Auto-Tag: Tagging-Data-By-Example in Data Lakes using Pre-training and Inferred Domain Patterns

Yeye He, Jie Song, Yue Wang, Surajit Chaudhuri
Vishal Anil, Blake Lassiter, Yaron Goland, Gaurav Malhotra
Microsoft Corporation

ABSTRACT

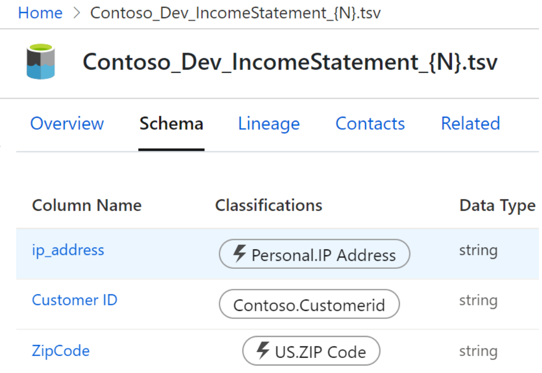
As data lakes become increasingly popular in large enterprises today, there is a growing need to tag or classify data assets (e.g., files and databases) in data lakes with additional metadata (e.g., semantic column-types), as the inferred metadata can enable a range of downstream applications like data governance (e.g., GDPR compliance), and dataset search. Given the sheer size of today's enterprise data lakes with petabytes of data and millions of data assets, it is imperative that data assets can be "auto-tagged", using lightweight inference algorithms and minimal user input.

In this work, we develop Auto-Tag, a corpus-driven approach that automates data-tagging of *custom* data types in enterprise data lakes. Using Auto-Tag, users only need to provide *one* example column to demonstrate the desired data-type to tag. Leveraging an index structure built offline using a lightweight scan of the data lake, which is analogous to pre-training in machine learning, Auto-Tag can infer suitable data patterns to best "describe" the underlying "domain" of the given column at an interactive speed, which can then be used to tag additional data of the same "type" in data lakes. The Auto-Tag approach can adapt to custom data-types, and is shown to be both accurate and efficient. Part of Auto-Tag ships as a "custom-classification" feature in a cloud-based data governance and catalog solution *Azure Purview*.

ACM Reference Format:
Yeye He, Jie Song, Yue Wang, Surajit Chaudhuri, Vishal Anil, Blake Lassiter, Yaron Goland, Gaurav Malhotra. 2021. Auto-Tag: Tagging-Data-By-Example in Data Lakes using Pre-training and Inferred Domain Patterns. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large enterprise data lakes are increasingly common today, often with petabytes of data and millions of data assets (e.g., flat files or databases). Given their sheer sizes, it has become increasingly important to *govern* and *catalog* data lakes, as evidenced by a growing number of offerings from startups and established vendors, such as Azure Purview [4], AWS Glue Catalog [3], Google Cloud Data Catalog [8], Alation [1], Waterline [12], Collibra [6], etc.


Home > Contoso_Dev_IncomeStatement_{N}.tsv
Contoso_Dev_IncomeStatement_{N}.tsv
Overview
Schema
Lineage
Contacts
Related

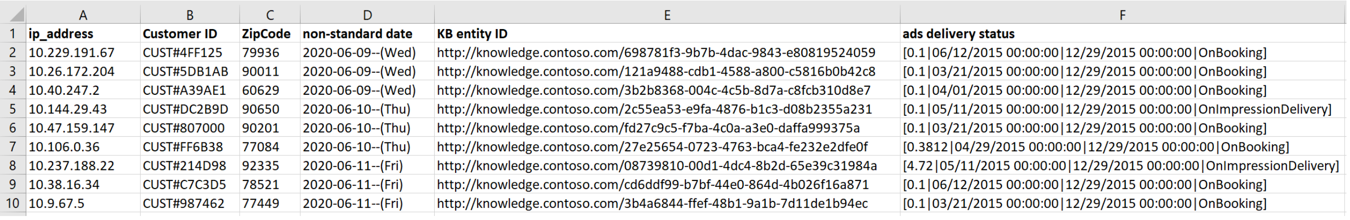
Column Name	Classifications	Data Type
ip_address	Personal.IP Address	string
Customer ID	Contoso.Customerid	string
ZipCode	US.ZIP Code	string

Figure 1: Azure Purview: sample columns in an example file "Contoso_Dev_IncomeStatement.tsv", are automatically tagged as "Personal.IP Address", "Contoso.CustomerId", "US.ZIP Code", etc.

A key challenge in governing data lakes is data *tagging* (also known as *classification*), which is the process of inferring rich metadata (e.g. semantic column-types) from data. Such inferred metadata are critical for downstream applications such as data governance and data discovery:

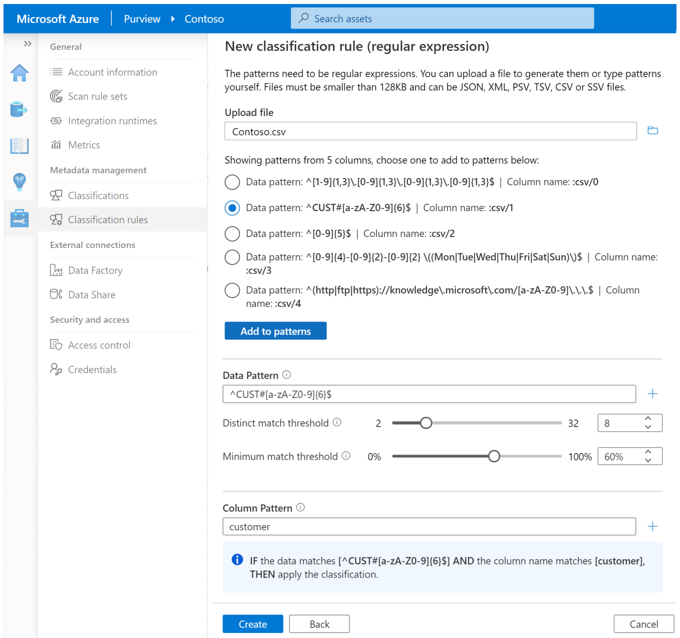
Data governance. Data protection regulations such as GDPR, PCI and CCPA impose strict requirements on how sensitive personal data can be retained and accessed. To ensure compliance, it is imperative that enterprises can automatically identify sensitive data assets in their data lakes, so that these data assets can be governed in accordance with regulatory requirements.

Data discovery. In order to improve the productivity of enterprise workers, it is increasingly important for enterprise workers to discover and leverage data assets relevant to their tasks, using self-service mechanisms such as data-set search. Given the large number of data assets in modern enterprise data lakes, and their nondescript, sometimes cryptic, nature, datasets search is clearly challenging (compared to the web search for exampled) [22]. Suitable metadata tags/classifications associated with data assets can significantly improve search relevance, and enhance the overall usefulness of enterprise data lakes.



