# BigGorilla: An Open-Source Ecosystem for Data Preparation and Integration

Chen Chen* Behzad Golshan* Alon Halevy* Wang-Chiew Tan* AnHai Doan†
*Megagon Labs †Univ. of Wisconsin
{chen,behzad,alon,wangchiew}@megagon.ai anhai@cs.wisc.edu

**Abstract**

*We present* BIGGORILLA*, an open-source resource for data scientists who need data preparation and integration tools, and the vision underlying the project. We then describe four packages that we contributed to* BIGGORILLA*:* KOKO *(an information extraction tool),* FLEXMATCHER *(a schema matching tool),* MAGELLAN *and* DEEPMATCHER *(two entity matching tools). We hope that as more software packages are added to* BIGGORILLA*, it will become a one-stop resource for both researchers and industry practitioners, and will enable our community to advance the state of the art at a faster pace.*

## 1 Introduction

Data preparation and data integration are long standing problems faced by industry and academia and considerable research has been carried out on different facets of these problems (e.g., [11, 12, 18, 26]). Yet, if a practitioner wishes to get her hands dirty, it is not obvious where to search for the appropriate tools that will suit her needs, or if they even exist. In contrast, the machine learning community has a "go-to" place, Scikit Learn [30], that provides tools for the practitioners working in Python. The lack of such a resource for data integration and preparation is a barrier to the dissemination of the technology developed by our community and limits its impact. Part of the reason that such a resource does not exist is that the space of data preparation and integration is large and is closely intertwined with other data management tools. For example, a typical data integration scenario might require several steps, such as data acquisition, extraction, cleaning, schema matching and mapping, entity matching, and workflow management. Moreover, today several of these steps also have to deal with data at scale. Furthermore, depending on the application, these steps often need to be adapted and repeated in different ways to suit the needs of the application's domain.

Motivated by the desire to create a one-stop resource for data scientists working on data preparation and integration, we initiated BIGGORILLA[1] and seeded it with a few high-quality packages. BIGGORILLA is an open-source ecosystem for data preparation and integration and is currently supported in Python, which is the most popular programming language used by data scientists and taught widely. The over-arching goal of BIGGORILLA is to become the go-to resource for data preparation and integration. In turn this means the website of BIGGORILLA should contain pertinent information for data scientists and pointers to relevant and


**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

[1]https://www.biggorilla.org. Data preparation and integration is a big, hairy, nasty problem and hence the name BIGGORILLA. The problem has also been referred to as the "800 pound gorilla of Big Data".

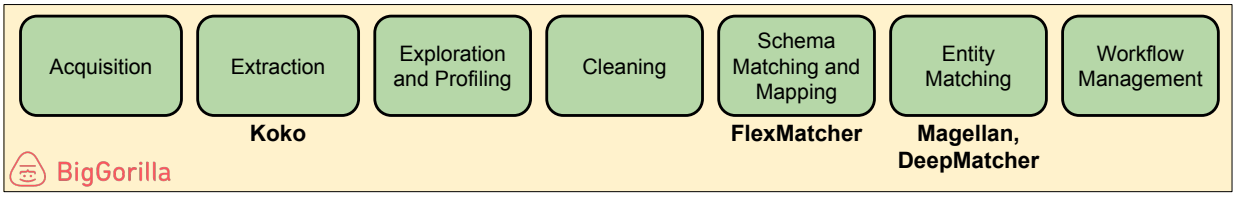


Figure 1: BIGGORILLA categorizes software into different components that are typical in a data preparation and data integration pipeline. The software packages that we contributed (KOKO, FLEXMATCHER, MAGELLAN, DEEPMATCHER) are listed under their respective components. The BIGGORILLA community will benefit as packages for more of these components get contributed.

freely available resources. By being open-source, BIGGORILLA also hopes to facilitate an active community and naturally surface a set of actively maintained software around data preparation and data integration.

BIGGORILLA cannot succeed without the contributions of the community at large. We hope to harness the knowledge of the community to add to and improve the content of the website over time. To begin this effort, we have listed on the website a number of popular software packages in each component. We hope the community will add to these lists, and in cases where they become long, will help us rank the packages by their quality and appropriateness for different contexts. In some cases, we plan to enlist a set of experts to curate a smaller set of software to place on top of the list.

Figure 1 shows the components that are often used in a typical data preparation and integration pipeline. While there are already several open-source packages in each component, the community will benefit from the availability of more packages particularly in data acquisition, cleaning, and workflow management.

The BIGGORILLA website also features a number of tutorials/examples that showcase how some of the software packages may be used. This is another way the community can contribute – we welcome example tasks that have been written which use BIGGORILLA components. Once a repository of examples is in place, it becomes easier for others to learn, through the examples, about data preparation and integration and how to use some of the software packages.

In the rest of this paper we describe four software packages that we contributed to BIGGORILLA: KOKO [38], an information extraction package, FLEXMATCHER [14] for schema matching, and MAGELLAN [21] and DEEPMATCHER for entity matching. We also describe the application of these packages to projects at several companies (including Recruit Holdings) and UW-Madison.

