AutoType: Synthesizing Type-Detection Logic for Rich Semantic Data Types using Open-source Code

Cong Yan*
University of Washington
congy@cs.washington.edu

Yeye He
Microsoft Research
yeyehe@microsoft.com

ABSTRACT

Given a table of data, existing systems can often detect basic *atomic* types (e.g., strings vs. numbers) for each column. A new generation of data-analytics and data-preparation systems are starting to automatically recognize rich semantic types such as date-time, email address, etc., for such metadata can bring an array of benefits including better table understanding, improved search relevance, precise data validation, and semantic data transformation. However, existing approaches only detect a limited number of types using regular-expression-like patterns, which are often inaccurate, and cannot handle rich semantic types such as credit card and ISBN numbers that encode semantic validations (e.g., checksum).

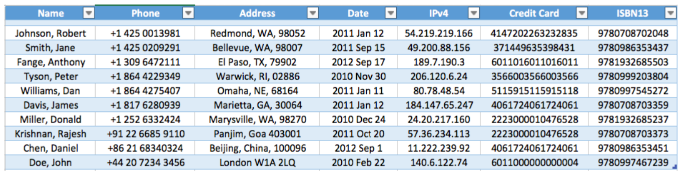
We developed AutoType, a system that can synthesize type-detection logic for rich data types, by leveraging code from open-source repositories like GitHub. Users only need to provide a set of positive examples for a target data type and a search keyword, our system will automatically identify relevant code, and synthesize type-detection functions using execution traces. We compiled a benchmark with 112 semantic types, out of which the proposed system can synthesize code to detect 84 such types at a high precision. Applying the synthesized type-detection logic on web table columns have also resulted in a significant increase in data types discovered compared to alternative approaches.

ACM Reference Format:
Cong Yan and Yeye He. 2018. AutoType: Synthesizing Type-Detection Logic for Rich Semantic Data Types using Open-source Code. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196888

1 INTRODUCTION

Metadata has long been recognized as an important aspect of data management. Classical metadata management problems, such as data provenance [31, 37, 39] and schema mappings [13, 26], have produced long and fruitful lines of research.

An important yet thus-far overlooked class of metadata is *semantic data types*. While basic *atomic types* (such as strings vs. numbers) is straightforward and widely-used, semantic data types are much more rich and fine-grained (e.g., `credit-card-number`, `UPC-code`, etc.). We observe that like atomic types, semantic data types can oftentimes also be automatically inferred from data values alone, both intuitively for humans and programatically using code.



Name	Phone	Address	Date	IPv4	Credit Card	ISBN13
Johnson, Robert	+1 425 0013981	Redmond, WA, 98052	2011 Jan 12	54.219.219.166	4147202263232835	9780708702048
Smith, Jane	+1 425 0209291	Bellevue, WA, 98007	2011 Sep 15	49.200.88.156	371449635398431	9780986353437
Fange, Anthony	+1 309 6472111	El Paso, TX, 79902	2012 Sep 17	189.7.190.3	6011016011016011	9781932685503
Tyson, Peter	+1 864 4229349	Warwick, RI, 02886	2010 Nov 30	206.120.6.24	3566003566003566	9780999203804
Williams, Dan	+1 864 4275407	Omaha, NE, 68164	2011 Jan 11	80.78.48.54	5115915115915118	9780997545272
Davis, James	+1 817 6280939	Marietta, GA, 30064	2011 Jan 12	184.147.65.247	4061724061724061	9780708703359
Miller, Donald	+1 252 6332424	Marysville, WA, 98270	2010 Dec 24	24.20.217.160	2223000010476528	9781932685237
Krishnan, Rajesh	+91 22 6685 9110	Panjim, Goa 403001	2011 Oct 20	57.36.234.113	2223000010476528	9780708703373
Chen, Daniel	+86 21 68340324	Beijing, China, 100096	2012 Sep 1	11.222.239.92	4061724061724061	9780986353451
Doe, John	+44 20 7234 3456	London W1A 2LQ	2010 Feb 22	140.6.122.74	6011000000000004	9780997467239

Figure 1: An example table with sales transactions. The semantic data types can often be inferred from values alone.

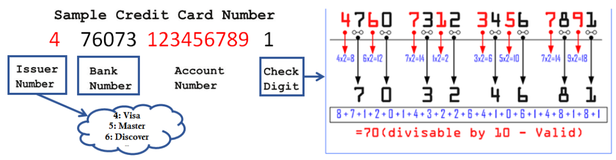


Figure 2: Credit card numbers encode rich semantic information and computes a checksum using Luhn's algorithm.

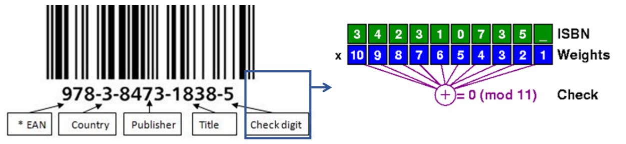


Figure 3: ISBN numbers also contain rich information and use a different GS1 checksum for data validation.

Consider the example spreadsheet in Figure 1, which contains sales transactions of a retailer. This table lists customer `names`, their `phone-numbers`, `addresses`, `date` and `time` of transactions, login `IP-addresses`, `credit-cards` used, and the `ISBN` (international standard for book identifiers) of books purchased. Assume for a moment that the column names at the top of the table are *not* given. As human beings, it is actually not that hard to determine the types of the first few columns by just looking at their values; and with some efforts one could write programs to recognize such types. The last two columns may appear cryptic to human eyes without column names. They can actually be algorithmically recognized as `credit-card-number` and `ISBN`, respectively, because their types encode rich semantics and require strict checksum validations.

Specifically, Figure 2 shows the structure of a valid `credit-card number`. The first digit indicates the card issuer (e.g., Visa, Master, etc.), and the next five digits indicate the issuing bank (e.g., Chase, BOA, etc.). The last digit is a check digit, whose value is calculated based on all previous digits using an algorithm known as Luhn's

*Work done at Microsoft Research.

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196888

Industry/Domain	Example Data Types
Science (Biology, Chemistry, etc.)	CAS registry, International Chemical Identifier, Hill notation, Protein ID, SMILE notation, . . .
Financial & Commerce	Credit card, UPC code, EAN code, ABA routing number, SWIFT code, CUSIP securities, . . .
Transportation	VIN number, Shipping container ISO 6346, UIC Wagon number, Maritime Ship Identifier, . . .
Technology & Telecommunication	IMEI code, IPv4, IPv6, MSISDN, OID, MAC address, URL, RGB color, CMYK color, . . .
Geo-location	Longitude/Latitude, Mailing address, UTM coordinates, MGRS, USNG coordinates, NAD83 . . .
Miscellaneous	Phone number, Person name, Date, Time, Email, National identification number, . . .

Table 1: Example semantic data types across different domains and industries.

checksum algorithm [7] (shown in the right part of Figure 2), so that incorrectly-entered credit card numbers can be quickly and inexpensively detected. Leveraging this semantic validation, one can safely conclude that the second-to-last column in Figure 1 is `credit-card` since all numbers contain valid card issuer/bank information, and pass the credit-card-specific checksum.

As another example, Figure 3 shows the structure of `ISBN-13` numbers. In this case sub-parts of the numbers can be decoded for rich semantics such as country names, publisher names, book titles, etc. The last digit is again a check digit, but in this case computed using a different GS1 algorithm [5] (an industry standard in business communication). With this type-specific validation, one can again determine that the last column is likely `ISBN` from values alone.

Semantic data types with sophisticated validation (where checksum is one example mechanism) are surprisingly common. These include many widely-used types such as `UPC-code` (unique identifiers for trade items), `US-VIN-number` (vehicle identifiers in US), as well as many others as listed in Table 1.

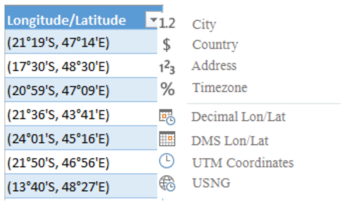
Our observation is that data types in most cases can be algorithmically detected from values, like all the columns in Figure 1. For types like `credit card`, there already exists validation functions like `bool isValidCreditCard()` (where the challenge is to find them), but for many other data types, validations are often implicit that require *re-purposing existing code snippets*. For example, there are code on GitHub that parse date strings (e.g., "`Sep 15, 2011`") into components such as year, month, day, etc. Our observation is that in this process, the validity of `date` strings is implicitly checked using domain-specific logic (e.g., "`Sep`" and "`Oct`" are valid months but "`Abc`" is not). Our key contribution is to automatically identify and reuse such logic from existing code to eventually synthesize new type-detection functions, without requiring developers to program such logic from scratch.

In the context of a commercial data preparation system, we identified a benchmark of 112 example data types whose automatic detection is beneficial[1]. Table 1 shows a subset of these types (with the full list in Appendix A). Note these types can by no means exhaustively capture all useful data types. In reality there is a long tail of concepts and data types specific to different industries (e.g., financial, pharmaceutical, etc.), as well as different languages and cultures (e.g., `dates` and `addresses` that are not in US English or the "en-us"). Writing type-detection logic from scratch for each such scenario is clearly costly and could not scale; automatically harvesting and reusing type-detection logic is thus crucial.

[1]We plan to release both the benchmark data, as well as synthesized type-detection functions to benefit the community at large. We are working with our product and legal teams to make this happen. These resources will be released in a future date on GitHub at https://github.com/congy/AutoType.

Applications of rich types. Automatically detecting rich data types enables a wide variety of applications and experiences, some of which are discussed below (more in Appendix C).

Table understanding. An obvious benefit of automatic type detection is to help users understand tables that they may otherwise be unfamiliar with. For instance, suppose in Figure 1 the column names are either not given or too generic/cryptic (which is often the case [20]). Users can better understand this table if columns can be automatically annotated with types (e.g., the last two columns are `credit-card` and `ISBN`).



Longitude/Latitude
(21°19'S, 47°14'E)
(17°30'S, 48°30'E)
(20°59'S, 47°09'E)
(21°36'S, 43°41'E)
(24°01'S, 45°16'E)
(21°50'S, 46°56'E)
(13°40'S, 48°27'E)

Figure 4: Once the column is detected as longitude / latitude, transformations can be automatically suggested, e.g. converting to city, timezone, and other coordinate systems.

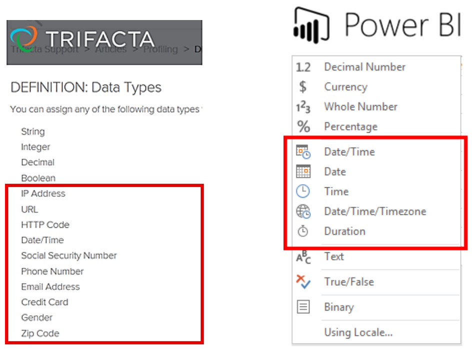
Table search. Keyword search of structured tables, both on the Web [15, 16] and in enterprises [20], are important and have led to influential systems like Google Web Table Search[2] and Microsoft Power Query Table Search[3]. However, a main challenge is that table column headers are often missing or non-descriptive, making keyword search difficult. If columns can be automatically enriched with with semantic types (e.g., `ip-address`, `credit card`, etc.), it is clearly beneficial to search relevance.

Data quality validation. One intended use of the type-detection logic produced by AutoType is in a commercial data preparation system, where the goal is to automatically detect column types and check data validity. For instance, if 99% of values in a column are detected as `ISBN` but a small fraction is not, the system can automatically flag outliers as potential errors for users to inspect.

Semantic data transformation. Another application in data preparation is to enable rich and type-specific transformations. In Figure 4 for example, once a column is detected as type `longitude/latitude`, users can click and select from a list of semantic transformations specific to this type. As we will show, AutoType can naturally support such transformations as it leverages existing code.

