B$ to be matched in the matching stage.
Apply classifier	This service lets you apply the final trained model on the survived pairs to get the predicted matches.
Cross validation	Performs cross-validation and learning of a best model over a sample of dataset.
Active learning	Composite Active Learning (AL) service that performs crowdsourced or user active learning on the sample feature vector set to learn a matcher M.
Falcon	Falcon service executes the end-to-end entity matching workflow on the two datasets to identify matching records.

Table 4: List of services in CloudMatcher.

But to make it work, her team must integrate data from multiple sources (e.g., Brazilian governments, private foundations, slaughterhouse records, etc.). This requires matching cattle ranches across the sources. For the past three years her team has hired a company to do such matching, but the accuracy remains unsatisfactory.

Since Mar 2018 a student from our group has been working with her team (1 programmer, 2 domain experts) to apply PyMatcher to match ranches. In Aug 2018 we showed that PyMatcher can achieve much higher recall than the company solution, while slightly reducing precision. As a result, PyMatcher was put into production. In Nov 2018, it was used to match the first batch of new data, which consists of millions of ranches, and achieved significantly better accuracy than the company solution (see the row "Land Use (UW)" in Table 1). The matching result has been sent to land use researchers all over the world. A paper is planned for a major land use journal, and the project received data science funding from UW (which funds only 10 of the 54 proposals received).

C LIST OF SERVICES OF CLOUDMATCHER

Table 4 shows the list of services that we currently provide.

D SYSTEM CHARACTERISTICS OF CLOUDMATCHER

CloudMatcher has been developed since June 2016, and CloudMatcher 2.0 is the latest version. It is a combination of Python (27K LOC) and Java (6.5K LOC). So far 7 developers have contributed to the code base (47K LOC), which also includes frontend written in HTML5, Javascript, CSS3 (13.5K LOC). CloudMatcher provides 18 basic services and 2 composite services. We use Hadoop for the most compute-intensive operations and HDFS as the shared filesystem.

To serve many users/requests, we use Nginx as the webserver, Gunicorn as the Python HTTP server and our web-app component is written using the popular Django web framework. We store state for the web application and the workflows/tasks in a PostgreSQL database. We use Celery as our task queue with RabbitMQ as the message broker (using binary-only AMQP message format). For portability, we run our Postgres instance and our Task Queue in Docker containers to make our development and production environments identical. The meta-manager is implemented using Twisted (an asynchronous event-driven engine for network application development) and we use Networkx library to handle workflow graphs/DAGs.

We typically deploy CloudMatcher on a 4-node Amazon EMR cluster, where each node has a 4-core Intel Xeon E5-2686 2.30GHz processor and 16GB of RAM.