## 2 Koko: A System for Scalable Semantic Querying of Text

KOKO [38] is an information extraction system for extracting tuples from text. It supports declarative specification of conditions on the surface syntax of sentences and on the structure of the dependency parse trees of sentences. While this idea has been explored in the past [35, 36], KOKO also supports conditions that are forgiving to linguistic variations of expressing concepts and allows to aggregate supporting evidence from the entire document in order to filter extractions. Furthermore, KOKO scales to millions of documents by exploiting a multi-indexing scheme and heuristics for efficient extractions.

In what follows, we present KOKO with some examples. The interested reader can find further details in [38].

**Surface syntax and conditions over dependency parse trees** Suppose we wish to extract foods that were mentioned as "*delicious*" in text. We could try to specify an extraction pattern that looks for the word "*delicious*" preceding a noun that is known to be in the category of foods. However, the preceding pattern would not work for the sentence "*I ate a delicious and salty pie with peanuts*" since the word "*delicious*" does not immediately precede the word "*pie*", and it also precedes the word "*peanuts*" which were not deemed delicious. Some sentences are

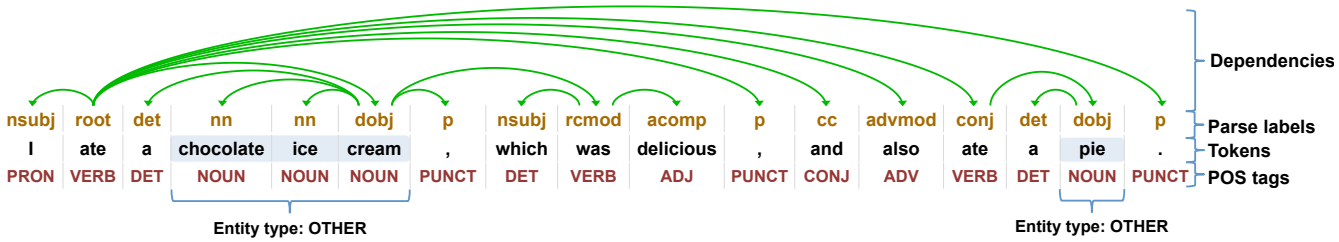


Figure 2: A sentence and its dependency tree annotated with parse labels [25], tokens, POS (Part-Of-Speech) tags [32], and entity types. This dependency tree is generated from Google Cloud NL API [19].

```
extract e:Entity, d:Str from input.txt if
(/ROOT:{
        a = //verb,
        b = a/dobj,
        c = b//"delicious",
        d = (b.subtree)
} (b) in (e))
```

Figure 3: Surface syntax and conditions over dependency parse trees.

```
extract x:Entity from "input.txt" if ()
satisfying x
        (str(x) contains "Cafe" {1}) or
        (str(x) contains "Roasters" {1}) or
        (x ", a cafe" {1}) or
        (x [["serves coffee"]] {0.5}) or
        (x [["employs baristas"]] {0.5})
with threshold 0.8
excluding (str(x) matches "[Ll]a Marzocco")
```

Figure 4: Similarity and aggregation conditions over the entire input.txt

even more nuanced. For example, "*I ate a chocolate ice cream, which was delicious, and also ate a pie*" the word "*delicious*" comes *after* "*ice cream*" which makes the extraction even more challenging.

The first example KOKO query (shown in Figure 3) can correctly extract "*ice cream*" and does not extract "*pie*". More precisely, it extracts pairs of entity $e$ and string $d$ from sentences (defined in the extract clause), where $e$ is an entity that is described as "*delicious*" and the string $d$ is the description of the entity where the word "*delicious*" occurs. The query operates simultaneously over the surface syntax and dependency parse tree of each input sentence in "input.txt". An example of a dependency parse tree of a sentence is shown in Figure 2. The query defines variables $a$, $b$, $c$, and $d$ within the block /ROOT:{ ... } under the if clause where the *paths* are defined w.r.t. the root of the dependency tree. Variables $a$, $b$, and $c$ bind to nodes in the dependency tree as given by the paths. For example, $a$ binds to a node that may occur arbitrarily deep under the root node of the tree. The syntax "*//*" denotes arbitrarily deep. On the other hand, $b$ is defined to be the "*dobj*" node directly under the $a$ node, and $c$ binds to the node with the token "*delicious*" that may occur arbitrarily deep under the $b$ node. For variables $d$ and $e$, which are defined in the first line of the query under the extract clause, each variable binds to a span of tokens. Outside the block, $(b)$ in $(e)$ is a constraint which asserts that the "*dobj*" (a parse label denoting direct object) token must be among the tokens that make up entity $e$. The query considers *every* combination of $a$, $b$, $c$, $d$ and $e$ bindings that are possible.

For the sentence in Figure 2, there is only one possible set of bindings for the variables in this query: $a$ = "*ate*", $b$ = "*cream*", $c$ = "*delicious*", $d$ = "*a chocolate ice cream, which was delicious*", and $e$ = "*chocolate ice cream*". The query returns the pair $(e,d)$.

**Similarity and aggregation conditions** The KOKO language allows for extractions that accommodate variations in linguistic expression and aggregation of evidence over the input. Specifically, it allows for expressions that "softly" match the text and then it aggregates multiple pieces of evidence before yielding an extraction.