	A	B	C	D	E	F
1	**ip_address**	**Customer ID**	**ZipCode**	**non-standard date**	**KB entity ID**	**ads delivery status**
2	10.229.191.67	CUST#4FF125	79936	2020-06-09--(Wed)	http://knowledge.contoso.com/698781f3-9b7b-4dac-9843-e80819524059	[0.1|06/12/2015 00:00:00|12/29/2015 00:00:00|OnBooking]
3	10.26.172.204	CUST#5DB1AB	90011	2020-06-09--(Wed)	http://knowledge.contoso.com/121a9488-cdb1-4588-a800-c5816b0b42c8	[0.1|03/21/2015 00:00:00|12/29/2015 00:00:00|OnBooking]
4	10.40.247.2	CUST#A39AE1	60629	2020-06-09--(Wed)	http://knowledge.contoso.com/3b2b8368-004c-4c5b-8d7a-c8fcb310d8e7	[0.1|04/01/2015 00:00:00|12/29/2015 00:00:00|OnBooking]
5	10.144.29.43	CUST#DC2B9D	90650	2020-06-10--(Thu)	http://knowledge.contoso.com/2c55ea53-e9fa-4876-b1c3-d08b2355a231	[0.1|05/11/2015 00:00:00|12/29/2015 00:00:00|OnImpressionDelivery]
6	10.47.159.147	CUST#807000	90201	2020-06-10--(Thu)	http://knowledge.contoso.com/fd27c9c5-f7ba-4c0a-a3e0-daffa999375a	[0.1|03/21/2015 00:00:00|12/29/2015 00:00:00|OnBooking]
7	10.106.0.36	CUST#FF6B38	77084	2020-06-10--(Thu)	http://knowledge.contoso.com/27e25654-0723-4763-bca4-fe232e2dfe0f	[0.3812|04/29/2015 00:00:00|12/29/2015 00:00:00|OnBooking]
8	10.237.188.22	CUST#214D98	92335	2020-06-11--(Fri)	http://knowledge.contoso.com/08739810-00d1-4dc4-8b2d-65e39c31984a	[4.72|05/11/2015 00:00:00|12/29/2015 00:00:00|OnImpressionDelivery]
9	10.38.16.34	CUST#C7C3D5	78521	2020-06-11--(Fri)	http://knowledge.contoso.com/cd6ddf99-b7bf-44e0-864d-4b026f16a871	[0.1|06/12/2015 00:00:00|12/29/2015 00:00:00|OnBooking]
10	10.9.67.5	CUST#987462	77449	2020-06-11--(Fri)	http://knowledge.contoso.com/3b4a6844-ffef-48b1-9a1b-7d11de1b94ec	[0.1|03/21/2015 00:00:00|12/29/2015 00:00:00|OnBooking]

Figure 2: An example spreadsheet from Contoso, with many enterprise-specific "custom" data-types.



New classification rule (regular expression)

The patterns need to be regular expressions. You can upload a file to generate them or type patterns yourself. Files must be smaller than 128KB and can be JSON, XML, PSV, TSV, CSV or SSV files.

Upload file

Contoso.csv

Showing patterns from 5 columns, choose one to add to patterns below:

○ Data pattern: ^[1-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$ | Column name: :csv/0

● Data pattern: ^CUST#[a-zA-Z0-9]{6}$ | Column name: :csv/1

○ Data pattern: ^[0-9]{5}$ | Column name: :csv/2

○ Data pattern: ^[0-9]{4}-[0-9]{2}-[0-9]{2} \((Mon|Tue|Wed|Thu|Fri|Sat|Sun)\)$ | Column name: :csv/3

○ Data pattern: ^(http|ftp|https)://knowledge\.microsoft\.com/[a-zA-Z0-9]\.\.\.$ | Column name: :csv/4

Add to patterns

Data Pattern ⓘ

^CUST#[a-zA-Z0-9]{6}$

Distinct match threshold ⓘ 2 32 8

Minimum match threshold ⓘ 0% 100% 60%

Column Pattern ⓘ

customer

ⓘ IF the data matches [^CUST#[a-zA-Z0-9]{6}$] AND the column name matches [customer], THEN apply the classification.

Create Back Cancel

Figure 3: Azure Purview UI for custom-classification by-example: after uploading an example data file, columns with inferred patterns will be suggested for user to inspect and approve.

Auto-tagging of "standard" data-types. Given the importance of data tagging, it is no surprise that leading vendors in this space all have features relating to automated data-tagging.

Figure 1 shows a data-tagging feature in Azure Purview [4]. Out of the box, the system can already recognize 100+ *standard* data types commonly found in the public domain [5].[1] In this particular example, it detects a few sample columns to be of type "IP Address", "US Zip Code", etc. Note that these are well-known data-types from the public domain, henceforth refer to as "*standard*" data-types.

Because standard data-types are well-known and their corresponding "data-taggers" can be reliably tested beforehand, auto-tagging features for standard data-types are readily available and work well "out-of-the-box" in today's data-catalog and governance vendors (using a combination of techniques like predefined regex patterns, bloom-filters, etc.).

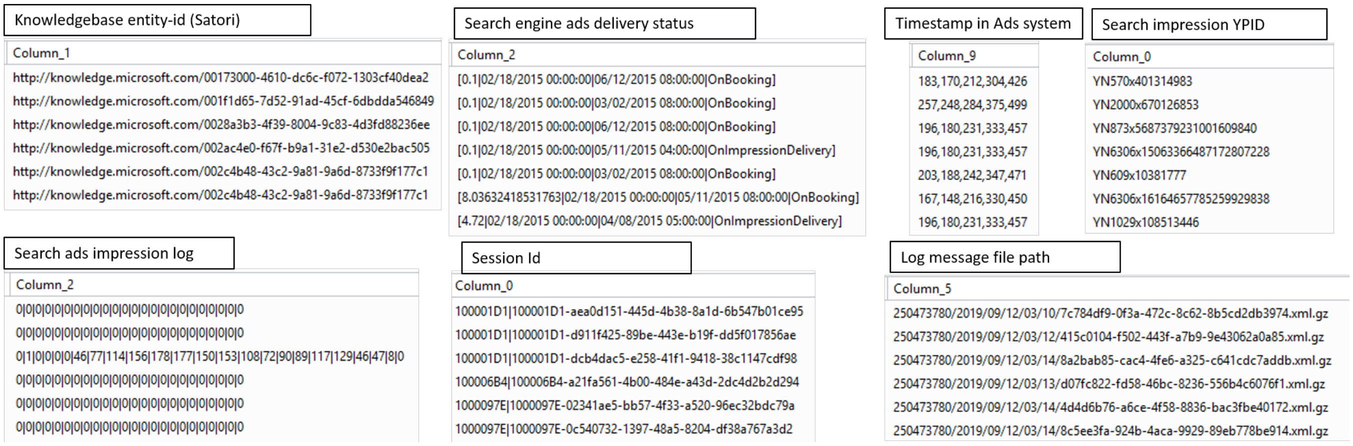
Auto-tagging of "custom" data-types. While standard data-types capture an important class of use cases, we observe that there is also a large number of "*custom*" enterprise data types, which are unique to different industries and enterprises.