Types-detection in existing commercial systems. Realizing the benefits of types, a new generation of commercial data systems start to introduce a limited number of semantic types. For instance, Trifacta[4] detects around 10 types (Figure 5(a)). Microsoft Power BI[5] also supports a few `date-time` types (Figure 5(b)).

[2]http://research.google.com/tables
[3]http://office.microsoft.com/powerbi
[4]https://docs.trifacta.com/display/PE/Supported+Data+Types, retrieved in 2017-09.
[5]https://docs.microsoft.com/en-us/power-bi/desktop-data-types, retrieved in 2017-09.



(a) Trifacta (b) Power BI

Figure 5: A limited number of types used in commercial systems (retrieved in 2017-09).

Our analysis suggests that at the time of this writing, existing systems often rely on manually-defined regex patterns to detect data types. As one could imagine, using syntactic patterns can be inaccurate (e.g., Trifacta detect 16 digit numbers with certain prefixes as `credit card` without validating checksum), and thus can lead to false-positive detection.

AutoType using open-source code. Our objective of AutoType is produce type-detection logic for a large number of semantic types, so that a data preparation system can leverage it to automatically detect types out-of-the-box. It quickly becomes clear that regex patterns alone are not sufficient (e.g., we would not want to predict all columns with 12-digit numbers to be `UPC`).

One alternative is to manually write detection code for each type from scratch. While engineers can write code for common types (e.g., `date-time` and `email-address`), this becomes difficult as we go into the long and fat tail of rich semantic types (such as ones in Table 1), which include notation standards in disciplines like chemistry and astronomy, as well as many industry-specific ISO standards (e.g., ISO 6346 for the container shipping industry[6]). Engineers often have to obtain a rudimentary level of domain-specific knowledge by reading specifications before they can start writing code, which is both costly and error-prone.

Our key observation is that for decades developers have been writing custom-code to process data of various types, and placed their code in places like the public GitHub or proprietary enterprise Git [3]. Since in many cases the logic to handle different data types already exist in such code, our goal is to design an *extensible search system*, which when given any target data type, can automatically *discover and synthesize corresponding type-detection logic using existing code.* Again since the 112 types tested here are by no means exhaustive, being able to easily extend to new types is a key requirement.

Using AutoType, developers of our data-preparation system only need to provide (1) a few positive examples of a target type and (2) a name for the type. (Alternatively, they may just point AutoType to an existing data column whose type needs to be detected, which would already have both information). AutoType automatically searches and synthesizes type-detection logic that developers can then manually inspect and verify.

[6]https://en.wikipedia.org/wiki/ISO_6346

In an enterprise setting, AutoType can be equally applicable, as administrators or data stewards can connect AutoType with their proprietary enterprise code repositories to find type-detection logic relevant to them. Logic so discovered can then be inspected and "registered" by administrators for the benefit of all enterprise users.

It is worth emphasizing that AutoType is intended to assist *technical users* like developers and administrators; *non-technical end-users* can benefit from applications enabled by rich semantic types transparently.

2 PROBLEM DEFINITION

We formally define our type-detection problem as follows.

Definition 1. *Program Synthesis for Type-detection.* Given a large collection of code repositories **R** (e.g., GitHub), a set of user-provided positive examples **P** for a target data type **T**, and a keyword name N describing **T**; automatically produce a ranked list of synthesized type-detection functions using code in **R**, whose execution would lead to a Boolean result for predicting whether an input string is of type **T**.

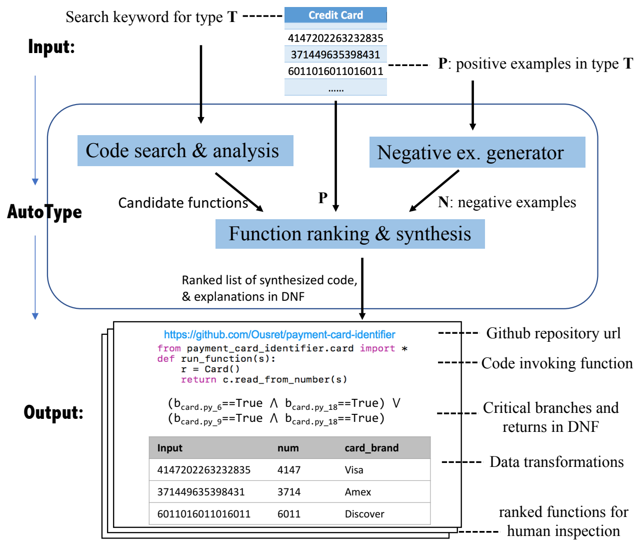
Note that the input required in Definition 1 is relatively easy to provide. For instance, the positive examples **P** and keyword name N are often readily available – users[7] only need to point AutoType to an existing data column (e.g., in Figure 1) and that is already sufficient. For **R** we use the popular open-source repository GitHub, but any code repository such as CodePlex [2] or proprietary Enterprise Git [3] can also be used.

Enumerable vs. un-enumerable types. In this work, we focus on using synthesized code to detect "types" whose value domains are too big to be exhaustively enumerated. All the columns/types in Figure 1 are in this category – these types often involve a numeric component, (e.g., `date` or `ip-address`), whose exhaustive enumeration is inefficient or unnecessary. Compact detection logic in the form of code is often the best for such types.

"Data types" can sometimes refer to types with categorical values that are enumerated, e.g., the list of all countries or state-abbreviations, etc. Such types often contain a limited number of constant string values, for which knowledge-bases [12, 40] and mapping tables [42] that exhaustively enumerate them is typically the best for detection [19, 36]. While AutoType can still find code to recognize such types (e.g., we find GitHub code that can parse `country-codes` or `airport-codes`, and return relevant information of input entities), in this work we do not focus on the enumerable types since they can already be handled reasonably well using knowledge-bases.

Data types vs. data semantics. At the risk of stating the obvious, we want to emphasize that in this work we make the distinction between *data types* and *data semantics*. We define data types to be type information that can be unambiguously inferred from data values alone, such as `date`, `address`, and `credit-card`. (For columns whose types cannot be uniquely identified we can fall back to primitive types like `string` and `number`). Note that this notion of types is different from the *semantics* of data, where an `address` can mean `mailing-address` vs. `billing-address`, or a number can be `GDP` vs.

[7]In this work, we use "users" to refer to technical users like developers when the context is clear.



https://github.com/Ousret/payment-card-identifier

```
from payment_card_identifier.card import *
def run_function(s):
    r = Card()
    return c.read_from_number(s)
```

$(b_{card.py\_6}{=}{=}True \wedge b_{card.py\_18}{=}{=}True) \vee$
$(b_{card.py\_9}{=}{=}True \wedge b_{card.py\_18}{=}{=}True)$

Input	num	card_brand
4147202263232835	4147	Visa
371449635398431	3714	Amex
6011016011016011	6011	Discover

Figure 6: Overview of AutoType architecture.

population. Column semantics is clearly useful but difficult to determine in the absence of contexts such as column headers. We thus consider data semantics to be beyond the scope of this work.

3 SYSTEM OVERVIEW

Figure 6 gives an overview of the AutoType system. The top part shows input to AutoType. As discussed in Definition 1, users only need to provide the name of a target type **T** (e.g., "credit card") as a search keyword, and a few positive example values **P** (e.g., valid credit card numbers).

In the first *code search and analysis* component, AutoType uses the search keyword to retrieve top-*k* relevant repositories, utilizing GitHub and Bing's Search API. Functions from these repositories are analyzed based on certain criteria (e.g., whether they can be compiled and executed) to produce candidate functions **F** (Section 4).

Using the set of positive examples **P**, we design automatic methods to dynamically generate negative examples **N** likely not in type **T**. Note that while we can also ask users to provide negative examples, unlike positive examples it is often difficult for humans to systematically enumerate negative examples (Section 6).

```
1  class CreditCard:
2    def read_from_number(s):
3      # Get 4-digit prefix of card numbers
4      num = int(s[:4])
5      # Visa starts with 4
6      if num/1000 == 4:
7        self.card_brand = 'Visa'
8      # Mastercard starts with 50-55
9      elif num/100>=50 and num/100<=55:
10       self.card_brand = 'Mastercard'
11     # Handle other types: Discover, JCB, ...
12     elif ...
13
14     # Next, validate credit-card checksum
15     temp_sum = ... # computing luhn sum
16     if temp_sum % 10 == 0:
17       self.cardnumber = s
18     # More processing for bank info, etc.
19     ...
20     return self
```

Listing 1: Code processing a credit card string

	Positive examples	Branches & Return objects
e_1^+	4147202263232835	b_6 == True, b_{18} == True, r_{24} = object{brand=..., number=...}
e_2^+	5115915115915118	b_6 == False, b_9 == True, b_{18} == True, r_{24} = object{brand=..., number=...}
	...	...

	Negative examples	Branches & Return objects
e_1^-	4147202263232836	b_6 == True, b_{18} == False, r_{24} = object{brand=..., number=...}
e_2^-	4147202263232736	b_6 == True, b_{18} == True, r_{24} = object{brand=..., number=...}
e_3^-	A147202263232835	b_4 = exception("Invalid literal for int")
	...	...

Figure 7: Positive and negative examples, and their corresponding execution traces (branches taken, objects returned) for running the function in Listing 1.

The main *Function ranking and synthesis* component takes as input the set of candidate functions **F**, positive examples **P** and negative examples **N**. It first modifies each function $F \in \mathbf{F}$ by injecting profiling logic, so that during execution, program internal states such as paths taken and objects returned can be recorded. We then execute each modified function F using examples in **P** and **N** as parameter. For example, we can execute the read_from_number() function in Listing 1, using positive/negative examples in Figure 7.

Using information such as key branches taken during execution, AutoType automatically generates an explanation in disjunctive-normal-form (DNF) [22], that can explain positive examples away from the negative ones. In the example of Listing 1, AutoType will be able to find a perfect explanation that tells positive and negative examples apart using the following DNF: (b_6 == True $\wedge$ b_{16} == True) $\vee$ (b_9 == True $\wedge$ b_{16} == True) $\vee$ ..., where b_i indicates the branch value on line i. This DNF clearly explains important execution states (in this case branches) that should be taken by valid credit card numbers.

Note that DNF explanations of a function is dependent on input examples used. For instance, if input examples only contain `Visa-credit-cards` instead, the DNF may be produced as (b_6 == True $\wedge$ b_{16} == True) accordingly.

AutoType analyzes across all candidate functions **F** from the previous step, and uses a holistic optimization formulation to rank results. Note that from the ranked functions, users can inspect DNF along with source code (e.g., variable names, comments, and detailed logic) to verify whether a function is indeed relevant to a target type (Section 5).

As a by-product of identifying relevant code, AutoType also produces semantic data transformations that are internal states of executing these functions (e.g., `card-brand`/`issuer` of `credit cards`, shown at the bottom of Figure 6).

In the following we describe components of AutoType in detail.

4 CODE SEARCH AND ANALYSIS

AutoType first searches and identifies candidate functions **F** from relevant code repositories.

4.1 Code search using keywords

AutoType uses the type name N as a keyword query to search for relevant repositories on GitHub and GithubGist [4] (which hosts a large number of diverse code snippets). We leverage both the GitHub search API as well the Bing search API (with queries like "`credit card site:github.com`"). We take the *union* of top-40 repositories returned by these two APIs since their results are often complementary. For the scope of this work, we only focus on Python repositories but the techniques discussed here can be easily adapted to other languages. The identifiers of top-returned repositories are then extracted for the complete repositories to be crawled locally using the GitHub clone.

Discussions. It is worth noting that the search keyword used in this step is not required to be precise, for it only serves as a crude pruning step to weed out irrelevant repositories. For instance, for type `IPv4` the query "IP address" works fine, and for `ISBN-13` the query "ISBN" also works. The positive examples **P** and negative examples **N** can more precisely describe the desired types, which help to pinpoint to relevant type-detection functions (Section 5).