Consider the task of extracting cafe names from blog posts. Cafe names are varied and the way cafes are described in different blog posts are also varied. Thus, it would be difficult to write rules that would extract them

accurately from multiple blogs. KOKO combines evidence from multiple mentions in the text and only extracts cafe names that have sufficiently high confidence. For example, if we see that an entity *employs baristas* and *serves espresso*, we may conclude that the combined evidence is sufficient to indicate that the entity is a cafe. However, if it only *employs baristas*, we may decide against that conclusion. In scouring for evidence, care is needed to accommodate linguistic variations on how these properties are expressed, such as *serves up delicious cappuccinos*, or *hired the star barista*. KOKO includes a semantic similarity operator that retrieves phrases that are linguistically similar to the one specified in the rule. Semantic similarity can be determined using paraphrase-based word embeddings. KOKO attaches a confidence value to the phrases matched by the similarity operator, and these confidence values can be aggregated from multiple pieces of evidence in a document. For example, if we see that an entity *serves great macchiatos* and *recently hired a barista*, that may match well to our conditions *serves espressos* and *employ baristas* respectively.

Figure 4 shows an example KOKO query where the aggregation conditions are specified under the satisfying clause (the if clause of the query is empty in this example). The first 3 conditions each have weight 1 while the last two conditions have weight 0.5 each. Intuitively, the weights specify how important a condition is to the overall collection of evidence. Aggregation conditions are either boolean conditions or *descriptors*. For example, the first three conditions are boolean conditions where the first two check whether the string $x$ contains the specified string ("*Cafe*" or "*Roasters*") or not and the last one checks whether $x$ is followed by the string "*, a cafe*" in the text. The remaining conditions are *descriptors* that check whether the span $x$ is followed by the phrase similar to "*serves coffee*" or "*employ baristas*". Descriptors accommodate the fact that different blogs are likely to describe cafes differently. Hence, it is highly unlikely that we will find exact matches of the phrase "*serves coffee*", which is why the descriptor conditions play an important role. The excluding condition ensures that $x$ does not match the string "*La*" (or "*la*") "*Marzocco*", which refers to an espresso machine manufacturer.

To summarize, a basic KOKO query has the following form, where there can be up to one satisfying clause for each output variable.

extract *output tuple* from *input.txt* if
  *variable declarations, conditions, and constraints*
[satisfying *output variable*
  *conditions for aggregating evidence*
with threshold $\alpha$]
[excluding *conditions*]

For every sentence in input.txt, if the extract clause is satisfied, KOKO will search input.txt to compute a score for every satisfying clause. It does so by computing, for every sentence, a score that reflects the degree of match according to the conditions and weights in the satisfying clause and then aggregating the scores of the sentences. For every variable, if the aggregated score of the satisfying clause for that variable from the collective evidence passes the threshold stated, then the result is returned.

More precisely, the score of a value $e$ for the variable under consideration is the weighted sum of confidences, computed as follows: score(e) = $w_1 * m_1(e) + \ldots + w_n * m_n(e)$ where $w_i$ denotes the weight of the $i$-th condition and $m_i(e)$ denotes the degree of confidence for $e$ based on condition $i$. The confidence for $e$ is computed for each sentence in the text and aggregated together.

While the idea of specifying extractions by leveraging dependency trees has been explored in the past [35, 36], KOKO is novel in that it provides a single declarative language that combines surface-level and tree patterns, aggregates and combines evidence, and uses novel indexing techniques to scale to large corpora.

**Multi-indices for the text** KOKO maintains several indexes on the text to process queries efficiently. The indices are built offline, created when the input text is first read. Indices can be persisted for subsequent use.

There are two types of indices in KOKO: *inverted index* and *hierarchy index*. We create *inverted indices* for words and entities and *hierarchy indices* for parse labels and POS (Part-Of-Speech) tags. Unlike indices of [5, 17] our hierarchy index is a compressed representation over dependency structure for parse labels and POS tags. By merging identical nodes, our hierarchy index reduces more than $99.7\%$ of the nodes for both parse labels and POS tags. Hierarchy indices are therefore highly space efficient and enable fast searching.

Intuitively a *word/entity index* maps words/entities to sentences that contain them along with relevant meta-

data. For a word index, every word points to the sentences that contain them, along with the token id and the first and last token id of the subtree rooted at the current token based on the dependency tree, and the depth of the token in the dependency tree. An entity index is defined similarly but the metadata contains information on the span of tokens that constitute the entity. The entity indices enable fast access to sentences that contain certain words and one can also reason about the ancestor-descendant relationship through the metadata that is stored.

A *hierarchy index* is a compact representation of all dependency trees, which provides fast access to the dependency structure of all sentences. There are two types of hierarchy indices: PL (parse label) hierarchy index and POS index. A hierarchy index is constructed by merging identical nodes of dependency trees of all sentences together. Starting at the root of all dependency trees, children nodes with the same label are merged, and then for each child node, all children nodes with the same labels are merged and so on. Hence, by construction, every node in a hierarchy index has a set of children nodes with distinct labels. Consequently, every node of the index can be identified through a unique path given by the sequence of labels from the root node to that node. Every node is annotated with a posting list, which tracks tokens of sentences that have the given path.

We leave the details of how these indices are exploited to [38]. Our experiments demonstrate that the indices are compact and enable us to speed up query processing by at least a factor of 7.