In the example of Figure 2, column B contains unique IDs that this company Contoso (a fictional company) assigns to their customers – in this case, these values have a prefix of "CUST#" followed by six hexadecimal characters (e.g., "CUST#0FF125"). Ideally, we want to tag columns of this type as a new custom data-type "Contoso.CustomerId", like shown in the second column of Figure 1. However, Contoso's way of identifying customers is likely unique, as other companies may devise their own unique-identifiers for customers – for example, another company may use customer-identifiers that have a string prefix of "C-" followed by a unique nine-digit number (e.g., "C-123456789"), while yet another company may choose to use other types of UUID. These different forms of customer-ids are custom-made (generated by some programs) and unique to each enterprise, which are thus not well-known "standard" data-types that a general-purpose data-catalog solution can possibly anticipate.

There are a large number of such custom data-types in today's enterprises. Figure 4 shows a few example custom data-types, harvested from a real production data lake at Microsoft [15]. Each column here has a distinctive data pattern, which uniquely identifies a custom data-type widely used inside the company. For instance, the first column is known as Knowledge-Base entity-id (Satori [21]), which is a unique ID assigned to real-world entities and used by the search engine Bing. Similarly, the second column encodes the delivery status of Bing ads, etc.

Methods developed for tagging "standard" data types are clearly inapplicable for these idiosyncratic "custom" data types, because they are unique to different enterprises, and unlikely to be found from the public domain or other enterprises.

[1] Note that many of these data-types are sensitive in nature, making them particularly relevant to data governance and catalog vendors.



Knowledgebase entity-id (Satori)

Column_1
http://knowledge.microsoft.com/00173000-4610-dc6c-f072-1303cf40dea2
http://knowledge.microsoft.com/001f1d65-7d52-91ad-45cf-6dbdda546849
http://knowledge.microsoft.com/0028a3b3-4f39-8004-9c83-4d3fd88236ee
http://knowledge.microsoft.com/002ac4e0-f67f-b9a1-31e2-d530e2bac505
http://knowledge.microsoft.com/002c4b48-43c2-9a81-9a6d-8733f9f177c1
http://knowledge.microsoft.com/002c4b48-43c2-9a81-9a6d-8733f9f177c1

Search engine ads delivery status

Column_2
[0.1|02/18/2015 00:00:00|06/12/2015 08:00:00|OnBooking]
[0.1|02/18/2015 00:00:00|03/02/2015 08:00:00|OnBooking]
[0.1|02/18/2015 00:00:00|06/12/2015 08:00:00|OnBooking]
[0.1|02/18/2015 00:00:00|05/11/2015 04:00:00|OnImpressionDelivery]
[0.1|02/18/2015 00:00:00|03/02/2015 08:00:00|OnBooking]
[8.03632418531763|02/18/2015 00:00:00|05/11/2015 08:00:00|OnBooking]
[4.72|02/18/2015 00:00:00|04/08/2015 05:00:00|OnImpressionDelivery]

Timestamp in Ads system

Column_9
183,170,212,304,426
257,248,284,375,499
196,180,231,333,457
196,180,231,333,457
203,188,242,347,471
167,148,216,330,450
196,180,231,333,457

Search impression YPID

Column_0
YN570x401314983
YN2000x670126853
YN873x5687379231001609840
YN6306x15063366487172807228
YN609x10381777
YN6306x16164657785259929838
YN1029x108513446

Search ads impression log

Column_2
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0
0|1|0|0|0|0|46|77|114|156|178|177|150|153|108|72|90|89|117|129|46|47|8|0
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0

Session Id

Column_0
100001D1|100001D1-aea0d151-445d-4b38-8a1d-6b547b01ce95
100001D1|100001D1-d911f425-89be-443e-b19f-dd5f017856ae
100001D1|100001D1-dcb4dac5-e258-41f1-9418-38c1147cdf98
100006B4|100006B4-a21fa561-4b00-484e-a43d-2dc4d2b2d294
1000097E|1000097E-02341ae5-bb57-4f33-a520-96ec32bdc79a
1000097E|1000097E-0c540732-1397-48a5-8204-df38a767a3d2

Log message file path

Column_5
250473780/2019/09/12/03/10/7c784df9-0f3a-472c-8c62-8b5cd2db3974.xml.gz
250473780/2019/09/12/03/12/415c0104-f502-443f-a7b9-9e43062a0a85.xml.gz
250473780/2019/09/12/03/14/8a2bab85-cac4-4fe6-a325-c641cdc7addb.xml.gz
250473780/2019/09/12/03/13/d07fc822-fd58-46bc-8236-556b4c6076f1.xml.gz
250473780/2019/09/12/03/14/4d4d6b76-a6ce-4f58-8836-bac3fbe40172.xml.gz
250473780/2019/09/12/03/14/8c5ee3fa-924b-4aca-9929-89eb778be914.xml.gz

Figure 4: Example "custom" data types, crawled from an enterprise data-lake at Microsoft. Each column has a distinctive data pattern in proprietary formats, encoding specific meanings. These custom data types are all common, occurring in at least 5000 columns in our sample crawl with 7M columns.

This motivates us to look into ways that can tag custom enterprise data-types, with minimal input from users.

Tagging custom-types by-examples. Unlike auto-tagging of standard data types, which can be expected to work out-of-box, we believe that tagging custom enterprise data types requires some amount of human input, (e.g. from data-owners or domain-experts), to (1) determine relevant data of interest to "tag", and (2) provide suitable and meaningful tags that can describe the underlying meaning of the custom data-type (i.e., an algorithm may infer values of the form "CID-12345" to be a unique data-type, but cannot be sure of its meaning).

We believe that a human-in-the-loop approach to tagging custom data-types have two key desiderata:

• *Low human-cost.* The system should require minimal input from enterprise users, ideally needing users to provide only one example column to demonstrate the custom data type of interest (e.g., an example column in Figure 4). Note that this is different from typical machine-learning tasks – asking users to repeatedly provide feedback in the form of positive/negative labels to tag *one* custom data type can be too costly in this setting.

• *Low execution-cost.* It is also important that any tag-inference algorithm needs to be lightweight, in order for the feature to be cost-effective on large enterprise data lakes. Although a deep analysis (e.g., a full scan) of the data lake will typically yield better predictive accuracy, the scale of the data (e.g., petabytes) makes a full scan too expensive. Thus, auto-tagging algorithms should only perform a *lightweight* scan (e.g. a row-wise sample per asset), in order for the feature to be viable in terms of COGS.

In Section 2, we will discuss why existing techniques (e.g., [16, 25, 38–40]) may be insufficient in such a setting, either due to high human-costs, or high execution-cost.

