FlashProfile: A Framework for Synthesizing Data Profiles

SASWAT PADHI*, University of California – Los Angeles, USA
PRATEEK JAIN, Microsoft Research, India
DANIEL PERELMAN, Microsoft Corporation, USA
OLEKSANDR POLOZOV, Microsoft Research, USA
SUMIT GULWANI, Microsoft Corporation, USA
TODD MILLSTEIN, University of California – Los Angeles, USA

We address the problem of learning a syntactic *profile* for a collection of strings, i.e. a set of regex-like patterns that succinctly describe the syntactic variations in the strings. Real-world datasets, typically curated from multiple sources, often contain data in various syntactic formats. Thus, any data processing task is preceded by the critical step of data format identification. However, manual inspection of data to identify the different formats is infeasible in standard big-data scenarios.

Prior techniques are restricted to a small set of pre-defined patterns (e.g. digits, letters, words, etc.), and provide no control over granularity of profiles. We define syntactic profiling as a problem of clustering strings based on *syntactic similarity*, followed by identifying patterns that succinctly describe each cluster. We present a technique for synthesizing such profiles over a given language of patterns, that also allows for interactive refinement by requesting a desired number of clusters.

Using a state-of-the-art inductive synthesis framework, PROSE, we have implemented our technique as FlashProfile. Across 153 tasks over 75 large real datasets, we observe a median profiling time of only ~0.7 s. Furthermore, we show that access to syntactic profiles may allow for more accurate synthesis of programs, i.e. using fewer examples, in programming-by-example (PBE) workflows such as Flash Fill.

CCS Concepts: • **Information systems → Clustering and classification**; *Summarization*; • **Software and its engineering → Programming by example**; *Domain specific languages*; • **Computing methodologies** → Anomaly detection;

Additional Key Words and Phrases: data profiling, pattern profiles, outlier detection, hierarchical clustering, pattern learning, program synthesis

ACM Reference Format:
Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 150 (November 2018), 28 pages. https://doi.org/10.1145/3276520

* Work done during an internship with PROSE team at Microsoft.

Authors' addresses: Saswat Padhi, Dept. of Computer Science, University of California – Los Angeles, CA, 90095, USA, padhi@cs.ucla.edu; Prateek Jain, Microsoft Research, Bangalore, India, prajain@microsoft.com; Daniel Perelman, Microsoft Corporation, Redmond, WA, 98052, USA, danpere@microsoft.com; Oleksandr Polozov, Microsoft Research, Redmond, WA, 98052, USA, polozov@microsoft.com; Sumit Gulwani, Microsoft Corporation, Redmond, WA, 98052, USA, sumitg@microsoft.com; Todd Millstein, Dept. of Computer Science, University of California – Los Angeles, CA, 90095, USA, todd@cs.ucla.edu.

2475-1421/2018/11-ART150
https://doi.org/10.1145/3276520

Reference ID
ISBN: ␣1-158-23466-X
not_available
doi: ␣10.1016/S1387-7003(03)00113-8
⋮
PMC9473786
ISBN: ␣0-006-08903-1
doi: ␣␣10.13039/100005795
PMC9035311
⋮
PMC5079771
ISBN: ␣2-287-34069-6

(a) Sample data

- `W_W` (5)
- `W: N.N/LN-N(N)N-D` (11)
- `W: D-N-N-L` (34)
- `W: N.N/N` (110)
- `W: D-N-N-D` (267)
- `WN` (1024)

Classes: [L]etter, [W]ord, [D]igit, [N]umber

(b) Profile from Ataccama One

- `doi:␣+10\.\d\d\d\d\d/\d+` (110)
- `.*` (113)
- `ISBN:␣0-\d\d\d-\d\d\d\d\d\d-\d` (204)
- `PMC\d+` (1024)

(c) Profile from Microsoft SSDT

- “not_available” (5)
- “doi:” ␣$^+$ “10.1016/” U D^4 “–” D^4 “(” D^2 “)” D^5 “–” D (11)
- “ISBN:” ␣ D “–” D^3 “–” D^5 “–X” (34)
- “doi:” ␣$^+$ “10.13039/” D^+ (110)
- “ISBN:” ␣ D “–” D^3 “–” D^5 “–” D (267)
- “PMC” D^7 (1024)

Classes: [U]ppercase, [D]igit

Superscripts indicate repetition of atoms.
Constant strings are surrounded by quotes.

(d) Default profile from FlashProfile

Fig. 1. Profiles for a set of references[1]— number of matches for each pattern is shown on the right

1 INTRODUCTION

In modern data science, most real-life datasets lack high-quality metadata — they are often incomplete, erroneous, and unstructured (Dong and Srivastava 2013). This severely impedes data analysis, even for domain experts. For instance, a merely preliminary task of *data wrangling* (importing, cleaning, and reshaping data) consumes 50 – 80% of the total analysis time (Lohr 2014). Prior studies show that high-quality metadata not only help users clean, understand, transform, and reason over data, but also enable advanced applications, such as compression, indexing, query optimization, and schema matching (Abedjan et al. 2015). Traditionally, data scientists engage in *data gazing* (Maydanchik 2007) — they manually inspect small samples of data, or experiment with aggregation queries to get a bird's-eye view of the data. Naturally, this approach does not scale to modern large-scale datasets (Abedjan et al. 2015).

Data profiling is the process of generating small but useful metadata (typically as a succinct summary) for the data (Abedjan et al. 2015). In this work, we focus on syntactic profiling, i.e. learning structural patterns that summarize the data. A syntactic profile is a disjunction of regex-like patterns that describe all of the syntactic variations in the data. Each pattern succinctly describes a specific variation, and is defined by a sequence of *atomic patterns* or *atoms*, such as digits or letters.

While existing tools, such as Microsoft SQL Server Data Tools (SSDT) (Microsoft 2017c), and Ataccama One (Ataccama 2017) allow pattern-based profiling, they generate a single profile that cannot be customized. In particular, **(1)** they use a small predetermined set of atoms, and do not allow users to supply custom atoms specific to their domains, and **(2)** they provide little support for controlling granularity, i.e. the number of patterns in the profile.

We present a novel application of program synthesis techniques to addresses these two key issues. Our implementation, FlashProfile, supports custom user-defined atoms that may encapsulate arbitrary pattern-matching logic, and also allows users to interactively control the granularity of generated profiles, by providing desired bounds on the number of patterns.

A Motivating Example. Fig. 1(a) shows a fragment of a dataset containing a set of references in various formats, and its profiles generated by Ataccama One (in Fig. 1(b)), Microsoft SSDT (in Fig. 1(c)), and our tool FlashProfile (in Fig. 1(d)). Syntactic profiles expose rare variations that are hard to notice by manual inspection of the data, or from simple statistical properties such as distribution of string lengths. For example, Ataccama One reveals a suspicious pattern “W_W”, which matches less than 0.5% of the dataset. SSDT, however, groups this together with other less frequent patterns into a “.*” pattern. Since SSDT does not provide a way of controlling the granularity of the profile, a user would be unable to further refine the “.*” pattern. FlashProfile shows that this pattern actually corresponds to missing entries, which read “not_available”.

[1] The full dataset is available at https://github.com/SaswatPadhi/FlashProfileDemo/tree/master/motivating_example.json.

For this dataset, although Ataccama One suggests a profile of the same granularity as from FlashProfile, the patterns in the profile are too coarse to be immediately useful. For instance, it may not be immediately obvious that the pattern `W: D-N-N-L` maps to ISBNs in the dataset. FlashProfile further qualifies the `W` (word) to the constant "ISBN", and restricts the [N]umber patterns to $\texttt{D}^3$ (short for Digit$^{\times 3}$) and $\texttt{D}^5$, and the final [L]etter to the constant "X".

FlashProfile also allows users familiar with their domains to define custom patterns, that cluster data in ways that are specific to the domain. For example, the two patterns for "doi" in Fig. 1(d) are vastly different — one contains letters and parentheses, whereas the other contains only digits. However, grouping them together makes the profile more readable, and helps spot outliers differing from the expected patterns. Fig. 2(a) shows a profile suggested by FlashProfile when provided with two custom atoms: ⟨DOI⟩ and ⟨ISBN10⟩,[2] with appropriate costs. Users may refine the profile to observe more specific variations within the DOIs and ISBNs. On requesting one more pattern, FlashProfile unfolds ⟨DOI⟩, since the DOIs are *more dissimilar* to each other than ISBNs, and produces the profile shown in Fig. 2(b).

- "not_available" (5)
- "doi:" ␣$^+$ ⟨DOI⟩ (121)
- "ISBN:" ␣ ⟨ISBN10⟩ (301)
- "PMC" $\texttt{D}^7$ (1024)

(a) Auto-suggested profile from FlashProfile

- "not_available" (5)
- "doi:" ␣$^+$ "10.1016/" U $\texttt{D}^4$ "-" $\texttt{D}^4$ "(" $\texttt{D}^2$ ")" $\texttt{D}^5$ "-" D (11)
- "doi:" ␣$^+$ "10.13039/" $\texttt{D}^+$ (110)
- "ISBN:" ␣ ⟨ISBN10⟩ (301)
- "PMC" $\texttt{D}^7$ (1024)

(b) A refined profile on requesting 5 patterns

Fig. 2. Custom atoms,[2] and refinement of profiles

Key Challenges. A key barrier to allowing custom atoms is the large search space for the desirable profiles. Prior tools restrict their atoms to letters and digits, followed by simple upgrades such as sequences of digits to numbers, and letters to words. However, this simplistic approach is not effective in the presence of several overlapping atoms and complex pattern-matching semantics. Moreover, a naïve exhaustive search over all profiles is prohibitively expensive. Every substring might be generalized in multiple ways into different atoms, and the search space grows exponentially when composing patterns as sequences of atoms, and a profile as a disjunction of patterns.

One approach to classifying strings into matching patterns might be to construct decision trees or random forests (Breiman 2001) with features based on atoms. However features are typically defined as predicates over entire strings, whereas atoms match specific substrings and may match multiple times within a string. Moreover, the location of an atomic match within a string depends on the lengths of the preceding atomic matches within that string. Therefore, this approach seems intractable since generating features based on atoms leads to an exponential blow up.

Instead, we propose to address the challenge of learning a profile by first *clustering* (Xu and Wunsch II 2005) — partitioning the dataset into *syntactically similar* clusters of strings and then learning a succinct pattern describing each cluster. This approach poses two key challenges: **(1)** efficiently learning patterns for a given cluster of strings over an arbitrary set of atomic patterns provided by the user, and **(2)** defining a suitable notion of *pattern-based similarity* for clustering, that is aware of the user-specified atoms. For instance, as we show in the motivating example (Fig. 1 and Fig. 2), the clustering must be sensitive to the presence of ⟨DOI⟩ and ⟨ISBN10⟩ atoms. Traditional character-based similarity measures over strings (Gomaa and Fahmy 2013) are ineffective for imposing a clustering that is susceptible to high-quality explanations using a given set of atoms.

Our Technique. We address both the aforementioned challenges by leveraging recent advances in *inductive program synthesis* (Gulwani et al. 2017) — an approach for learning programs from incomplete specifications, such as input-output examples for the desired program.

[2] ⟨DOI⟩ is defined as the regex `10.\d{4,9}/[-._;()/:A-Z0-9a-z]+`.
⟨ISBN10⟩ is defined as the regex `\d-\d{3}-\d{5}-[0-9Xx]`.

First, to address challenge (1), we present a novel domain-specific language (DSL) for patterns, and define a specification over a given set of strings. Our DSL provides constructs that allow users to easily augment it with new atoms. We then give an efficient synthesis procedure for learning patterns that are consistent with the specification, and a cost function to select compact patterns that are not overly general, out of all patterns that are consistent with a given cluster of strings.

Second, we observe that the cost function for patterns induces a natural *syntactic dissimilarity measure* over strings, which is the key to addressing challenge (2). We consider two strings to be similar if both can be described by a low-cost pattern. Strings requiring overly general / complex patterns are considered dissimilar. Typical clustering algorithms require computation of all pairwise dissimilarities (Xu and Wunsch II 2005). However, in contrast to standard clustering scenarios, computing dissimilarity for a pair of strings not only gives us a numeric measure, but also a pattern for them. That this allows for practical performance optimizations. In particular, we present a strategy to approximate dissimilarity computations using a small set of carefully sampled patterns.

To summarize, we present a framework for syntactic profiling based on clustering, that is parameterized by a pattern learner and a cost function. Fig. 3 outlines our interaction model. In the default mode, users simply provide their dataset. Additionally, they may control the performance *vs.* accuracy trade-off, define custom atoms, and provide bounds on the number of patterns. To enable efficient refinement of profiles based on the given bounds, we construct a *hierarchical clustering* (Xu and Wunsch II 2005, Section IIB) that may be *cut* at a suitable height to extract the desired number of clusters.

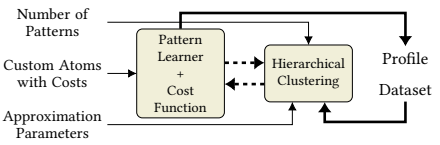


Fig. 3. FlashProfile's interaction model: *dashed* edges denote internal communication, and *thin* edges denote optional parameters to the system.

Evaluation. We have implemented our technique as FlashProfile using PROSE (Microsoft 2017d), also called FlashMeta (Polozov and Gulwani 2015), a state-of-the-art inductive synthesis framework. We evaluate our technique on 75 publicly-available datasets collected from online sources.[3] Over 153 tasks, FlashProfile achieves a median profiling time of 0.7s, 77% of which complete in under 2s. We show a thorough analysis of our optimizations, and a comparison with state-of-the-art tools.

Applications in PBE Systems. The benefits of syntactic profiles extend beyond data understanding. An emerging technology, programming by examples (PBE) (Gulwani et al. 2017; Lieberman 2001), provides end users with powerful semi-automated alternatives to manual data wrangling. For instance, they may use a tool like Flash Fill (Gulwani 2011), a popular PBE system for data transformations within Microsoft Excel and Azure ML Workbench (Microsoft 2017a,b). However, a key challenge to the success of PBE is finding a representative set of examples that best discriminates the desired program from a large space of possible programs (Mayer et al. 2015). Typically users provide the desired outputs over the first few entries, and Flash Fill then synthesizes the *simplest generalization* over them. However, this often results in incorrect programs, if the first few entries are not representative of the various formats present in the entire dataset (Mayer et al. 2015).

Instead, a syntactic profile can be used to select a representative set of examples from syntactically dissimilar clusters. We tested 163 scenarios where Flash Fill requires more than one input-output example to learn the desired transformation. In 80% of them, the examples belong to different syntactic clusters identified by FlashProfile. Moreover, we show that a *profile-guided interaction model* for Flash Fill, which we detail in § 6, is able to complete 86% of these tasks requiring the *minimum* number of examples. Instead of the user having to select a representative set of examples, our dissimilarity measure allows for proactively requesting the user to provide the desired output on an entry that is *most discrepant* with respect to those previously provided.

[3] All public datasets are available at: https://github.com/SaswatPadhi/FlashProfileDemo/tree/master/tests.

In summary, we make the following major contributions:

(§2) We formally define syntactic profiling as a problem of clustering of strings, followed by learning a succinct pattern for each cluster.