4.2 Code analysis for candidate functions

For each crawled repository, AutoType uses Python's built-in AST parser [8] to analyze all .py source files and obtain abstract syntax trees (ASTs). We then extract functions from AST, and identify the ones that are (1) suitable for single-parameter invocations, and (2) actually compilable and executable.

```
F(s);
a=classA(); a.F(s);
a=classA(s); a.F();
F(); #Within F(), replace sys.argv with s
F(); #Within F(), replace input() with s
fp=open('f.txt','w'); fp.write(s); F('f.txt');
```

Listing 2: Examples of invoking functions F that potentially accept one parameter string s.

For the first criterion, we focus on functions suitable for single-parameter invocations. Using AST-level information, AutoType handles six variants of functions that can accept a single parameter, as discussed below (each of which corresponds to an example snippet in Listing 2):

(1) Non-class function that takes a single parameter;
(2) In-class, single-parameter member function, whose class has parameter-less constructors;
(3) In-class, parameter-less member function, whose class has single-parameter constructors;
(4) Non-class, parameter-less function that takes an implicit parameter from system arguments;
(5) Non-class, parameter-less function that takes an implicit parameter from console input;
(6) Non-class, parameter-less function that takes an implicit parameter by reading input from files.

In addition to functions, AutoType also directly execute code snippets outside functions in python files. In the interest of space we leave more details of this step and its extensions (e.g. invoking functions with multiple parameters) to Appendix D.

For the functions that can be invoked in these ways, we then test whether they are actually compilable and executable. One challenge in programmatically compiling and executing Python projects is that most of them have external dependencies (other Python packages). To address this, AutoType parses the `requirements.txt` file if available (which describes required dependencies), as well as exception messages thrown during execution, to automatically identify and install missing packages using pip. This execute-parse-install-rerun process may loop for multiple times (each time with a different exception) before the function can successfully execute.

5 FUNCTION SYNTHESIS AND RANKING

Given candidate functions **F**, AutoType executes each $F \in \mathbf{F}$ using positive examples **P** and negative examples **N**, in order to find salient features from execution traces to differentiate **P** and **N**.

In AutoType **P** is given by users and **N** is automatically generated. However, to better focus on the core ranking part, for now we will assume that **N** is already given, and defer the discussion on automatically generating **N** to Section 6.

5.1 Profile execution traces

As discussed in Section 3, our key idea is that given a target type **T**, if a function $F \in \mathbf{F}$ handles data of type **T**, then its internal program states can often be used to reliably differentiate positive examples from negative ones, because the positive examples (or the input values expected by F) would likely follow paths corresponding to normal data processing, whereas the negative examples that F cannot handle would likely error out and thus take different paths. This observation was illustrated with the example in Figure 7 and is very common across data/functions tested.

Specifically, we record each branch value and return value during execution. Branch values determine whether a branch is taken and directly control program paths (illustrated as "b_i == True/False"), while return values often reflect important execution results (illustrated as r_i). Note that this tracing is inter-procedure: traces of all functions invoked directly and indirectly by F will all be recorded.

To obtain complete execution traces of F across the full call stack (e.g., including other functions invoked by F), we programmatically instrumentcompiled Python byte-code to dump all branch comparisons and return values. More details about the instrumentation can be found in Appendix D.2.

Note that while our implementation modifies byte-code which is specific to Python, this approach can be easily applied to any other languages that supports programmatic instrumentation (e.g., Pin for C/C++ [27] or BTrace [1] for Java). Furthermore, for any programming language that supports programmatic debugging (e.g., Javascript [6] and Ruby [9]), one could also step through source code line-by-line, and record internal execution states, to also achieve implementation without code modification.

5.2 Rank functions by DNF

Using the observation that branches and return values are useful features, in this step we rank functions $F \in \mathbf{F}$ based on how well they can differentiate **P** from **N**.

Featurization. For each example $e \in \mathbf{P} \cup \mathbf{N}$, the execution of $F \in \mathbf{F}$ with parameter e creates traces that include branch/return events, denoted as $T(e)$. First, we *featurize* events in $T(e)$ to make them suitable for reasoning.

In principle, we can log all events and model them as *sequences* to preserve order information; or as *multi-sets* that are unordered but retain the occurrence counts (e.g., for loops); or simply as *sets*. We find that for function ranking, the set-based featurization is already expressive enough. Furthermore, given that the number of examples in **P** is often limited (e.g., around 20), it can be advantageous to use set-based model to avoid data sparsity.

For each branch/return event, we create binary features and treat them as a set. Specifically, for branch conditions, each branch b_i is translated into two binary features, b_i == `True` and b_i == `False`, corresponding to each branch condition.

For return values r_i, if r_i is already an atomic value (e.g., bool, number) we simply print its value; if r_i is of collection type (e.g., `dict, list`), we print the length of r_i; if r_i is a composite object, we print whether r_i is `None` (i.e., Python's equivalent of `Null`). We then create simple binary featurizations: for boolean r_i, we just use r_i = `True` and r_i = `False`, for numbers and lengths we use $r_i = 0$ and $r_i \neq 0$. Note that an alternative strategy is to dump constituent atomic values of all objects (recursively if needed), and create more complex featurizations. However given the large number of features and a relative small number of examples, to avoid data sparsity we choose to use the simpler featurization.

With these, each execution of a function $F \in \mathbf{F}$ over one example $e \in \mathbf{P} \cup \mathbf{N}$, is featurized as a set of binary clauses as shown in Figure 7. We denote each such trace as $T(e)$.

Example 1. Executing function F in Listing 1 using e_1^+ in Figure 7 produces a trace: $T(e_1^+) = \{b_6 == \text{True},\ b_{16} == \text{True}, r_{20} \neq \text{None}\}$. Similarly, $T(e_2^+) = \{b_6 == \text{False}, b_9 == \text{True},\ b_{16} == \text{True}, r_{20} \neq \text{None}\}$, etc. Note that exceptions, such as ones encountered by e_3^- in Figure 7, is also recorded in the trace.

Ranking as Optimization. Given the featurized trace $T(e)$ for each execution, we are now ready to formulate the function ranking problem. Recall that our formulation has two objectives: (1) quantitatively determine how well $F \in \mathbf{F}$ differentiates **P** and **N**, so that we can compare across all candidate functions **F**; (2) generate human-understandable explanations to assist users understand whether F can indeed work as expected to process target type **T**.

Observing that the desired explanations in programs are often disjunctions of multiple conjunctive literal features, we propose to use disjunctive-normal-form (DNF) [22] as our generated explanations, which is a widely-used canonical logical form. The intuition for using DNF is that each conjunction of literals often correspond to a specific program-path/return-conditions that a subset of positive examples satisfy, and using a disjunction corresponds to taking a union of these subsets.

Example 2. In the example of Listing 1, one possible DNF explanation is (b_6 == True ∧ b_{16} == True) ∨ (b_9 == True ∧ b_{16} == True) ∨..., whose conjunctive clauses correspond to sub-paths that valid credit card input will pass through.

Note that valid credit card input may pass through branch conditions in Listing 1 that may not be relevant for type-detection. We thus define a notion to allow a partial conjunctive clause C to "cover" an example $e \in \mathbf{P} \cup \mathbf{N}$. Define $B(F) = \bigcup_{e \in \mathbf{P} \cup \mathbf{N}} T(e)$ as the union of all possible binary clauses from executions of F.

Definition 2. *Cover.* A conjunctive clause $C = c_1 \wedge c_2 \wedge \ldots, \wedge c_m$ with $c_i \in B(F)$, is said to *cover* an example $e \in \mathbf{P} \cup \mathbf{N}$, if $c_i \subseteq T(e), \forall c_i \in C$. We denote this by $C \subseteq T(e)$. We further define the set of examples covered by C as $Cov(C) = \cup_{e \in \mathbf{P} \cup \mathbf{N}} \{e | C \subseteq T(e)\}$.

Example 3. Given that $T(e_2^+) = \{b_6 == \text{False}, b_9 == \text{True},\ b_{16} == \text{True}, r_{20} \neq \text{None}\}$, the conjunctive clause $C = (b_9 == \text{True} \wedge b_{16} == \text{True})$ *covers* e_2^+, because the set of literals in C is a subset of $T(e_2^+)$.

Intuitively, a clause C covers an example e if the conditions specified in C is a sub-part of, and consistent with $T(e)$.

Given the definition of cover, intuitively we want to find an ideal DNF explanation, such that the union of its conjunctive clauses cover all of **P**, and none of **N**.

However, recall that in our problem setting, because users are not required to provide **N**, we automatically generate **N** by dynamically mutating positive examples (to be discussed in Section 6). In some cases some **N** so generated may accidentally be positive examples. For example, in Figure 7 when we mutate digits in e_1^+ to generate e_2^-, the resulting e_2^- actually passes the checksum (the chance of this happening is $\frac{1}{10}$). As such, while we want to ideally generate an DNF that covers all of **P** and none of **N**, in practice this is hard to guarantee, and we could use "no **N**" as a "hard" constraint. We arrive at an optimization formulation that maximizes the coverage of **P**, while limiting the coverage of **N** as a soft-constraint.

Definition 3. *Best-DNF-Cover.* Given a function $F \in \mathbf{F}$, positive examples **P** and negative examples **N**.Find a logical clause in DNF using literals from $B(F)$, that can cover as many examples in **P** as possible, subject to a budget of covering at most $\theta|\mathbf{N}|$ examples in **N**, for some given $\theta \in [0, 1]$.

$$\max_{\substack{\forall D \in \text{DNF}(B(F)),\\ D = C_1 \vee C_2 \ldots \vee C_n}} \left| \bigcup_{i \in [n]} Cov(C_i) \cap \mathbf{P} \right| \tag{1}$$

$$\text{s.t.} \quad \left| \bigcup_{i \in [n]} Cov(C_i) \cap \mathbf{N} \right| \leq \theta |\mathbf{N}| \tag{2}$$

This formulation reflects the consideration that not all generated negative examples in **N** are truly negative, and a θ fraction of negative examples may be covered.

Definition 4. *Best-k-Concise-DNF-Cover.* Given a function $F \in \mathbf{F}$, positive examples **P** and negative examples **N**. Find a logical clause in DNF using literals from $B(F)$, whose constituent conjunctions contain up to k literals, that can cover as many examples in **P** as possible, subject to a budget of covering at most $\theta|\mathbf{N}|$ examples in **N**, for some given $\theta \in [0, 1]$.

The optimization formulation of *Best-k-Concise-DNF-Cover* will be exactly Equation (1) and (2), plus an additional constraint $n \leq k$.

The consideration behind using DNF (instead of full execution paths) and restricting the number of clauses is that otherwise complicated DNF (with hundreds of literals) is hard for humans to understand. Furthermore, very specific DNF recording full-path information often reduces the *generalizability* [32] of the resulting DNF. For example, it may be too brittle to handle slight variation in positive examples, thus hurting the quality of synthesized validation function in recognizing unseen positive examples.

In practice, we observe that for most cases, a few key branches or return conditions can succinctly capture the common characteristics of positive examples in a target **T**. We thus propose a k-concise version of Problem 3 defined below, requiring each conjunctive

Algorithm 1 Generate DNF for *Best-k-Concise-Cover*

```
INPUT: $\mathbf{P}, \mathbf{N}, \theta, F$
1: Partition $c \in B(F)$ into groups $\{G_1, ..., G_m\}$, where $c_x \in G_i \wedge c_y \in G_i$ iff $Cov(c_x) = Cov(c_y)$
2: $S = \{c_1, ..., c_m | c_i \in G_i\}$
3: $L = \{C | C \in 2^S, |C| \leq k\}$
4: $cur \leftarrow \emptyset$
5: while $|Cov(cur)| < |\mathbf{P}|$ and $best \neq \emptyset$ do
6:     $L' = \{C | C \in L, |Cov_{\mathbf{N}}(cur \vee C)| \leq \theta |\mathbf{N}|\}$
7:     $best \leftarrow \arg\max_{C \in L'} |Cov_{\mathbf{P}}(cur \vee C)| - |Cov_{\mathbf{P}}(cur)|$
8:     $cur \leftarrow cur \vee best$
   return $cur$
```

clause to include at most k literals. We find in our experiments that limiting the complexity of DNF improves function ranking.