In this work, we develop an initial version of this auto-tagging feature called Auto-Tag. Unlike standard machine-learning, Auto-Tag has the advantage of requiring only *one* labeled example column (low human costs), and unlike content-based or dictionary-based approaches, does not require a full scan of data files (low execution costs), because patterns can be reliably generalized from small samples.

Using a variant of an algorithm we develop in [34], we first perform lightweight (row-wise sampled) scans of data lakes offline, to build a succinct index structure that is analogous to pre-training in machine-learning. At online inference time, given an example column of interest users point us to, Auto-Tag leverages the offline index to produce relevant data patterns that can accurately describe the underlying domain of the custom type of interest.

Figure 3 shows a by-example auto-tagging feature in Azure Purview, which uses this technology to auto-tag custom data-types. From the UI, users can easily upload a data file with target columns of interest. Leveraging a succinct index structure built offline, a list of suggested data-tagging patterns can be produced at an interactive speed, so that users can pick the desired pattern corresponding to the column of interest, inspect the suggested pattern, before approving the data-tagging rule. The tagging-rule so created would then be used to tag additional columns in the data lake matching the given pattern during data scans.

Our experiments using real data from a production enterprise data lake at Microsoft [15] suggest that Auto-Tag is both accurate and cost-efficient, for tagging custom data-types. We report experimental results in Section 4.

2 RELATED WORKS

Auto-tagging of data assets is an important topic, given the abundance of data in data lakes today. We will review related data-tagging techniques below[2], and discuss why they are not immediately applicable to our problem (high human-cost or execution cost).

We emphasize that different classes of techniques below are often suitable for orthogonal types of data (e.g., natural-language content vs. machine-generated data), and thus do not subsume each other.

Data-tagging by value-patterns. It is reported that a substantial fraction of enterprise data columns have regex-like patterns [34], for which pattern-based approaches are the most suitable.

There are many existing techniques from the *data profiling* literature, which infers patterns based on example data-values. These include research prototypes like Potter's wheel [32], X-System [26], LearnPads [19, 20], FlashProfile [29], and commercial implementations like Microsoft SQL Server SSIS [10], Trifacta [11], Ataccama [2].

As we will highlight in Section 3, the goal of data-profiling is distinctively different from data-tagging – it aims to find patterns to succinctly summarize *given data values only*, which tend to produce overly-specific (or under-generalized) patterns, which yield low recall when used for auto-tagging. There is significant room for improvement, and is the focus of our corpus-driven Auto-Tag approach.

Data-tagging by machine learning models. Machine-learning or deep-learning based approaches, such as Auto-EM [40], Sherlock [25] and Sato [39], have been developed to tag columns with natural-language content (e.g., company-names, people-names, etc.). Such approaches, however, are often a poor fit for machine-generated data (e.g., GUID, employee-ID, etc.), and would complement pattern-based approaches. Such approaches also typically require a non-trivial amount of labeled data, increasing the cost of adoption for tagging custom enterprise data-types (high human costs).

Data-tagging by value-overlap. Techniques have also been developed to tag columns based on value overlap in enterprise tables [16] and web tables [35, 36], where the idea is that if a substantial fraction of values in a given column match a known dictionary of values (e.g., a known list of department-names or product-names), then the column can be tagged accordingly.

When such "dictionaries of values" for enterprise concepts are not known a priori, techniques are developed to harvest such "dictionaries" for data-tagging. These techniques are known in the literature as *set expansion* [23, 31, 37], *concept discovery* [27, 28], and more broadly knowledge-base construction [14, 18, 21, 24]. These approaches, however, typically require full-scans for high recall, thus introducing high execution-costs.

Data-tagging by synthesized programs. Because values of certain data types (e.g., credit-card numbers, UPC codes) have unique signatures such as check-sums, which can only be detected via specific program-logic, program-synthesis based data-tagging have been proposed, which synthesize type-detection functions using existing code [38]. Such approaches, however, requires the presence of a enterprise-specific code repository to be effective.

3 AUTO-TAG BY-EXAMPLES

Given the need of low human-costs and execution-costs discussed above, in this work we set out to solve the auto-tagging problem, for string-valued custom-types with syntactic patterns (our prior study [34] suggests that this is an important class accounting for around 40% string-valued columns in a production data lake).

We will first briefly describe the pattern language used.

3.1 Preliminary: Pattern Language

We use a standard pattern language (similar to [32]). Other languages can also be plugged in Auto-Tag to produce corresponding patterns.

Figure 5 shows a standard generalization hierarchy, where leaf-nodes represent the English alphabet, and intermediate nodes (e.g., `<digit>`, `<letter>`) represent *token* that values can generalize into. A pattern is a sequence of (leaf or intermediate) tokens, and for a given value v, this hierarchy induces a space of all patterns consistent with v, denoted by $\mathbf{P}(v)$. For instance, for a value v = "`9:07`", we could generate $\mathbf{P}(v)$ = {"`<digit>:<digit>{2}`", "`<digit>+:<digit>{2}`", "`<digit>:<digit>+`", "`<num>:<digit>+`", "`9:<digit>{2}`", ...}, among many other options.

Given a column C for which patterns need to be generated, we define the space of *candidate patterns*, denoted by $\mathbf{P}(C)$, as the set of patterns consistent with values $v \in C$. We use an in-house implementation to produce patterns based on the hierarchy in Figure 5 (other hierarchies and languages can be applied similarly in Auto-Tag).

3.2 Find Suitable "Domain" Patterns

Given the pattern language $\mathbf{P}$ described above, and given a data lake, consisting of a large collection of tables $\mathbf{T}$ (which can be flat files such as .csv, .tsv, .xls, .json, as well as database files and database tables, etc.), at a high level our auto-tagging problem can be stated as follows.

Definition 1. Auto-tag by-examples. Given a data lake of tables $\mathbf{T}$, users demonstrate a desired action to tag data of

[2] We focus on techniques that produce tags based on *data-values in columns*. Orthogonal techniques leveraging other types of information also exist, e.g., program-flows [33].

type t, by providing one example column $C \in \mathbf{T}$ consisting of a set of values $C = \{v_i\}$ that are of type t, and a tag $n(t)$ describing this type t. Let $\mathbf{P}(C)$ be the set of all data-patterns consistent with C, our goal is to select a suitable pattern $p \in \mathbf{P}(C)$, such that for any data column $D \in \mathbf{T}$, if $p \in \mathbf{P}(D)$ or p also matches the column D, then D is also likely of type t (can be tagged as $n(t)$).

Example 1. As a concrete example, users provide an example column C_1 shown in Figure 8, as well as a tag $n(C_1)$, say "*date*". The system should suggest a suitable pattern "`<letter>{3} <digit>{2} <digit>{4}`" that best describe the underlying data "domain". Once this is reviewed and approved by users, it can be used to tag additional columns in $\mathbf{T}$ matching the same pattern (with tag "*date*").