(§3) We show a hierarchical clustering technique that uses pattern learning to measure dissimilarity of strings, and give performance optimizations that further exploit the learned patterns.

(§4) We present a novel DSL for patterns, and give an efficient synthesis procedure with a cost function for selecting desirable patterns.

(§5) We evaluate FlashProfile's performance and accuracy on large real-life datasets, and provide a detailed comparison with state-of-the-art tools.

(§6) We present a profile-guided interaction model for Flash Fill, and show that data profiles may aid PBE systems by identifying a representative set of inputs.

2 OVERVIEW

Henceforth, the term *dataset* denotes a set of strings. We formally define a syntactic profile as:

Definition 2.1. Syntactic Profile — Given a dataset $\mathcal{S}$ and a desired number k of patterns, syntactic profiling involves learning (1) a partitioning $\mathcal{S}_1 \sqcup \ldots \sqcup \mathcal{S}_k = \mathcal{S}$, and (2) a set of patterns $\{P_1, \ldots, P_k\}$, where each P_i is an expression that describes the strings in $\mathcal{S}_i$. We call the disjunction of these patterns $\widetilde{P} = P_1 \vee \ldots \vee P_k$ a *syntactic profile* of $\mathcal{S}$, which describes all the strings in $\mathcal{S}$.

The goal of syntactic profiling is to learn a set of patterns that summarize a given dataset, but is neither too specific nor too general (to be practically useful). For example, the dataset itself is a trivial overly specific profile, whereas the regex ".*" is an overly general one. We propose a technique that leverages the following two key subcomponents to generate and rank profiles:[4]

- a pattern learner $\mathcal{L}: 2^{\mathbb{S}} \rightarrow 2^{\mathscr{L}}$, which generates a set of patterns over an arbitrary pattern language $\mathscr{L}$, that are consistent with a given dataset.
- a cost function $C: \mathscr{L} \times 2^{\mathbb{S}} \rightarrow \mathbb{R}_{\geq 0}$, which quantifies the suitability of an arbitrary pattern (in the same language $\mathscr{L}$) with respect to the given dataset.

Using $\mathcal{L}$ and C, we can quantify the suitability of clustering a set of strings together. More specifically, we can define a minimization objective $O: 2^{\mathbb{S}} \rightarrow \mathbb{R}_{\geq 0}$ that indicates an aggregate cost of a cluster. We can now define an *optimal syntactic profile* that minimizes O over a given dataset $\mathcal{S}$:

Definition 2.2. Optimal Syntactic Profile — Given a dataset $\mathcal{S}$, a desired number k of patterns, and access to a pattern learner $\mathcal{L}$, a cost function C for patterns, and a minimization objective O for partitions, we define: (1) the optimal partitioning $\widetilde{\mathcal{S}}_{opt}$ as one that minimizes the objective function O over all partitions, and (2) the optimal syntactic profile $\widetilde{P}_{opt}$ as the disjunction of the least-cost patterns describing each partition in $\widetilde{\mathcal{S}}_{opt}$. Formally,

$$\widetilde{\mathcal{S}}_{opt} \stackrel{\text{def}}{=} \underset{\substack{\{\mathcal{S}_1, \ldots, \mathcal{S}_k\} \\ \text{s.t. } \mathcal{S} = \bigsqcup_{i=1}^{k} \mathcal{S}_i}}{\arg\min} \sum_{i=1}^{k} O(\mathcal{S}_i) \qquad \text{and} \qquad \widetilde{P}_{opt} \stackrel{\text{def}}{=} \bigvee_{\mathcal{S}_i \in \widetilde{\mathcal{S}}_{opt}} \underset{P \in \mathcal{L}(\mathcal{S}_i)}{\arg\min}\ C(P, \mathcal{S}_i)$$

Ideally, we would define the aggregate cost of a partition as the minimum cost incurred by a pattern that describes it entirely. This is captured by $O(\mathcal{S}_i) \stackrel{\text{def}}{=} \min_{P \in \mathcal{L}(\mathcal{S}_i)} C(P, \mathcal{S}_i)$. However, with this objective, computing the optimal partitioning $\widetilde{\mathcal{S}}_{opt}$ is intractable in general. For an arbitrary learner $\mathcal{L}$ and cost function C, this would require exploring all k-partitionings of the dataset $\mathcal{S}$.[5]

[4] We denote the universe of all strings as $\mathbb{S}$, the set of non-negative reals as $\mathbb{R}_{\geq 0}$, and the power set of a set X as 2^X.

[5] The number of ways to partition a set $\mathcal{S}$ into k non-empty subsets is given by Stirling numbers of the second kind (Graham et al. 1994), $\left\{ {|\mathcal{S}| \atop k} \right\}$. When $k \ll |\mathcal{S}|$, the asymptotic value of $\left\{ {|\mathcal{S}| \atop k} \right\}$ is given by $\frac{k^{|\mathcal{S}|}}{k!}$.

Instead, we use an objective that is tractable and works well in practice — the aggregate cost of a cluster is given by the maximum cost of describing any two strings belonging to the cluster, using the best possible pattern. Formally, this objective is $\widehat{O}(\mathcal{S}_i) \stackrel{\text{def}}{=} \max_{x,y \in \mathcal{S}_i} \min_{P \in \mathcal{L}(\{x,y\})} C(P, \{x, y\})$. This objective is inspired by the *complete-linkage* criterion (Sørensen 1948), which is widely used in clustering applications across various domains (Jain et al. 1999). To minimize $\widehat{O}$, it suffices to only compute the costs of describing (at most) $|\mathcal{S}|^2$ pairs of strings in $\mathcal{S}$.

We outline our main algorithm Profile in Fig. 4. It is parameterized by an arbitrary learner $\mathcal{L}$ and cost function C. Profile accepts a dataset $\mathcal{S}$, the bounds $[m, M]$ for the desired number of patterns, and a sampling factor θ that decides the efficiency *vs.* accuracy trade-off. It returns the generated partitions paired with the least-cost patterns describing them: $\{\langle \mathcal{S}_1, P_1\rangle, \ldots, \langle \mathcal{S}_k, P_k\rangle\}$, where $m \leqslant k \leqslant M$.

```
func Profile⟨ℒ,C⟩(𝒮: String[ ], m: Int, M: Int, θ: Real)
output: P̃, a partitioning of 𝒮 with the associated patterns
        for each partition, such that m ⩽ |P̃| ⩽ M
1· H ← BuildHierarchy⟨ℒ,C⟩(𝒮, M, θ)   ;   P̃ ← {}
2· for all X ∈ Partition(H, m, M) do
3·     ⟨Pattern: P, Cost: c⟩ ← LearnBestPattern⟨ℒ,C⟩(X)
4·     P̃ ← P̃ ∪ {⟨Data: X, Pattern: P⟩}
5· return P̃
```

Fig. 4. Our main profiling algorithm

At a high level, we partition a dataset using the cost of patterns to induce a syntactic dissimilarity measure over its strings. For large enough θ, we compute all $O(|\mathcal{S}|^2)$ pairwise dissimilarities, and generate the partitioning $\widetilde{\mathcal{S}}_{opt}$ that minimizes $\widehat{O}$. However, many large real-life datasets have a very small number of syntactic clusters, and we notice that we can closely approximate $\widetilde{\mathcal{S}}_{opt}$ by sampling only a few pairwise dissimilarities. We invoke BuildHierarchy, in line 1, to construct a hierarchy H over $\mathcal{S}$ with accuracy controlled by θ. The hierarchy H is then *cut* at a certain height to obtain k clusters by calling Partition in line 2 — if $m \neq M$, k is heuristically decided based on the quality of clusters obtained at various heights. Finally, using LearnBestPattern, we learn a pattern P for each cluster X, and add it to the profile $\widetilde{P}$.

In the following subsections, we explain the two main components: **(1)** BuildHierarchy for building a hierarchical clustering, and **(2)** LearnBestPattern for pattern learning.

2.1 Pattern-Specific Clustering

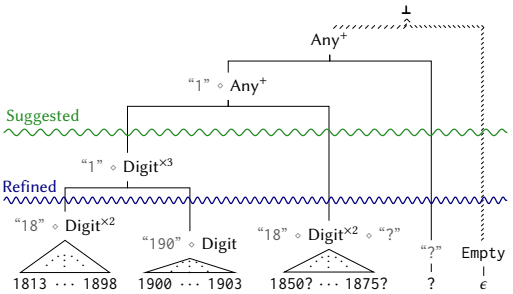
BuildHierarchy uses an agglomerative hierarchical clustering (AHC) (Xu and Wunsch II 2005, Section IIB) to construct a hierarchy (also called a dendrogram) that depicts a nested grouping of the given collection of strings, based on their syntactic similarity. Fig. 5 shows such a hierarchy over an incomplete and inconsistent dataset containing years, using the default set of atoms listed in Fig. 6. Once constructed, a hierarchy may be split at a suitable height to extract clusters of desired granularity, which enables a natural form of refinement — supplying a desired number of clusters. In contrast, flat clustering methods like k-means (MacQueen et al. 1967) generate a fixed partitioning within the same time complexity. In Fig. 5(b), we show a heuristically suggested split with 4 clusters, and a refined split on a request for 5 clusters. A key challenge to clustering is defining an appropriate pattern-specific measure of dissimilarity over strings, as we show below.

Example 2.3. Consider the pairs: $p = \{$ "1817", "1813?" $\}$ and $q = \{$ "1817", "1907" $\}$. Selecting the pair that is syntactically *more similar* is ambiguous, even for humans. The answer depends on the user's application — it may make sense to either cluster homogeneous strings (containing only digits) together, or to cluster strings with a longer common prefix together.

A natural way to resolve this ambiguity is to allow users to express their application-specific preferences by providing custom atoms, and then to make the clustering algorithm sensitive to the available atoms. Therefore we desire a dissimilarity measure that incorporates the user-specified atoms, and yet remains efficiently computable, since typical clustering algorithms compute dissimilarities between all pairs of strings (Xu and Wunsch II 2005).

Year
1900
1877
ϵ
1860
?
1866
ϵ
1893
⋮
1888 ?
1872

(a) Dataset [6]



(b) A hierarchy based on default atoms from Fig. 6

Fig. 5. A hierarchy with suggested and refined clusters: Leaf nodes represent strings, and internal nodes are labelled with patterns describing the strings below them. Atoms are concatenated using " ⋄ ". A dashed edge denotes the absence of a pattern that describes the strings together.

Lower $[a\text{–}z]$	Bin $[01]$
Upper $[A\text{–}Z]$	Digit $[0\text{–}9]$
⟨TitleCaseWord⟩ Upper ⋄ Lower+	Hex $[a\text{–}fA\text{–}F0\text{–}9]$
Alpha $[a\text{–}zA\text{–}Z]$	AlphaDigit $[a\text{–}zA\text{–}Z0\text{–}9]$
$\backslash s$	AlphaDigitSpace $[a\text{–}zA\text{–}Z0\text{–}9\backslash s]$
DotDash $[.\text{–}]$	Punct $[.,:?/\text{–}]$
AlphaDash $[a\text{–}zA\text{–}Z\text{–}]$	Symb $[\text{–}.,:?/@\#\$\%\&\cdots]$
AlphaSpace $[a\text{–}zA\text{–}Z\backslash s]$	Base64 $[a\text{–}zA\text{–}Z0\text{–}9+\backslash =]$

Fig. 6. Default atoms in FlashProfile, with their regex: We also allow "Any" atom that matches any character.

Syntactic Dissimilarity. Our key insight is to leverage program synthesis techniques to efficiently learn patterns describing a given set of strings, and induce a dissimilarity measure using the learned patterns — overly general or complex patterns indicate a high degree of syntactic dissimilarity.

In §3.1, we formally define the dissimilarity measure η as the minimum cost incurred by any pattern for describing a given pair of strings, using a specified pattern learner $\mathcal{L}$ and cost function C. We evaluate our measure η in §5.1, and demonstrate that for estimating syntactic similarity it is superior to classical character-based measures (Gomaa and Fahmy 2013), and simple machine-learned models such as random forests based on intuitive features.

Adaptive Sampling and Approximation. While η captures a high-quality syntactic dissimilarity, with it, each pairwise dissimilarity computation requires learning and scoring of patterns, which may be expensive for large real datasets. To allow end users to quickly generate approximately correct profiles for large datasets, we present a two-stage sampling technique. **(1)** At the top-level, FlashProfile employs a Sample–Profile–Filter cycle: we sample a small subset of the data, profile it, and filter out data that is described by the profile learned so far. **(2)** While profiling each sample, our BuildHierarchy algorithm approximates some pairwise dissimilarities using previously seen patterns. We allow end users to control the degree of approximations using two optional parameters.

Our key insight that, unlike typical clustering scenarios, computing dissimilarity between a pair of strings gives us more than just a measure — we also learn a pattern. We test this pattern on other pairs to approximate their dissimilarity, which is typically faster than learning new patterns. Our technique is inspired by counter-example guided inductive synthesis (CEGIS) (Solar-Lezama et al. 2006). CEGIS was extended to sub-sampling settings by Raychev et al. (2016), but they synthesize a single program and require the outputs for all inputs. In contrast, we learn a disjunction of several programs, the outputs for which over a dataset, i.e. the partitions, are unknown a priori.

Example 2.4. The pattern "PMC" ⋄ Digit×7 learned for the string pair {"PMC2536233", "PMC4901429"}, also describes the string pair {"PMC4901429", "PMC2395569"}, and may be used to accurately estimate their dissimilarity without invoking learning again.

However this sampling needs to be performed carefully for accurate approximations. Although the pattern "1" ⋄ Digit×3 learned for {"1901", "1875"}, also describes {"1872", "1875"}, there exists another pattern "187" ⋄ Digit, which indicates a much lower syntactic dissimilarity. We propose an adaptive algorithm for sampling patterns based on previously observed patterns and strings in the dataset. Our sampling and approximation algorithms are detailed in §3.2 and §3.3 respectively.

[6] Linda K. Jacobs, The Syrian Colony in New York City 1880-1900. http://bit.ly/LJacobs

2.2 Pattern Learning via Program Synthesis

An important aspect of our clustering-based approach to profiling, described in §2.1, is its generality. It is agnostic to the specific language $\mathscr{L}$ in which patterns are expressed, as long as appropriate pattern learner $\mathcal{L}$ and cost function C are provided for $\mathscr{L}$.

LearnBestPattern, listed in Fig. 7, first invokes $\mathcal{L}$ to learn a set V of patterns each of which describes all strings in $\mathcal{S}$. If pattern learning fails,[7] in line 2, we return the special pattern $\perp$ and a very high cost ∞. Otherwise, we return the pattern that has the minimum cost using C w.r.t. $\mathcal{S}$. LearnBestPattern is used during clustering to compute pairwise dissimilarity and finally compute the least-cost patterns for clusters.

```
func LearnBestPattern_(ℒ,C)(𝒮: String[ ])
output: The least-cost pattern and its cost, for 𝒮
1· V ← ℒ(𝒮)
2· if V = {} then return ⟨Pattern: ⊥, Cost: ∞⟩
3· P ← arg min_{P∈V} C(P, 𝒮)
4· return ⟨Pattern: P, Cost: C(P, 𝒮)⟩
```

Fig. 7. Learning the best pattern for a dataset

A natural approach to learning patterns is *inductive program synthesis* (Gulwani et al. 2017), which generalizes a given specification to desired programs over a domain-specific language (DSL). We propose a rich DSL for patterns, and present an efficient inductive synthesizer for it.

