PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

IDENTIFIABILITY OF JSON SCHEMA FROM POSITIVE EXAMPLES

JAIME ESTEBAN CASTRO RETAMAL

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
JUAN L. REUTTER

Santiago de Chile, November 2018

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

IDENTIFIABILITY OF JSON SCHEMA FROM POSITIVE EXAMPLES

JAIME ESTEBAN CASTRO RETAMAL

Members of the Committee:
JUAN L. REUTTER
DOMAGOJ VRGOČ
PABLO BARCELÓ
EDUARDO AGOSÍN

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, November 2018

Dedicated to those who are going
through difficult days

ACKNOWLEDGEMENTS

I have a lot of reasons to express my gratitude to Juan Reutter, my advisor for these two years and a half. He has been a very supportive person, in both personal and academic scopes. He has not only guided me on this research with dedication, but he has been an example of goodwill, good intentions, patience, and responsible work. I am especially indebted to him for trusting me, even on certain occasions when I did not believe in myself so much. I am sure that without Juan I would have dropped out of the master's program. At last, Juan also allowed me to do an internship at Google and giving lectures at the university, which were invaluable for my professional development.

I want to thank Domagoj Vrgoč and Pablo Barceló for accepting to be part of the committee and for their willingness. I also want to thank Eduardo Agosín, for his kindness and for making this last part of the process simple.

Special mention to all my friends at DCC, for the good times in all these years, and for being a vital part of my personal growth. I also was surprised to see how many of you noticed that I was almost finishing my degree.

Of course, I give my thanks to my mother and father, for taking care of me for all these years, for letting me pursue my studies with nearly no worries, and for allowing me to persevere on what I think is the best for me.

Finally, I would like to manifest my gratitude to Natalia Hanckes, who helped me to understand myself, and helped me to bring tranquillity and equilibrium to my life when anxiety was hitting me very hard. Without her help, I could not have performed well on my daily activities, including, finishing this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

JSON (JavaScript Object Notation) is arguably the most popular data format for API requests and responses. Despite the existence of JSON Schema, a standardized schema specification for JSON format, most APIs documentation relies solely on human-readable examples with non-evident edge cases. In the current situation, developers should understand by themselves if the API can produce a response with null or missing fields. At the same time, programmers cannot take advantage of automatic request/response integrity validation, automatic generation of examples for testing purposes, autocompletion when working with IDEs, among other benefits that an explicit data specification can deliver. The problem becomes evident when we take into account how fragile are implementations with a minimal change in the request/response data.

In this thesis, we explore the theoretical boundaries to learn JSON Schema from examples. We start by choosing a theoretical framework on what we think definitions and results should base on. Next, we use that framework to analyze the learnability of different classes of schemas and identify those with favorable results.

Then, we look at one of the few schema repositories to examine how is JSON Schema used in practice. With theoretical results and real data, we select a set of JSON Schema features that we can learn under the selected framework and we provide an algorithm to perform the before-mentioned task. Finally, we show the results and the presented challenges in the process of applying the learning algorithm over three sets of examples from OpenWeatherMap, GitHub, and Twitter, and then we present a little improvement out of the original framework.

Keywords: JSON, API, JSON schema, Formal languages, Identifiability in the limit, Schema inference.

RESUMEN

JSON (JavaScript Object Notation) es probablemente el formato de datos más usado para hacer *requests* u obtener *responses* desde APIs. A pesar de la existencia de JSON Schema, un estándar para definir la estructura de documentos tipo JSON, la gran mayoría de la documentación de APIs se basa solamente en ejemplos donde no están claros los casos bordes que pueden existir. En esta situación, los desarrolladores deben dilucidar por ellos mismos si es posible recibir valores nulos o si existen datos opcionales. Al mismo tiempo, los programadores no pueden aprovechar los beneficios de tener un esquema de datos explícito, como por ejemplo, la posibilidad de efectuar una validación de datos, la creación de ejemplos automática para realizar *testing*, autocompletado en IDEs, entre otros. El problema se hace más evidente si se toma en cuenta la fragilidad de las implementaciones cuando se hacen cambios pequeños en el formato de las *requests* o *responses*.

En primer lugar, se analizan los límites teóricos de la inferencia de JSON Schema a partir de ejemplos. Para ello se selecciona un marco teórico sobre el cual se crean las definiciones y se presentan los resultados. Luego, en base a este marco teórico, se analiza la factibilidad de aprender distintas clases de JSON Schema y se identifican aquellas que tienen buenos resultados.

Después, se examina uno de los pocos repositorios con esquemas JSON para entender cómo se usa la especificación en la práctica. Con esos antecedentes se propone un conjunto de características que se pueden inferir a partir de ejemplos bajo el marco teórico seleccionado. Además, se entrega un algoritmo para realizar la tarea de aprendizaje de esquemas. Finalmente, se muestran resultados de aplicar el algoritmo sobre tres conjuntos de ejemplos provenientes de OpenWeatherMap, GitHub, y Twitter.

Palabras Claves: JSON, API, JSON schema, Lenguajes formales, Identificación en el límite, Inferencia de esquemas.

1. INTRODUCTION

JSON (JavaScript Object Notation) is a semi-structured document format for exchanging data consisting of attribute-value pairs, arrays, strings, boolean values, and numbers. JSON has become the *lingua franca* for data storage and exchange among software developers.

The most frequent context where JSON plays a crucial role is in modern web and mobile applications. When a user wants to use a service, the user needs to execute a client application that connects to a server and exchanges information with it; with this purpose the provider of the service exposes an API (Application Programming Interface) which permits client applications to communicate with the servers. Although the format of the exchanged data is up to the service provider, JSON documents are a common choice because of its readability for developers and machines (Suárez Barría, 2016).