One challenge is that $\mathbf{P}(C)$ is large (there are many ways to "generalize" a column C into patterns). For a simple column of date-time strings like in Figure 6, and using a standard generalization hierarchy as in Figure 5, one could produce over 3 billion possible patterns. For example, the first part (digit "`9`" for month) alone can be generalized in 7 different ways shown in the top-right box of Figure 6, and the cross-product at each position creates a large space (3.3 billion patterns) for this seemingly simple column.

Given a large space of candidates $\mathbf{P}(C)$, the key is to:

(1) Not "*under-generalize*": or use overly restrictive patterns, which lead to low recall for data-tagging; and

(2) Not "*over-generalize*": or use overly generic patterns (e.g. the trivial "`.*`"), which lead to low precision.

These are the key reasons why related techniques like *pattern-profiling* (e.g., Potter's Wheel [32], PADS [19], X-System [26], FlashProfile [30], etc.) are not directly applicable to data-tagging, because they have very different objectives.

Specifically, the goal of pattern-profiling is to succinctly "summarize" a given set of values in column C, so that users can quickly understand what is in C without needing to scroll/inspect the entire C. Such techniques explicitly consider *only* values in C, *without needing to consider values not present in* C (e.g., other valid values that are in the same "domain" as C).

For example, classical pattern profiling methods like Potter's Wheel [32] and FlashProfile [30] would correctly generate a desirable pattern "`Mar <digit>{2} 2019`" for C_1 in Figure 8, which is valuable from a pattern-profiling's perspective as it succinctly describes all given values in C_1. However, this pattern is not suitable for data-tagging, as it under-generalizes and would miss many similar date-time columns like "`Apr 01 2019`", thus yielding low recall. A more appropriate data-tagging pattern should instead describe the entire "domain" of possible values for this data-type, e.g., "`<letter>{3} <digit>{2} <digit>{4}`".

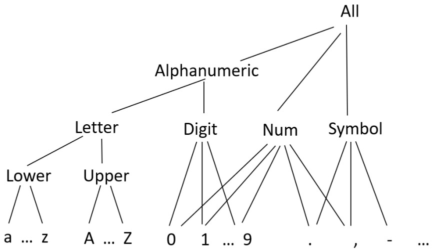


Figure 5: Example generalization hierarchy.

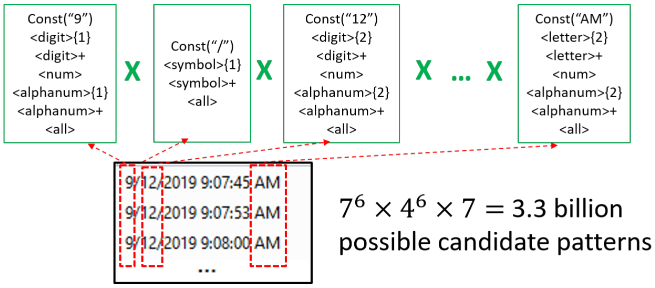


Figure 6: Possible ways to generalize a column of date-time strings, using the hierarchy in Figure 5.

A key challenge here is to select suitable patterns from $\mathbf{P}(C)$, when only one example column C is given. This is intuitively difficult if we only look at C – for the date examples in Figure 8, we as humans know the ideally-generalized pattern for this type, but for data from proprietary domains with ad-hoc formats (e.g., Figure 4), even humans may find it hard and need to use additional evidence to reason about ideal patterns to describe the corresponding "domain" (e.g., by inspecting similar-looking columns in the lake $\mathbf{T}$).

Following this intuition, we propose a corpus-driven approach Auto-Tag that leverages summary statistics of $\mathbf{T}$ (with similar-looking columns) to determine the best pattern, which we describe next.

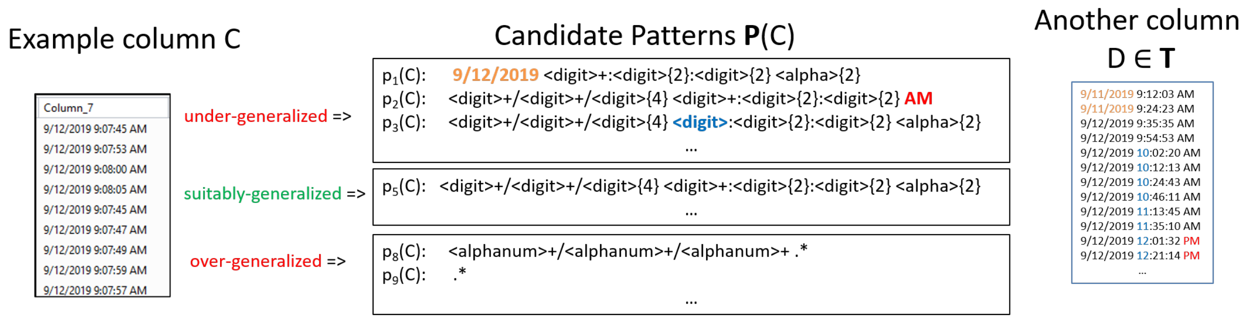
3.3 Auto-Tag: Estimate Pattern Quality

Intuitively, a pattern $p(C) \in \mathbf{P}(C)$ is a good domain pattern if it captures all valid values from the same domain, and a "bad" pattern if it under-generalizes or over-generalizes.

Avoid under-generalization. We show that it is possible to infer whether $p(C)$ under-generalizes, using summary statistics from $\mathbf{T}$ (without human input).

Example 2. The left of Figure 7 shows a query column C for which domain patterns need to be generated. A few candidate patterns in $\mathbf{P}(C)$ are listed in the middle. In this example, we know that $p_1(C), p_2(C), p_3(C)$ are "bad" because they under-generalize the domain (too "narrow").

We show that this can be inferred using $\mathbf{T}$ alone. Specifically, $p_1(C)$ likely under-generalizes the domain, because we can find many columns like $D \in \mathbf{T}$ shown on the right



Column_7
9/12/2019 9:07:45 AM
9/12/2019 9:07:53 AM
9/12/2019 9:08:00 AM
9/12/2019 9:08:05 AM
9/12/2019 9:07:45 AM
9/12/2019 9:07:47 AM
9/12/2019 9:07:49 AM
9/12/2019 9:07:59 AM
9/12/2019 9:07:57 AM

```
p1(C):   9/12/2019 <digit>+:<digit>{2}:<digit>{2} <alpha>{2}
p2(C):   <digit>+/<digit>+/<digit>{4} <digit>+:<digit>{2}:<digit>{2} AM
p3(C):   <digit>+/<digit>+/<digit>{4} <digit>:<digit>{2}:<digit>{2} <alpha>{2}
...
p5(C):   <digit>+/<digit>+/<digit>{4} <digit>+:<digit>{2}:<digit>{2} <alpha>{2}
...
p8(C):   <alphanum>+/<alphanum>+/<alphanum>+ .*
p9(C):   .*
...
```

```
9/11/2019 9:12:03 AM
9/11/2019 9:24:23 AM
9/12/2019 9:35:35 AM
9/12/2019 9:54:53 AM
9/12/2019 10:02:20 AM
9/12/2019 10:12:13 AM
9/12/2019 10:24:43 AM
9/12/2019 10:46:11 AM
9/12/2019 11:13:45 AM
9/12/2019 11:35:10 AM
9/12/2019 12:01:32 PM
9/12/2019 12:21:14 PM
...
```

Figure 7: Given a column C, leverage tables in a data lake T, to infer whether candidate patterns $p(C)$ over-generalize or under-generalize.

of Figure 7 that are "impure" – these columns contain values that match $p_1(C)$, as well as values that do not (e.g., "`9/11/2019 09:12:03 AM`", where the day part does not match $p_1(C)$). A large number of "impure" columns likely indicate under-generalizations.