Language for Patterns. Our DSL $\mathscr{L}_{\text{FP}}$ is designed to support efficient synthesis using existing technologies while still being able to express rich patterns for practical applications. A pattern is an arbitrary sequence of atomic patterns (atoms), each containing low-level logic for matching a sequence of characters. A pattern $P \in \mathscr{L}_{\text{FP}}$ *describes* a string s, i.e. $P(s) = \texttt{True}$, iff the atoms in P match contiguous non-empty substrings of s, ultimately matching s in its entirety. FlashProfile uses a default set of atoms listed in Fig. 6, which may be augmented with new regular expressions, constant strings, or ad hoc functions. We formally define our language $\mathscr{L}_{\text{FP}}$ in §4.1.

Pattern Synthesis. The inductive synthesis problem for pattern learning is: given a set of strings $\mathcal{S}$, learn a pattern $P \in \mathscr{L}_{\text{FP}}$ such that $\forall s \in \mathcal{S}$: $P(s) = \texttt{True}$. Our learner $\mathcal{L}_{\text{FP}}$ decomposes the synthesis problem for P over the strings in $\mathcal{S}$ into synthesis problems for individual atoms in P over appropriate substrings. However, a naïve approach of tokenizing each string to (exponentially many) sequences of atoms, and computing their intersection is simply impractical. Instead, $\mathcal{L}_{\text{FP}}$ computes the intersection incrementally at each atomic match, using a novel decomposition technique.

$\mathcal{L}_{\text{FP}}$ is implemented using PROSE (Microsoft 2017d; Polozov and Gulwani 2015), a state-of-the-art inductive synthesis framework. PROSE requires DSL designers to define the logic for decomposing a synthesis problem over an expression to those over its subexpressions, which it uses to automatically generate an efficient synthesizer for their DSL. We detail our synthesis procedure in §4.2.

Cost of Patterns. Once a set of patterns has been synthesized, a variety of strategies may be used to identify the most desirable one. Our cost function C_{FP} is inspired by *regularization* (Tikhonov 1963) techniques that are heavily used in statistical learning to construct generalizations that do not overfit to the data. C_{FP} decides a trade-off between two opposing factors: **(1)** *specificity:* prefer a pattern that is not general, and **(2)** *simplicity:* prefer a compact pattern that is easy to interpret.

Example 2.5. The strings {"Male", "Female"} are matched by the patterns Upper ⋄ Lower$^+$, and Upper ⋄ Hex ⋄ Lower$^+$. Although the latter is more specific, it is overly complex. On the other hand, the pattern Alpha$^+$ is simpler and easier to interpret, but is overly general.

To this end, each atom in $\mathscr{L}_{\text{FP}}$ has a fixed *static cost* similar to fixed *regularization hyperparameters* used in machine learning (Bishop 2016), and a dataset-driven *dynamic weight*. The cost of a pattern is the weighted sum of the cost of its constituent atoms. In §4.3, we detail our cost function C_{FP}, and provide some guidelines on assigning costs for new atoms defined by users.

[7] Pattern learning may fail, for example, if the language $\mathscr{L}$ is too restrictive and no pattern can describe the given strings.

3 HIERARCHICAL CLUSTERING

We now detail our clustering-based approach for generating syntactic profiles and show practical optimizations for fast approximately-correct profiling. In §3.1 – §3.4, we explain these in the context of a small chunk of data drawn from a large dataset. In §3.5, we then discuss how profile large datasets by generating profiles for as many chunks as necessary and combining them.

Recall that our first step in PROFILE is to build a hierarchical clustering over the data. The BUILDHIERARCHY procedure, listed in Fig. 8, constructs a hierarchy H over a given dataset $\mathcal{S}$, with parameters M and θ. M is the maximum number of clusters in a desired profile. The *pattern sampling factor* θ decides the performance *vs.* accuracy trade-off while constructing H.

func BUILDHIERARCHY$_{(\mathcal{L},C)}(\mathcal{S}$: String[ ], M: Int, θ: Real)
output: A hierarchical clustering over $\mathcal{S}$
1 · $D \leftarrow$ SAMPLEDISSIMILARITIES$_{(\mathcal{L},C)}(\mathcal{S}, \lceil \theta M \rceil)$
2 · $A \leftarrow$ APPROXDMATRIX$(\mathcal{S}, D)$
3 · **return** $AHC(\mathcal{S}, A)$

Fig. 8. Building an approximately-correct hierarchy[8]

Henceforth, we use *pair* to denote a pair of strings. In line 1 of BUILDHIERARCHY, we first sample pairwise dissimilarities, i.e. the best patterns and their costs, for a small set (based on the θ factor) of string pairs. Specifically, out of all $O(|\mathcal{S}|^2)$ pairs within $\mathcal{S}$, we adaptively sample dissimilarities for only $O(\theta M|\mathcal{S}|)$ pairs by calling SAMPLEDISSIMILARITIES, and cache the learned patterns in D. We formally define the dissimilarity measure in §3.1, and describe SAMPLEDISSIMILARITIES in §3.2. The cache D is then used by APPROXDMATRIX, in line 2, to complete the dissimilarity matrix A over $\mathcal{S}$, using approximations wherever necessary. We describe these approximations in §3.3. Finally, a standard agglomerative hierarchical clustering (AHC) (Xu and Wunsch II 2005, Section IIB) is used to construct a hierarchy over $\mathcal{S}$ using the matrix A.

3.1 Syntactic Dissimilarity

We formally define our syntactic dissimilarity measure as follows:

Definition 3.1. Syntactic Dissimilarity — For a given pattern learner $\mathcal{L}$ and a cost function C over an arbitrary language of patterns $\mathscr{L}$, we define the syntactic dissimilarity between strings $x, y \in \mathbb{S}$ as the minimum cost incurred by a pattern in $\mathscr{L}$ to describe them together, i.e.

$$\eta(x,y) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x = y \\ \infty & \text{if } x \neq y \ \wedge\ V = \{\} \\ \min\limits_{P \in V} C\big(P, \{x,y\}\big) & \text{otherwise} \end{cases}$$

where $V = \mathcal{L}\big(\{x,y\}\big) \subseteq \mathscr{L}$ is the set of patterns that describe strings x and y, and ∞ denotes a high cost for a failure to describe x and y together using patterns learned by $\mathcal{L}$.

The following example shows some candidate patterns and their costs encountered during dissimilarity computation for various pairs. The actual numbers depend on the pattern learner and cost function used, in this case FlashProfile's $\mathcal{L}_{\text{FP}}$ and C_{FP}, which we describe in §4. However, this example highlights the desirable properties for a natural measure of syntactic dissimilarity.

Example 3.2. For three pairs, we show the shortcomings of classical character-based similarity measures. We compare the Levenshtein distance (LD) (Levenshtein 1966) for these pairs against the pattern-based dissimilarity η computed with our default atoms from Fig. 6. On the right, we also show the least-cost pattern, and two other randomly sampled patterns that describe the pair.

First, we compare two dates both using the same syntactic format "YYYY-MM-DD":

(a) `1990-11-23` / `2001-02-04` LD = 8 *vs.* $\eta = 4.96$

4.96	Digit$^{\times 4}$ ◦ "-" ◦ Digit$^{\times 2}$ ◦ "-" ◦ Digit$^{\times 2}$
179.9	Hex^{+} ◦ Symb ◦ Hex^{+} ◦ "-" ◦ Hex^{+}
46482	Digit^{+} ◦ Punct ◦ Any^{+}

[8] $\lceil x \rceil$ denotes the *ceiling* of x, i.e. $\lceil x \rceil = \min\{m \in \mathbb{Z} \mid m \geqslant x\}$.

Syntactically, these dates are very similar — they use the same delimiter "-", and have same width for the numeric parts. The best pattern found by FlashProfile captures exactly these features. However, Levenshtein distance for these dates is higher than the following dates which uses a different delimiter and a different order for the numeric parts:

(b) `1990-11-23` / `29/05/1923` LD = 5 *vs.* $\eta = 30.2$

30.2	$\text{Digit}^+ \diamond \text{Punct} \diamond \text{Digit}^{\times 2} \diamond \text{Punct} \diamond \text{Digit}^+$
318.6	$\text{Digit}^+ \diamond \text{Symb} \diamond \text{Digit}^+ \diamond \text{Symb} \diamond \text{Digit}^+$
55774	$\text{Digit}^+ \diamond \text{Punct} \diamond \text{Any}^+$

The Levenshtein distance is again lower for the following pair containing a date and an ISBN code:

(c) `1990-11-23` / `899-2119-33-X` LD = 7 *vs.* $\eta = 23595$

23595	$\text{Digit}^+ \diamond \text{"-"} \diamond \text{Digit}^+ \diamond \text{"-"} \diamond \text{Any}^+$
55415	$\text{Digit}^+ \diamond \text{Punct} \diamond \text{Any}^+$
92933	Any^+

The same trend is also observed for Jaro-Winkler (Winkler 1999), and other measures based on edit distance (Gomaa and Fahmy 2013). Whereas these measures look for exact matches on characters, pattern-based measures have the key advantage of being able to generalize substrings to atoms.

3.2 Adaptive Sampling of Patterns

Although η accurately captures the syntactic dissimilarity of strings over an arbitrary language of patterns, it requires pattern learning and scoring for every pairwise dissimilarity computation, which is computationally expensive. While this may not be a concern for non-realtime scenarios, such as profiling large datasets on cloud-based datastores, we provide a tunable parameter to end users to be able to generate approximately correct profiles for large datasets in real time.

Besides a numeric measure of dissimilarity, computing η over a pair also generates a pattern that describes the pair. Since the patterns generalize substrings to atoms, often the patterns learned for one pair also describe many other pairs. We aim to sample a subset of patterns that are likely to be sufficient for constructing a hierarchy accurate until M levels, i.e. $1 \leqslant k \leqslant M$ clusters extracted from this hierarchy should be identical to k clusters extracted from a hierarchy constructed without approximations. Our SampleDissimilarities algorithm, shown in Fig. 9, is inspired by the seeding technique of k-means++ (Arthur and Vassilvitskii 2007). Instead of computing all pairwise dissimilarities for pairs in $\mathcal{S} \times \mathcal{S}$, we compute the dissimilarities for pairs in $\rho \times \mathcal{S}$, where set ρ is a carefully selected small set of *seed strings*. The patterns learned during this process are likely to be sufficient for accurately estimating the dissimilarities for the remaining pairs.

SampleDissimilarities takes a dataset $\mathcal{S}$ and a factor $\widehat{M}$, and it samples dissimilarities for $O(\widehat{M}|\mathcal{S}|)$ pairs. It iteratively selects a set ρ containing $\widehat{M}$ strings that are most dissimilar to each other. Starting with a random string in ρ, in each iteration, at line 6, it adds the string $x \in \mathcal{S}$ such that it is as dissimilar as possible, even with its most-similar neighbor in ρ. In the end, the set D only contains dissimilarities for pairs in $\mathcal{S} \times \rho$, computed at line 5. Recall that, $\widehat{M}$ is controlled by the pattern sampling factor θ. In line 1 of BuildHierarchy (in Fig. 8), we set $\widehat{M} = \lceil \theta M \rceil$.

func $\text{SampleDissimilarities}_{\langle \mathcal{L}, C \rangle}(\mathcal{S}: \text{String}[\,], \widehat{M}: \text{Int})$
output: A dictionary mapping $O(\widehat{M}|\mathcal{S}|)$ pairs of strings from $\mathcal{S}$, to the best pattern describing each pair and its cost

1· $D \leftarrow \{\}$; $a \leftarrow$ a random string in $\mathcal{S}$; $\rho \leftarrow \{a\}$
2· **for** $i \leftarrow 1$ **to** $\widehat{M}$ **do**
3· **for all** $b \in \mathcal{S}$ **do**
4· $D[a, b] \leftarrow \text{LearnBestPattern}_{\langle \mathcal{L}, C \rangle}(\{a, b\})$
▶ Pick the most dissimilar string w.r.t. strings already in ρ.
5· $a \leftarrow \arg\max_{x \in \mathcal{S}} \min_{y \in \rho} D[y, x].\text{Cost}$
6· $\rho \leftarrow \rho \cup \{a\}$
7· **return** D

Fig. 9. Adaptively sampling a small set of patterns

Since the user may request up to at most M clusters, θ must be at least 1.0, so that we pick at least one seed string from each cluster to ρ. Then, computing the dissimilarities with *all* other strings in the dataset would ensure we have a good distribution of patterns that describe intra– and inter– cluster dissimilarities, even for the finest granularity clustering with M clusters.

Example 3.3. Consider the dataset containing years in Fig. 5(a). Starting with a random string, say "1901"; the set ρ of seed strings grows as shown below, with increasing $\widehat{M}$. At each step, NN (nearest neighbor) shows the new string added to ρ paired with its most similar neighbor.

$\widehat{M} = 2 \mid NN = \langle \epsilon, \text{"1901"} \rangle \quad \rho = \{\text{"1901"}, \epsilon\}$
$\widehat{M} = 3 \mid NN = \langle \text{"?"}, \text{"1901"} \rangle \quad \rho = \{\text{"?"}, \text{"1901"}, \epsilon\}$
$\widehat{M} = 4 \mid NN = \langle \text{"1875?"}, \text{"1901"} \rangle \quad \rho = \{\text{"1875?"}, \text{"?"}, \text{"1901"}, \epsilon\}$
$\widehat{M} = 5 \mid NN = \langle \text{"1817"}, \text{"1875?"} \rangle \quad \rho = \{\text{"1817"}, \text{"1875?"}, \text{"?"}, \text{"1901"}, \epsilon\}$
$\widehat{M} = 6 \mid NN = \langle \text{"1898"}, \text{"1817"} \rangle \quad \rho = \{\text{"1898"}, \text{"1817"}, \text{"1875?"}, \text{"?"}, \text{"1901"}, \epsilon\}$

3.3 Dissimilarity Approximation

Now we present our technique for completing a dissimilarity matrix over a dataset $\mathcal{S}$, using the patterns sampled from the previous step. Note that, for a large enough value of the pattern sampling factor, i.e. $\theta \geqslant \frac{|\mathcal{S}|}{M}$, we would sampled all pairwise dissimilarities and no approximation would be necessary. For smaller values of θ, we use the patterns learned while computing η over $\rho \times \mathcal{S}$ to approximate the remaining pairwise dissimilarities in $\mathcal{S} \times \mathcal{S}$. The key observation here is that, testing whether a pattern describes a string is typically much faster than learning a new pattern.