Developers also use JSON documents as a mean to store data; for instance, MongoDB and CouchDB are two NoSQL databases which support JSON documents and queries over them (Jing, Haihong, Guan, & Jian, 2011). JSON documents are also useful to store user configurations, for example, text editors such as Visual Studio Code and Atom store user configuration in JSON files.

The growing popularity of JSON precipitated the creation of JSON Schema (JSON-schema-org, 2018), a way to specify what a JSON document should contain (Pezoa, Reutter, Suarez, Ugarte, & Vrgoč, 2016). The JSON Schema specification is currently in its seventh draft (Wright & Andrews, 2018) and there is a growing body of applications and tools using JSON Schema.

To observe how JSON and JSON schema work together, look at the JSON file in Figure 1.1 that describes a person. It is reasonable to think that all the fields are mandatory and that all the fields should have the same type as the example. With that in mind, Figure 1.2 shows a possible schema to describe people.

```
{
    "name": "John",
    "surname": "Doe",
    "birthdate": "1990-01-07",
    "siblings": true
}
```

Figure 1.1. Example of a JSON document with person's data.

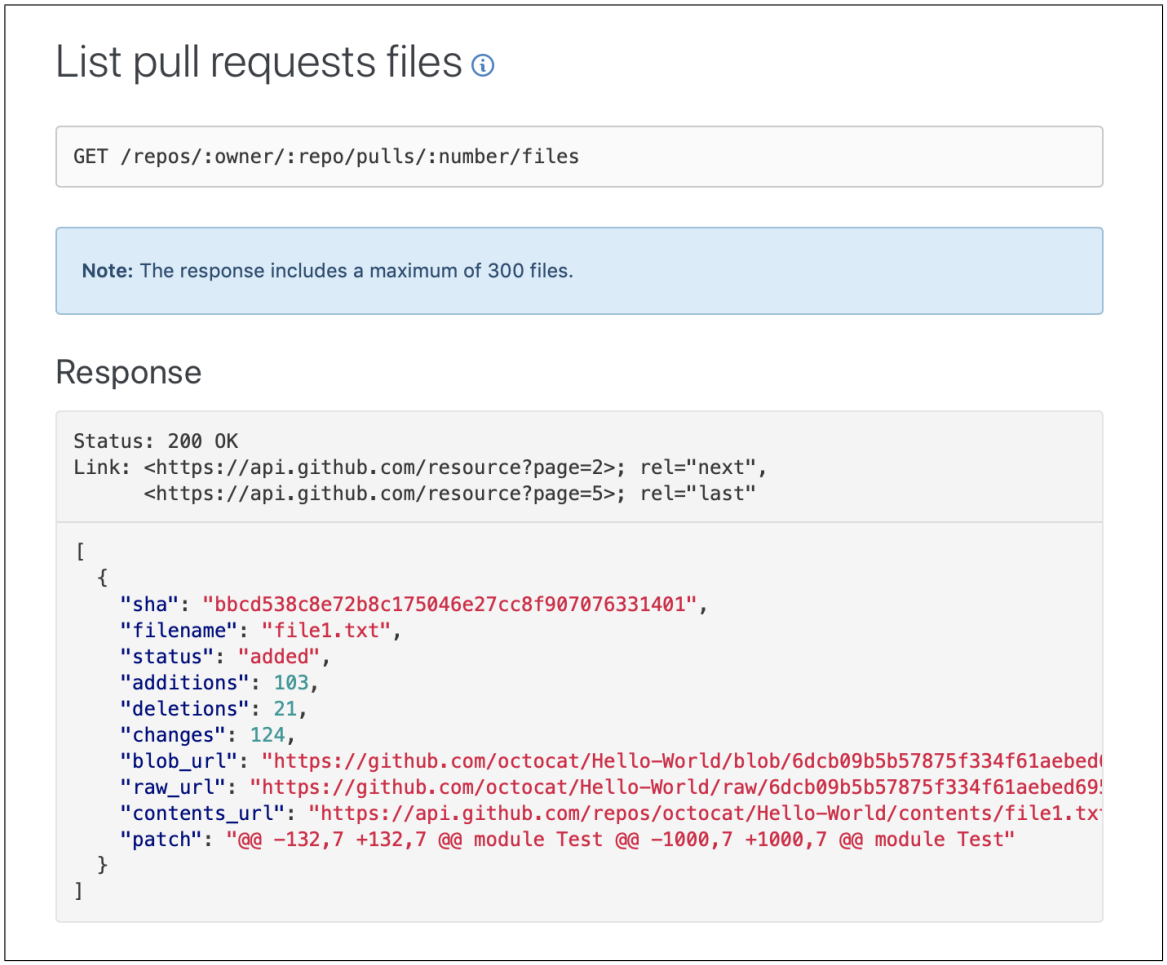
```
{
    "type": "object",
    "properties": {
        "name": {"type": "string"},
        "surname": {"type": "string"},
        "birthdate": {"type": "string"},
        "siblings": {"type": "boolean"}
    },
    "required": ["name", "surname", "birthdate", "siblings"]
}
```

Figure 1.2. Example of a JSON schema for people's data.

One of the most important uses of JSON Schema is in documentation and standarization of APIs. Developers can use JSON Schema to specify the shape of JSON documents that are expected along API requests and responses. JSON Schema is also the backbone of more complete frameworks for standirizing APIs such as Open API Initiative (The Linux Foundation, 2018), an endeavour seeking to build an open documentation of APIs available worldwide.

Despite the extensive use of JSON documents and the existence of JSON Schema as a method to specify validations on the content of the documents, few of the publicly available APIs provide schemas to validate the requests or responses. Indeed, most of their documentation relies solely on examples or, in the best of the cases, on sparse information across the documentation or huge tables. Clearly, this approach has several drawbacks, and such a documentation is bound to leave gray areas and spaces whenever a particular request or response was not covered by a specific example.