It is important to contrast KOKO with machine learning approaches for recognizing occurrences of typed entities in text (e.g., [8, 28]). Machine learning models are typically specific to a particular domain and do not easily generalize to other domains (e.g., to extract restaurant names instead of cafe names). Also, learning methods usually require significant training data to obtain a model with sufficient precision and recall. In contrast, KOKO does not require training data but relies on user-defined conditions and KOKO is *debuggable* where users can discover the reasons that led to an extraction, which is important for certain applications [6]. Finally, KOKO can be used to create training data for weak-supervision based machine learning approaches.

# 3 FlexMatcher

*Schema Matching* refers to the problem of finding correspondences between the elements of different database schemas. Schema matching is one of the critical steps in the data integration pipeline, and thus has received a lot of attention by researchers in the database community (e.g., see [4, 33, 34]). Many of the proposed solutions for schema matching use machine-learning techniques to find the correct correspondences between the input schemas. FLEXMATCHER is an open-source implementation of such a solution in Python. More specifically, FLEXMATCHER is a system for matching the schemas of multiple data sources to a single mediated schema. FLEXMATCHER uses a collection of supervised machine-learning techniques to train a model that can predict the correct correspondence between the schema of a new data source and the mediated schema. The idea is that after observing a small set of data sources that have been manually mapped to the mediated schema, FLEXMATCHER should be able to predict the correspondences for a new data source. For the most part, FLEXMATCHER is an adaptation of the LSD system proposed by [9]. We provide a brief description of FLEXMATCHER's design and operation next.

**The architecture of FlexMatcher** Figure 5 illustrates the architecture of FLEXMATCHER and describes the training process through a simple working example. Assume that our mediated schema consists of only two attributes: *company name* (associated with color blue) and *salary* (associated with color yellow). Now given multiple datasets listing the salaries of individuals from different companies, our goal is to automatically match the schemas of new datasets to the mediated schema.

As Figure 5 shows, the input to the system is a collection of tables. Note that the input tables can vary in size both in terms of the number of columns as well as the number of available data points. Each table used for training FLEXMATCHER should be accompanied by its correspondences from its columns to the mediated schema. These correspondences are demonstrated with colored arrows on the input tables in Figure 5. For instance, the first two columns in the second table are mapped to attributes company, salary while the last

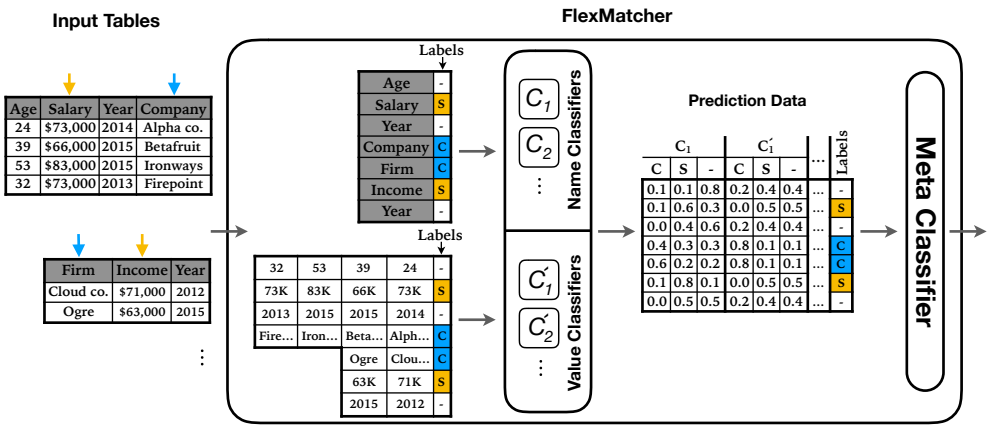


Figure 5: FLEXMATCHER's architecture and training process of the base classifiers and meta-classifier.

column does not map to any attribute of interest in the mediated schema.

Given the input tables, FLEXMATCHER then creates two training datasets. The first dataset (placed on top) consists of the names of columns along with their associated correspondences. In contrast, the other dataset only contains the values under each column along with their associated correspondences. These two training datasets are then fed to an arsenal of classifiers which we refer to as *base* classifiers. Base classifiers are responsible for both extracting relevant features as well as building a model for predicting the correspondences. Note that the base classifiers are divided into two groups: classifiers on column names and classifiers on column values. Naturally, the *name classifiers* are efficient at extracting features from the text and detecting text similarities, while the *value classifiers* are more suitable for extracting features from a collection of values that can be *string*, *numeric*, or *categorical*. Value classifiers can even extract features based on the distribution of the input values (such as min, max, mean, and etc.) to make accurate predictions.

Given that FLEXMATCHER uses a collection of base classifiers, it needs to determine how to combine the prediction of all classifiers to reach a final prediction. To achieve this, FLEXMATCHER first uses each base classifiers to predict the training data using a 5-fold cross-validation technique. This means the data is sliced into 5 folds, and then each fold is predicted using the other folds as training data for the classifier. The output of this step, is the probability distribution over the attributes in our mediated schema for each column. The probability distributions reported by each base classifiers are then organized into a table which is marked as *prediction data* in Figure 5. Finally, FLEXMATCHER trains a *meta-classifier* on the prediction data. The meta-classifier is a simple *logistic regression* model that learns how to combine the prediction of each base-classifier.

Once the training process is complete, FLEXMATCHER can make predictions for new data sources by (1) making prediction based on column names using name classifiers, (2) making predictions based on data points via value classifiers, and (3) combining the prediction of all base-classifiers using the pre-trained meta classifier.