Example 4. Suppose in a *Best-k-Concise-DNF-Cover* problem with $k = 2$, $\theta = 0.2$ and $\mathbf{P}$ and $\mathbf{N}$ as shown in Figure 7. As can be verified, one best cover is (b_6 == True $\wedge$ b_{16} == True) $\vee$ (b_9 == True $\wedge$ b_{16} == True), which covers all $\mathbf{P}$ while satisfying the budget of mistake coverage on $\mathbf{N}$.

Theorem 4. The problem in Definition 4 is NP-hard. Furthermore, it cannot be approximated within a factor of $2^{(\log n)^{\delta}}$, for some δ, unless $3SAT \in DTIME(2^{n^{\frac{3}{4}+\epsilon}})$.

The hardness result can be obtained using a reduction from set-union knapsack (a proof of this is in Appendix F). In light of the complexity and inapproximability, we propose a greedy algorithm for *Best-DNF-Cover*.

We first partition literals in $B(F)$: literals with identical coverage of $\mathbf{P} \bigcup \mathbf{N}$ are merged in groups, denoted as $G_1(F), ..., G_m(F)$. Usually literals in each group represent redundant features in sub-paths taken by a subset of examples (there may be hundreds of such features, many of which are redundant). We then pick one literal from each group into a candidate set S, and enumerate all conjunctive clauses up to length k using literals in S, which is effectively a subset of the power-set of S, denoted as $L = \{C | C \in 2^S, |C| \leq k\}$ (the space allowed by Definition 4). This has a total complexity of $O(|S|^k)$ (where k is a small fixed constant, and $|S|$ is much smaller than the number of raw features).

We then greedily pick the best conjunctive clause from L that achieves the most additional coverage of positive examples, without violating the constraint on the negative examples. This algorithm is illustrated in Algorithm 1. We use $Cov_{\mathbf{P}}$ and $Cov_{\mathbf{N}}$ as shorthand notations for coverage of positive and negative examples, respectively. Note that other heuristics, such as picking the conjunctive clause with the best positive-to-negative ratio, are also possible choices.

Ranking-by-DNF. Given functions $\mathbf{F}$, positive examples $\mathbf{P}$ and negative ones $\mathbf{N}$, for each function $F \in \mathbf{F}$, we generate $DNF(F)$ as formulated in Definition 4. Functions are ranked by the positive-example coverage $Cov_{\mathbf{P}}(C_i)$, with the negative coverage $Cov_{\mathbf{N}}(C_i)$ used as a tie-breaker.

5.3 Synthesize validation function from DNF

If users are satisfied with DNF generated by AutoType, and select a top-returned function F, the DNF associated with F, DNF(F), will be used to automatically synthesize type-validation function based on F. When AutoType runs on a new input s on F, it records the trace and featurizes into $T(s)$, and check whether $T(s)$ is consistent with the DNF(F). The pseudo-code of this step is in Appendix G.

5.4 Manual Verification of Functions

For developers, it is often important to manually verify the correctness of functions synthesized by AutoType. Conceptually, a function may be incorrect because: (1) it is not related to the target data type at all; and (2) it intends to process the target type but does not work as designed (e.g., buggy, not comprehensive, etc.).

From our experience, the first category of false-positives is easy to detect, because variable names, function names and comments often provide sufficient contexts. The second category is more difficult to catch, but the positive/negative examples AutoType leverages are very effective in pruning out such functions.

We should note that as good engineering practices, developers should always build comprehensive unit tests for corner cases, irrespective of whether the code is written by themselves, taken from open-source repositories, or synthesized automatically. For example for `credit-card` it would be good to have unit tests for different types of card numbers (e.g., Visa, Master, Amex, etc.) that should return true, and a few others that should return false. (Note that these should be built independent of examples used by AutoType). Since these are required even if developers choose to write code from scratch, they are not additional burdens imposed by AutoType.

In addition to function correctness, developers also need to verify license and security aspects of open-source code. We leave such discussions to Appendix E.

6 GENERATE NEGATIVE DATA

So far we have assumed that the negative examples $\mathbf{N}$ are given as discussed in Section 5. In practice, since it is difficult for users to systematically provide representative negative examples, AutoType obviates the burden by automatically generating examples that are likely negative.

Our first observation is that $\mathbf{N}$ cannot be trivially generated (e.g., random strings) such that they are entirely different from $\mathbf{P}$. For example, suppose the target type $\mathbf{T}$ is in the numeric domain (e.g., `credit-card`). If we were to use random strings like "`ABC123.?`" as $\mathbf{N}$, then almost all functions that accept an `int`, denoted as $\mathbf{F}_{int}$, will process $\mathbf{P}$ normally, while throwing exceptions for the random $\mathbf{N}$. Thus all $\mathbf{F}_{int}$ will create significantly different traces between $\mathbf{N}$ and $\mathbf{P}$, as if they can all detect $\mathbf{P}$, which is clearly incorrect. The issue is that randomly generated $\mathbf{N}$ is so different from $\mathbf{P}$ that they are not useful in telling which functions are truly relevant.

In theory, if we were able to really test all "representative" data values covering the entire space of possible input, it would allow us to know that certain functions are more "selective" than general $\mathbf{F}_{int}$ and hence potentially more relevant (conceptually using an argument reminiscent of version-space algebra). However, in practice function executions are expensive (e.g. some may invoke web service calls), such that running millions of $\mathbf{N}$ across thousands of candidate functions is not viable.

Our observation is that small "mutations" of $\mathbf{P}$ of a given type $\mathbf{T}$ can often generate informative negative examples that are more discriminative. For example, by randomly mutating digits in valid `credit-card` numbers and using them as $\mathbf{N}$, with high likelihood we produce invalid numbers (e.g., due to checksum), thus allowing a true credit-card-checksum function to be selected. In comparison, general $\mathbf{F}_{int}$ functions accept any integers and would not be able to tell the small difference between this **N** and **P**.

As we will see below, the optimal strategy to "mutate" **P** to generate true **N** varies across data types in **T**. However, there exists a strict hierarchy of strategies ordered by the amount of mutations injected. This allows us to dynamically test different strategies, starting from the strategy with the least amount of mutation.

We observe that in most data types, punctuation often serve as structural delimiters (e.g., dots in `ip-address`, spaces in `date`, dashes in `phone-number`, etc.), while non-punctuation characters typically contain real content. In addition, each data type **T** often has its type-specific "alphabet", defined as follows.

Definition 5. Let Σ be the complete alphabet with all punctuation Σ_P, and non-punctuation $\Sigma_{\overline{P}}$ (which includes both numbers and letters), respectively. For a string s, let $C(s)$ be the set of characters in s. Given **P** of a type **T**, we define $\Sigma(\mathbf{P}) = \bigcup_{s \in \mathbf{P}} C(s)$ as the *inferred alphabet* of **T**, any $c \in \Sigma(\mathbf{P})$ as an *in-alphabet character.*

We can additionally define $\Sigma_{\overline{P}}(\mathbf{P}) = \bigcup_{s \in \mathbf{P}} (C(s) \cap \Sigma_{\overline{P}})$ as the *in-alphabet, non-punctuation characters* for a given **P**.

Example 5. Given a set of valid `IPv4-addresses` **P** (e.g., the `IPv4` column in Figure 1), the inferred alphabet is $\Sigma(\mathbf{P}) = \{0, \ldots, 9\} \cup \{\text{"."}\}$. The in-alphabet, non-punctuation characters is $\Sigma_{\overline{P}}(\mathbf{P}) = \{0, \ldots, 9\}$, representing address information; while "." is the only punctuation used for structuring addresses into segments.

For the `date` type with examples of the format of "`Jan 01, 2011`", the inferred alphabet is $\Sigma(\mathbf{P}) = \{0, \ldots, 9\} \cup \{a, b, \ldots\} \cup \{\text{" "}, \text{","}\}$ (space and comma). The punctuation is again structural to separate components such as year, month and day.

With these, we now describe a hierarchy of three mutation strategies, strictly ordered by the amount of mutations injected into **P**.

S1. Mutate-preserve-structure. Given an example $s \in \mathbf{P}$, with some fixed probability p, this strategy replaces each *in-alphabet non-punctuation character* from s, with another *in-alphabet non-punctuation character* randomly drawn from $\Sigma_{\overline{P}}(\mathbf{P})$.

Within our hierarchy, S1 performs the least amount of mutation to positive examples. By replacing non-punctuation characters in $s \in \mathbf{P}$ with other in-alphabet non-punctuation characters, while leaving structural components (i.e. punctuation) intact, this will likely generates new positive examples as opposed to negative ones. For example, for many types such as `IPv6`, `phone-numbers`, `JSON`, and `XML`, S1 still generates positive data with high likelihood.

However, for a class of data types that maintain strong internal consistency, such as types with internal checksum like `credit-card`, `ISBN`, `VIN`, etc., S1 will likely produce enough negative examples (e.g., with a probability $\frac{9}{10}$ for `credit-card`) such that AutoType can already identify relevant functions. Similarly, for other types with global structures (but not checksum) such as `address`, where small changes to an address like "`1 Wall St., New York, NY 10286`" can often lead to an invalid address "`1 W`**`z`**`ll St., New `**`A`**`ork, NY `**`9`**`0286`", good semantic address-parsers can again differentiate between **P** and **N**. For these types we can stop after trying S1, but for remaining types where no functions can differentiate between **P** and **N** generated using S1, we move on to S2 to inject more mutations.

S2. Mutate-preserve-alphabet. Given an example $s \in \mathbf{P}$, with some fixed probability p, this strategy replaces each *in-alphabet character* (punctuation and non-punctuation) in s with another *in-alphabet character* randomly drawn from $\Sigma(\mathbf{P})$.

S2 differs from S1 in that we are now mutating punctuation characters in addition to non-punctuation, which for some types may break the internal structure maintained through punctuation. For example, for types like `date` and `IPv6`, mutating positive example $s \in \mathbf{P}$ will likely generate negative examples when punctuation in s is altered. So for these types S2 is sufficient as AutoType will find functions that can differentiate between to **P** and **N** so generated.

However, types like `gene-sequences` and `Roman-numeral` contain only non-punctuation characters drawn from type-specific alphabets (e.g., "ACGT" for `gene-sequences`, "XVILCDM" for `Roman-numeral`, etc.). For such types, S2 still does not generate negative examples and we will resort to S3 below.

S3. Mutate-random. Given an example $s \in \mathbf{P}$, with some fixed probability p this strategy replaces each *in-alphabet character* in s with *any random character* drawn from the full English alphabet Σ.

In S3, by replacing characters with random ones, we inject the most amount of mutations and will generate negative examples for types like `gene-sequences` and `Roman-numeral` (for which S2 fails).

Proposition 1. *Given any $s \in \mathbf{P}$, let $S1(s)$, $S2(s)$, $S3(s)$ be the spaces of mutations with policy S1, S2 and S3, respectively. The following set-containment relationship holds: $S1(s) \subseteq S2(s) \subseteq S3(s)$.*

This proposition shows that S1, S2 and S3 form a hierarchy strictly ordered by the amount of mutations on **P** (a proof of this is in Appendix H).

Such a hierarchical relationship is useful, because given a new target type **T**, we do not known a priori which strategy is appropriate for **T**. However, because the strategies are ordered, we can test each in turn with an increasing amount of mutation, until the right strategy is used and true **N** generated, at which point *Best-k-Concise-Cover* returns relevant functions as opposed to $\emptyset$.

Algorithm 2 summarizes this process. Note that in *Generate-N-by-Mutation*, we generate a large number of negative examples for each positive example to improve robustness.