Consider for example GitHub, a hosting service for projects using the Git version control system. The only documentation to get the files associated with a *pull request* is the example shown in Figure 1.3. One can argue that the example is simple enough to be understood by any developer and that is why GitHub does not provide a schema; however, some questions can still arise: which are the possible values associated to `"status"`? If the actual pull request has more than 300 files, would we have some way to get all of them? These questions are essential to ensure a smooth connection between this API and a system application, and yet they are typically not answered in documentation with examples. Indeed, to figure out the answers one may require a mix between trial and error, and knowledge on the problem domain.



List pull requests files

```
GET /repos/:owner/:repo/pulls/:number/files
```

Note: The response includes a maximum of 300 files.

Response

```
Status: 200 OK
Link: <https://api.github.com/resource?page=2>; rel="next",
      <https://api.github.com/resource?page=5>; rel="last"
```


```
[
  {
    "sha": "bbcd538c8e72b8c175046e27cc8f907076331401",
    "filename": "file1.txt",
    "status": "added",
    "additions": 103,
    "deletions": 21,
    "changes": 124,
    "blob_url": "https://github.com/octocat/Hello-World/blob/6dcb09b5b57875f334f61aebed
    "raw_url": "https://github.com/octocat/Hello-World/raw/6dcb09b5b57875f334f61aebed69
    "contents_url": "https://api.github.com/repos/octocat/Hello-World/contents/file1.tx
    "patch": "@@ -132,7 +132,7 @@ module Test @@ -1000,7 +1000,7 @@ module Test"
  }
]
```


Figure 1.3. GitHub documentation for listing *pull request* files

The actual response of the endpoint shown in Figure 1.3 includes up to 30 files. To retrieve the other files, the endpoint provides two links in the HTTP header of the response: a link to go to the next portion of the data, and another to go to the last one. The limit of 300 files does apply since the API only generates up to 10 portions of data. There is also another detail that is impossible to infer from the example: the property `"patch"` is optional since it is applicable only for text files.

Even if a human is wise enough to discover the needed answers for the situation shown in Figure 1.3, one can find the same situation for much larger responses, where nesting, nulls, and optional values can appear depending on the requested data. It is worth noting that this situation does not apply only for GitHub but for almost all the public APIs.

Using a more generic example, consider a weather application providing an API to inform about current weather. Upon prompted with a specific location, the API responds with the JSON document in Figure 1.4. The document provides the current temperature, a description of current weather, the amount of rainfall in the last three hours and the coordinates of the location.

```
{
  "temperature": 10,
  "description": "light rain",
  "precipitation_3h_mm": 1.33,
  "coords": [51, 0]
}
```

Figure 1.4. Example for a weather API response

If this is the only example in the documentation of this API, users looking to work with it will have several issues. Suppose that we need to look for places without rain. How can we derive from this API that it is not raining in a location? Should we look for a value close to 0 under `"precipitation_3h_mm"`? But what if this key:value pair is simply not present when the weather is not rainy? We can immediately answer those questions if the API provides a JSON Schema such as the one in the Figure 1.5. There we have the pair

`"type": "object"`, so we only validate JSON objects. The `"required"` field indicates that validated objects must have pairs with keys `"temperature"`, `"description"` and `"coords"`, but `"precipitation_3h_mm"` is optional. Furthermore, the `"properties"` field specifies how the values of these keys should look like: temperature must be a number, description must be a string, precipitation must be a number greater than 0.01 and coords must be an array. We infer that the value `"precipitation_3h_mm"` only shows up when rain is present in the last three hours.

```
{
    "type": "object",
    "properties": {
        "temperature": {"type": "number"},
        "description": {"type": "string"},
        "precipitation_3h_mm": {
            "type": "number",
            "minimum": 0.01
        },
        "coords": {"type": "array"}
    },
    "required": ["temperature", "description", "coords"]
}
```

Figure 1.5. Possible schema for weather API response shown in Figure 1.4

Without JSON schemas designed to validate requests or responses, developers face tough times designing applications for both client and server. They not only have to write a program to parse the document controlling all the possible cases without any guarantee of integrity, but they also have to collect or handcraft samples for testing their code. That does not take in consideration how fragile is the implementation if there is a minor change in the JSON format.

Furthermore, without JSON schemas we are not taking advantage of the autocompletion features of most Integrated Development Environments (IDEs). With a schema, IDEs would be able to suggest types or name fields when operating with specific data.

Currently, there are some efforts to standardize API definitions based on JSON schema. Those efforts are not enough since manually creating schemas for all the endpoints for all the public APIs is prohibitively expensive. Thus, we need to extract JSON schemas automatically from existing data.

Indeed, there are also some efforts to learn schemas from provided examples. However, the previous work focuses only on getting schemas or another alternative model from input data. They do not provide foundations nor a theoretical framework on what any inference process should base on. As a result, the previous work offer algorithms that are very complicated to understand, and with different approaches that cannot take advantage of pre-existent knowledge. The lack of a clear framework for learning schemas has also discouraged the integration of Machine Learning algorithms since it is not clear where exact learning is achievable or where a probabilistic learning approach is the only way to get good results.

In this thesis, we propose a theoretical framework to study the learnability of JSON schemas from positive samples. Although the full specification for JSON schema is not identifiable under this framework, we devise fragments of the language that are identifiable. Afterward, we define a class of schemas with some desirable properties: first, it is identifiable our the proposed framework; second, it is useful in real-world scenarios; and third, we can provide a simple algorithm to identify it.

We believe that providing clear and concise foundations to infer JSON schemas would help to put in clear what are the remaining challenges on this topic, where a probabilistic learning algorithm would help, and what are the theoretical boundaries for this task. Progress on this line would promote more developers to automatize schema extraction duties and to take advantage of the numerous benefits of JSON schema.