We can show that $p_2(C)$ also likely under-generalizes the domain, as it makes many columns like D "impure" (the "`PM`" part does not match $p_2(C)$).

The same can be said about $p_3(C)$ (values like "`10:02:20 AM`" are inconsistent with $p_3(C)$ because they have two-digit hours, whereas $p_3(C)$ uses a single `<digit>`).

Using $p_5(C)$ to describe the domain, on the other hand, would not yield many "impure" columns in $\mathbf{T}$, suggesting that it does not under-generalize the domain.

Intuitively, we can use the *impurity* of p on data columns $D \in \mathbf{T}$, measured as the fraction of values in D not matching p, to infer whether p is an under-generalization:

Definition 2. The *impurity* of a candidate pattern p on a data column $D \in \mathbf{T}$, is defined as:

$$\mathrm{Imp}_D(p) = \frac{|\{v|v \in D, p \notin \mathbf{P}(v)\}|}{|\{v|v \in D\}|} \tag{1}$$

Example 3. In Figure 7, $\mathrm{Imp}_D(p_1)$ can be calculated as $\frac{2}{12}$, since the first 2 values (with "`9/11/2019`") out of 12 do not match p_1. Similarly, $\mathrm{Imp}_D(p_3)$ can be calculated as $\frac{8}{12}$, since the last 8 values in D (with two-digit hours) do not match p_3.

Finally, $\mathrm{Imp}_D(p_5)$ is $\frac{0}{12}$, since all values in D match p_5.

We note that if $p(C)$ is used to tag data in the same domain as C, then $\mathrm{Imp}_D(p)$ directly corresponds to expected false-negative-rate (FNR), or recall-loss for data-tagging tasks.

Definition 3. The expected *false-negative-rate* (FNR) of using pattern $p(C)$ to tag a data column D drawn from the same domain as C, denoted by $\mathrm{FNR}_D(p)$, is defined as:

$$\mathrm{FNR}_D(p) = \frac{\mathrm{FN}_D(p)}{\mathrm{TP}_D(p) + \mathrm{FN}_D(p)} \tag{2}$$

Where $\mathrm{TP}_D(p)$ and $\mathrm{FN}_D(p)$ are the number of false-positive detection and true-negative detection of p on D, respectively. Note that since D is from the same domain as C, ensuring that $\mathrm{TP}_D(p)$ and $\mathrm{FN}_D(p) = |D|$, $\mathrm{FNR}_D(p)$ can be rewritten as:

$$\mathrm{FNR}_D(p) = \frac{|\{v|v \in D, p \notin \mathbf{P}(v)\}|}{|\{v|v \in D\}|} = \mathrm{Imp}_D(p) \tag{3}$$

Thus allowing us to estimate $\mathrm{FNR}_D(p)$ using $\mathrm{Imp}_D(p)$.

Example 4. Continue with Example 3, it can be verified that the expected FNR of using p as the domain pattern for D, directly corresponds to the impurity $\mathrm{Imp}_D(p)$ – e.g., using p_1 to tag D has $\mathrm{FNR}_D(p_1) = \mathrm{Imp}_D(p_1) = \frac{2}{12}$; while using p_5 to tag D has $\mathrm{FNR}_D(p_5) = \mathrm{Imp}_D(p_5) = 0$, etc.

Based on $\mathrm{FNR}_D(p)$ defined for one column $D \in \mathbf{T}$, we can in turn define the estimated FNR on the entire corpus $\mathbf{T}$, using all column $D \in \mathbf{T}$ where some value $v \in D$ matches p:

Definition 4. Given a corpus $\mathbf{T}$, we estimate the FNR of pattern p on $\mathbf{T}$, denoted by $\mathrm{FNR}_{\mathbf{T}}(p)$, as:

$$\mathrm{FNR}_{\mathbf{T}}(p) = \underset{D \in \mathbf{T}, v \in D, p \in \mathbf{P}(v)}{\mathrm{avg}} \mathrm{FNR}_D(p) \tag{4}$$

Example 5. Continue with the p_5 in Example 3 and Example 4. Suppose there are 5000 data columns $D \in \mathbf{T}$ where some value $v \in D$ matches p_5. Suppose 2000 columns out of the 5000 have $\mathrm{FNR}_D(p_5) = 0$, and the remaining 3000 columns have $\mathrm{FNR}_D(p_5) = 50\%$. The overall $\mathrm{FNR}_{\mathbf{T}}(p_5)$ can be calculated as $\frac{3000*50\%}{5000} = 30\%$, using Equation (4).

Avoid over-generalization. So far we have focused on avoiding under-generalization. Similarly we should also avoid over-generalization, such as p_8 and p_9 shown in Figure 8. We achieve this by measuring *coverage* of pattern p over $\mathbf{T}$.

Definition 5. The *coverage* of a candidate pattern p on $\mathbf{T}$, is defined as:

$$\mathrm{Cov}(p) = |\{D|D \in \mathbf{T}, p \in \mathbf{P}(D)\}| \tag{5}$$

Intuitively, among all candidate patterns that do not under-generalize (using impurity-based estimates), we should pick the pattern with the least coverage, which is the least likely to over-generalize.

Example 6. Recall that in Example 2, we infer that $p_1(C)$, $p_2(C)$ and $p_3(C)$ likely under-generalize (thus can be excluded), while $p_5(C)$, $p_8(C)$ and $p_9(C)$ do not. For the remaining patterns, given a data lake with 10M columns, we find the coverage of $p_5(C)$, $p_8(C)$ and $p_9(C)$ to be 20K, 500K and 10M, respectively. We can then pick $p_5(C)$ as the suitable pattern for auto-tagging, as it does not under-generalize, and at the same time is the least likely to over-generalize.

Given this intuition, we formalize pattern-inference as an optimization problem below.