Example 6. Suppose we would like to detect `IPv6` addresses, and have `4f:45b6:336:d336:e41b:8df4:696:e2` as a positive example. We start with S1, which will not touch punctuation "`:`", but randomly replace non-punctuation with `[0-9a-f]`. This will still create positive examples. When we treat such examples as **N**, *Best-k-Concise-Cover*(**P**, **N**, F) returns no result as **N** and **P** cannot be differentiated.

We then try S2, which mutates any character (including "`:`") with in-alphabet characters (including "`:`") . S2 is likely to produce true negative examples (due to incorrect number of "`:`"), with which *Best-k-Concise-Cover()* will return relevant `IPv6` functions, and at that point we can stop without using S3.

In addition to replacement-based mutations discussed above, there are orthogonal strategies, such as altering example lengths, which can also be used in addition to strategies discussed above.

Algorithm 2 Generate **N** dynamically

INPUT: P, F

```
1: for Si ∈ [S1, S2, S3] do
2:     N ⟵ Generate-N-by-Mutation(P, Si)
3:     R ⟵ {F|F ∈ F, Best-k-Concise-Cover(P, N, F) ≠ ∅}
4:     if R ≠ ∅ then return R ranked by coverage
```

7 ADDITIONAL APPLICATIONS

7.1 Semantic transformations

Semantic transformations, as illustrated with the example in Figure 4, are transformations specific to a data types once the type is detected. Automatically recommending contextual and type-specific transformations is clearly useful.

Our observation is that when functions process data of type **T**, they often (1) produce necessary intermediate results, and (2) perform additional transformations that are relevant, both of which can be harvested for semantic transformations. For instance, the function in Listing 1 produces `card-issuer` in `self.card_brand` when processing credit card numbers.

Given this, once relevant functions are identified for a target type **T**, AutoType can suggest candidate transformations by leveraging intermediate variables in these functions. These transformations can then be shown in tabular form for each **P** shown in Figure 6, for humans to inspect and identify relevant semantic transformations.

We test AutoType on 20 popular types and manually check candidate transformations so generated. Many transformations are indeed non-trivial to write and useful to have. We leave the detail of this step to Appendix B.

7.2 Type detection in tables

Another application of the synthesized type-detection logic is to detect data types for tables in the wild, where type-semantics are often missing (e.g., column may use very generic headers like "name" or missing altogether [20]). With type-detection logic, we can automatically infer that tables columns are of type `IPv4`, date, ISBN, etc., as shown in Figure 1. Such meta-data will clearly benefit applications like table search and table understanding.

We test AutoType on type-detection by using web-tables [15, 16], which are known to be a rich source of diverse tables. We synthesize type-detection logic for 20 popular data types and run them on a sample of web table columns. Compared to alternatives like Regex-based and keyword-based methods, synthesized type-detection functions can significantly improve both precision and recall of column type detection (Section 9).

8 EXPERIMENTAL EVALUATION

We first evaluate the quality of type-detection code synthesized.

8.1 Experimental setup

Test cases. We identify a benchmark of 112 semantic data types (listed in Appendix A) that are regarded as useful for a commercial data preparation system. Some types like `date-time` have multiple formats/sub-types (e.g.,"`Jan 01, 2017`" vs. "`2017-01-01`"). We create a separate test case for each sub-type, as well as a test case with data mixed from different sub-types.

For each test case we use as input around 20 positive examples (taken randomly from the web), as well as a canonical type name.

Computing environment. We implement a prototype of AutoType in Python 2.7 that searches over public open-source repositories GitHub and Gist. Experiments were run with Azure virtual machines with 16 2.6GHz processors.

Methods compared.

- **DNF-concise (DNF-S):** This is our AutoType approach formulated using Definition 4. We use Algorithm 1, with $k = 3$ and $\theta = 0.3$.
- **DNF-complete (DNF-C):** This is a variant of our approach **DNF-S**, but uses the formulation in Definition 3 without requiring k-conciseness.
- **Return-Only (RET):** This is an additional variant of **DNF-S** that treats functions as *black boxes* and uses only return values (no branches). This allows us to quantify the importance of features derived from program internal logic flows.
- **Keyword match (KW):** We implement a TF-IDF-style [30] keyword-search method that treats each function as a "document", and ranks functions using the search keyword as query. This seemingly naive approach is actually very close to what humans would do manually, because they would likely use a search engine to retrieve top-ranked functions for inspection.
- **Logistic-regression (LR):** Since the positive/negative examples are effectively labeled data, and branches/returns are used as features, we using a conventional machine-learning model, logistic-regression, to rank functions based on regression scores.

Evaluation metric. We evaluate the ranked list of synthesized code using standard IR metrics: *precision@K*, and *normalized discounted cumulative gain (NDCG)* [25].

Precision@K is normally defined as $\frac{\text{\#-relevant-above-K}}{K}$, the number of relevant items above position K divided by K.

One approach to define relevance is to have a human judge inspect the source code of F, and as long as F *intends* to process type **T**, we label F as relevant. However, we find that this often *overstates* the utility of F, for some code on GitHub is not implemented as well as others (e.g., a function that claims to process `IPv4` may only check if the input consists of numbers separated by "." without validating if there are four segments, each below 256).

So in addition to having a human judge to read source code and give a true/false intention score $I(F)$ for each F, we use a hold-out set of 10 positive examples of **T** (separate from the 20 used for training) denoted as $\mathbf{P}_{test}$, which should all pass if F is good; as well as 1000 truly negative examples sampled from web tables denoted as $\mathbf{N}_{test}$ (inspected by humans), for which F should all return false. In reality F is often imperfect, and we define a quality score of F as $Q(F) = 0.5\frac{|\text{pass-in-}\mathbf{P}_{test}|}{|\mathbf{P}_{test}|} + 0.5\frac{|\text{pass-in-}\mathbf{N}_{test}|}{|\mathbf{N}_{test}|}$ (essentially treating $\mathbf{P}_{test}$ and $\mathbf{N}_{test}$ as unit-tests).

With these, the final relevance is computed as $rel(F) = I(F)Q(F)$. Note that if F is not even intended for **T** (our human judge labels $I(F)$ as 0), then the overall relevance is 0 regardless of $Q(F)$.

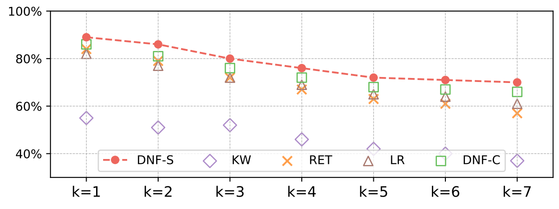
Using $rel(F)$, we can compute precision@K as usual.

Similarly, we compute $NDCG_p = \frac{DCG_p}{IDCG_p}$, where $DCG_p = \frac{\sum_{i=1}^{p} rel_i}{\log_2(i+1)}$, in which rel_i is the relevance of F at position i (and $IDCG$ is computed similarly). Note that NDCG gives a normalized score in [0, 1] relative to ideal ranking.

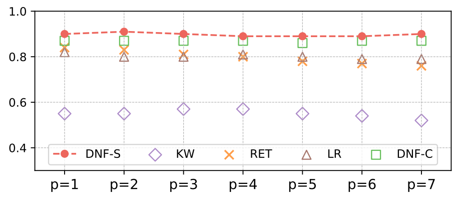
We also report *relative recall* of each method using the *pooling* methodology from IR [29]. Like evaluating search engines, we "pool" top-k relevant results from all methods, and treat their union as a proxy of ground truth. The *relative recall* of a method is defined as $\frac{\text{\#-relevant-top-k}}{\text{\#-total-relevant}}$. In our experiment we use $k = 7$.

8.2 Synthesis quality comparison

8.2.1 Ranking quality. Figure 8(a) and Figure 8(b) show results for precision@K evaluation, and NDCG, respectively. Table 8(c) shows the result of relative recall (from top-7). DNF-S scores the best across all ranking methods, with top-1 precision reaching 90%. This shows the effectiveness of the k-concise DNF formulation.



(a) Precision@K comparison



(b) NDCG comparison

DNF-S	0.88
KW	0.52
RET	0.78
LR	0.79
DNF-C	0.80

(c) Recall comparison

Figure 8: Ranking quality evaluations

DNF-C computes complex DNF with full path information that are often less understandable for humans. Overall its quality is comparable to DNF-S, except that in a few cases the DNFs generated are too specific to "generalize" well.

The quality of RET is considerably lower than DNF-S, because it does not use program internal path information, and will miss certain relevant functions (e.g., function in Listing 1). The quality gap is thus more pronounced as k grows.

The quality of LR is reasonable (since it uses identical features as DNF-S), but still inferior to DNF-S. We believe this is because DNF-based methods use problem-specific properties (e.g, union of conjunctions of literals is suitable to describe program executions) to quickly converge to good representations with limited examples. In comparison, while ML models are highly generic and expressive, the generality comes at a cost of requiring more training examples, which likely leads to the performance gap. Furthermore, the explainability and understand-ability of DNF-based approaches over ML models (e.g., fractional weights over a large number of features) is also an important consideration.

KW uses keyword-matches only and is substantially worse. This shows the importance of internal program states over keywords.

8.2.2 Coverage analysis. Overall, AutoType is able to find functions for 84 out of 112 types tested. We manually inspect and label

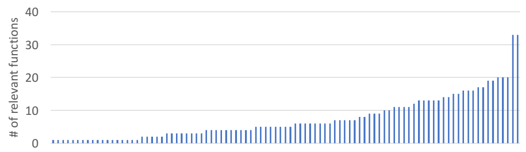


Figure 9: Distribution of correct functions returned

up to 33 functions returned for each type. The distribution of relevant functions is shown in Figure 9. We find 7.4 functions for each type on average.

After inspecting functions discovered, we find that most are not initially written for data validation. Instead, many of them are written to parse strings into program-internal representations, or to convert values of one data type to another, etc.

The large number of functions returned for each type shows the creativity of developers and the variety ways in which code can be re-purposed for validation. For instance, to validate `IPv4`, some synthesized functions use complex regex, some use the `inet` service from UNIX, and others invoke the `WHOIS` web service (which not only validates addresses but also obtains registration information such as IP owners).

Another example is `person-name`. Although this type cannot be precisely defined and validated programmatically (it is difficult to exhaustively compile all first/last names), AutoType finds interesting functions that can be re-purposed for type-detection, including ones that look up names using facebook/instagram to retrieve profiles; and ones that predict genders based on names . .

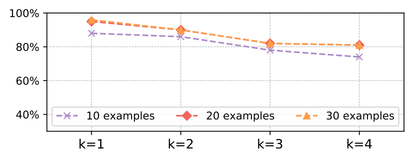
Note that in many cases, it is hard to program type-detection code from scratch. For example, `chemical-formulas` follow complex rules, `addresses` require lookup of reference data for validation, etc. AutoType provides an easy alternative to search and reuse.

For the 28 types that AutoType could not synthesize type-detection logic, we find two main reasons why they are not covered. First, for 24 relatively niche data types (e.g., Library of Congress Classification), we could not find relevant code in Python2 after extensive manual search. It is possible that such Python code may not exist on GitHub. We find that for 12 types (e.g., `National Provider Identifier, ISWC`), validation functions exist but are written in other languages (e.g., Python3 or Java).

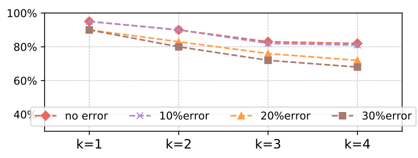
Second, for the remaining 4 types (`SQL query, TAF, ISNI, CIR`), relevant functions exist but AutoType is not able to use them, because certain complicated invocations (e.g., `a = foo1(); b = foo2(a); c = foo3(b, s)` where `s` is the example string) are not currently handled by AutoType.