Next, we highlight some aspects of the design of FLEXMATCHER that allows practitioners to easily modify the tool to improve the performance in different settings.

One of the most important feature of FLEXMATCHER is that it allows the user to add/remove base classifiers (of either type) as they see fit. This enables practitioners to both benefit from the existing classifiers in FLEXMATCHER while having the option to add classifiers that can work with features that are expected to be more helpful in a particular domain. For instance, if we know that in our setting the salaries are always reported with the currency symbol (such as $ or €), then we can simply add a new classifier that predicts a column as "salary" if it can detect any currency symbol in the data. To make the process of adding and removing classifiers easy, FLEXMATCHER is designed to be compatible with python's machine learning ecosystem. More precisely, all the base-classifiers are expected to comply with python's *scikit-learn* package [30] which implements a great

number of machine learning techniques. Thus, one can easily deploy different models from the scikit-learn package and use them as part of FLEXMATCHER.

Following our last example, it is important to mention that base classifiers do not need to be efficient at detecting all attributes in the mediated schema. Clearly, the "salary" classifier we just proposed can not distinguish between "company name" or "last name". Nevertheless, the meta classifier learns to trust this classifier if it predicts a column as "salary" and ignore its predictions in other scenarios. As a result, users can easily build new classifiers that are focused and tailored to a single attribute, and thus enhance the overall performance of FLEXMATCHER in their setting.

FLEXMATCHER also allows users to enforce cardinality constraints on the predicted correspondences. For instance, we can enforce that at most a single column should be mapped to each of the attributes in the mediated schema. These constraints are handled by solving a *maximum bipartite-matching* problem. More precisely, the goal is to match the columns to the attributes in the mediated schema such that (a) the cardinality constraints are met, and (b) the sum of FLEXMATCHER's confidence over all mapped attributes is maximized. To achieve this, FLEXMATCHER applies the Hungarian algorithm to find the optimal matching.

Finally, FLEXMATCHER has the ability to cope with absence of values in the input data-sources. In this case, the system only deploys the name classifiers to make a prediction based on the common patterns that it observes in column names. Alternatively if the data source miss column names, then FLEXMATCHER would only deploy the value classifiers to predict the correspondences. This makes FLEXMATCHER more robust to such data-quality issues.

# 4 Entity Matching Packages

## 4.1 String Similarity Measures and Similarity Joins

String matching is a fundamental operation in many data science tasks, such as data exploration, data profiling, data cleaning, information extraction, schema and entity matching. As a result, over the past few decades, string matching has received significant attention and many solutions have been proposed [11, 39]. At present, BIGGORILLA contains two packages related to string matching: PY_STRINGMATCHING and PY_STRINGSIMJOIN.

PY_STRINGMATCHING: Given two sets of strings $A$ and $B$, *string matching* is the problem of finding all pairs $(a \in A, b \in B)$ that match, i.e., refer to the same real-world entity, such as "Michael J. Williams" and "Williams, Michael". So far the most common solution is to return as matches all pairs $(a, b) \in A \times B$ that satisfy a predicate of the form $sim[t(a), t(b)] \geq \epsilon$. Here $sim(a, b)$ is a string similarity measure, $t$ is a tokenizer, and $\epsilon \in [0, 1]$ is a pre-specified threshold. For example, given the predicate *jaccard*[*3gram*($a$), *3gram*($b$)] $> 0.8$, we tokenize strings $a$ and $b$ into sets of 3-grams $S_a$ and $S_b$ respectively, compute the Jaccard score $|S_a \cap S_b|/|S_a \cup S_b|$, then return true if this score exceeds 0.8 and return false otherwise.

Numerous string similarity measures have been developed, such as Jaccard, edit distance, TF/IDF, cosine, etc. [11, 39]. PY_STRINGMATCHING implements a broad range of these similarity measures, together with a set of well-known tokenizers. The rationale for developing this package and a comparison with nine existing string similarity measure packages are provided in the package's developer manual [16].

PY_STRINGSIMJOIN: This package performs a *string similarity join* between two sets $A$ and $B$, using a predicate such as *jaccard*[*3gram*($a$), *3gram*($b$)] $> 0.8$ as the join condition [39]. Applying the join predicate to all pairs in $A \times B$ is often impractical because $A \times B$ can be very large. To address this, current work typically builds an index $I$ over a table, say $A$ (sometimes multiple indexes are built, over both tables). For each string $b \in B$, it then consults $I$ to locate only a relatively small set of strings in $A$ that can potentially match with $b$. For example, suppose $I$ is an inverted index that, given a token $t$, returns the IDs of all strings in $A$ that contain $t$. Then for a string $b \in B$, we can consult $I$ to find only those strings $a$ in $A$ that share at least a token with $b$, then apply the join predicate only to these $(a, b)$ pairs. Numerous such indexing techniques have been developed,

e.g., inverted index, size filtering, prefix filtering, etc. [11, 39]. The package PY_STRINGSIMJOIN implements a variety of string similarity joins using these indexing techniques. Together, these two packages provide a basis for us to build more advanced packages, such as MAGELLAN, a package to perform end-to-end entity matching, which we describe next.

### 4.2 End-to-End Entity Matching with Magellan

The MAGELLAN system focuses on entity matching (EM), the problem of identifying 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 [7, 13]. MAGELLAN introduces a new template for research and system development for EM.