1.1. Summary of contributions

First, we provide a theoretical framework to study the learnability problem from positive samples for JSON schema. Precisely, we use language identification in the limit framework by Gold et al. (1967), and we explain why it is suitable for this problem, taking into consideration previous work for this document schema and others.

Then, we prove that it is not possible to learn in the limit any arbitrary JSON schema. This is expected, as for example regular expressions are known not to be learnable in this framework either. Thus we provide a theoretical analysis of which classes of JSON Schema can be learned in the limit, arriving at a very interesting picture of different classes that can be learned, that are incomparable to each other, and that can be combined through limited boolean combinations and nesting. We also identify what are the problematic features in JSON Schema that lead to unlearnability.

We then propose one particular class of schemas as a natural alternative for our learning algorithm. The decision of the schema class to learn is based not only on theory, but also on an empirical study about which features in JSON schema are popular in schemas available worldwide, which is also a contribution on its own. We test the practical applicability of our class by learning the schema of three popular APIs: Open Weather, GitHub and Twitter. We show that the class is general enough to obtain a realistic schema for these APIs, while at the same time the learned schema is more than just a description of all the examples.

A limitation of our framework is that our class of schemas cannot specify the `"patternProperties"` keyword, used to specify that in al pairs `k:v` in an object, if k belongs to a regular expression then v must conform to a certain schema. However, this keyword is useful to produce more compact schemas avoiding redundancy. Thus, as a final contribution, we improve our learning algorithm with a component that decides when to learn `"patternProperties"`, that uses a heuristic based on a distance notion between schemas. We use this algorithm to learn a schema for Wikidata, which takes advantage of

`"patternProperties"`, and compare it with the schema of the vanilla algorithm. Both algorithms produce almost equivalent schemas, but indeed the schemas produced with the heuristic variant can be much more compressed.

1.2. Related work

There are some efforts to standardize API definitions based on JSON schema. One of them is the API Discovery Service (Google LLC, 2018), which is a service from Google that provides metadata about Google APIs in JSON schema format. Another example is the OpenAPI Initiative (The Linux Foundation, 2018), which describes itself as a specification that "defines a standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic". A third example is Schema Store (Kristensen et al., 2018), which is a repository of JSON schemas created by the community for specific services or files.

Regarding schema extraction from examples, one tool to perform this task is JSONschema.net (Wootton, 2017). JSONschema.net is a web application that infers a schema from an example, whose primary goal is to make schema generation quick and straightforward. To achieve that purpose, it provides a user interface to customize the space of possible target schemas. However, the application only accepts one example, limiting its potential use in scenarios where some fields are optional or where there are boolean combinations of different schema types.

Another development is Schema Guru (Snowplow Analytics Ltd, 2016), which is a tool to derive JSON schemas from a set of JSON instances. Unlike JSONschema.net, this tool allows more than one example which is useful when there are optional fields. By default, string schemas have both length bounds, and number schemas have both minimum and maximum values. It also identifies some common regular expressions, like base64

encoding, IP addresses, ISO country codes, among others. Nevertheless, there is no insight on the limitations that this program could have, nor explanations on why it supports some features and others not.

From the scientific community, we have a couple of attempts to infer JSON schemas or another way to model JSON data. The work of Wischenbart et al. (2012) aims to integrate data from user profiles extracted from Facebook, LinkedIn, and Google+. Their approach is to generate a single data model for Eclipse Modeling Framework (EMF). They do it by via extracting a JSON schema for each service and then merging them. Izquierdo and Cabot (2013) made a similar work but creating a set of rules to address the need of automatic schema discovery for APIs.

Klettke, Störl, and Scherzinger (2015) propose an algorithm to infer a JSON schema from a MongoDB database. They generate a directed graph from the documents stored in the database and derive a schema from that graph. Finally, they use the graph to generate statistics and detect outliers in the DB.

The work of Wang et al. (2015) propose an algorithm to extract schemas from JSON data stores. They use a hierachical data structure to compare and generate distinct schemas for the database objects. They use the created structure to answer schema and attribute existence queries.

Baazizi, Ben Lahmar, Colazzo, Ghelli, and Sartiani (2017) focus on inferring a type data from JSON massive datasets. For this, they identify a minilanguage to represent a sort of JSON schema. Their algorithm works in a two-step process: first, they map every single value of the input to a JSON type (i.e., object, array, number, boolean, and others); then they make type fusions following a set of rules trading precision for human-readability. Their fusion process emphasized commutativity and associativity in order to use a map-reduce programming model.

The last research on this topic, at best of our knowledge, comes from Frozza, dos Santos Mello, and da Costa (2018). They construct an intermediate representation for

each document describing the type of the data they have, and then they construct a JSON schema from their structure. They make a benchmark with other previous works, claiming that their approach is as good or better than previous work.

As we said before, the previous work regarding schema extracting from JSON data do not provide foundations nor a theoretical framework on what any inference process should base on. Furthermore, some works do not take into consideration the efforts to formalize JSON schema specification from Pezoa et al. (2016) and Bourhis, Reutter, Suárez, and Vrgoč (2017).

On the contrary, for XML documents there are works that establish a theoretical framework built from the notion of learnability in the limit, and construct learnable classes of DTD and XSD (Bex, Neven, Schwentick, & Tuyls, 2006; Bex, Neven, & Vansummeren, 2007; Bex, Neven, Schwentick, & Vansummeren, 2010; Bex, Gelade, Neven, & Vansummeren, 2010).

1.3. Thesis outline and structure

This Chapter motivated the problem we intend to solve and showed the current state-of-art. It also summarized the contributions of this work.