The ApproxDMatrix procedure, listed in Fig. 10, uses the dictionary D of patterns from SampleDissimilarities to generate a matrix A of all pairwise dissimilarities over $\mathcal{S}$. Lines 7 and 8 show the key approximation steps for a pair $\{x, y\}$. In line 7, we test the patterns in D, and select a set V of them containing only those which describe both x and y. We then compute their new costs relative to $\{x, y\}$, in line 8, and select the least cost as an approximation of $\eta(x, y)$. If V turns out to be empty, i.e. no sampled pattern describes both x and y, then, in line 10, we call LearnBestPattern to compute $\eta(x, y)$. We also add the new pattern to D for use in future approximations.

```
func ApproxDMatrix_(ℒ,C)(𝒮: String[ ],
                         D: String × String ↦ Pattern × Real)
output: A matrix A of all pairwise dissimilarities over strings in 𝒮
1·  A ← {}
2·  for all x ∈ 𝒮 do
3·    for all y ∈ 𝒮 do
4·      if x = y then A[x, y] ← 0
5·      else if ⟨x, y⟩ ∈ D then A[x, y] ← D[x, y].Cost
6·      else
▶         Select the least cost pattern that describes x and y.
7·        V ← {P | ⟨Pattern: P, Cost: ·⟩ ∈ D ∧ P(x) ∧ P(y)}
8·        if V ≠ {} then A[x, y] ← min_{P ∈ V} C(P, {x, y})
9·        else
▶           Compute η(s, y), and store the learned pattern.
10·         D[x, y] ← LearnBestPattern_(ℒ,C)({x, y})
11·         A[x, y] ← D[x, y].Cost
12· return A
```

Fig. 10. Approximating a complete dissimilarity matrix

Although $\theta = 1.0$ ensures that we pick a seed string from each final cluster, in practice we use a θ that is slightly greater than 1.0. This allows us to sample a few more seed strings, and ensures a better distribution of patterns in D at the cost of a negligible performance overhead. In practice, it rarely happens that no sampled pattern describes a new pair (at line 9, Fig. 10), since seed patterns for inter-cluster string pairs are usually overly general, as we show in the example below.

Example 3.4. Consider a dataset $\mathcal{S} = \{$"07-jun", "aug-18", "20-feb", "16-jun", "20-jun"$\}$. Assuming $M = 2$ and $\theta = 1.0$ (i.e. $\widehat{M} = 2$), suppose we start with the string "20-jun". Then, following the SampleDissimilarities algorithm shown in Fig. 9, we would select $\rho = \{$ "20-jun", "aug-18" $\}$, and would sample the following seed patterns into D based on patterns defined over our default atoms (listed in Fig. 6) and constant string literals:

(a) D["20-jun","07-jun"] $\mapsto$ Digit$^{\times 2}$ $\diamond$ "-jun", and
(b) D["20-jun","20-feb"] $\mapsto$ "20-" $\diamond$ Lower$^{\times 3}$,
(c) D["20-jun","16-jun"] $\mapsto$ Digit$^{\times 2}$ $\diamond$ "-jun", and
(d) D["20-jun","aug-18"], D["aug-18","07-jun"], D["aug-18","20-feb"], D["aug-18","16-jun"] $\mapsto$ AlphaDigit^{+} $\diamond$ "-" $\diamond$ AlphaDigit^{+}.

Next, we estimate η ("16-jun", "20-feb") using these patterns. None of *(a)* — *(c)* describe the pair, but *(d)* does. However, it is overly general compared to the least-cost pattern, $\text{Digit}^{\times 2} \diamond$ "-" $\diamond \text{Lower}^{\times 3}$.

As in the case above, depending on the expressiveness of the pattern language, for a small θ the sampled patterns may be too specific to be useful. With a slightly higher $\theta = 1.25$, i.e. $\widehat{M} = \lceil \theta M \rceil = 3$, we would also select "07-jun" as a seed string in ρ, and sample the desired while computing D["07-jun","20-feb"]. We evaluate the impact of θ on performance and accuracy in §5.2.

3.4 Hierarchy Construction and Splitting

Once we have a dissimilarity matrix, we use a standard agglomerative hierarchical clustering (AHC) (Xu and Wunsch II 2005, Section IIB) algorithm, as outlined in Fig. 11. Note that AHC is not parameterized by $\mathcal{L}$ and C, since it does not involve learning or scoring of patterns any more.

func AHC($\mathcal{S}$: String[ ], A: String × String ↦ Real)
output: A hierarchy over $\mathcal{S}$ using dissimilarity matrix A
1 · $H \leftarrow \{\{s\} \mid s \in \mathcal{S}\}$
2 · **while** $|H| > 1$ **do**
3 · $\quad \langle X, Y \rangle \leftarrow \arg\min_{X,Y \in H} \widehat{\eta}(X, Y \mid A)$
4 · $\quad H \leftarrow (H \setminus \{X, Y\}) \cup \{\{X, Y\}\}$
5 · **return** H

Fig. 11. A standard algorithm for AHC

We start with each string in a singleton set (leaf nodes of the hierarchy). Then, we iteratively join the least-dissimilar pair of sets, until we are left with a single set (root of the hierarchy). AHC relies on a *linkage criterion* to estimate dissimilarity of sets of strings. We use the classic complete-linkage (also known as further-neighbor linkage) criterion (Sørensen 1948), which has been shown to be resistant to outliers, and yield useful hierarchies in practical applications (Jain et al. 1999).

Definition 3.5. Complete-Linkage — For a set $\mathcal{S}$ and a dissimilarity matrix A defined on $\mathcal{S}$, given two arbitrarily-nested clusters X and Y over a subset of entities in $\mathcal{S}$, we define the dissimilarity between their contents (the *flattened* sets $\overline{X}, \overline{Y} \subseteq \mathcal{S}$, respectively) as:

$$\widehat{\eta}(X, Y \mid A) \stackrel{\text{def}}{=} \max_{x \in \overline{X}, y \in \overline{Y}} A[x, y]$$

Once a hierarchy has been constructed, our PROFILE algorithm (in Fig. 4) invokes the PARTITION method (at line 2) to extract k clusters within the provided bounds $[m, M]$. If $m \neq M$, we use a heuristic based on the *elbow* (also called *knee*) method (Halkidi et al. 2001): between the top m^{th} and the M^{th} nodes, we split the hierarchy till the knee — a node below which the average intra-cluster dissimilarity does not vary significantly. A user may request $m = k = M$, in which case PARTITION simply splits the top k nodes of the hierarchy to generate k clusters.

3.5 Profiling Large Datasets

To scale our technique to large datasets, we now describe a second round of sampling. Recall that in SAMPLEDISSIMILARITIES, we sample dissimilarities for $O(\theta M|\mathcal{S}|)$ pairs. However, although θM is very small, $|\mathcal{S}|$ is still very large for real-life datasets. In order to address this, we run our PROFILE algorithm from Fig. 4 on small chunks of the dataset, and combine the generated profiles.

We outline our BIGPROFILE algorithm in Fig. 12. This algorithm accepts a new *string sampling factor* $\mu \geqslant 1$, which controls the size of chunks profiled in each iteration, until we have profiled all the strings in $\mathcal{S}$. In §5.3, we evaluate the impact of μ on performance and accuracy.

We start by selecting a random subset X of size $\lceil \mu M \rceil$ from $\mathcal{S}$ in line 3. In line 4, we obtain a profile $\widetilde{P}'$ of X, and merge it with the global profile $\widetilde{P}$ in line 5. We repeat this loop with the remaining strings in $\mathcal{S}$ that do not match the global profile. While merging $\widetilde{P}$ and $\widetilde{P}'$ in line 5, we may exceed the maximum number of patterns M, and may need to *compress* the profile.

func BIGPROFILE$_{\langle \mathcal{L}, C \rangle}$($\mathcal{S}$: String[ ], m: Int, M: Int, θ: Real, μ: Real)
output: A profile $\widetilde{P}$ that satisfies $m \leqslant |\widetilde{P}| \leqslant M$
1 · $\widetilde{P} \leftarrow \{\}$
2 · **while** $|\mathcal{S}| > 0$ **do**
3 · $\quad X \leftarrow$ SAMPLERANDOM($\mathcal{S}$, $\lceil \mu M \rceil$)
4 · $\quad \widetilde{P}' \leftarrow$ PROFILE$_{\langle \mathcal{L}, C \rangle}(X, m, M, \theta)$
5 · $\quad \widetilde{P} \leftarrow$ COMPRESSPROFILE$_{\langle \mathcal{L}, C \rangle}(\widetilde{P} \cup \widetilde{P}', M)$
6 · $\quad \mathcal{S} \leftarrow$ REMOVEMATCHINGSTRINGS($\mathcal{S}$, $\widetilde{P}$)
7 · **return** $\widetilde{P}$

Fig. 12. Profiling large datasets

For brevity, we elide the details of SampleRandom and RemoveMatchingStrings, which have straightforward implementations. In Fig. 13 we outline CompressProfile. It accepts a profile $\widetilde{P}$ and shrinks it to at most M patterns. The key idea is to repeatedly merge the most similar pair of patterns in $\widetilde{P}$. However, we cannot compute the similarity between patterns. Instead, we estimate it using syntactic similarity of the associated data partitions. The profile $\widetilde{P}$ must be of the same type as returned by Profile, i.e. a set of pairs, each containing a data partition and its pattern. In line 2, we first identify the partitions $\langle X, Y\rangle$ which are the most similar, i.e. require the least cost pattern for describing them together. We then merge X and Y to Z, learn a pattern describing Z, and update $\widetilde{P}$ by replacing X and Y with Z and its pattern. This process is repeated until the total number of patterns falls to M.

func CompressProfile$_{\langle\mathcal{L},C\rangle}(\widetilde{P}$: ref Profile, M: Int)
output: A compressed profile $\widetilde{P}$ that satisfies $|\widetilde{P}| \leqslant M$

1 · **while** $|\widetilde{P}| > M$ **do**
▶ Compute the most similar partitions in the profile so far.
2 · $\langle X, Y\rangle \leftarrow \arg\min_{X,Y\in\widetilde{P}} [\text{LearnBestPattern}_{\langle\mathcal{L},C\rangle}(X.\text{Data} \cup Y.\text{Data})].\text{Cost}$
▶ Merge partitions $\langle X, Y\rangle$, and update $\widetilde{P}$.
3 · $Z \leftarrow X.\text{Data} \cup Y.\text{Data}$
4 · $P \leftarrow \text{LearnBestPattern}_{\langle\mathcal{L},C\rangle}(Z).\text{Pattern}$
5 · $\widetilde{P} \leftarrow (\widetilde{P} \setminus \{X, Y\}) \cup \{\langle \text{Data: } Z, \text{Pattern: } P\rangle\}$
6 · **return** $\widetilde{P}$

Fig. 13. Limiting the number of patterns in a profile

Theorem 3.6 (Termination). *Over an arbitrary language $\mathscr{L}$ of patterns, assume an arbitrary learner $\mathcal{L}: 2^{\mathbb{S}} \to 2^{\mathscr{L}}$ and a cost function $C: \mathscr{L} \times 2^{\mathbb{S}} \to \mathbb{R}_{\geq 0}$, such that for any finite dataset $\mathcal{S} \subset \mathbb{S}$, we have: (1) $\mathcal{L}(\mathcal{S})$ terminates and produces a finite set of patterns, and (2) $C(P, \mathcal{S})$ terminates for all $P \in \mathscr{L}$. Then, the* BigProfile *procedure (Fig. 12) terminates on any finite dataset $\mathcal{S} \subset \mathbb{S}$, for arbitrary valid values of the optional parameters m, M, θ and μ.*

Proof. We note that in BigProfile, the loop within lines 2 – 6 runs for at most $\frac{|\mathcal{S}|}{\lceil \mu M\rceil}$ iterations, since at least $\lceil \mu M\rceil$ strings are removed from $\mathcal{S}$ in each iteration. Therefore, to prove termination of BigProfile, it is sufficient to show that Profile and CompressProfile terminate.

First, we note that termination of LearnBestPattern immediately follows from (1) and (2). Then, it is easy to observe that CompressProfile terminates as well: (1) the loop in lines 1 – 5 runs for at most $|\widetilde{P}| - M$ iterations, and (2) LearnBestPattern is invoked $O(|\widetilde{P}|^2)$ times in each iteration.

The Profile procedure (Fig. 4) makes at most $O\big((\mu M)^2\big)$ calls to LearnBestPattern (Fig. 7) to profile the $\lceil \mu M\rceil$ strings sampled in to X — at most $O\big((\mu M)^2\big)$ calls within BuildHierarchy (Fig. 8), and $O(M)$ calls to learn patterns for the final partitions. Depending on θ, BuildHierarchy may make many fewer calls to LearnBestPattern. However, it makes no more than 1 such call per pair of strings in X, to build the dissimilarity matrix. Therefore, Profile terminates as well. □

4 PATTERN SYNTHESIS

We now describe the specific pattern language, leaning technique and cost function used to instantiate our profiling technique as FlashProfile. We begin with a brief description our pattern language in §4.1, present our pattern synthesizer in §4.2, and conclude with our cost function in §4.3.

4.1 The Pattern Language $\mathscr{L}_{\text{FP}}$

Fig. 14(a) shows the formal syntax for our pattern language $\mathscr{L}_{\text{FP}}$. Each pattern $P \in \mathscr{L}_{\text{FP}}$ is a predicate defined on strings, i.e. a function P: String $\to$ Bool, which embodies a set of constraints over strings. A pattern P *describes* a given string s, i.e. $P(s) = \texttt{True}$, iff s satisfies all constraints imposed by P. Patterns in $\mathscr{L}_{\text{FP}}$ are composed of atomic patterns:

Definition 4.1. Atomic Pattern (or Atom) — An atom, α: String $\to$ Int is a function, which given a string s, returns the length of the longest prefix of s that satisfies its constraints. Atoms only match non-empty prefixes. $\alpha(s) = 0$ indicates match failure of α on s.

$$\begin{aligned}\text{Pattern } P[s] &:= \texttt{Empty}(s)\\ &\mid P[\texttt{SuffixAfter}(s, \alpha)]\\ \text{Atom } \alpha &:= \text{Class}_c^z \mid \text{RegEx}_r\\ &\mid \text{Funct}_f \mid \text{Const}_s\end{aligned}$$

$c \in$ power set of characters
$f \in$ functions String $\to$ Int
$r \in$ regular expressions
$s \in$ set of strings $\mathbb{S}$
$z \in$ non-negative integers

(a) Syntax of $\mathscr{L}_{\text{FP}}$ patterns.

$$\frac{}{\texttt{Empty}(\epsilon) \;\Downarrow\; \texttt{true}}$$

$$\frac{s = s_0 \circ s_1 \qquad \alpha(s) = |s_0| > 0}{\texttt{SuffixAfter}(s, \alpha) \;\Downarrow\; s_1}$$

$$\frac{}{\text{Funct}_f(s) \;\Downarrow\; f(s)}$$

$$\frac{|s| > 0 \qquad s_0 = s \circ s_1}{\text{Const}_s(s_0) \;\Downarrow\; |s|}$$

$$\frac{L = \{n \in \mathbb{N} \mid r \triangleright s[0:n]\}}{\text{RegEx}_r(s) \;\Downarrow\; \max L}$$

$$\frac{s = s_0 \circ s_1 \qquad \forall x \in s_0 : x \in c \qquad s_1 = \epsilon \vee s_1[0] \notin c}{\text{Class}_c^0(s) \;\Downarrow\; |s_0|}$$

$$\frac{s = s_0 \circ s_1 \qquad \forall x \in s_0 : x \in c \qquad |s_0| = z > 0 \qquad s_1 = \epsilon \vee s_1[0] \notin c}{\text{Class}_c^z(s) \;\Downarrow\; z}$$

(b) Big-step semantics for $\mathscr{L}_{\text{FP}}$ patterns: We use the judgement $E \;\Downarrow\; v$ to indicate that the expression E evaluates to a value v.