The current research template for EM focuses largely on developing algorithmic solutions for *blocking* and *matching*, two important steps in the EM process. However, a recent paper [21] argues that EM processes often involve many other "pain points", and proposes a new research template in which we move beyond examining just blocking and matching. Instead, we would (a) develop a step-by-step how-to guide that captures the entire EM process, (b) examine the guide to identify all true pain points of the EM process, then (c) develop solutions and tools for the pain points. Several recent works [10, 20] confirm that there are indeed many other pain points including data labeling, debugging, EM-centric cleaning, and defining the notion of match. Furthermore, the research argues that solving these pain points is critical for developing practical EM tools, and that addressing them raises many novel research challenges.

Most current EM systems are built as *stand-alone monolithic systems*. However, in [21] we argue that such systems are very difficult to extend, customize, and combine. We observe that many EM steps essentially perform data science tasks, and that there already exist vibrant ecosystems of open-source data science tools (e.g., those in Python and R), which are being used heavily by data scientists to solve these tasks. Thus, we propose to *develop EM tools within such data science ecosystems*. This way, the EM tools can easily exploit other tools in the ecosystems, and at the same time make such ecosystems better at solving EM problems.

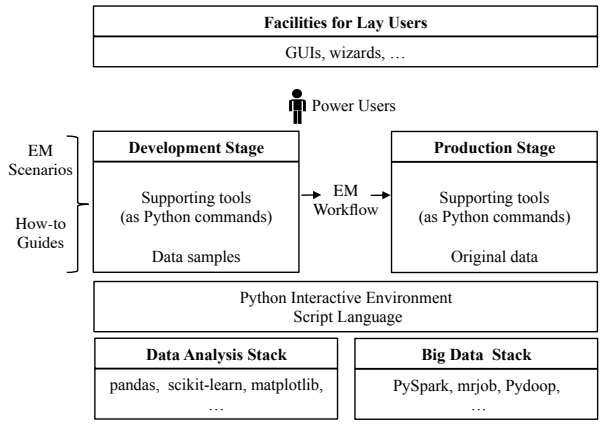


Figure 6: The MAGELLAN architecture.

Figure 6 shows the MAGELLAN architecture. The system targets a set of EM scenarios. For each EM scenario it provides a how-to guide. The guide proposes that the user solve the scenario in two stages: development and production. In the development stage, the user 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). Since tools are built into the Python ecosystem, MAGELLAN can borrow functionalities (e.g., cleaning, extraction, visualization) from other Python packages in the ecosystem. Finally, the MAGELLAN is currently geared toward users who are knowledgeable in programming. In future facilities for lay users (e.g., GUIs, wizards) can be provided so that lay user actions can be translated into sequences of commands for MAGELLAN.

We describe the capabilities of MAGELLAN next, which shows that it goes beyond blocking and matching.

- *Loading tables:* MAGELLAN relies on the pandas package to read tables on file into data frames. As a result, it can read a wide variety of files, including csv, json, XML, etc.

- *Browsing, exploration, profiling, and cleaning:* MAGELLAN uses the pandas-profiling, matplotlib, and other popular Python packages to help the user visualize and profile the tables. It can connect to OPENREFINE, a stand-alone data exploration and cleaning tool, to help the user explore and clean the tables.

- *Downsampling and blocking:* If two tables $A$ and $B$ to be matched have been read into memory and they are big, then the user can downsample them into smaller tables $A'$ and $B'$, which would be easier to work with. MAGELLAN also provides the user with a variety of blocking methods (e.g., attribute equivalence, overlap, rule-based, etc.). More blocking methods are being developed. Finally, MAGELLAN also provides a blocking debugger, which can be used to examine how good the current blocker is [24].
- *Feature generation:* Both the blocking step and the matching step (described later) use a set of features, such as *edit_dist*($A'$.*city*, $B'$.*city*) and *jaccard*[*3gram*($A'$.*name*), *3gram*($B'$.*name*)]. MAGELLAN uses the two packages PY_STRINGMATCHING and PY_STRINGSIMJOIN described in the previous section to create these features. We are also currently working on helping lay users learn blocking rules.
- *Sampling and labeling:* After blocking, the user can take a sample of tuple pairs and label them as match/no-match. MAGELLAN provides several labeling tools, including a Web-based tool that enables collaborative labeling. We are currently working on a tool to help debug the labels.
- *Selecting and debugging the matchers:* Using the labeled data, the user selects a learning-based matcher (from the library of learning methods in scikit-learn), evaluate its accuracy, and debug if necessary. MAGELLAN currently provides tools to debug decision trees and random forests. We have also recently developed a deep learning based matcher that works well for textual and dirty tuples [27] (see the next section).
- *Clustering and merging matches:* Once a matcher has been selected, the user can apply it to the tuple pairs in the output of the blocker to predict matches. We are currently working on tools to cluster the tuples (such that all tuples within a single cluster match) and to merge tuples within each cluster into a single canonical tuple.
- *Adding matching rules to the workflow:* MAGELLAN also allows the user to easily add matching rules to various steps in the EM workflow.