With respect to the next chapters, Chapter 2 introduces some preliminaries to complete the theoretical framework we need, including but not limiting to an extract of JSON schema formalization and language identification in the limit framework. Then, in Chapter 3 we give a formal analysis of lernability for interesting JSON schema classes. Next, in Chapter 4 we analize which features of JSON schemas are present on real-world schemas to create a suitable JSON schema class identifiable in the limit; we also provide a simple algorithm to identify the created class. Afterward, Chapter 5 shows the results of the algorithm on three real-world samples. In light of the observed in Chapter 5, Chapter 6 offers an improvement to create more concise schemas. Finally, we present concluding remarks and some future improvements and lines of research in Chapter 7.

2. PRELIMINARIES

2.1. JSON schema specification

We first present how JSON schema works. The most recent draft of the specification is in its seventh version (Wright & Andrews, 2018), but we will deal with a core fragment as described in Pezoa et al. (2016), which is equivalent to the full specification.

Every JSON schema is a JSON document itself. A schema can specify the type of document among these: numbers, strings, booleans, nulls, objects, and arrays. For each type, there is a specific set of optional keywords that constrain the data the document can contain. All the types share the mandatory keyword `"type"` which determines the type against the document will be validated. For instance, a document of the form {`"type":"string", ...`} specifies string values, {`"type":"number", ...`} specifies numeric values, {`"type":"object", ...`} specifies objects, and {`"type":"array", ...`} specifies arrays. We now summarize each of the six different types; for further details we refer to Pezoa et al. (2016) and Bourhis et al. (2017).

- **Number schemas**: Number schemas represent numeric data, which can be integers or floats. If the schema describes integers, it includes the key-value pair `"type":"integer"`, otherwise it includes the pair `"type":"float"`. To simplify the presentation of this work we use only with integers unless otherwise stated, although the vast majority of the results are easily extendable to use floating-point numbers with limited precision. It is possible to specify minimum value including the key-value pair `"minValue"`$:i$, for $i \in \mathbb{Z}$, which validates only numbers greater or equal than i. The key-value pair `"maxValue"`$:j$, for $j \in \mathbb{Z}$, has a similar behavior. Additionally, the pair `"multipleOf"`$:k$ specifies that a number should be multiple of k, for $k \in \mathbb{N} \setminus \{0\}$. Thus for example {`"type":"integer", "minimum":12, "multipleOf":4`} describes the numbers 12, 16, 20, and so on.

- **String schemas**: String schemas represent string data. They include the key-value pair `"type":"string"`. These schemas can include the key-value pairs `"minLength"`:i and `"maxLength"`:j, for $i, j \in \mathbb{N}$, which validates only against those strings whose length is equal or between i and j. Finally, they may feature the pair `"pattern"`:*regex*, for *regex* a regular expresion over the alphabet Σ, which only validates against strings that are in $\mathscr{L}(regex)$. For example, {`"type":"string"`} and {`"type":"string", "pattern":"^(0|1)+$"`} are string schemas. The first schema validates against any string, and the second only against strings built from 0 or 1.
- **Boolean schemas**: Boolean schemas represent boolean values, i.e., `true` and `false`. They feature the pair `"type":"boolean"` without the ability to put more restrictions.
- **Null schemas**: Null schemas represent the value `null`, which is often utilized to show that a there is no value for a field. They only have the key-value pair `"type":"null"`.
- **Object schemas**: Object schemas represent JSON objects, which themselves have a key-value structure. Besides the key-value pair `"type":"object"`, they may feature the following:
  - `"minProperties"`:i and `"maxProperties"`:j, which specify that an object should have at least i pairs and/or at most j of them.
  - `"required"`:$[k_1, \ldots, k_n]$, where each k_i is a string value. This keyword enforces that all the properties k_i must be present in objects validating against the schema.
  - `"properties"`:$\{k_1 : S_1, \ldots, k_m : S_m\}$, where each k_i is a key and each S_i is another JSON schema. This keyword establishes that the value associated to the keyword k_i should validate against the schema S_i.
  - `"patternProperties"`:$\{r_1 : S_1, \ldots, r_l : S_l\}$, where each r_i is a regular expression over Σ and each S_i is a JSON schema. This keyword determines that a value under a key in the language of r_i should validate against S_i.

```
{
    "type": "object",
    "properties": {
        "name": {"type": "string"}
    },
    "patternProperties": {
        "^a(b|c)a$": {
            "type": "number",
            "multipleOf": 2
        }
    },
    "required": ["name"],
    "additionalProperties": false
}
```

Figure 2.1. Object schema example

  - `"additionalProperties"`:S, where S is a JSON schema. This keyword specifies the schema for all the values associated to keys not present in `"properties"` and keys not belonging to the language of any regular expression in `"patternProperties"`. In this work we will use $S =$ `true` to refer the schema that accepts any JSON value, and $S =$ `false` to make reference to the schema that does not accept any JSON value.

  For example, the schema in Figure 2.1 specifies objects where the value under `"name"` must be a string, the value under any key of the form `^a(b|c)a$` must be an even number, the key `"name"` must be present and there cannot be any other keys in the document.

- **Array schemas**: Array schemas represent JSON arrays, which contain a collection of JSON values. This kind of schemas include the pair `"type":"array"` and zero or more of the following key-value pairs:
  - `"items"`:S or `"items"`:$[S_1, \ldots, S_n]$, where S and each S_i are JSON schemas. This keyword indicates the schema that a element should satisfy. If it is

specified an array of schemas, then the element i of the array should validate against the schema S_i. Otherwise, all the elements should comply with S.

- `"additionalItems"`:S, where S is a JSON schema. In case `"items"` specifies an array of n schemas, each element whose position i is greater than n should be verified against the schema S. If `"items"` specifies a single schema, `"additionalItems"` keyword is ignored. In this work we will use $S =$ `true` and $S =$ `false` as defined for `"additionalProperties"` in object schemas.
- `"minItems"`:i and `"maxItems"`:j, for $i, j \in \mathbb{N}$, indicate that the arrays must have at least i elements and/or at most j items.
- `"uniqueItems"`:`true` enforces that all the elements in the array have to be different.