Fig. 14. Formal syntax and semantics of our DSL $\mathscr{L}_{\text{FP}}$ for defining syntactic patterns over strings[9]

We allow the following four kinds of atoms in $\mathscr{L}_{\text{FP}}$:

(a) *Constant Strings*: A Const_s atom matches only the string s as the prefix of a given string. For brevity, we denote $\text{Const}_{\text{“str”}}$ as simply “str” throughout the text.

(b) *Regular Expressions*: A RegEx_r atom returns the length of the longest prefix of a given string, that is matched by the regex r.

(c) *Character Classes*: A Class_c^0 atom returns the length of the longest prefix of a give string, which contains only characters from the set c. A Class_c^z atom with $z > 0$ further enforces a fixed-width constraint — the match $\text{Class}_c^z(s)$ fails if $\text{Class}_c^0(s) \neq z$, otherwise it returns z.

(d) *Arbitrary Functions*: A Funct_f atom uses the function f that may contain arbitrary logic, to match a prefix p of a given string and returns $|p|$.

Note that, although both constant strings and character classes may be expressed as regular expressions, having separate terms for them has two key benefits:

- As we show in the next subsection, we can *automatically infer* all constant strings, and some character class atoms (namely, those having a fixed-width). This is unlike regular expression or function atoms, which we do not infer and they must be provided a priori.
- These atoms may leverage more efficient matching logic and do not require regular expression matching in its full generality. Constant string atoms use equality checks for characters, and character class atoms use set membership checks.

We list the default set of atoms provided with FlashProfile, in Fig. 6. Users may extend this set with new atoms from any of the aforementioned kinds.

Example 4.2. The atom Digit is Class_D^1 with $D = \{0, \ldots, 9\}$. We write Class_D^0 as Digit^+, and Class_D^n as $\text{Digit}^{\times n}$ for clarity. Note that, $\text{Digit}^{\times 2}$ matches “04/23” but not “2017/04”, although Digit^+ matches both, since the longest prefix matched, “2017”, has length $4 \neq 2$.

Definition 4.3. Pattern — A pattern is simply a sequence of atoms. The pattern Empty denotes an empty sequence, which only matches the empty string ϵ. We use the concatenation operator “ $\diamond$ ” for sequencing atoms. For $k > 1$, the sequence $\alpha_1 \diamond \alpha_2 \diamond \ldots \diamond \alpha_k$ of atoms defines a pattern that is realized by the $\mathscr{L}_{\text{FP}}$ expression:

$$\texttt{Empty}(\texttt{SuffixAfter}(\cdots \texttt{SuffixAfter}(s, \alpha_1) \cdots, \alpha_k)),$$

which matches a string s, iff

$$s \neq \epsilon \;\bigwedge\; \forall i \in \{1, \ldots, k\} : \alpha_i(s_i) > 0 \;\bigwedge\; s_{k+1} = \epsilon,$$

where $s_1 \overset{\text{def}}{=} s$ and $s_{i+1} \overset{\text{def}}{=} s_i[\alpha_i(s_i) :]$ is the remaining suffix of the string s_i after matching atom α_i.

[9] $a \circ b$ denotes the concatenation of strings a and b, and $r \triangleright x$ denotes that the regex r matches the string x in its entirety.

Throughout this section, we use $s[i]$ to denote the i^{th} character of s, and $s[i:j]$ denotes the substring of s from the i^{th} character, until the j^{th}. We omit j to indicate a substring extending until the end of s. In $\mathscr{L}_{\text{FP}}$, the `SuffixAfter(`s, α`)` operator computes $s[\alpha(s):]$, or fails with an error if $\alpha(s) = 0$. We also show the formal semantics of patterns and atoms in $\mathscr{L}_{\text{FP}}$, in Fig. 14(b).

Note that, we explicitly forbid atoms from matching empty substrings. This reduces the search space by an exponential factor, since an empty string may trivially be inserted between any two characters within a string. However, this does not affect the expressiveness of our final profiling technique, since a profile uses a disjunction of patterns. For instance, the strings matching a pattern $\alpha_1 \diamond (\epsilon \mid \alpha_2) \diamond \alpha_3$ can be clustered into those matching $\alpha_1 \diamond \alpha_3$ and $\alpha_1 \diamond \alpha_2 \diamond \alpha_3$.

4.2 Synthesis of $\mathscr{L}_{\text{FP}}$ Patterns

Our pattern learner $\mathcal{L}_{\text{FP}}$ uses inductive program synthesis (Gulwani et al. 2017) for synthesizing patterns that describe a given dataset $\mathcal{S}$ using a specified set of atoms $\mathfrak{V}$. For the convenience of end users, we automatically *enrich* their specified atoms by including: (1) all possible Const atoms, and (2) all possible fixed-width variants of all Class atoms specified by them. Our learner $\mathcal{L}_{\text{FP}}$ is instantiated with these enriched atoms derived from $\mathfrak{V}$, denoted as $\widehat{\mathfrak{V}}$:

$$\begin{aligned} \widehat{\mathfrak{V}} \stackrel{\text{def}}{=} \mathfrak{V} \quad &\cup \quad \{\text{Const}_s \mid s \in \mathbb{S}\} \\ &\cup \quad \{\text{Class}_c^z \mid \text{Class}_c^0 \in \mathfrak{V} \ \bigwedge \ z \in \mathbb{N}\} \end{aligned} \tag{1}$$

Although $\widehat{\mathfrak{V}}$ is very large, as we describe below, our learner $\mathcal{L}_{\text{FP}}$ efficiently explores this search space, and also provides a completeness guarantee on patterns possible over $\widehat{\mathfrak{V}}$.

We build on PROSE (Microsoft 2017d), a state-of-the-art inductive program synthesis library, which implements the FlashMeta (Polozov and Gulwani 2015) framework. PROSE uses *deductive reasoning* — reducing a problem of synthesizing an expression to smaller synthesis problems for its subexpressions, and provides a robust framework with efficient algorithms and data-structures for this. Our key contribution towards $\mathcal{L}_{\text{FP}}$ are efficient *witness functions* (Polozov and Gulwani 2015, §5.2) that enable PROSE to carry out the deductive reasoning over $\mathscr{L}_{\text{FP}}$.

An inductive program synthesis task is defined by: (1) a *domain-specific language* (DSL) for the target programs, which in our case is $\mathscr{L}_{\text{FP}}$, and (2) a *specification* (Polozov and Gulwani 2015, §3.2) (spec) that defines a set of constraints over the output of the desired program. For learning patterns over a collection $\mathcal{S}$ of strings, we define a spec φ, that simply requires a learned pattern P to describe all given strings, i.e. $\forall s \in \mathcal{S}: P(s) = \texttt{True}$. We formally write this as:

$$\varphi \stackrel{\text{def}}{=} \bigwedge_{s \in \mathcal{S}} [s \rightsquigarrow \texttt{True}]$$

We provide a brief overview of the deductive synthesis process here, and refer the reader to FlashMeta (Polozov and Gulwani 2015) for a detailed discussion. In a deductive synthesis framework, we are required to define the logic for reducing a spec for an expression to specs for its subexpressions. The reduction logic for specs, called witness functions (Polozov and Gulwani 2015, §5.2), is domain-specific, and depends on the semantics of the DSL. Witness functions are used to recursively reduce the specs to terminal symbols in the DSL. PROSE uses a succinct data structure (Polozov and Gulwani 2015, §4) to track the valid solutions to these specs at each reduction and generate expressions that satisfy the initial spec. For $\mathscr{L}_{\text{FP}}$, we describe the logic for reducing the spec φ over the two kinds of patterns: `Empty` and P`[SuffixAfter(`s, α`)]`. For brevity, we elide the pseudocode for implementing the witness functions — their implementation is straightforward, based on the reductions we describe below.

For `Empty(`s`)` to satisfy a spec φ, i.e. describe all strings $s \in \mathcal{S}$, each string s must indeed be ϵ. No further reduction is possible since s is a terminal. We simply check, $\forall s \in \mathcal{S}: s = \epsilon$, and reject `Empty(`$s$`)` if $\mathcal{S}$ contains at least one non-empty string.

The second kind of patterns for non-empty strings, $P[\texttt{SuffixAfter}(s, \alpha)]$, allows for complex patterns that are a composition of an atom α and a pattern P. The pattern $P[\texttt{SuffixAfter}(s, \alpha)]$ contains two unknowns: **(1)** an atom α that matches a non-empty prefix of s, and **(2)** a pattern P that matches the remaining suffix $s[\alpha(s) :]$. Again, note that this pattern must match all strings $s \in \mathcal{S}$. Naïvely considering all possible combinations of α and P leads to an exponential blow up.

First we note that, for a fixed α the candidates for P can be generated recursively by posing a synthesis problem similar to the original one, but over the suffix $s[\alpha(s) :]$ instead of each string s. This reduction style is called a *conditional witness function* (Polozov and Gulwani 2015, §5.2), and generates the following spec for P assuming a fixed α:

$$\varphi_\alpha \overset{\text{def}}{=} \bigwedge_{s \in \mathcal{S}} \left[s[\alpha(s) :] \rightsquigarrow \texttt{True} \right] \tag{2}$$

However, naïvely creating φ_α for all possible values of α is infeasible, since our set $\widehat{\mho}$ of atoms is unbounded. Instead, we consider only those atoms (called *compatible atoms*) that match some non-empty prefix for *all* strings in $\mathcal{S}$, since ultimately our pattern needs to describe all strings. Prior pattern-learning approaches (Raman and Hellerstein 2001; Singh 2016) learn complete patterns for each individual string, and then compute their intersection to obtain patterns consistent with the entire dataset. In contrast, we compute the set of atoms that are compatible with the entire dataset at each step, which allows us to generate this intersection in an incremental manner.

Definition 4.4. Compatible Atoms — Given a universe $\mho$ of atoms, we say a subset $A \subseteq \mho$ is compatible with a dataset $\mathcal{S}$, denoted as $A \propto \mathcal{S}$, if all atoms in A match each string in $\mathcal{S}$, i.e.

$$A \propto \mathcal{S} \quad \text{iff} \quad \forall\, \alpha \in A :\ \forall\, s \in \mathcal{S} :\ \alpha(s) > 0$$

We say that a compatible set A of atoms is *maximal* under the given universe $\mho$, denoted as $A = \max_{\propto}^{\mho}[\mathcal{S}]$ iff $\forall\, X \subseteq \mho :\ X \propto \mathcal{S} \Rightarrow X \subseteq A$.

Example 4.5. Consider a dataset with Canadian postal codes: $\mathcal{S}$ = { "V6E3V6", "V6C2S6", "V6V1X5", "V6X3S4" }. With $\mho$ = the default atoms (listed in Fig. 6), we obtain the enriched set $\widehat{\mho}$ using Equation (1). Then, the maximal set of atoms compatible with $\mathcal{S}$ under $\widehat{\mho}$, i.e. $\max_{\propto}^{\widehat{\mho}}[\mathcal{S}]$ contains 18 atoms, such as "V6", "V", Upper, Upper$^+$, AlphaSpace, AlphaDigit$^{\times 6}$, etc.

For a given universe $\mho$ of atoms and a dataset $\mathcal{S}$, we invoke the GetMaxCompatibleAtoms method outlined in Fig. 15 to efficiently compute the set $\Lambda = \max_{\propto}^{\widehat{\mho}}[\mathcal{S}]$, where $\widehat{\mho}$ denotes the enriched set of atoms based on $\mho$ given by Equation (1). Starting with $\Lambda = \mho$, in line 1, we iteratively remove atoms from Λ, that are not compatible with $\mathcal{S}$, i.e. fail to match at least one string $s \in \mathcal{S}$, at line 4. At the same time, we maintain a hashtable C, which maps each Class atom to its width at line 6. C is used to enrich $\mho$ with fixed-width versions of Class atoms that are already specified in $\mho$. If the width of a Class atom is not constant over all strings in $\mathcal{S}$, we remove it from our hashtable C, at line 7. For each remaining Class atom α in C, we add a fixed-width variant for α to Λ. In line 8, we invoke RestrictWidth to generate the fixed-width variant for α with width $C[\alpha]$.

```
func GetMaxCompatibleAtoms(S: String[ ], ℧: Atom[ ])
output: The maximal set of atoms that are compatible with S
1·  C ← {} ; Λ ← ℧
2·  for all s ∈ S do
3·      for all α ∈ Λ do
▶           Remove incompatible atoms.
4·          if α(s) = 0 then Λ.Remove(α) ; C.Remove(α)
5·          else if α ∈ Class then
▶               Check if character class atoms maintain a fixed width.
6·              if α ∉ C then C[α] ← α(s)
7·              else if C[α] ≠ α(s) then C.Remove(α)
▶   Add compatible fixed-width Class atoms.
8·  for all α ∈ C do Λ.Add(RestrictWidth(α, C[α]))
▶   Add compatible Const atoms.
9·  L ← LongestCommonPrefix(S)
10· Λ.Add(Const_L[0:1], Const_L[0:2], ..., Const_L)
11· return Λ
```

Fig. 15. Computing the maximal set of compatible atoms

Finally, we also enrich Λ with Const atoms — we compute the longest common prefix L across all strings, and add every prefix of L to Λ, at line 12. Note that, GetMaxCompatibleAtoms does not explicitly compute the entire set $\widehat{\mathfrak{U}}$ of enriched atoms, but performs simultaneous pruning and enrichment on $\mathfrak{U}$ to ultimately compute their maximal compatible subset, $\Lambda = \max_{\propto}^{\widehat{\mathfrak{U}}}[\mathcal{S}]$.

In essence, the problem of learning an expression $P\,[\texttt{SuffixAfter}(s, \alpha)]$ with spec φ is reduced to $\left|\max_{\propto}^{\widehat{\mathfrak{U}}}[\mathcal{S}]\right|$ subproblems for P with specs $\{\varphi_\alpha \mid \alpha \in \max_{\propto}^{\widehat{\mathfrak{U}}}[\mathcal{S}]\}$, where φ_α is as given by Equation (2), and $\widehat{\mathfrak{U}}$ denotes the enriched set of atoms derived from $\mathfrak{U}$ by Equation (1). Note that these subproblems are recursively reduced further, until we match all characters in each string, and terminate with `Empty`. Given this reduction logic as witness functions, PROSE performs these recursive synthesis calls efficiently, and finally combines the atoms to candidate patterns. We conclude this subsection with a comment on the soundness and completeness of $\mathscr{L}_{\text{FP}}$.

Definition 4.6. Soundness and $\mathfrak{U}$-Completeness — We say that a learner for $\mathscr{L}_{\text{FP}}$ patterns is *sound* if, for any dataset $\mathcal{S}$, every learned pattern P satisfies $\forall s \in \mathcal{S} : P(s) = \texttt{True}$.