As described, MAGELLAN relies heavily on packages in the Python ecosystem. In addition to developing the above capabilities, we are also developing methods and tools to scale up the execution of MAGELLAN commands, to manage data larger than memory, to specify EM workflows, to perform incremental execution of workflows, and to track provenance, among others. Finally, MAGELLAN provides detailed guidance to users on the challenges that commonly arise during EM, such as how to collaboratively label a dataset (so that the labelers can quickly converge to a match definition), how to decide when to stop the blocking step, and how to determine the size of the sample to be labeled.

### 4.3 Deep Learning for Semantic Matching Tasks

The entity matching problem discussed in the previous section is just one case of a *semantic matching* problem. In the above case, matching means that the two data instances refer to the same real world entity. A different example of semantic matching is answer selection (a major sub-task of question answering), which determines if a candidate sentence or paragraph correctly answers a given question. Here the comparison is between two natural language text blobs, a question and an answer candidate. Other types of semantic matching tasks include textual entailment, entity linking, and coreference resolution.

Deep learning (DL) solutions have been shown to achieve state-of-the-art performance for these tasks [37, 15]. While DL models solutions for these tasks seem highly specialized at first glance, they do in fact share several commonalities. Based on a comprehensive review [27], we note that all of these models share the following major components:

- *Language representation:* This component maps words in each data instance into word embeddings, i.e., vector representations of words. For example, it would convert a question into a sequence of vectors.
- *Similarity representation:* This component automatically learns a vector representation that captures the similarity of the two data instances given their language representations. This involves summarizing each data instance and then comparing them.
- *Classification:* This component takes the similarity representations and uses those as features for a classifier that determines if the two data instances match.

These similarities present a great opportunity - a system that can address one of these tasks can easily tackle other matching tasks. However, developing such a system is a cumbersome endeavor. Even when using popular deep learning frameworks, many aspects such as data processing, training mechanics, model construction, etc. need to be handled manually. To simplify building deep learning models for semantic matching tasks, we developed DEEPMATCHER, a Python package based on the model architecture described above. It is broadly applicable to matching tasks such as entity matching, question answering, textual entailment, DEEPMATCHER is built on top of PyTorch [29], a deep learning framework that is great for research as it enables rapid development iterations. Once a model has been finalized and trained, it can then be optimized for fast inference in production settings.

In order to help users get started, DEEPMATCHER comes built-in with four representative DL models, based on four major categories of neural networks used in matching tasks [27]. The four models vary in complexity and provide a trade-off between training time and model expressiveness:

- *SIF:* This model first aggregates information about the words present in each data instance, and then compares them. It does not take word sequence into account. Intuitively, data instances that contain similar words are considered matches.
- *RNN:* This model first performs a sequence-aware summarization of each data instance, and then compares them. Intuitively, data instances that contain similar word sequences are considered matches.
- *Attention:* This model first performs an alignment between words in each data instance, then performs a word-by-word comparison based on alignment, and finally aggregates this information to perform classification. Intuitively, data instances that contain *aligning words* are considered matches.
- *Hybrid:* This model first performs an alignment between word sequences in each data instance, then compares the aligning word sequences, and finally aggregates this information to perform classification. Intuitively, data instances that contain *aligning word sequences* are considered matches.

While DEEPMATCHER provides these four representative solutions out of the box, the package is designed to be easily extensible. It provides several options for each component described earlier. Users can create their own models using the provided building blocks and optionally use custom modules. The four built-in models can also be thoroughly customized. It is built to be both simple enough for deep learning beginners and also highly flexible for advanced users.

There are four main steps in using DEEPMATCHER. First, the user preprocesses all data instances. This involves tokenization, loading word embeddings, etc. Second, the user constructs a neural network model for matching. Third, the user trains the neural network model on the preprocessed training dataset. Finally, the user may apply the trained model over the test set or use it to make predictions. Without customization, this entire pipeline requires less than 10 lines of code. A version of this package was used to perform an extensive empirical evaluation of deep learning approaches for the task of entity matching, compared to MAGELLAN [27]. Apart from entity matching, DEEPMATCHER can be used for other semantic matching tasks, such as answer selection for question answering, and textual entailment.

# 5 Applications

BigGorilla has been used in several projects within Recruit Holdings. Some examples include mining of restaurant/salon names, matching schemas of real estate records, and entity extraction from textual statements. In addition, there are tutorials on various components of BigGorilla that can be used as course assignments for a data science course.

In one of the applications, KOKO was used to extract names of companies from text and the result was fed into Magellan to determine entity matching across two datasets. Prior to KOKO, extraction of company names was carried out manually and with the help of KOKO, the amount of human labor is reduced and some work is expended instead to verify whether the entity names are extracted correctly.

FlexMatcher has been used to match schemas of real estate records from different agencies. FlexMatcher semi-automates the schema matching process with data-driven approaches, and improves accuracy by identifying a number of correct matching instances missed by human workers. For example, FlexMatcher successfully matched two columns containing similar data – i.e., "*mls_subpremise*" and "*L_Address2 Unit #*" – although the syntax of the column names are quite different.

Magellan has been used in several projects at Recruit Holdings to help improve the accuracy and efficiency of their entity matching tasks [1]. For example, one restaurant-advertising company collects restaurant information (e.g., reviews and ratings) from online resources regularly. The collected information is then consolidated with an existing restaurant database. Magellan is used to match between restaurant names, which reduces the cost of matching significantly, with 4% increase in entity matching accuracy at the same time. In other projects, Magellan has also been used to match salon/nail shop names, with similar performance improvement.