For example, the schema in Figure 2.2 validates against arrays of at least 2 elements, where the first two are strings and the remaining ones, if they exists, are numbers.

```
{
  "type": "array",
  "items": [{"type": "string"}, {"type": "string"}],
  "additionalItems": {"type": "number"},
  "uniqueItems": true
}
```

Figure 2.2. Array schema example

Another essential feature of JSON schema is the ability to construct boolean combinations of schemas. For instance, `"anyOf"`:$[S_1, \ldots, S_n]$ is a schema that accepts a value if and only if at least one schema S_i accepts it. Similarly, `"allOf"`:$[S_1, \ldots, S_m]$ accepts a value if and only if all the schemas S_i accept it. We can also indicate that a value **should not** be valid with respect a schema S using `"not"`:S. For example, the schema

`{"not":{"type":"integer","multipleOf":2}}` validates against any odd number, or any document which is not a number. Finally, one can indicate that a value must be in a fixed collection of values with `"enum"`$:[J_1,\ldots,J_l]$, where each J_i is a valid JSON value.

From now on we will denote that the document J validates against the schema S as $J \models S$. On the contrary, if the document J does not validate against the schema S we denote it as $J \not\models S$. We use the same notation and idea for sets of documents, thus we say that the set D validates against the schema S if and only if $J_i \models S, \forall J_i \in D$, and we denote it as $D \models S$. We also say that the schema S is consistent with X if and only if $X \models S$.

2.2. Language learning framework

In this section, we explain the language identification problem and then we present the learning framework with we will work. We provide justifications of our choice using related work.

Learning languages is a classic problem discussed in linguistics and computer science. People tend to compare the remarkable ability of humans to learn natural languages after being exposed to a finite amount of examples. Children are capable of constructing well-formed sentences in their mother tongue even at an early age (Mohri, Rostamizadeh, Talwalkar, & Bach, 2012).

In computer science, we try to learn a language, potentially with an infinite amount of words. Actually, we attempt to learn a computational representation that allows to generate any element of the target language (Mohri et al., 2012). A learning framework determines the input of the learning problem and the expected output. In the next paragraphs we will talk about the *identification in the limit framework*.

In the identification in the limit framework, the language learning problem consists on identifying exactly a target language L after receiving a finite amount of examples.

Definition 2.1 (Adapted from Mohri et al. (2012))**.** *Let Σ be a finite alphabet. A class of languages $\mathscr{C}$ is "identifiable in the limit" if there exists an algorithm A that identifies any language $L \in \mathscr{C}$ after examining a finite number of tagged examples, and its hypothesis remains unchanged thereafter.*

The tags in Definition 2.1 refers to a mean to distinguish positive examples (strings which belong to the target language) from negative ones. Thus, the definition does not restrict the nature of the example set: the learner algorithm can receive either positive examples, negative examples, or a mix of both. In certain situations it is necessary to restrict the input to only positive examples, for instance, Gold et al. (1967) argue that children rarely receive negative feedback when they make grammatical errors.

Definition 2.2. *Let Σ be a finite alphabet. A class of languages $\mathscr{C}$ is "identifiable in the limit from positive examples" if there exists an algorithm A that identifies any language $L \in \mathscr{C}$ after examining a finite number of positive examples, and its hypothesis remains unchanged thereafter.*

Intuitively, for each identifiable language, there exists a set of examples which allows an algorithm to identify it unambiguously, which we call *characteristic set*. Thus, a learner algorithm should identify the target language with any superset of the characteristic set. When we only consider positive examples, the supersets should also be subsets of the target language. Indeed, the following lemma summarizes this idea.

Lemma 2.1 (Denis, Lemay, and Terlutte (2002))**.** *A class of language $\mathscr{C}$ is identifiable in the limit from positive data if there exists an algorithm A that takes as input a set of words and outputs the representation of a language, and if we can associate with each language L of $\mathscr{C}$ a characteristic positive sample D_L such that, for every sample $D \subseteq L$ with $D_L \subseteq D$, $A(D)$ is a representation of L.*

Gold et al. (1967) proved that any class containing all finite languages and at least one infinite language is not identifiable in the limit from positive examples. They also

showed the immediate consequence: regular expressions are not identifiable in the limit from positive data.

The previous definitions do not restrict the size of the input set, nor the learning time. Besides, a relative basic class of languages is not identifiable in the limit from positive samples. Then, one may wonder why the identifiability in the limit from positive examples framework is suitable to study the learnability problem for JSON schema.

We take into account that the main difficulty when learning a schema is avoiding an overfitted description of the examples. Indeed, assume we are given a set $J_1, \ldots, J_n$ of examples, and that we wish to learn a JSON Schema out of this set. The learned schema must of course validate all the examples, but we must take caution in preventing that the learned schema validates *only* those given documents. Instead, we need to find a way to learn a schema S that is at the same time specific enough to validate our examples, but general enough to validate other similar documents. Note that this is fundamentally different from the task of schema extraction, denoted as schema inference in (Baazizi, Ben Lahmar, et al., 2017; Baazizi, Colazzo, Ghelli, & Sartiani, 2017). In this other task, the focus are large JSON datasets for which one needs to obtain "complete structural information about input data" (Baazizi, Ben Lahmar, et al., 2017). Hence the focus is on much more precision, but at the same time using an algorithm that does not generalizes as well as a learning algorithm, specially when given fewer examples.

Also, this framework has demonstrated to be useful to study the inference of underlying schemas from other types of semistructured documents, such as XML. Works of Bex et al. (2006); Bex et al. (2007); Bex, Neven, et al. (2010); and Bex, Gelade, et al. (2010) consistently used this framework to analyze the learnability of DTDs and XML schemas from data.