We say that a learner for $\mathscr{L}_{\text{FP}}$, instantiated with a universe $\mathfrak{U}$ of atoms is $\mathfrak{U}$-*complete* if, for any dataset $\mathcal{S}$, it learns every possible pattern $P \in \mathscr{L}_{\text{FP}}$ over $\mathfrak{U}$ atoms that satisfy $\forall s \in \mathcal{S} : P(s) = \texttt{True}$.

Theorem 4.7 (Soundness and $\widehat{\mathfrak{U}}$-Completeness of $\mathcal{L}_{\text{FP}}$). *For an arbitrary set $\mathfrak{U}$ of user-specified atoms,* FlashProfile*'s pattern learner $\mathcal{L}_{\text{FP}}$ is sound and $\widehat{\mathfrak{U}}$-complete, where $\widehat{\mathfrak{U}}$ denotes the enriched set of atoms obtained by augmenting $\mathfrak{U}$ with constant-string and fixed-width atoms, as per Equation (1).*

Proof. Soundness is guaranteed since we only compose compatible atoms. $\widehat{\mathfrak{U}}$-completeness follows from the fact that we always consider the *maximal* compatible subset of $\widehat{\mathfrak{U}}$. □

Due to the $\widehat{\mathfrak{U}}$-completeness of $\mathcal{L}_{\text{FP}}$, once the set $\mathcal{L}_{\text{FP}}(\mathcal{S})$ of patterns over $\mathcal{S}$ has been computed, a variety of cost functions may be used to select the most suitable pattern for $\mathcal{S}$ amongst all possible patterns over $\widehat{\mathfrak{U}}$, without having to invoke pattern learning again.

4.3 Cost of Patterns in $\mathscr{L}_{\text{FP}}$

Our cost function C_{FP} assigns a real-valued score to each pattern $P \in \mathscr{L}_{\text{FP}}$ over a given dataset $\mathcal{S}$, based on the structure of P and its behavior over $\mathcal{S}$. This cost function is used to select the most desirable pattern that represents the dataset $\mathcal{S}$. `Empty` is assigned a cost of 0 regardless of the dataset, since `Empty` can be the only pattern consistent with such datasets. For a pattern $P = \alpha_1 \diamond \ldots \diamond \alpha_k$, we define the cost $C_{\text{FP}}(P, \mathcal{S})$ with respect to a given dataset $\mathcal{S}$ as:

$$C_{\text{FP}}(P, \mathcal{S}) = \sum_{i=1}^{k} Q(\alpha_i) \cdot W(i, \mathcal{S} \mid P)$$

C_{FP} balances the trade-off between a pattern's specificity and complexity. Each atom α in $\mathscr{L}_{\text{FP}}$ has a statically assigned cost $Q(\alpha) \in (0, \infty]$, based on a priori bias for the atom. Our cost function C_{FP} computes a sum over these static costs after applying a data-driven weight $W(i, \mathcal{S} \mid P) \in (0, 1)$:

$$W(i, \mathcal{S} \mid \alpha_1 \diamond \ldots \diamond \alpha_k) = \frac{1}{|\mathcal{S}|} \cdot \sum_{s \in \mathcal{S}} \frac{\alpha_i(s_i)}{|s|},$$

where $s_1 \stackrel{\text{def}}{=} s$ and $s_{i+1} \stackrel{\text{def}}{=} s_i[\alpha_i(s_i) :]$ denotes the remaining suffix of s_i after matching with α_i, as in Definition 4.3. This dynamic weight is an average over the fraction of length matched by α_i across $\mathcal{S}$. It gives a quantitative measure of how well an atom α_i generalizes over the strings in $\mathcal{S}$. With a sound pattern learner, an atomic match would never fail and $W(i, \mathcal{S} \mid P) > 0$ for all atoms α_i.

Example 4.8. Consider $\mathcal{S} = \{\texttt{"Male"}, \texttt{"Female"}\}$, that are matched by $P_1 = \text{Upper} \diamond \text{Lower}^+$, and $P_2 = \text{Upper} \diamond \text{Hex} \diamond \text{Lower}^+$. Given FlashProfile's static costs: $\{\text{Upper} \mapsto 8.2,\ \text{Hex} \mapsto 26.3,\ \text{Lower}^+ \mapsto$

9.1}, the costs for these two patterns shown above are:

$$C_{\text{FP}}(P_1, \mathcal{S}) = 8.2 \times \frac{1/4 + 1/6}{2} + 9.1 \times \frac{3/4 + 5/6}{2} = 8.9$$

$$C_{\text{FP}}(P_2, \mathcal{S}) = 8.2 \times \frac{1/4 + 1/6}{2} + 26.3 \times \frac{1/4 + 1/6}{2} + 9.1 \cdot \frac{2/4 + 4/6}{2} = 12.5$$

P_1 is chosen as best pattern, since $C_{\text{FP}}(P_1, \mathcal{S}) < C_{\text{FP}}(P_2, \mathcal{S})$.

Note that although Hex is a more specific character class compared to Upper and Lower, we assign it a higher static cost to avoid strings like "face" being described as $\text{Hex}^{\times 4}$ instead of $\text{Lower}^{\times 4}$. $\text{Hex}^{\times 4}$ would be chosen over $\text{Lower}^{\times 4}$ only if we observe some other string in the dataset, such as "f00d", which cannot be described using $\text{Lower}^{\times 4}$.

Static Cost (Q) for Atoms. Our learner $\mathcal{L}_{\text{FP}}$ automatically assigns the static cost of a Const_s atom to be proportional to $1/|s|$, and the cost of a Class_c^z atom, with width $z \geqslant 1$, to be proportional to $\frac{Q(\text{Class}_c^0)}{z}$. However, static costs for other kinds of atoms must be provided by the user.

Static costs for our default atoms, listed in Fig. 6, were seeded with the values based on their estimated *size* — the number of strings the atom may match. Then they were penalized (e.g. the Hex atom) with empirically decided penalties to prefer patterns that are more *natural* to users. We describe our *quality* measure for profiles in §5.2, which we have used to decide the penalties for the default atoms. In future, we plan to automate the process of penalizing atoms by designing a learning procedure which tries various perturbations to the seed costs to optimize profiling quality.

5 EVALUATION

We now present experimental evaluation of the FlashProfile tool which implements our technique, focusing on the following key questions:

(§5.1) How well does our syntactic similarity measure capture similarity over real world entities?
(§5.2) How accurate are the profiles? How do sampling and approximations affect their quality?
(§5.3) How fast is FlashProfile, and how does its performance depend on the various parameters?
(§5.4) Are the profiles natural and useful? How do they compare to those from existing tools?

Implementation. We have implemented FlashProfile as a cross-platform C# library built using Microsoft PROSE (Microsoft 2017d). It is now publicly available as part of the PROSE NuGet package.[10] All of our experiments were performed with PROSE 2.2.0 and .NET Core 2.0.0, on an Intel i7 3.60 GHz machine with 32 GB RAM running 64-bit Ubuntu 17.10.

Test Datasets. We have collected 75 datasets from public sources,[11] spanning various domains such as names, dates, postal codes, phone numbers, etc. Their sizes and the distribution of string lengths is shown in Fig. 16. We sort them into three (overlapping) groups:

- CLEAN (25 datasets): Each of these datasets, uses a *single format* that is *distinct* from other datasets. We test syntactic similarity over them — strings from the same dataset must be labeled as similar.

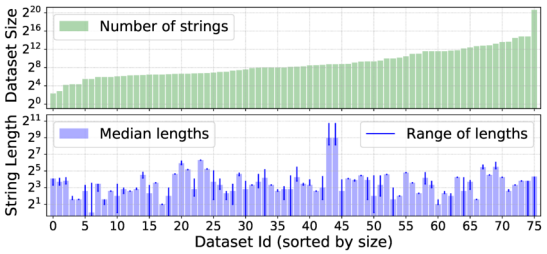


Fig. 16. Number and length of strings across datasets[11]

[10] FlashProfile has been publicly released as the `Matching.Text` module within the PROSE library. For more information, please see: https://microsoft.github.io/prose/documentation/matching-text/intro/.
[11] All public datasets are available at: https://github.com/SaswatPadhi/FlashProfileDemo/tree/master/tests.

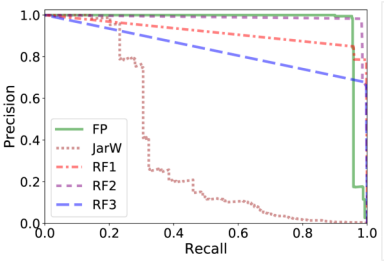
- Domains (63 datasets): These datasets belong to *mutually-distinct domains* but may exhibit multiple formats. We test the quality of profiles over them — a profile learned over fraction of a dataset should match rest of it, but should not be too general as to also match other domains.
- All (75 datasets): We test FlashProfile's performance across all datasets.

5.1 Syntactic Similarity

We evaluate the applicability of our dissimilarity measure from Definition 3.1, over real-life entities. From our Clean group, we randomly pick two datasets and select a random string from each. A good similarity measure should recognize when the pair is drawn from the same dataset by assigning them a lower dissimilarity value, compared to a pair from different datasets. For example, the pair {"A. Einstein", "I. Newton"} should have a lower dissimilarity value than {"A. Einstein", "03/20/1998"}. We instantiated FlashProfile with only the default atoms listed in Fig. 6 and tested 240400 such pairs. Fig. 17 shows a comparison of our method against two simple baselines: **(1)** a character-wise edit-distance-based similarity measure (JarW), and **(2)** a machine-learned predictor (*RF*) over intuitive syntactic features.

For evaluation, we use the standard precision-recall (PR) (Manning et al. 2008) measure. In our context, precision is the fraction of pairs that truly belongs to the same dataset, out of all pairs that are labeled to be "similar" by the predictor. Recall is the fraction of pairs retrieved by the predictor, out of all pairs truly drawn from same datasets. By varying the threshold for labelling a pair as "similar", we generate a PR curve and measure the area under the curve (AUC). A good similarity measure should exhibit high precision and high recall, and therefore have a high AUC.

First, we observed that character-based measures (Gomaa and Fahmy 2013) show poor AUC, and are not indicative of syntactic similarity. Levenshtein distance (Levenshtein 1966), used for string clustering by OpenRefine (Google 2017), a popular data-wrangling tool, exhibits a negligible AUC over our benchmarks. Although the Jaro-Winkler distance (Winkler 1999), indicated as JarW in Fig. 17(a), shows a better AUC, it is quite low compared to both our and machine-learned predictors.



(a) Precision-Recall curves **(b)** Features[12]

Predictor	FP	JarW	RF_1	RF_2	RF_3
AUC	96.28%	35.52%	91.73%	98.71%	76.82%

Fig. 17. Similarity prediction accuracy of FlashProfile (FP) *vs.* a character-based measure (JarW), and random forests ($RF_{1\ldots3}$) trained on different distributions

Our second baseline is a standard random forest (Breiman 2001) model *RF* using the syntactic features listed in Fig. 17(b), such as difference in length, number of digits, etc. We train RF_1 using $\sim 80,000$ pairs with $(1/25)^2 = 0.16\%$ pairs drawn from same datasets. We observe from Fig. 17(a) that the accuracy of *RF* is quite susceptible to changes in the distribution of the training data. RF_2 and RF_3 were trained with 0.64% and 1.28% pairs from same datasets, respectively. While RF_2 performs marginally better than our predictor, RF_1 and RF_3 perform worse.

5.2 Profiling Accuracy

We demonstrate the accuracy of FlashProfile along two dimensions:

- *Partitions:* Our sampling and approximation techniques preserve partitioning accuracy
- *Descriptions:* Profiles generated using $\mathcal{L}_{\text{FP}}$ and C_{FP} are natural, not overly specific or general.

For these experiments, we used FlashProfile with only the default atoms.

[12] len returns string length, begin⟨X⟩ checks if both strings begin with a character in X, cnt⟨X⟩ counts occurrences of characters from X in a string, and $\Delta[f]$ computes $|f(s_1) - f(s_2)|^2$ for a pair of strings s_1 and s_2.

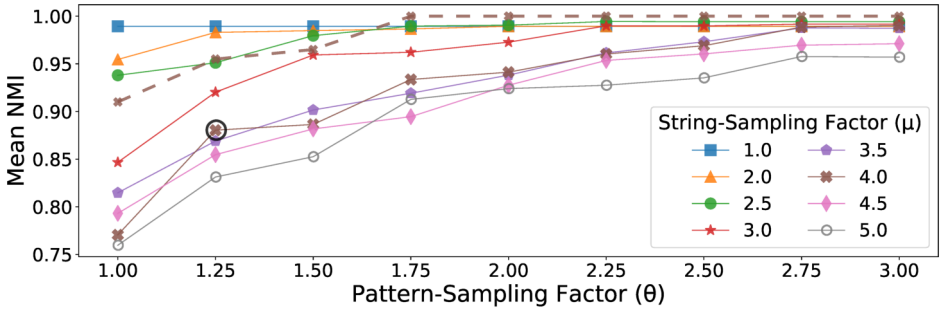


Fig. 18. FlashProfile's partitioning accuracy with different $\langle \mu, \theta \rangle$-configurations

Partitioning. For each $c \in \{2, \ldots, 8\}$, we measure FlashProfile's ability to repartition $256c$ strings — 256 strings collected from each of c randomly picked datasets from Clean. Over 10 runs for each c, we pick different sets of c files, shuffle the $256c$ strings, and invoke FlashProfile to partition them into c clusters. For a fair distribution of strings across each run, we ignore one dataset from the Clean group which had much longer strings (> 1500 characters) compared to other datasets (10 – 100 characters). We experiment with different values of $1.0 \leqslant \mu \leqslant 5.0$ (*string-sampling factor*, which controls the size of chunks given to the core Profile method), and $1.0 \leqslant \theta \leqslant 3.0$ (*pattern-sampling factor*, which controls the approximation during hierarchical clustering).

We measure the precision of clustering using *symmetric uncertainty* (Witten et al. 2017), which is a measure of normalized mutual information (NMI). An NMI of 1 indicates the resulting partitioning to be identical to the original clusters, and an NMI of 0 indicates that the final partitioning is unrelated to the original one. For each $\langle \mu, \theta \rangle$-configuration, we show the mean NMI of the partitionings over $10c$ runs (10 for each value of c), in Fig. 18. The NMI improves with increasing θ, since we sample more dissimilarities, resulting in better approximations. However, the NMI drops with increasing μ, since more pairwise dissimilarities are approximated. Note that the number of string pairs increases quadratically with μ, but reduces only linearly with θ. This is reflected in Fig. 18 – for $\mu > 4.0$, the partitioning accuracy does not reach 1.0 even for $\theta = 3.0$. FlashProfile's default configuration $\langle \mu = 4.0, \theta = 1.25 \rangle$, achieves a median NMI of 0.96 (mean 0.88) (indicated by a circled point). The dashed line indicates the median NMIs with $\mu = 4.0$. The median NMI is significantly higher than the mean, indicating our approximations were accurate in most cases. As we explain below in §5.3, with $\langle \mu = 4.0, \theta = 1.25 \rangle$, FlashProfile achieves the best performance *vs.* accuracy trade-off.