3.4 Problem Formulation: CMDT

We now describe the basic version of Auto-Tag as follows. Given an input query column C and a background corpus $\mathbf{T}$, we need to produce a domain pattern $p(C)$, such that $p(C)$ is expected to have a low FNR but also with few false positives. We formulate this as an optimization problem, called Coverage-Minimizing version of Data-Tagging (CMDT), defined as:

$$\text{(CMDT)} \qquad \min_{p \in \mathbf{P}(C)} \text{Cov}_{\mathbf{T}}(p) \tag{6}$$

$$\text{s.t. } \text{FNR}_{\mathbf{T}}(p) \leq r \tag{7}$$

$$\text{Cov}_{\mathbf{T}}(p) \geq m \tag{8}$$

Equation (7) states that the expected recall loss of using p as the domain pattern for C, estimated from $\text{FNR}_{\mathbf{T}}(p)$, is lower than a given threshold r. Equation (8) is an optional constraint that requires the coverage of p, $\text{Cov}_{\mathbf{T}}(p)$, defined as the number of columns in $\mathbf{T}$ that match p, to be greater than a given threshold m (otherwise the custom data-type may be too niche to be interesting).

The domain pattern p we produce for C is then the minimizer of CMDT from the space of all candidate patterns $\mathbf{P}(C)$ (Equation (6)), which as discussed minimizes the chance of over-generalization (and false-positives in auto-tagging) for a given recall constraint.

We should note that the CMDT formulation is closely related to the FMDV problem in [34]. The two problems share the same problem structure but use different objective functions (tailored to data-tagging and data-validation, respectively). We leverage similar vertical-cut and horizontal-cut algorithms in [34], and also optimization methods (lightweight scan with offline indexing). Together, these mechanisms achieve (1) interactive response time and (2) cost effectiveness (by scanning a small fraction of rows per file). We refer readers to [34] for details of the algorithms in the interest of space.

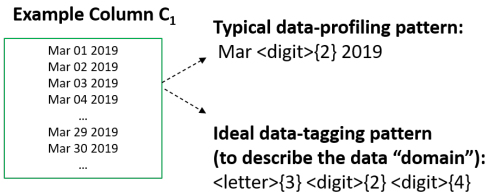

Example Column C_1
Mar 01 2019
Mar 02 2019
Mar 03 2019
Mar 04 2019
...
Mar 29 2019
Mar 30 2019
...
Typical data-profiling pattern:
Mar <digit>{2} 2019
Ideal data-tagging pattern
(to describe the data "domain"):
<letter>{3} <digit>{2} <digit>{4}

Figure 8: Example showing different patterns produced for data-profiling vs. data-tagging, because the two have very different objectives.

4 EXPERIMENTS

We implement our offline indexing algorithm in a Map-Reduce-like language called Scope [41] with UDFs in C#, executed on a production cluster [15].

4.1 Benchmark Evaluation

Data set. We evaluate algorithms using a real corpus $\mathbf{T}$ with 7.2M data columns, crawled from a production data lake at Microsoft [41].

Evaluation methodology. We randomly sample 1000 columns from $\mathbf{T}$ to produce a benchmark set of columns, denoted by $\mathbf{B}$. We use the first 1000 values of each column to control column size variations.

Given a benchmark $\mathbf{B}$ with 1000 columns, $\mathbf{B} = \{C_i | i \in [1, 1000]\}$, we manually assign a ground-truth tag-id for each column. This produces clusters of columns in the same data-type and should be assigned same tags.

We then evaluate precision and recall of patterns generated on $\mathbf{B}$ as follows. For each column $C_i \in \mathbf{B}$, we use the first 10% of values in C_i (or 100 values) as the "training data", denoted by C_i^{train}, from which patterns need to be generated. Each algorithm A can observe C_i^{train} and "learn" pattern $A(C_i^{\text{train}})$. The inferred pattern is denoted as $A(C_i^{\text{train}})$.

To test recall of $A(C_i^{\text{train}})$ when C_i is used for auto-tagging, denoted by $R_A(C_i)$, we use the remaining 90% of values from C_i, as well as other columns in $\mathbf{B}$ with the same ground-truth cluster-id. These are data columns drawn from the same data-type as C_i, which we expect $A(C_i^{\text{train}})$ to match. We take chunks of 100 values from these columns as column-units, and compute recall by testing the fraction of column-units that can be correctly tagged (under different matching thresholds).

To test precision, denoted by $P_A(C_i)$, we use columns in $\mathbf{B}$ with a different ground-truth cluster-id, which are from a different data-type as C_i. We know that it is a false-positive if $A(C_i^{\text{train}})$ were to tag these columns. We compute precision accordingly, by taking chunks of 100 values from these columns as column-units, and compute the fraction of column-units that not incorrectly tagged by $A(C_i^{\text{train}})$.

The overall precision/recall on benchmark **B** is the average across all cases: $P_A(\mathbf{B}) = \text{avg}_{C_i \in \mathbf{B}} P_A(C_i)$, and $R_A(\mathbf{B}) = \text{avg}_{C_i \in \mathbf{B}} R_A(C_i)$. Both of these are between 0 and 1 as usual.

4.2 Methods Compared

We compare the following algorithms using benchmark **B**, by reporting precision/recall numbers $P_A(\mathbf{B})$ and $R_A(\mathbf{B})$.

Auto-Tag. This is our proposed approach using CMDT.

Potter's Wheel (PWheel) [32]. This is an influential pattern-profiling method that finds the best pattern based on minimal description length (MDL).

SQL Server Patterns [10]. SQL Server has a data-profiling feature in SSIS and Data Tools. We invoke it programmatically to produce regex patterns for each column.

XSystem [26]. This recent approach develops a flexible branch and merge strategy to pattern profiling. We use the authors' implementation on GitHub [13] to produce patterns.

FlashProfile [29]. FlashProfile is a recent approach to pattern profiling, which clusters similar values by a distance score. We use the authors' implementation in NuGet [7].

We also compare with **Grok** [9], which is a popular approach that uses a collection of curated regex patterns to detect common types in log messages; **schema-matching** [17] based methods (followed by pattern-profiling); and a simple **Value-Union** [16] method that is more suitable for natural-language content. These methods are not as effective (e.g., producing overly-generic patterns with low precision), and are omitted from the results (to ensure we can zoom in on the competitive methods in the figures).

4.3 Experimental Results

We evaluate different methods based on tagging quality (precision/recall), latency, and memory footprint.

Quality. Figure 9 shows precision/recall of all methods using the enterprise benchmark **B** with 1000 randomly sampled test cases. It can be seen that Auto-Tag is substantially better than other methods in both precision and recall.

Among all the baselines, we find data profiling techniques like `PWheel` and `FlashProfile` to also be of high-precision. However, these techniques tend to under-generalize and produce lower recall (because as discussed, data profiling techniques aim to optimize for a fundamentally different objective compared to data-tagging).

Latency. Given that it is important to produce regex suggestions at interactive speed (for users to inspect and verify), we compare the mean and max latency of different methods on 1000 benchmark test columns. It can be seen that Auto-Tag is clearly interactive, where the max latency is 0.663 second.

In comparison, methods like `FlashProfile` and `XSystem` use expensive clustering, which on average take 6-7 seconds per input column, where the max latency per column is close to 6 minutes for both methods. Note that for both `FlashProfile` and `XSystem` we use authors' original implementations [7, 13].