8.2.3 Sensitivity to seed examples. We first evaluate how the number of positive examples affects quality. We choose 20 popular types (listed in Appendix I) and evaluate their precision-at-K with 10 to 30 examples, shown in Figure 10(a). We can see that there is not much quality gain from 20 to 30 examples, while using 10 examples results in a small drop of quality (8%). This shows that although more examples are generally better, AutoType also does not require a large number of examples.

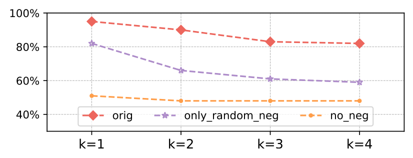
We then intentionally add "noise" (i.e., incorrect positive examples) into the input. On the same 20 types, we increase the ratio of incorrect examples from 0% to 30%. Figure 10(b) shows 10% of errors has virtually no effect on result quality, underlining the robustness



(a) Varying the number of provided positive examples.



(b) Adding noise into positive examples.



(c) Not using/using only random negative examples.

Figure 10: Sensitivity to user-provided positive examples and systematically-generated negative examples.

of the AutoType formulation. Injecting additional noise does affect result quality as expected.

8.2.4 Effectiveness of negative example generation. We test the effectiveness of our hierarchical generation of negative examples by comparing to 1) using random strings as negative examples, 2) using no negative examples at all (where functions are ranked by how many positive examples share the same path). Figure 10(c) shows that our strategy significantly outperforms both baselines.

Additional experiments, such as sensitivity to different input search keywords, can be found in Appendix J.

8.3 Comparison with related systems

To the best of our knowledge, there are no existing systems designed directly for type detection, so we discuss a comparison with two related data transformation systems that use the Program-by-Example (PBE) paradigm.

Transform-Data-by-Example (TDE)[8] [24]. TDE is intended to automate data transformation by reusing relevant code to synthesize new programs consistent with given input-output examples. While not intended for type detection, we force a comparison using positive/negative examples as input, and True/False as output. Note that a small output domain like this is known to be problematic for PBE systems. As expected, out of 20 popular types tested, TDE is able to find functions for 4 types, presumably because the output alone is not informative enough to guide the search. After changing output to related transformations (e.g., a different `date-time` format), TDE finds relevant code (though not precisely for type detection) for half of the cases.

DataXFormer [33]. DataXFormer is also a PBE system that uses search engines to find relevant web tables and services. Since the online system is no longer available[9], we perform the following manual analysis to estimate its coverage upper bound. We search 112 types on Google (using queries like "*type-name* parser" "*type-name* validation"), out of which 76 types have related web services. Since not all web services can be automatically found, parsed, and invoked by DataXFormer this is its recall upper bound.

We inspect web services returned for a sample of 20 types, by observing how they behave when valid vs. invalid data are entered. In 65% cases human readable messages like "invalid", "not correct", "not valid" etc. are returned, suggesting that the system may be able to automatically leverage such services when corresponding keywords are used as output-examples. For remaining services there are no clear indication of data validity.

Overall, we find adapting PBE systems to type detection non-trivial, and the black-boxes (e.g., services) returned are not ideal for understanding and reuse. AutoType on the other hand, produces white-box code snippets for developers to inspect at high precision and recall.

9 TYPE-DETECTION IN TABLES

Additionally, we apply synthesized type-detection logic on a large set of table columns to evaluate its effectiveness.

9.1 Experiment setup

Data set. We use a large corpus of web tables extracted from Bing index [16], and randomly sample 60K table columns for the experiment of column type detection. We again pick 20 popular data types as the target for detection.

Methods compared.

- **DNF-concise (DNF-S):** We use the top-1 function synthesized by AutoType. A column is predicted to be of type **T** if over 80% of its values are accepted by the type-detection function (to account for dirty values such as meta-data mixed in columns).
- **Keyword match (KW):** We choose a number of search keywords for each type (e.g., "url" and "website" for type `url`). Columns with these search keywords as headers are returned as results.
- **Regular expression validation (REGEX):** We also test regular expressions that can sometimes be used to detect data types. Specifically, we automatically generate regex from positive examples **P** used by AutoType, using techniques described in Potter's Wheel [35]. Like in DNF-S, if over 80% of values in a column C is consistent with the regex of type **T**, we predict C to be of type **T**.

Evaluation metric. We manually verify columns detected as of type **T** from each method. For many columns, we are confident about their types by looking at values: e.g., "`459 Euclid Rd, Utica NY`" is an `address`, "`(502) 107-2133`" is a `phone-number`, etc. Certain types require algorithmic verification (e.g., `ISBN, credit card`) and are verified using ground-truth algorithms. Remaining cases are verified based on column header.

After verifying correctness of columns returned, we compute the precision of each type-detection method. Since it is hard to manually inspect all 60K columns to compute recall, for each type we take the union of correct columns returned by all three methods as the ground truth to compute a relative recall.

9.2 Type-detection quality

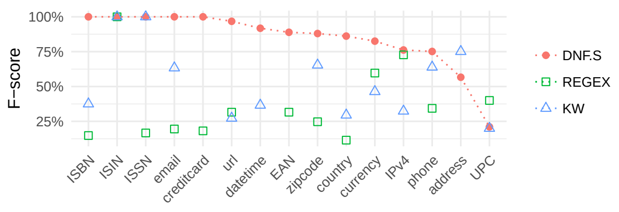


Figure 11: F-score on column-type detection

[8]https://www.microsoft.com/en-us/research/project/transform-data-by-example/
[9]http://www.dataxformer.org, last tested in 2018-03.

	datetime	address	country	phone	currency	email	zipcode	IPv4	url	ISBN	UPC	EAN	ISIN	ISSN	creditcard
DNF-S	**2958** (**0.88**)	157 (**0.80**)	**132** (**0.87**)	**68** (0.69)	**26** (**1.00**)	**37** (**1.00**)	**22** (**0.81**)	8 (**0.80**)	**15** (**1.00**)	**12** (**1.00**)	**2** (0.12)	**4** (**0.80**)	**1** (**1.00**)	**1** (**1.00**)	**1** (**1.00**)
KW	779 (0.65)	**333** (0.63)	38 (0.37)	54 (0.62)	19 (0.42)	19 (0.83)	17 (0.59)	**11** (0.19)	3 (0.50)	3 (0.75)	1 (0.14)	0 (-)	**1** (**1.00**)	**1** (**1.00**)	0 (-)
REGEX	0 (-)	0 (-)	46 (0.07)	17 (**1.00**)	17 (0.85)	4 (**1.00**)	**22** (0.14)	8 (0.73)	3 (**1.00**)	2 (0.13)	1 (**0.50**)	3 (0.20)	**1** (**1.00**)	**1** (0.09)	**1** (0.10)
Union-all	3069	358	155	82	37	37	23	11	16	12	3	4	1	1	1

Table 2: Recall of each method, reported as the number of true-positive columns, and corresponding precision in parenthesis.

Valid columns are found for 15 types out of the 20 tested. Figure 11 shows the F-score of three methods, and Table 2 presents the detailed result for each type. DNF-S has the best F-score in 13 cases. It detects types for 3444 columns in total, with an average precision of 85%.

Keyword-only match based on column headers results in many false-positives. In addition, column headers are often missing or non-descriptive, leading to many false-negatives.

REGEX is not accurate for complex types requiring algorithmic verification (e.g., `ISBN`, `ISSN`) or reference lookup (e.g., `zipcode` and `country`). It also fails to generate a regex from examples containing mixed format (mixed `date time` and `address`) due to data heterogeneity. In addition, regex often fail to generalize when the input data cover a subset of possible examples in a target type. For example, input examples provided for `ISBN` only contain digits (e.g., "`9784063641561`"), while real `ISBN` data can also be formatted as "`978-4-06-364156-1`". Regex trained from given input is not able to recognize `ISBN` variations, while the function returned by AutoType can handle input in a more robust manner (e.g., removing "`-`") before computing checksum.

Now we discuss results of DNF-S. Among all false-positive columns detected, 57% can be attributable to certain implicit assumptions in the code we use. For example, the `UPC` validation function computes checksum without verifying data length, thus returning `ISBN` columns as the two share the same checksum algorithm. Such code can be easily modified and enhanced with minimum efforts. The remaining 43% cases is due to data ambiguity. Such columns are able to pass a perfectly-written type-detection function, but still not of the target type for column headers indicate otherwise. For example, a column whose header is "`version number`", with data like "`7.74.0.0`" is detected as `IPv4`; or a column with header "`temperature range`" and values like "`4-11`" is detected as `date`, etc.

Reasons behind false-negatives detection are more complicated. Firstly, 12% columns have composite values with sub-strings that can be detected as a target type. For example, "`524 Lake, Salem, OR, (843) 389-9216`" is composed of both `address` and `phone number`, "`ISBN 9784063641677`" contains a sub-string that is `ISBN`, etc. Most functions are only able to process clean input and not extraneous information. Techniques like [18] can be employed as a pre-processing step to segment input values in columns before invoking type-detection logic. Other reasons include code quality (17%), and data quality in columns (16%). The remaining cases are in `mailing address`, where the top-1 validation code returned by AutoType and employed here uses a address-parsing service that would not handle partial addresses like "`100 Main Street`".

10 RELATED WORK

Despite its wide applicability, type detection is largely treated as a manual and ad-hoc process (e.g., with regex-like rules), and not systematically studied to the best of our knowledge.

Recent Program-by-Example (PBE) systems, such as Transform-Data-by-Example (TDE) [24] and DataXFormer [33], also leverage existing source code and web services. Conceptually they can be used to find relevant code, by using positive/negative examples as input-examples of PBE, and "true/false" or "valid/invalid" as output-examples of PBE. Our experiment suggests that PBE systems are not best suited for synthesizing type-detection logic, because the output domain is small (binary) and non-descriptive, which is known to be difficult for PBE.

There is a long and fruitful line of research on search source code using natural language queries (e.g., [28, 38]), which are conceptually similar to the search engines and GitHub search API we used. We find the keyword search paradigm to be ineffectively when used directly for the specific task of identifying relevant type-detection logic, as keywords are often ambiguous as compared to examples.

We note that the general idea of leveraging code analysis and execution paths has been applied in other contexts in the programming language community. For instance, in statistical debugging and fault localization [17, 43], variable values and code paths are utilized to find predictors that can differentiate between successful runs and faulty runs (e.g., those that cause crashes and exceptions). Predictors can then be given to developers to localize bugs.

Another related area of research in programming language is test case generation for test coverage [14]. The idea is to use the set of branches that need to be taken to reach a particular code region, to induce constraints on input values that need to be satisfied, which can then be solved to reverse-engineer desired input data to generate coverage tests.

11 CONCLUSIONS AND FUTURE WORK

In this work we take the initial step towards synthesizing type-detection functions for rich semantic data types. There are a few promising directions for future work. One direction of interest is to better explain top-returned functions such that users can verify more easily. Another dimension is to explore rich applications that semantic types can enable, in the context of information extraction and table search for example. We hope that our research will serve as a springboard for future work, improving and employing our techniques for a rich variety of applications.

REFERENCES

[1] Btrace - a tracing tool for java. https://github.com/btraceio/btrace.
[2] Codeplex. https://www.codeplex.com/.
[3] Enterprise version of the proprietary github. https://enterprise.github.com/home.
[4] GitHubGist. https://gist.github.com/.
[5] GS1 check digit. https://www.gs1.org/how-calculate-check-digit-manually.
[6] Javascript debugger statement. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/debugger.
[7] Luhn algorithm. https://en.wikipedia.org/wiki/Luhn_algorithm.
[8] Python built-in ast parser. https://docs.python.org/2/library/ast.html.
[9] Ruby debugging. http://guides.rubyonrails.org/debugging_rails_applications.html.
[10] Trifacta built-in data types. https://docs.trifacta.com/display/PE/Supported+Data+Types.
[11] A. Arulselvan. A note on the set union knapsack problem. *Discrete Applied Mathematics*, 169:214–218, 2014.
[12] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. *The semantic web*, pages 722–735, 2007.
[13] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[15] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *Proceedings of VLDB*, (1), 2008.