Descriptions. We evaluate the suitability of the automatically suggested profiles, by measuring their overall precision and recall. A natural profile should not be too specific – it should generalize well over the dataset (high true positives), but not beyond it (low false positives).

For each dataset in our Domains, we profile a randomly selected 20% of its strings, and measure: (1) the fraction of the remaining dataset described by it, and (2) the fraction of an equal number of strings from other datasets, matched by it. Fig. 19 summarizes our results. The lighter and darker shades indicate the fraction of true positives and false positives respectively. The white area at the top indicates the fraction of false negatives – the fraction of the remaining 80% of the dataset that is not described by the profile. We observe an overall precision of 97.8%, and a recall of 93.4%. The dashed line indicates a mean true positive rate of 93.2%, and the dotted line shows a mean false positive rate of 2.3%; across all datasets.

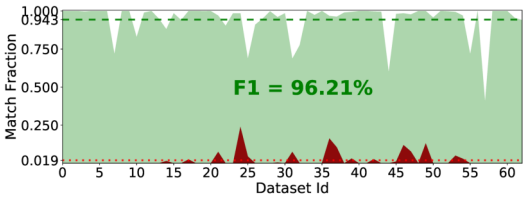
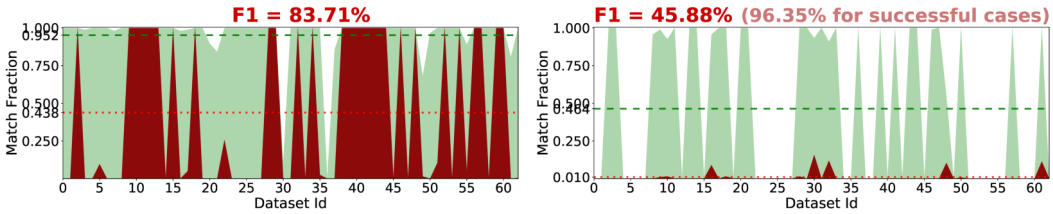


Fig. 19. Quality of descriptions at $\langle \mu = 4.0, \theta = 1.25 \rangle$



(a) Microsoft SSDT (Microsoft 2017c) **(b)** Ataccama One (Ataccama 2017)

Fig. 20. Quality of descriptions from current state-of-the-art tools

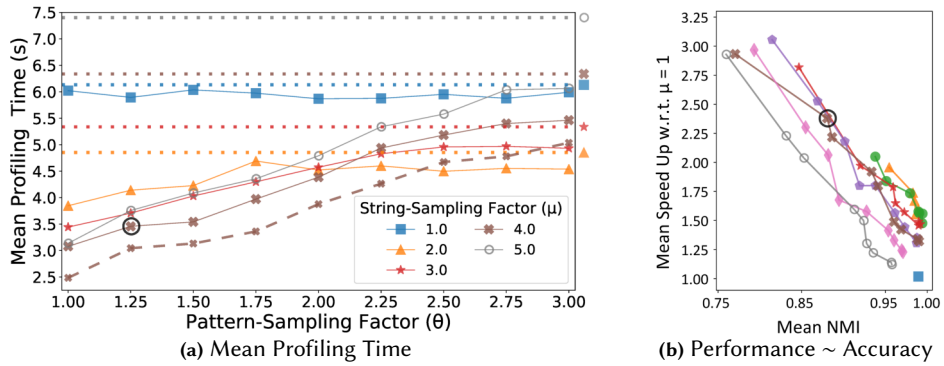
We also perform similar quality measurements for profiles generated by Microsoft SSDT (Microsoft 2017c) and Ataccama One (Ataccama 2017). We use "Column Pattern Profiling Tasks" with `PercentageDataCoverageDesired` = 100 within SSDT, and "Pattern Analysis" feature within the Ataccama One platform. We summarize the per-dataset description quality for SSDT in Fig. 20(a), and for Ataccama One in Fig. 20(b). We observe a low overall F1 score for both tools.

While SSDT has a very high false positive rate, Ataccama One has a high failure rate. For 27 out of 63 datasets, SSDT generates ".*" as one of the patterns, and it fails to profile one dataset that has very long strings (up to 1536 characters). On the other hand, Ataccama One fails to profile 33 datasets. But for the remaining 30 datasets, the simple atoms (digits, numbers, letters, words) used by Ataccama One seem to work well — the profiles exhibit high precision and recall. Note that, this quantitative measure only captures the specificity of profiles, but not their readability. We present a qualitative comparison of profiles generated by these tools in §5.4.

5.3 Performance

We measure the mean profiling time with various $\langle \mu, \theta \rangle$-configurations, and summarize our findings in Fig. 21(a). The dotted lines indicate profiling time without pattern sampling, i.e. $\theta \to \infty$, for different values of the μ factor. The dashed line shows the median profiling time for different values of θ with our default $\mu = 4.0$. We also show the performance-accuracy trade off in Fig. 21(b) by measuring the mean speed up of each configuration w.r.t. $\langle \mu = 1.0, \theta = 1.0 \rangle$. We select the *Pareto optimal* point $\langle \mu = 4.0, \theta = 1.25 \rangle$ as FlashProfile's default configuration. It achieves a mean speed up of 2.3× over $\langle \mu = 1.0, \theta = 1.0 \rangle$, at a mean NMI of 0.88 (median = 0.96).

As one would expect, the profiling time increases with θ, due to sampling more patterns and making more calls to $\mathcal{L}_{\text{FP}}$. The dependence of profiling time on μ however, is more interesting. Notice that with $\mu = 1$, the profiling time is *higher* than any other configurations, when pattern sampling is enabled, i.e. $\theta \neq \infty$ (solid lines). This is due to the fact that FlashProfile learns very specific profiles with $\mu = 1$ with very small samples of strings, which do not generalize well over the remaining data. This results in many Sample–Profile–Filter iterations. Also note that with



(a) Mean Profiling Time **(b)** Performance ~ Accuracy

Fig. 21. Impact of sampling on performance (using the same colors and markers as Fig. 18)

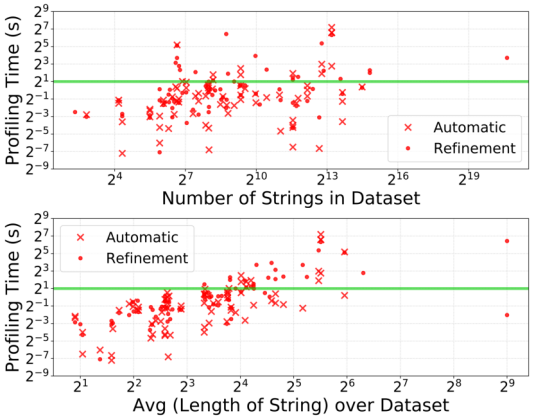


Fig. 22. Performance over real-life datasets

Zip Code
99518
61021-9150
2645
83716
⋮
K0K 2C0
14480
S7K7K9

(a) Dataset

LDL DLD
LDLDLD
N-N
N

(b) A1

\w\w\w \w\w\w
\d\d\d\d\d
\d\d\d\d
.*

(c) SSDT

U D U ␣ D U D
“61” D^3 “–” D^4
“S7K7K9”
D^+
ϵ

(d) FP

U D U ␣ D U D
“61” D^3 “–” D^4
“S7K7K9”
D^5
D^4
ϵ

(e) FP_6

Most frequent pattern from Potter's Wheel = `int`

Fig. 23. Profiles for a dataset with zip codes[13]

pattern-sampling enabled, the profiling time decreases with μ until $\mu = 4.0$ as, and then increases as profiling larger samples of strings becomes expensive.

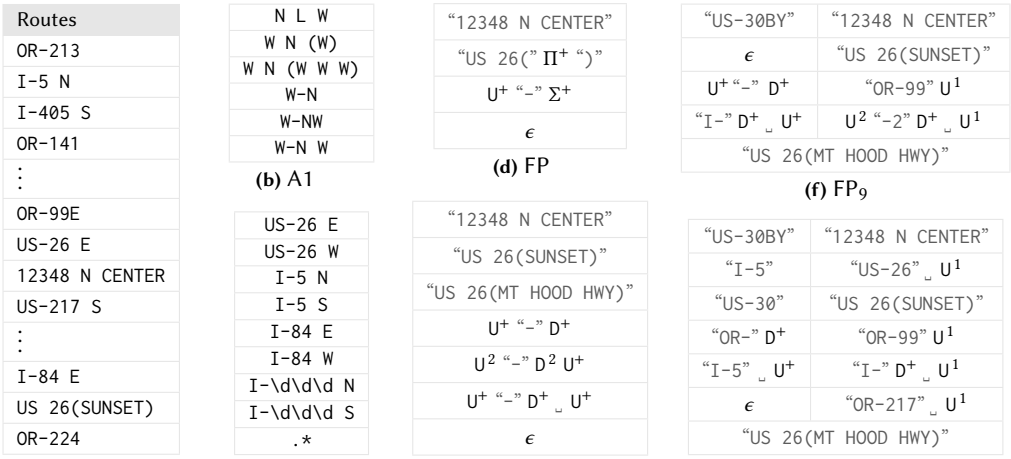
Finally, we evaluate FlashProfile's performance on end-to-end real-life profiling tasks on all 75 datasets, that have a mixture of clean and dirty datasets. Over 153 tasks – 76 for automatic profiling, and 77 for refinement, we observe a median profiling time of 0.7 s. With our default configuration, 77% of the requests are fulfilled within 2 seconds – 70% of automatic profiling tasks, and 83% of refinement tasks. In Fig. 22 we show the variance of profiling times w.r.t. size of the datasets (number of strings in them), and the average length of the strings in the datasets (all axes being logarithmic). We observe that the number of string in the dataset doesn't have a strong impact on the profiling time. This is expected, since we only sample smaller chunks of datasets, and remove strings that are already described by the profile we have learned so far. We repeated this experiment with 5 dictionary-based custom atoms: ⟨DayName⟩, ⟨ShortDayName⟩, ⟨MonthName⟩, ⟨ShortMonthName⟩, ⟨US_States⟩, and noticed an increase of ∼ 0.02 s in the median profiling time.

5.4 Comparison of Learned Profiles

We compare the profiles learned by FlashProfile to the outputs from 3 state-of-the-art tools: **(1)** Ataccama One (Ataccama 2017): a dedicated profiling tool, **(2)** Microsoft's SSDT (Microsoft 2017c) a feature-rich IDE for database applications, and **(3)** Potter's Wheel (Raman and Hellerstein 2001): a tool that detects the most frequent data pattern and predicts anomalies in data. Fig. 23 and Fig. 24 show the observed outputs. We list the output of Ataccama One against A1, the suggested profile from FlashProfile against FP, and the one generated on requesting k patterns against FP_k. For brevity, we **(1)** omit the concatenation operator “ ⋄ ” between atoms, and **(2)** abbreviate Digit ↦ D, Upper ↦ U, AlphaSpace ↦ Π, AlphaDigitSpace ↦ Σ.

First, we observe that SSDT generates an overly general “.*” pattern in both cases. Ataccama One generates a very coarse grained profile in both cases, which although explains the pattern of special characters, does not say much about other syntactic properties, such as common prefixes, or fixed-length patterns. With FlashProfile, one can immediately notice in Fig. 23(d), that “S7K7K9” is the only Canadian zip code which does not have a space in the middle, and that some US zip codes have 4 digits instead of 5 (probably the leading zero was lost while interpreting it as a number). Similarly, one can immediately observe that in Fig. 24(d), “12348 N CENTER” is not a route. Similarly the pattern “US 26(” Π^+ “)” indicates that it is the only entry with a space instead of a dash between the “US” and “26”.

[13] Dataset collected from a database of vendors across US and Canada: https://goo.gl/PGS2pL



(a) Dataset **(b)** A1 **(c)** SSDT **(d)** FP **(e)** FP_7 **(f)** FP_9 **(g)** FP_{13}

Most frequent pattern from Potter's Wheel = `IspellWord int space AllCapsWord`

Fig. 24. Profiles for a dataset containing US routes[14]

In many real-life scenarios, simple statistical profiles are not enough for data understanding or validation. FlashProfile allows users to gradually drill into the data by requesting profiles with a desired granularity. Furthermore, they may also provide custom atoms for domain-specific profiling.

6 APPLICATIONS IN PBE SYSTEMS

In this section, we discuss how syntactic profiles can improve programming-by-example (PBE) (Gulwani et al. 2017; Lieberman 2001) systems, which synthesize a desired program from a small set of input-output examples. For instance, given an example "`Albert Einstein`" ⇝ "`A.E.`", the system should learn a program that extracts the initials for names. Although many PBE systems exist today, most share criticisms on low usability and confidence in them (Lau 2009; Mayer et al. 2015).

Examples are an inherently under-constrained form of specifying the desired program behavior. Depending on the target language, a large number of programs may be consistent with them. Two major challenges faced by PBE systems today are: **(1)** obtaining a set of examples that accurately convey the desired behavior to limit the space of synthesized programs, and **(2)** ranking these programs to select the ones that are *natural* to users.

In a recent work, Ellis and Gulwani (2017) address **(2)** using data profiles. They show that incorporating profiles for input-output examples significantly improves ranking, compared to traditional techniques which only examine the structure of the synthesized programs. We show that data profiles can also address problem **(1)**. Raychev et al. (2016) have presented *representative data samplers* for synthesis scenarios, but they require the outputs for all inputs. In contrast, we show a novel interaction model for proactively requesting users to supply the desired outputs for syntactically different inputs, thereby providing a representative set of examples to the PBE system.

Significant Inputs. Typically, users provide outputs for only the first few inputs of target dataset. However, if these are not representative of the entire dataset, the system may not learn a program that generalizes over other inputs. Therefore, we propose a novel interaction model that requests the user to provide the desired outputs for *significant* inputs, incrementally. A significant input is one that is syntactically the most dissimilar with all previously labelled inputs.

[14] Dataset collected from https://portal.its.pdx.edu/fhwa

We start with a syntactic profile $\widetilde{P}$ for the input dataset and invoke the OrderPartitions function, listed in Fig. 25, to order the partitions identified in $\widetilde{P}$ based on mutual dissimilarity, i.e. each partition $\mathcal{S}_i$ must be as dissimilar as possible with (its most-similar neighbor within) the partitions $\{\mathcal{S}_1, \ldots, \mathcal{S}_{i-1}\}$. It is a simple extension of our SampleDissimilarities procedure (Fig. 9) to work with sets of strings instead of strings. We start with the partition that can be described with the minimum-cost pattern. Then, from the remaining partitions, we iteratively select the one that is most dissimilar to those previously selected. We define the dissimilarity between two partitions as the cost of the best (least-cost) pattern required to describe them together.