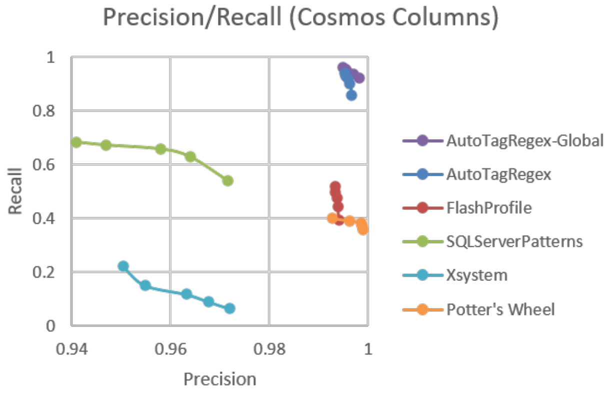


Figure 9: Precision/Recall on 1000 randomly sampled cosmos data. Results are scaled to test columns that have patterns.

Memory footprint. We also evaluate the average and max memory usage for pattern-learning per input column. Auto-Tag avoids expensive bottom-up enumeration and is lightweight, which uses an average of 1.9MB memory (2.8MB max). In comparison, clustering-based pattern-profiling methods like `FlashProfile` takes 162MB memory on average, with a max memory usage of 7.9GB.

Method	mean-latency (ms)	max-latency (ms)
Auto-Tag	**12**	**663**
FlashProfile	7076	359382
XSystem	6411	346996

Table 1: Mean/max latency on 1000 benchmark cases.

5 CONCLUSIONS

Observing the need to data-tagging for custom data types in enterprise data lakes, we propose a corpus-driven Auto-Tag approach to infer relevant data patterns. This is shown to be accurate and cost-effective, when evaluated on real enterprise data from a production data lake.

REFERENCES

[1] Alation Data Catalog. https://www.alation.com/.

[2] ataccama: Data Profiling. https://www.ataccama.com/product/data-discovery-and-profiling.

[3] AWS Glue Data Catalog. https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html.

[4] Azure Purview. https://azure.microsoft.com/en-us/services/purview/.

[5] Azure Purview: 100+ standard data-types for auto-tagging. https://docs.microsoft.com/en-us/azure/purview/supported-classifications.

[6] Collibra Data Catalog. https://www.collibra.com/data-catalog.

[7] FlashProfile package. https://www.nuget.org/packages/Microsoft.ProgramSynthesis.Extraction.Text/.

[8] Google Cloud Data Catalog. https://cloud.google.com/data-catalog.

[9] Grok Patterns. https://github.com/elastic/elasticsearch/blob/master/libs/grok/src/main/resources/patterns/grok-patterns.

[10] SSIS: Data Profiling. https://docs.microsoft.com/en-us/sql/integration-services/control-flow/data-profiling-task?view=sql-server-ver15.

[11] Trifacta: Data Profiling. https://www.trifacta.com/data-profiling/.

[12] Waterline Data Catalog. https://www.waterlinedata.com/.

[13] XSystem Code. https://bitbucket.org/andrewiilyas/xsystem-old/src/outlier-detection/.

[14] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250, 2008.

[15] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[16] E. Cortez, P. A. Bernstein, Y. He, and L. Novik. Annotating database schemas to help enterprise search. *Proceedings of the VLDB Endowment*, 8(12):1936–1939, 2015.

[17] H.-H. Do, S. Melnik, and E. Rahm. Comparison of schema matching evaluations. In *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 221–237. Springer, 2002.

[18] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–610, 2014.

[19] K. Fisher and R. Gruber. Pads: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices*, 40(6):295–304, 2005.

[20] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. *ACM SIGPLAN Notices*, 43(1):421–434, 2008.

[21] Y. Gao, J. Liang, B. Han, M. Yakout, and A. Mohamed. Building a large-scale, accurate and fresh knowledge graph. In *SigKDD*, 2018.

[22] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Managing google's data lake: an overview of the goods system. *IEEE Data Eng. Bull.*, 39(3):5–14, 2016.

[23] Y. He and D. Xin. Seisa: set expansion by iterative similarity aggregation. In *Proceedings of the 20th international conference on World wide web*, pages 427–436, 2011.

[24] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013.

[25] M. Hulsebos, K. Hu, M. Bakker, E. Zgraggen, A. Satyanarayan, T. Kraska, Ç. Demiralp, and C. Hidalgo. Sherlock: A deep learning approach to semantic data type detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1500–1508, 2019.

[26] A. Ilyas, J. M. da Trindade, R. C. Fernandez, and S. Madden. Extracting syntactical patterns from databases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 41–52. IEEE, 2018.

[27] K. Li, Y. He, and K. Ganjam. Discovering enterprise concepts using spreadsheet tables. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1873–1882, 2017.

[28] M. Ota, H. Müller, J. Freire, and D. Srivastava. Data-driven domain discovery for structured datasets. *Proceedings of the VLDB Endowment*, 13(7):953–967, 2020.

[29] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. Millstein. Flashprofile: a framework for synthesizing data profiles. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[30] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. D. Millstein. Flashprofile: Interactive synthesis of syntactic profiles. 2017.

[31] P. Pantel, E. Crestan, A. Borkovsky, A.-M. Popescu, and V. Vyas. Web-scale distributional similarity and entity set expansion. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 938–947, 2009.

[32] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, volume 1, 2001.

[33] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing. Bootstrapping privacy compliance in big data systems. In *2014 IEEE Symposium on Security and Privacy*, pages 327–342. IEEE, 2014.

[34] J. Song and Y. He. Auto-validate: Unsupervised data validation using data-domain patterns inferred from data lakes. In *SIGMOD 2021 (to appear)*.

[35] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, and G. Miao. Recovering semantics of tables on the web. 2011.

[36] J. Wang, H. Wang, Z. Wang, and K. Q. Zhu. Understanding tables on the web. In *International Conference on Conceptual Modeling*, pages 141–155. Springer, 2012.

[37] R. C. Wang and W. W. Cohen. Language-independent set expansion of named entities using the web. In *Seventh IEEE international conference on data mining (ICDM 2007)*, pages 342–350. IEEE, 2007.

[38] C. Yan and Y. He. Synthesizing type-detection logic for rich semantic data types using open-source code. In *Proceedings of the 2018 International Conference on Management of Data*, pages 35–50, 2018.

[39] D. Zhang, Y. Suhara, J. Li, M. Hulsebos, Ç. Demiralp, and W.-C. Tan. Sato: Contextual semantic type detection in tables. *arXiv preprint arXiv:1911.06311*, 2019.

[40] C. Zhao and Y. He. Auto-em: End-to-end fuzzy entity-matching using pre-trained deep models and transfer learning. In *The World Wide Web Conference*, pages 2413–2424, 2019.

[41] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. Scope: parallel databases meet mapreduce. *The VLDB Journal*, 21(5):611–636, 2012.