Furthermore, the related work for extracting schema data from JSON documents has not used negative examples (Wootton, 2017; Snowplow Analytics Ltd, 2016; Wischenbart et al., 2012; Izquierdo & Cabot, 2013; Klettke et al., 2015; Wang et al., 2015; Baazizi,

Ben Lahmar, et al., 2017; Frozza et al., 2018). Intuitively, this is because not all negative example is useful to recognize a language. Thus, negative examples should be constructed carefully, requiring human intervention which goes against the goal of economizing human effort.

2.2.1. Definitions for JSON schema learnability in the limit from positive sample

In this subsection, we provide specific definitions for JSON schema under the presented framework. Let JSON denote the set of all JSON documents, and let $\mathscr{S}$ be a subclass of all JSON schemas. We say that the class $\mathscr{S}$ is *learnable in the limit* if there is a computable learning function $A : 2^{\mathsf{JSON}} \to \mathscr{S}$ that assigns a schema in $\mathscr{S}$ to each finite set of JSON documents, and such that the following holds:

(i) For every set $D = \{J_1, \ldots, J_n\}$ of documents, the learned schema $A(D)$ is consistent with D.

(ii) For each schema $S \in \mathscr{S}$ there is a set D_S of JSON documents such that for every set D', with $D_S \subseteq D'$ and D' consistent with S, we have that $A(D') = A(D_S)$; furthermore, each document accepted by $A(D_S)$ is also accepted by S.

The first condition is a soundness guarantee: The learned schema must always validate the examples which are fed to it. The second condition is how we ensure the generality of the learned schemas, and can be intuitively understood as follows. Imagine that we are trying to learn a schema S. We do not have access to S, but can request a number of examples of JSON documents that are accepted by S. Then, if the above conditions hold for a learning algorithm A, once we request all examples in D_S we will know that the output of our learning algorithm will not change when more examples are provided. The set D_S is the *characteristic set* of S.

From now on, we will refer to the presented framework as "identification in the limit", "identification in the limit from positive data" and "Gold's framework" indistinctly.

3. LEARNABILITY OF JSON SCHEMAS

In this section we establish the learnability of JSON Schema. We study each type of schema by separate, providing learnable subsets whenever the full class is not learnable in the limit. JSON values `true`, `false` and `null` are only specified in JSON schema by enumeration, and these are trivial to learn. Thus we only focus on numbers, strings, arrays and objects.

3.1. Numbers

We begin by analyzing how to learn numeric JSON schemas. In JSON, numeric schemas are mostly specified in terms of intervals (by specifying maximum and/or minimum values), possibly with the additional restriction of being a multiple of a specific number.

The first observation we make is that, under Gold's framework, it is not possible to learn both closed intervals and infinite sets of numbers at the same time. Moreover, it is not feasible to identify infinite intervals bounded on one side and unbounded intervals simultaneously. To formalize this claim, we define the following classes of numeric schemas.

(i) $\mathscr{N}_{\_}$ is the class of schemas that only use `"multipleOf"`.
(ii) $\mathscr{N}_{\text{min}}$ is the class of schemas using keywords `"minimum"` and `"multipleOf"`.
(iii) $\mathscr{N}_{\text{max}}$ is the class of schemas using instead `"maximum"` and `"multipleOf"`.
(iv) $\mathscr{N}_{\text{min-max}}$ is the class of schemas using all the keywords `"minimum"`, `"maximum"` and `"multipleOf"`.

The class $\mathscr{N}_{\text{min-max}}$ defines closed intervals, but all the remaining ones define infinite sets of numbers. Note also that the inclusion of `"multipleOf"` in all those classes is without loss of generality for integers, since `"multipleOf":1` is equivalent to not having

this keyword at all. As promised, the following proposition shows that finite and infinite sets cannot be mixed if one hopes for learnability.

PROPOSITION 3.1.

(i) *Neither* $(\mathscr{N}_{min} \cup \mathscr{N}_{\_})$ *nor* $(\mathscr{N}_{max} \cup \mathscr{N}_{\_})$ *are learnable in the limit.*

(ii) *Neither* $(\mathscr{N}_{min} \cup \mathscr{N}_{min\text{-}max})$ *nor* $(\mathscr{N}_{max} \cup \mathscr{N}_{min\text{-}max})$ *are learnable in the limit.*

(iii) *Let S be a single schema from* $\mathscr{N}_{\_}$*. Then* $\mathscr{N}_{min\text{-}max} \cup S$ *is also not learnable in the limit.*

To prove this proposition, we assume a learner algorithm exists for each class and then we fool it using a set of documents for which it as to return two schemas at the same time, leading to a contradiction.

PROOF FOR PROPOSITION 3.1.1. We begin proving that $\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\_}$ is not learnable in the limit.

By contradiction, let us assume that this schema class is identifiable in the limit. Let S be a schema from $\mathscr{N}_{\_}$ and let D_S be its characteristic set. Now, let S' be a $\mathscr{N}_{\text{min}}$ schema constructed in this way: set the `"multipleOf"` to have the same value as S, and set `"minimum"` to be the smallest element of D_S. Also, let $D_{S'}$ be the characteristic set of the constructed schema S'.

Let us check what happens if the algorithm that learns $\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\_}$ schemas receives $D = D_S \cup D_{S'}$. On one side D is superset of D_S and subset of S, then it should identify S. On the other hand, D is superset of $D_{S'}$ and subset of S', then it should learn S'. As a result, there is a contradiction. The process to demonstrate that $(\mathscr{N}_{\text{max}} \cup \mathscr{N}_{\_})$ is not learnable in the limit is analogous.

□

PROOF FOR PROPOSITION 3.1.2. We begin proving that $\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\text{min-max}}$ is not learnable in the limit.