[16] K. Chakrabarti, S. Chaudhuri, Z. Chen, K. Ganjam, Y. He, and W. Redmond. Data services leveraging bing's data assets. *IEEE Data Eng. Bull.*, 39(3):15–28, 2016.

[17] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 34–44. IEEE, 2009.

[18] X. Chu, Y. He, K. Chakrabarti, and K. Ganjam. Tegra: Table extraction by global record alignment. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1713–1728. ACM, 2015.

[19] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of SIGMOD*, 2015.

[20] E. Cortez, P. A. Bernstein, Y. He, and L. Novik. Annotating database schemas to help enterprise search. *Proceedings of VLDB*, (12), 2015.

[21] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. imap: discovering complex semantic matches between database schemas. In *Proceedings of SIGMOD*, 2004.

[22] M. Hazewinkel, editor. *Encyclopedia of Mathematics*. Springer Science+Business Media B.V. / Kluwer Academic Publishers, 2001.

[23] Y. He, K. Chakrabarti, T. Cheng, and T. Tylenda. Automatic discovery of attribute synonyms using query logs and table corpora. In *Proceedings of WWW*. International World Wide Web Conferences Steering Committee, 2016.

[24] Y. He, K. Ganjam, K. Lee, Y. Wang, V. Narasayya, S. Chaudhuri, X. Chu, and Y. Zheng. Transform-Data-by-Example (TDE): Extensible Data Transformation in Excel. In *SIGMOD*, 2018.

[25] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[26] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *Proceedings of PODS*, 2005.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, 2005.

[28] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 260–270. IEEE, 2015.

[29] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[30] C. D. Manning, P. Raghavan, and H. Schütze. Scoring, term weighting and the vector space model. *Introduction to information retrieval*, 100:2–4, 2008.

[31] J. L. McCarthy. Metadata management for large statistical databases. In *Proceedings of VLDB*, 1982.

[32] T. M. Mitchell. Machine learning., 1997.

[33] J. Morcos, Z. Abedjan, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: An interactive data transformation tool. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 883–888, 2015.

[34] F. Naumann, C.-T. Ho, X. Tian, L. M. Haas, and N. Megiddo. Attribute classification using feature analysis. In *Proceedings of ICDE*, 2002.

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

[36] D. Ritze, O. Lehmberg, and C. Bizer. Matching html tables to dbpedia. In *Proceedings of WIMS*, page 10. ACM, 2015.

[37] A. Sen. Metadata management: past, present and future. *Decision Support Systems*, (1), 2004.

[38] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz. Sando: an extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 15. ACM, 2012.

[39] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 2005.

[40] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *Proceedings of WWW*, 2007.

[41] A. Talaika, J. Biega, A. Amarilli, and F. M. Suchanek. Ibex: harvesting entities from the web using unique identifiers. In *Proceedings of WebDB*, 2015.

[42] Y. Wang and Y. He. Synthesizing mapping relationships using table corpus. In *Proceedings of SIGMOD*, 2017.

[43] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112. ACM, 2006.

APPENDIX

A FULL LIST OF DATA TYPES TESTED

We roughly cluster semantic types tested based on their domains. Type names below in italic are the ones for which AutoType cannot find relevant functions for type-detection.

Science (Biology, chemistry, etc): Simplified Molecular-Input Line-Entry (SMILE), International Chemical Identifier (InChI), CAS registration number, FASTA sequence, FASTQ gene sequence, chemical formula, Uniprot, Ensembl gene ID, *Life Science Identifier (LSID), IUPAC number, EVMPD, Anatomical Therapeutic Chemical, SNPID number, International Code of Zoological Nomenclature*

Health: drug name, DEA number, ICD9, ICD10, HL7 message, *HCPCS code, FDA drug code, Active Ingredient Group number*

Financial & commerce: Stock Exchange Daily Official List (SEDOL), UPC barcode, CUSIP number, Stock ticker, ABA routing number, EAN barcode, ASIN book number, IBAN number, bitcoin address, EDIFACT message, FIX message, GTIN number, credit card number, currency, SWIFT message, *NATO stock number*

Technology & communication: IPv4 address, IPv6 address, URL, IMEI number, MAC address, MD5, MSISDN, Notice To Airmen, *AIS message, NMEA 0183 International Standard Text Code (ISTC)*

Transportation: Vehicle Identification Number (VIN), UIC wagon number, International Maritime Organization number (IMO)

Geo location: Long/lat, US zipcode, UK postal code, Canada postal code, MGRS coordinate, Global Location Number (GLN), UTM, airport code, us state abbreviation, country code, *geojson, TAF message, International Geo Sample Number (IGSN)*

Publication: ISBN, ISIN, ISSN, Bibcode, ISAN, ISWC, DOI, ISRC, ISMN, ORCID, ONIX publishing protocol, *Library of Congress Classification (LCC), ISO 690 citation, APA citation, National Bibliography Number (NBN), Electronic Textbook Track Number (ETTN)*

Personal Information: phone number, email address, person name, mailing address, Legal Entity Identifier (LEI), US Social Security Number (SSN), Chinese Resident ID, Employer Identification Number (EIN), NHS number, PubChem ID, *Personal Identifiable Information (PII), National Provider Identifier (NPI), FEI identifier*

Other: book name, HEX color format, RGB color format, CMYK color format, HSL color format, UNIX time, http status code, roman number, HTML, JSON, XML, date time, *SQL statement, Reuters instrument code, OID number, Global Unique Identifier, International Standard Name Identifier (ISNI)*

B SEMANTIC TRANSFORMATIONS

As mentioned in Section 7.1, AutoType collects intermediate variables when running relevant functions to search for candidate transformations. Specifically, AutoType dumps out all atomic variables (e.g., `int, float, string`), and decomposes composite ones (e.g., class objects, list, dict) to dump their constituent atomic values.

Across a set of relevant functions returned by AutoType, such an approach often produces a rich variety of possible transformations for **T**; AutoType additionally filter variables that are of low entropy (e.g., producing the same value across **P**) when necessary.

We use AutoType to retrieve data transformations for 20 popular types. We manually check these transformations produced from top 10 functions for each type, shown in Table 3. There are 7 meaningful transformations for each data type on average.

type:	transformations	type:	transformations
email:	domain, IPv4,...	us zipcode:	long/lat, city, state, country,...
url:	scheme, domain, host,...	VIN:	manufacturer, serial, year, region,...
phone:	country	date time:	year, month, day, hour,...
DOI:	bookname	person name:	first/last name, facebook profile,...
ISBN:	DOI, EAN, language,...	MAC address:	IPv6, EUI-64-int format,...
IPv4:	country/city, DNS zone,...	address:	country, city, long/lat, zipcode,...
IPv6:	start ip, end ip,...	credit card:	card brand, card length
IBAN:	country, countrycode	stock ticker:	company, latest/highest price,...
country:	full name, code,...	chemical formula:	average mass, atoms,...
UPC:	product, manufacturer	hexcolor:	rgb, cmyk, colorname,...

Table 3: Examples of semantic data transformations.

As can be seen from Table 3, many interesting transformations exist. For example, the vehicle type, manufacturer and serial number information can be extracted from VIN numbers; molecular mass can be computed from chemical formula; country name from phone numbers, etc. Interestingly, there also exist transformations that convert one data type into other types, for instance, from UPC to EAN code, HSL color format to RGB format, UTM coordinate to longitude and latitude, etc., all of which require non-trivial transformation logic that are hard to program from scratch. This again demonstrates the advantage of the AutoType approach that reuses existing code.

C MORE APPLICATIONS OF DATA TYPES

Data type is a key type of of metadata. In addition to the scenarios discussed in the introduction, it has been used as an important ingredient in many other problems, including schema mapping [34], outlier detection [10], and information extraction [41]. We will review some additional applications below.

Schema mapping. Authors in [21, 34] observe that table column types (e.g., date-time) can be used as an important signal to produce accurate schema mappings, for columns of compatible types are more likely to be matches. Since type detection is not a focus of these schema mapping work, manually-defined regex are typically used to detect a small number of data types.

Outlier Detection. Given a relational table, it is intuitive that if most values in a column (e.g., 95% of them) can be recognized as values of some data type, it is reasonable to believe that the remaining values are potential errors/outliers. This is exactly the approach taken by commercial systems such as Trifacta [10]. However, as we discuss in the Appendix, these commercial systems typically uses predefined regex-like patterns to detect data types, which can often lead to false-positives (e.g., \d{16} for credit cards).

Information Extraction. While most applications of type detection have been in the context of tabular relational tables, type detection logic also applies to unstructured text data for the purpose of information extraction (IE). For instance, authors in [41] develop hand-crafted logic to extract unique identifiers such as ISBN, GTIN and DOI from web documents. Because these identifiers require strict (and different) checksum/validation algorithms, the detection is likely to be high precision. The AutoType system proposed in this work can be easily used for IE scenarios, and in fact subsumes the types recognized in [41]. Using automatically-discovered type detection logic provides an interesting alternative to complement the mainstream text-pattern based extraction, which we believe can greatly extend the reach of existing IE systems.

D CODE ANALYSIS

D.1 Invoking functions

Beside various ways of invoking existing functions, AutoType also tries to run code snippets that do not live in functions. Since each python file can be executed standalone as a script, once AutoType detects python code outside of functions in a file, it runs the file directly. Before running the file, AutoType also tries to feed input example by replacing system argument or user input, as described in Section 4.2. We add an additional feeding-input method for such script files based on the observation that data processed by these files are often hard-coded, as the example in Listing 3 shows. Each assignment of a constant value (e.g., line 1 in Listing 3) can be replaced with an input parameter as one new "function". If a modified file runs successfully without exception, it will be saved as a candidate function.

```
1 card_number='4111111111111111'
2 for c in card_number:
3   ... #code to validate a card number
```

Listing 3: code snippet with hard-coded input string

This approach helps AutoType find many useful functions, especially for code from GitHub Gist which hosts many small snippets.

AutoType also considers multi-parameter functions if the input string can be split, for instance, `long/lat` which can be split into two parts, `rgb color` into three, etc. For these types, AutoType tries functions that take the same number of argument as the number input components.

D.2 Byte-Code Injection

AutoType instruments python bytecode to obtain the status of every branch and return. To do so, AutoType first reads bytecode and finds out all branch and return instructions (e.g., `POP_JUMP_IF_TRUE`, `RETURN_VALUE`, etc). Immediately before every such instruction, AutoType adds bytecode that saves stack status (as python interpretor runs instructions on a stack), dumps the stack top, and then reumes the stack status. Since these actions are performed just before the branch or return, the stack top is the value that a branch depends on, or the return value. Besides stack top, the bytecode AutoType injects also dumps the filename and line number of the corresponding branch/return (used as identifier). When running on a concrete input, the instrumented code dumps every branch/return values along its execution, which AutoType will process afterwards. Since all bytecode in the repository are instrumented, when a function calls another in the same repository, the tracing is naturally inter-procedure. Note that the library functions are not instrumented due to the lack of source code, and AutoType only analyzes branches and returns from the repository where the source code is available.

D.3 Other engineering challenges

There are additional engineering challenges in implementing AutoType, some of which we discuss below.

Storing repositories: Some repositories on GitHub are rather large, which are expensive to clone and analyze. We consider only repositories smaller than 2G and with less than 500 python files to make end-to-end latency acceptable.

Long running functions: Some open-source functions can takes a long time to execute (due to bad design, unexpected input, etc.). AutoType uses a separate thread to monitor each function run and terminates a function when it takes over 30 seconds.

System-level sand-boxing: Some open-source code perform system-level modifications (e.g., create/delete files, issue system calls, change system settings, etc). Restricting function execution in sandboxes is an important consideration to restrict unexpected or sometimes malicious system modifications.