BigGorilla also contains tutorials which showcase key components of the data integration pipeline and can be used for educational purposes. Currently, there are tutorials that showcase roughly 3 steps (extraction, schema matching, and entity matching and some cleaning) of the data integration pipeline. These tutorials illustrate KOKO, FlexMatcher and Magellan through an example problem that analyzes aviation data from multiple data sources, to produce statistics, such as the airlines that have the most accidents in history. These tutorials are freely available for use and can be easily adapted to be used as course assignments for educational purposes.

The packages PY_STRINGMATCHING, PY_STRINGSIMJOIN, and MAGELLAN have also been successfully used in five domain science projects at UW-Madison (in economics, biomedicine, environmental science [3, 22, 23, 31]), and at several other companies (e.g., Johnson Control, Marshfield Clinic, WalmartLabs). For example, at WalmartLabs they improved the recall of a deployed EM solution by 34%, while reducing precision slightly by 0.65%. They have also been used by 400+ students to match real-world data in five data science classes at UW-Madison (e.g., [2]).

# 6 Conclusion

The goal of BIGGORILLA is to become a one-stop open source resource for data scientists working on data preparation and integration, and grow an active community supporting this effort. We have described four software packages (KOKO, FLEXMATCHER, MAGELLAN, and DEEPMATCHER) which we have contributed to BIGGORILLA. We hope that as more software packages are added to BIGGORILLA, it will further benefit both researchers and industry practitioners in understanding the state-of-the-art in data preparation and integration, what is freely available for use, and perhaps more importantly, what more can be done.

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

[4] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

[5] S. Bird, Y. Chen, S. B. Davidson, H. Lee, and Y. Zheng. Designing and evaluating an xpath dialect for linguistic queries. In *ICDE*, page 52, 2006.

[6] L. Chiticariu, Y. Li, and F. R. Reiss. Rule-based information extraction is dead! long live rule-based information extraction systems! In *EMNLP*, pages 827–832, 2013.

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

[8] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, 2011.

[9] A. Doan, P. Domingos, and A. Halevy. Learning to match the schemas of data sources: A multistrategy approach. *Mach. Learn.*, 50(3):279–301, Mar. 2003.

[10] A. Doan et al. Human-in-the-loop challenges for entity matching: A midterm report. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2017, Chicago, IL, USA, May 14, 2017*, 2017.

[11] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann, 1st edition, 2012.

[12] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.

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

[14] FlexMatcher (Schema Matching package in Python). https://github.com/biggorilla-gh/flexmatcher,

[15] M. Francis-Landau, G. Durrett, and D. Klein. Capturing semantic similarity for entity linking with convolutional neural networks. NAACL, 2016.

[16] P. S. G.C. et al. Py_stringmatching's developer manual, pages.cs.wisc.edu/˜anhai/py_stringmatching/v0.2.0/dev-manual-v0.2.0.pdf, 2017.

[17] S. Ghodke and S. Bird. Fast query for large treebanks. In *HLT-NAACL*, pages 267–275, 2010.

[18] B. Golshan, A. Y. Halevy, G. A. Mihaila, and W. Tan. Data integration: After the teenage years. In *PODS*, 2017.

[19] Google Cloud Natural Language API. https://cloud.google.com/natural-language/,

[20] Y. Govind et al. Cloudmatcher: A cloud/crowd service for entity matching. In *KDD Workshop on Big Data as a Service (BIGDAS-17)*, 2017.

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

[22] 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.

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

[24] H. Li et al. Matchcatcher: A debugger for blocking in entity matching. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, 2018.

[25] R. T. McDonald, J. Nivre, Y. Quirmbach-Brundage, Y. Goldberg, D. Das, K. Ganchev, K. B. Hall, S. Petrov, H. Zhang, O. Täckström, C. Bedini, N. B. Castelló, and J. Lee. Universal dependency annotation for multilingual parsing. In *ACL*, pages 92–97, 2013.

[26] R. J. Miller. The future of data integration. In *KDD*, pages 3–3.

[27] S. Mudgal et al. Deep learning for entity matching: A design space exploration. In *SIGMOD-18*, 2018.

[28] D. Nadeau and S. Sekine. A survey of named entity recognition and classification. *Lingvisticæ Investigationes*, 30(1):2–36, 2007.

[29] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

[32] S. Petrov, D. Das, and R. T. McDonald. A universal part-of-speech tagset. In *LREC*, pages 2089–2096, 2012.

[33] E. Peukert, J. Eberius, and E. Rahm. A self-configuring schema matching system. In *ICDE*, pages 306–317, 2012.

[34] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[35] Tregex, tsurgeon, semgrex. http://nlp.stanford.edu/software/tregex.html,

[36] M. A. Valenzuela-Escárcega, G. Hahn-Powell, and M. Surdeanu. Odin's runes: A rule language for information extraction. In *LREC*, 2016.

[37] S. Wang and J. Jiang. A compare-aggregate model for matching text sequences. ICLR, 2017.

[38] X. Wang, A. Feng, B. Golshan, A. Halevy, G. Mihaila, H. Oiwa, and W.-C. Tan. Scalable semantic querying of text *(to appear)*. *PVLDB*, 2018.

[39] M. Yu et al. String similarity search and join: A survey. *Front. Comput. Sci.*, 10(3):399–417, 2016.