Once we have an ordered set of partitions, $\langle \mathcal{S}_1, \ldots, \mathcal{S}_{|\widetilde{P}|} \rangle$, we request the user to provide the desired output for a randomly selected input from each partition in order. Since PBE systems like Flash Fill are interactive, and start synthesizing programs right from the first example, the user can inspect and skip over inputs for which the output is correctly predicted by the synthesized program. After one cycle through all partitions, we restart from partition $\mathcal{S}_1$, and request the user to provide the output for a new random string in each partition.

```
func OrderPartitions_⟨ℒ,C⟩(P̃: Profile)
output: A sequence of partitions ⟨S_1, ..., S_|P̃|⟩ over S
► Select with the partition that has the minimum cost.
1· ρ ← ⟨(arg min_{X∈P̃} C(X.Pattern, X.Data)).Data⟩
2· while |ρ| < |P̃| do
►     Pick the most dissimilar partition w.r.t. those in ρ.
3·    T ← arg max_{Z∈P̃} min_{X∈ρ}
4·            (LearnBestPattern_⟨ℒ,C⟩(Z.Data ∪ X)).Cost
5·    ρ.Append(T.Data)
6· return ρ
```

Fig. 25. Ordering partitions by mutual dissimilarity

We evaluate the proposed interaction model over 163 Flash Fill benchmarks[15] that require more than one example to learn the desired string-transformation program. Fig. 26 compares the number of examples required originally, to that using our interaction model. Seven cases that timeout due to the presence of extremely long strings have been omitted.

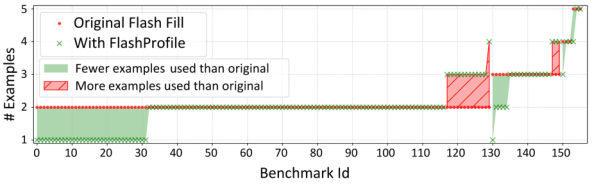


Fig. 26. Examples needed with and without FlashProfile

Over the remaining 156 cases, we observe that, Flash Fill (1) requires a single example per partition for 131 (= 80%) cases, and (2) uses the minimal set[16] of examples to synthesize the desired program for 140 (= 86%) cases — 39 of which were improvements over Flash Fill. Thus, (1) validates our hypothesis that our partitions indeed identify representative inputs, and (2) further indicates that our interaction model is highly effective. Selecting inputs from partitions ordered based on mutual syntactic dissimilarity helps Flash Fill converge to the desired programs with fewer examples. Note that, these results are based on the default set of atoms. Designing custom atoms for string-transformation tasks, based on Flash Fill's semantics is also an interesting direction.

Although the significant inputs scenario is similar to *active learning*, which is well-studied in machine-learning literature (Hanneke 2014), typical active-learning methods require hundreds of labeled examples. In contrast, PBE systems deal with very few examples (Mayer et al. 2015).

7 RELATED WORK

There has been a line of work on profiling various aspects of datasets — Abedjan et al. (2015) present a recent survey. Traditional techniques for summarizing data target statistical profiles (Cormode et al. 2012), such as sampling-based aggregations (Haas et al. 1995), histograms (Ioannidis 2003), and wavelet-based summaries (Karras and Mamoulis 2007). However, pattern-based profiling

[15] These benchmarks are a superset of the original set of Flash Fill (Gulwani 2011) benchmarks, with many more real-world scenarios collected from customers using products powered by PROSE (Microsoft 2017d).

[16] By *minimal*, we mean that there is no smaller set of examples with which Flash Fill can synthesize the desired program.

is relatively underexplored, and is minimally or not supported by state-of-the-art data analysis tools (Ataccama 2017; Google 2017; Microsoft 2017c; Trifacta 2017).

To our knowledge, no existing approach learns syntactic profiles over an extensible language and allows refinement of generated profiles. We present a novel dissimilarity measure which is the key to learning refinable profiles over arbitrary user-specified patterns. Microsoft's SQL Server Data Tools (SSDT) (Microsoft 2017c) learns rich regular expressions but is neither extensible not comprehensive. A dedicated profiling tool Ataccama One (Ataccama 2017) generates comprehensive profiles over a very small set of base patterns. Google's OpenRefine (Google 2017) does not learn syntactic profiles, but it allows clustering of strings using character-based similarity measures (Gomaa and Fahmy 2013). In §5 we show that such measures do not capture syntactic similarity. While Potter's Wheel (Raman and Hellerstein 2001) does not learn a complete profile, it learns the most frequent data pattern over arbitrary user-defined *domains* that are similar to our atomic patterns.

Application-Specific Structure Learning. There has been prior work on learning specific structural properties aimed at aiding data wrangling applications, such as data transformations (Raman and Hellerstein 2001; Singh 2016), information extraction (Li et al. 2008), and reformatting or text normalization (Kini and Gulwani 2015). However, these approaches make specific assumptions regarding the target application, which do not necessarily hold when learning general purpose profiles. Although profiles generated by FlashProfile are primarily aimed at data understanding, in §6 we show that they may aid PBE applications, such as Flash Fill (Gulwani 2011) for data transformation. Bhattacharya et al. (2015) also utilize hierarchical clustering to group together sensors used in building automation based on their tags. However, they use a fixed set of domain-specific features for tags and do not learn a pattern-based profile.

Grammar Induction. Syntactic profiling is also related to the problem of learning regular expressions, or more generally grammars from a given set of examples. De la Higuera (2010) present a recent survey on this line of work. Most of these techniques, such as L-Star (Angluin 1987) and RPNI (Oncina and García 1992), assume availability of both positive and negative examples, or a membership oracle. Bastani et al. (2017) show that these techniques are either too slow or do not generalize well and propose an alternate strategy for learning grammars from positive examples. When a large number of negative examples are available, genetic programming has also been shown to be useful for learning regular expressions (Bartoli et al. 2012; Svingen 1998). Finally, LearnPADS (Fisher et al. 2008; Zhu et al. 2012) also generates a syntactic description, but does not support refinement or user-specified patterns.

Program Synthesis. Our techniques for sampling-based approximation and finding representative inputs relate to prior work by Raychev et al. (2016) on synthesizing programs from noisy data. However, they assume a single target program and the availability of outputs for all inputs. In contrast, we synthesize a disjunction of several programs, each of which returns `True` only on a specific partition of the inputs, which is unknown a priori.

FlashProfile's pattern learner uses the PROSE library (Microsoft 2017d), which implements the FlashMeta framework (Polozov and Gulwani 2015) for inductive program synthesis, specifically programming-by-examples (PBE) (Gulwani et al. 2017; Lieberman 2001). PBE has been leveraged by recent works on automating repetitive text-processing tasks, such as string transformation (Gulwani 2011; Singh 2016), extraction (Le and Gulwani 2014), and format normalization (Kini and Gulwani 2015). However, unlike these applications, data profiling does not solicit any (output) examples from the user. We demonstrate a novel application of a supervised synthesis technique to solve an unsupervised learning problem.

8 CONCLUSION

With increasing volume and variety of data, we require powerful profiling techniques to enable end users to understand and analyse their data easily. Existing techniques generate a single profile over pre-defined patterns, which may be too coarse grained for a user's application. We present a framework for learning syntactic profiles over user-defined patterns, and also allow refinement of these profiles interactively. Moreover, we show domain-specific approximations that allow end users to control accuracy *vs.* performance trade-off for large datasets, and generate approximately correct profiles in realtime on consumer-grade hardware. We instantiate our approach as FlashProfile, and present extensive evaluation on its accuracy and performance on real-life datasets. We also show that syntactic profiles are not only useful for data understanding and manual data analysis tasks, but can also help existing PBE systems.

ACKNOWLEDGMENTS

The lead author is thankful to the PROSE team at Microsoft, especially to Vu Le, Danny Simmons, Ranvijay Kumar, and Abhishek Udupa, for their invaluable help and support. We also thank the anonymous reviewers for their constructive feedback on earlier versions of this paper.

This research was supported in part by an internship at Microsoft, by the National Science Foundation (NSF) under Grant No. CCF-1527923, and by a Microsoft Research Ph.D. Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF or of the Microsoft Corporation.

REFERENCES

Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling Relational Data: A Survey. *VLDB J.* 24, 4 (2015), 557–581. https://doi.org/10.1007/s00778-015-0389-y

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

David Arthur and Sergei Vassilvitskii. 2007. k-means++: The Advantages of Careful Seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, Nikhil Bansal, Kirk Pruhs, and Clifford Stein (Eds.). SIAM, 1027–1035. http://dl.acm.org/citation.cfm?id=1283383.1283494

Ataccama. 2017. Ataccama One Platform. https://www.ataccama.com/.

Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. 2012. Automatic Generation of Regular Expressions from Examples with Genetic Programming. In *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012, Companion Material Proceedings*, Terence Soule and Jason H. Moore (Eds.). ACM, 1477–1478. https://doi.org/10.1145/2330784.2331000

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 95–110. https://doi.org/10.1145/3062341.3062349

Arka Aloke Bhattacharya, Dezhi Hong, David E. Culler, Jorge Ortiz, Kamin Whitehouse, and Eugene Wu. 2015. Automated Metadata Construction to Support Portable Building Applications. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments, BuildSys 2015, Seoul, South Korea, November 4-5, 2015*, David Culler, Yuvraj Agarwal, and Rahul Mangharam (Eds.). ACM, 3–12. https://doi.org/10.1145/2821650.2821667

Christopher M. Bishop. 2016. *Pattern Recognition and Machine Learning*. Springer New York. http://www.worldcat.org/oclc/1005113608

Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324

Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases* 4, 1-3 (2012), 1–294. https://doi.org/10.1561/1900000004

Colin De la Higuera. 2010. *Grammatical inference: learning automata and grammars.* Cambridge University Press.

Xin Luna Dong and Divesh Srivastava. 2013. Big Data Integration. *PVLDB* 6, 11 (2013), 1188–1189. http://www.vldb.org/pvldb/vol6/p1188-srivastava.pdf

Kevin Ellis and Sumit Gulwani. 2017. Learning to Learn Programs from Examples: Going Beyond Program Structure. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia,*

August 19-25, 2017, Carles Sierra (Ed.). ijcai.org, 1638–1645. https://doi.org/10.24963/ijcai.2017/227

Kathleen Fisher, David Walker, and Kenny Qili Zhu. 2008. LearnPADS: automatic tool generation from ad hoc data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 1299–1302. https://doi.org/10.1145/1376616.1376759

Wael H Gomaa and Aly A Fahmy. 2013. A survey of text similarity approaches. *International Journal of Computer Applications* 68, 13 (April 2013), 13–18. https://doi.org/10.5120/11638-7118

Google. 2017. OpenRefine: A free, open source, powerful tool for working with messy data. http://openrefine.org/.

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics - A Foundation for Computer Science, 2nd Edition).* Addison-Wesley. http://www.worldcat.org/oclc/992331503

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. 1995. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.*, Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio (Eds.). Morgan Kaufmann, 311–322. http://www.vldb.org/conf/1995/P311.PDF

Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. 2001. On Clustering Validation Techniques. *J. Intell. Inf. Syst.* 17, 2-3 (2001), 107–145. https://doi.org/10.1023/A:1012801612483

Steve Hanneke. 2014. Theory of Disagreement-Based Active Learning. *Found. Trends Mach. Learn.* 7, 2-3 (June 2014), 131–309. https://doi.org/10.1561/2200000037

Yannis E. Ioannidis. 2003. The History of Histograms (abridged). In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer (Eds.). Morgan Kaufmann, 19–30. http://www.vldb.org/conf/2003/papers/S02P01.pdf

Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. 1999. Data Clustering: A Review. *ACM Comput. Surv.* 31, 3 (1999), 264–323. https://doi.org/10.1145/331499.331504

Panagiotis Karras and Nikos Mamoulis. 2007. The Haar+ Tree: A Refined Synopsis Data Structure. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 436–445. https://doi.org/10.1109/ICDE.2007.367889

Dileep Kini and Sumit Gulwani. 2015. FlashNormalize: Programming by Examples for Text Normalization. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, Qiang Yang and Michael Wooldridge (Eds.). AAAI Press, 776–783. http://ijcai.org/Abstract/15/115

Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (2009), 65–67. https://doi.org/10.1609/aimag.v30i4.2262

Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 542–553. https://doi.org/10.1145/2594291.2594333

Vladimir I Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics Doklady*, Vol. 10. 707–710. http://adsabs.harvard.edu/abs/1966SPhD...10..707L

Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning for Information Extraction. In *2008 Conference on Empirical Methods in Natural Language Processing, EMNLP 2008, Proceedings of the Conference, 25-27 October 2008, Honolulu, Hawaii, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*. ACL, 21–30. http://www.aclweb.org/anthology/D08-1003

Henry Lieberman. 2001. *Your wish is my command: Programming by example.* Morgan Kaufmann.

Steve Lohr. 2014. For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights. *New York Times* 17 (2014). https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html

James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA., 281–297.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press.

Arkady Maydanchik. 2007. *Data Quality Assessment.* Technics Publications. https://technicspub.com/data-quality-assessment/

Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the*

28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015, Celine Latulipe, Bjoern Hartmann, and Tovi Grossman (Eds.). ACM, 291–301. https://doi.org/10.1145/2807442.2807459

Microsoft. 2017a. Azure Machine Learning By-Example Data Transform. https://www.youtube.com/watch?v=9KG0Sc2B2KI.

Microsoft. 2017b. Data Transformations "By Example" in the Azure ML Workbench. https://blogs.technet.microsoft.com/machinelearning/2017/09/25/by-example-transformations-in-the-azure-machine-learning-workbench/.

Microsoft. 2017c. Microsoft SQL Server Data Tools (SSDT). https://docs.microsoft.com/en-gb/sql/ssdt.

Microsoft. 2017d. Program Synthesis using Examples SDK. https://microsoft.github.io/prose/.

José Oncina and Pedro García. 1992. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition* 5, 99-108 (1992), 15–20.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 107–126. https://doi.org/10.1145/2814270.2814310

Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 381–390. http://www.vldb.org/conf/2001/P381.pdf

Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 761–774. https://doi.org/10.1145/2837614.2837671

Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming By Example for Syntactic String Transformations. *PVLDB* 9, 10 (2016), 816–827. http://www.vldb.org/pvldb/vol9/p816-singh.pdf

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, John Paul Shen and Margaret Martonosi (Eds.). ACM, 404–415. https://doi.org/10.1145/1168857.1168907

Thorvald Sørensen. 1948. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Biol. Skr.* 5 (1948), 1–34.

Borge Svingen. 1998. Learning Regular Languages using Genetic Programming. *Proc. Genetic Programming* (1998), 374–376.

Andrei N Tikhonov. 1963. Solution of Incorrectly Formulated Problems and the Regularization Method. In *Dokl. Akad. Nauk.*, Vol. 151. 1035–1038.

Trifacta. 2017. Trifacta Wrangler. https://www.trifacta.com/products/wrangler/.

William E Winkler. 1999. The State of Record Linkage and Current Research Problems. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.4336

Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2017. *Data Mining: Practical Machine Learning Tools and Techniques, 4th Edition.* Elsevier Science & Technology. http://www.worldcat.org/oclc/1007085077

Rui Xu and Donald C. Wunsch II. 2005. Survey of Clustering Algorithms. *IEEE Trans. Neural Networks* 16, 3 (2005), 645–678. https://doi.org/10.1109/TNN.2005.845141

Kenny Qili Zhu, Kathleen Fisher, and David Walker. 2012. LearnPADS++ : Incremental Inference of Ad Hoc Data Formats. In *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings (Lecture Notes in Computer Science)*, Claudio V. Russo and Neng-Fa Zhou (Eds.), Vol. 7149. Springer, 168–182. https://doi.org/10.1007/978-3-642-27694-1_13