E SECURITY, PRIVACY AND LICENSES

There are a few practical considerations in using open-source software (OSS) code. Licensing terms is often a big constraint, for code with restrictive licenses such as GPL can prevent it from being used in commercial systems. While some GitHub projects describe license terms in standard manners (e.g. in "License.txt"), many do not have such information and would require manual inspections.

Code security and data privacy is clearly also important. While we can perform limited sand-boxing for isolation at execution, ultimately rigorous security reviews are necessary (both manual and automatic ones are crucial).

F PROOF OF THEOREM 1

Proof. We prove the hardness of this problem using a reduction from set-union-knapsack (SUKP) [11]. Recall that in SUKP, we have a universe of elements $\mathcal{U} = \{e_1, e_2, \ldots, e_n\}$. Given a set of items $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, with $S_i \subseteq \mathcal{U}$. Each item S_i has a non-negative profit $P(S_i)$, and each item e_i has a cost $C(e_i)$. The problem is to select a subset of items $T \subseteq \mathcal{S}$ to maximize the total profit $P(T) = \sum_{S_i \in T} P(S_i)$, subject to the constraint that the cost calculated based on the union of elements is no greater than some fixed budget B, or $C(T) = \sum_{e \in \cup_{S_i \in T}(S_i)} C(e) \leq B$. SUKP is known to be NP-hard and inapproximable [11].

We show a reduction from SUKP. For any given problem in SUKP, we construct a problem in *Best-k-Concise-Cover* with $k = 1$ as follows. For each element e_1, we construct N_i which is $C(e_i)$ number of negative examples with unit weight. For each item S_i, we construct a new single-literal clause $L(S_i)$ that is associated with $P(S_i)$ number of new positive examples and all the negative examples in $\bigcup_{e_j \in S_i} N_j$. Furthermore, set $\theta = \frac{\sum_{e_i \in \mathcal{U}} C(e_i)}{B}$.

This process ensures that we can translate any SUKP problem instance to *Best-1-Concise-Cover* problem. Assume that we could solve *Best-1-Concise-Cover*, then we would be able to solve SUKP known to be NP-hard, creating a contradiction. Also notice that the reduction is approximation-preserving, with the same objective function and constraints. Thus *Best-k-Concise-Cover* is NP-hard and cannot be approximated within a factor of $2^{(\log n)^{\delta}}$, for some δ, unless $3SAT \in DTIME(2^{n^{\frac{3}{4}+\epsilon}})$, as shown in [11]. □

G ALGORITHM FOR SYNTHESIZING VALIDATION FUNCTION

Instead of directly using the concise DNF, we extend it to use more literals to validate future data more precisely: for any literal c_i in DNF, generate DNF-E by replacing c_i with $c_{i_1} \wedge c_{i_2} \wedge \ldots \wedge c_{i_k}$ where $c_i \in G_i$ (the literal group mentioned in Algorithm 1) and

Algorithm 3 Pseudo-code for synthesized bool F'(s)

```
INPUT: data value $s$ to validate
1: DNF-E = DNF
2: for literal $c_i \in$ DNF-E and $c_i \in G_i$ do
3:     replace $c_i$ with $c_{i_1} \wedge c_{i_2} \wedge \ldots$, where $G_i = \{c_{i_1}, c_{i_2}, \ldots\}$
4: run $F(s)$
5: $T(s) \longleftarrow$ collect trace from $F(s)$ and create features
6: if $\wedge T(s) \rightarrow$ DNF-E then return True
7: else return False
```

$G_i = \{c_{i_1}, \ldots, c_{i_k}\}$. Intuitively, a literal in DNF represents a group, which is usually a sub-path that many input positive examples take. Extending this literal to a conjunction of full sub-path restricts a future data to pass the validation only when it takes exactly the same sub-path, instead of hitting any literal on the sub-path.

After getting the extended DNF, DNF-E, AutoType runs a new input s on F, records the trace and featurizes into $T(s)$. Then it checks whether the conjunction of all literals in $T(s)$ is consistent with DNF-E (i.e, $\wedge T(s) \rightarrow$ DNF-E). If yes, AutoType returns True, otherwise False. The pseudo-code is shown in Algorithm 3.

Example 7. Taking the DNF generated in Example 4, and assuming literal groups are as follow: $G_1 = \{b_6 ==\text{True}\}$, $G_2 = \{b_6 ==\text{False}, b_9 ==\text{True}\}$, $G_3 = \{b_{16} ==\text{True}\}$,.... AutoType generates DNF-E as $(b_6 == \text{True} \wedge b_{16} == \text{True}) \vee ((b_6 == \text{False} \wedge b_9 == \text{True}) \wedge b_{16} == \text{True})$.

When a new valid card number d comes in, assuming it is a Visa card, $T(d) = T(e_1^+)$. Apparently $\wedge T(d) \rightarrow$ DNF-E, and the validation function returns True. For a non-valid card number d', the function either throws exception, has b_{16} untaken (fails checksum validation), or has both b_6 and b_9 untaken (without valid card brand), so $T(d') \nrightarrow$ DNF-E and the validation will return False.

H PROOF OF PROPOSITION 1

Proof. For a given string s, the fact that $S1(s) \subseteq S2(s) \subseteq S3(s)$ holds follows from the definition of the three strategies. Specifically, we note that strategy S1 can replace only *non-punctuation characters*, whereas S2 replaces *any character*, which is a super-set. Furthermore note that the S1 uses *in-alphabet non-punctuation character* to replace selected characters whereas S2 uses *in-alphabet characters*, which is again a super-set. This ensures that the space of possible mutations $S2(s)$ for any s is a super-set of $S1(s)$.

Similarly, we can show that $S3(s)$ is a super-set of $S1(s)$, because S3 replaces selected characters with *random character*, which is a super-set of *in-alphabet characters* used by S2.

These together prove that $S1(s) \subseteq S2(s) \subseteq S3(s)$. □

I EXPERIMENT DETAILS

Three different keywords used for sensitivity analysis. Table 4 shows the alternative keywords we used in Figure 12.

Popular types (20) used in sensitivity and type detection experiments: datetime, address, country code, phone number, currency, email address, zipcode, IPv4 address, url address, ISBN, UPC barcode, EAN code, ISIN, ISSN, creditcard, IPv6 address, IBAN number, VIN number, stock symbol, airport code.

type	keyword1	keyword2	keyword3
ISBN	ISBN	international standard book number	ISBN13
IPv4	IPv4	IPv4 address	ip address v4
SWIFT	SWIFT message	Society for Worldwide Interbank Financial Telecommunication	SWIFT
zipcode	US zipcode	zipcode	US postal code
SEDOL	SEDOL	stock exchange daily official list	SEDOL number
ISIN	ISIN	ISIN number	international securities identification number
VIN	VIN	Vehicle Identification Number	VIN number
RGB	RGB color	RGB	RGB color code
FASTA	FASTA sequence	FASTA gene sequence	FASTA
DOI	DOI identifier	digital number identifier	DOI number

Table 4: Alternative input keywords tested

These types are used in Section 8.2.3 and 8.2.4 (for sensitivity analysis), Section 8.3 (for comparison with TDE and DataXFormer), and Section 9 (for column type detection).

J SENSITIVITY TO INPUT KEYWORDS

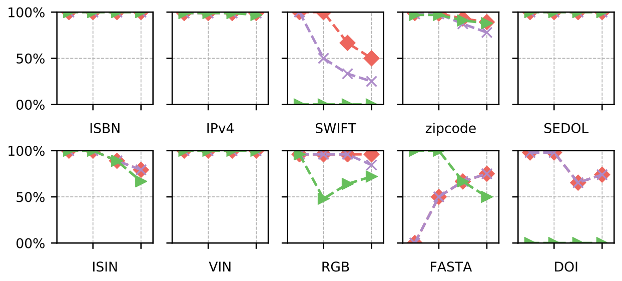


Figure 12: Evaluation with different keyword for 10 types. Evaluated with precision@K (where K = $\{1, 2, 3, 4\}$).

In order to understand the sensitivity of AutoType to different input keywords, we sample 10 data types for which we can come up with at least 3 different keywords to describe them (skipping data types like "credit card" with only one or two reasonable type names). All input keywords can be found in Table 4 in the Appendix I.

Figure 12 shows the resulting quality. For 6 out of 10 data types tested, quality results are insensitive to input keywords, presumably because these alternative ways of describing data types are well-established and widely-used in source code and comments (e.g., "ISBN" and "international standard book number" and "ISBN13"). In three remaining cases, result quality degrade substantially for one input keyword; and in one case quality degradation happens for two input keywords.

We find two main reasons for the quality degradation: (1) Ambiguity. For example "SWIFT" is a data type used in the financial industry for message exchanges, but there is a popular programming language by the same name. As such, a search of "SWIFT" via GitHub or Bing search API retrieves results dominated by code in the swift language, leading to quality degradation. A search of "SWIFT message" disambiguates the intent and performs much better. (2) Non-standard keyword description. Some keywords we come up with are not so standard. For example, for "DOI" we tested "digital object identifier", "DOI identifier", and "DOI number". It appears that "DOI identifier" is the more standard name (I in the name stands for identifier), while "DOI number" is not, and that in turn leads to sub-optimal search results.

This experiments suggest that our approach can indeed be sensitive to input keywords. However this is not entirely surprising, for even commercial search engines can balk at "bad" queries (e.g. "SWIFT") and would require users to iteratively refine their queries. Suggest alternative keywords automatically (leveraging synonym systems such as [23]) is a good direction for future work.

K SENSITIVITY ANALYSIS OF LR

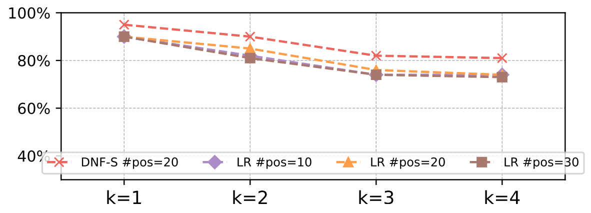


Figure 13: Varying the #positive examples under LR method

We also compared DNF-S with LR as described in Section 8.1 with different number of positive examples, varied from 10 to 30. Figure 13 shows the result. With a small number of examples, the LR method shows very little changes (less than 4% variance), and underperforms DNF-S under all settings.

Note that in this LR based ranking method, we did not add any regularization terms (e.g., limiting the number of features it uses). We would expect the LR method works better with regularization, since overfitting might be a big issue with a small number of examples. However, it is hard to find a good regularization that works well for all functions, where the number of features varies greatly with different functions. Since we do not require large amount of examples provided by users, we believe using DNF is a better approach than LR.

L EFFICIENCY ANALYSIS

Since AutoType intends to help technical users to search and reuse type-detection code, which is largely an offline process, optimizing efficiency is not prioritized when we build AutoType. As mentioned in Appendix D.3, AutoType searches up to 40 repositories from two search engines and runs each function up to 30 seconds. Besides, AutoType terminates after running 60 minutes for each type.

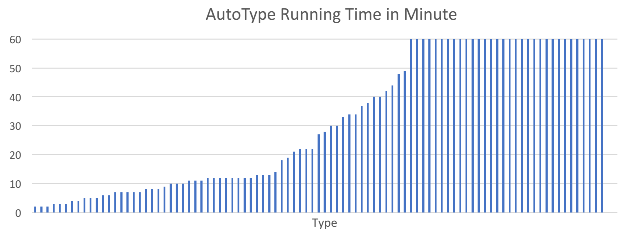


Figure 14: Distribution of running time.

Figure 14 shows the distribution of execution time. For 25 such types the process finishes relatively fast (within 10 minutes), while for 32 types it takes over 60-minutes. The longer-running ones often correspond to popular types like zipcode and ISBN, which often have a large amount of relevant code repositories that AutoType needs to churn through.