By contradiction, let us assume that this schema class is identifiable in the limit. Let S be a schema from $\mathscr{N}_{\text{min}}$ and let D_S be its characteristic set, which in this case is simply a set of numbers. Now, let S' be a $\mathscr{N}_{\text{min-max}}$ schema constructed in this way: set the `"minimum"` and `"multipleOf"` to have the same value as S, and set `"maximum"` to be the maximum element of D_S. Also, let $D_{S'}$ be the characteristic set of the constructed schema S'.

Let us check what happens if the algorithm that learns $\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\text{min-max}}$ schemas receives $D = D_S \cup D_{S'}$. On one side D is superset of D_S and subset of S, then it should identify S. On the other hand, D is superset of $D_{S'}$ and subset of S', then it should learn S'. As a result, there is a contradiction.

We realize that the reasoning to prove that $\mathscr{N}_{\text{max}} \cup \mathscr{N}_{\text{min-max}}$ is not identifiable in the limit is very similar. Given a schema in $\mathscr{N}_{\text{max}}$ and its characteristic set, it is sufficient with constructing another schema in $\mathscr{N}_{\text{min-max}}$ with the same values for `"maximum"` and `"multipleOf"`, and constraining the `"minimum"` to the smallest element of the characteristic set. We get the same contradiction as before. □

PROOF FOR PROPOSITION 3.1.3. Following the same idea as the previous proofs, let D_S be the characteristic set of S. We construct a S' schema of $\mathscr{N}_{\text{min-max}}$, where the `"minimum"` and `"maximum"` are chosen to be the smallest and greatest element of D_S, and `"multipleOf"` to the same as S. Then, we encounter with the same problem: what if the algorithm receives $D_S \cup D_{S'}$? Like before, the algorithm should provide S and S' as result at the same time. Contradiction. □

So what can we do? It is easy to devise algorithms to learn schemas in one of the classes $\mathscr{N}_{\_}$, $\mathscr{N}_{\text{min}}$, $\mathscr{N}_{\text{max}}$ or $\mathscr{N}_{\text{min-max}}$. For $\mathscr{N}_{\_}$ we just keep track of the GCD of what we are learning, and for open intervals we additionally track the smallest (or greatest) number we have seen, and for closed intervals we track both the smallest and greatest numbers.

Interestingly, we can also show that the class $(\mathscr{N}_{\min} \cup \mathscr{N}_{\max})$ of all intervals closed on one side is also learnable in the limit. Summing up, we have the following positive results.

PROPOSITION 3.2.

(i) *All of $\mathscr{N}_{\_}$, $\mathscr{N}_{min}$, $\mathscr{N}_{max}$ or $\mathscr{N}_{min\text{-}max}$ are learnable in the limit.*
(ii) *The class $(\mathscr{N}_{min} \cup \mathscr{N}_{max})$ is also learnable in the limit.*

SKETCH. The first point of Proposition 3.2 is straightforward: we create an algorithm for each of those classes, and we show that for each language in the hypothesis space there exists a characteristic set complying with the identification in the limit framework.

For the second point we need a more complicated algorithm: we create an injective map between from each schema in $(\mathscr{N}_{\min} \cup \mathscr{N}_{\max})$ to a characteristic set compound of two numbers: one indicating the bound itself, and another number (depending on the first one) indicating the *direction* of the interval.

Let S be a schema in $\mathscr{N}_{\min} \cup \mathscr{N}_{\max}$, and let $bound(S)$ be a function that extracts the lower or upper bound of the interval that S represents. The mapping is as follows:

$$f(S) = \begin{cases} f_{\min}(S) & S \in \mathscr{N}_{\min} \\ f_{\max}(S) & S \in \mathscr{N}_{\max} \end{cases}$$

$$f_{\min}(S) = \begin{cases} bound(S), k \times (2 \times bound(S) + 1) & bound(S) \geq 0 \\ bound(S), -k \times (2 \times bound(S)) & bound(S) < 0 \end{cases}$$

$$f_{\max}(S) = \begin{cases} bound(S), -k \times (2 \times bound(S) + 1) & bound(S) \geq 0 \\ bound(S), k \times (2 \times bound(S)) & bound(S) < 0 \end{cases}$$

As example, for {"type":"integer", "minimum":3, "multipleOf":1} the characteristic set is $\{3, 7\}$ and for {"type":"integer", "maximum":3, "multipleOf":1} is $\{3, -7\}$.

Note that the characteristic sets do not translate into relevant numbers from a practical point of view, and thus this algorithm is not very useful in practice. For more details of this proof we refer to the Appendix A.1. □

3.2. Strings

We continue analyzing how to learn string JSON schemas. In JSON, string schemas are mostly specified in terms of length limits (in other words, intervals of lengths), possibly with the additional restriction of being accepted by a specific regular expression.

We have mentioned that schemas specifying regular expressions in full cannot be learned in the limit (Gold et al., 1967). Given that any algorithm for learning a subset of regular expressions can be translated into an algorithm for learning string schemas, we do not focus on comparing these restrictions (for reference see e.g. (Bex, Gelade, et al., 2010; Freydenberger & Kötzing, 2015; De La Higuera, 1997; Fernau, 2009; Ilyas, da Trinidade, Fernandez, & Madden, 2017)). Instead we provide two easy learnable classes: we can either learn complete patterns by bounding the length of strings, or we can just be content with their minimal length. Formally, define the following classes of schemas.

(i) $\mathscr{S}_{\text{min}}$ is the class of schemas using only the keyword "minLength".
(ii) $\mathscr{S}^{\text{pattern}}_{\text{min-max}}$ is the class of schemas using keywords "minLength", "maxLength", and "pattern".

Note that $\mathscr{S}_{\text{min}}$ includes the schema that accepts all the strings, since it is equivalent to setting "minLength" to zero. Also note that $\mathscr{S}^{\text{pattern}}_{\text{min-max}}$ only defines finite sets, because the length is always constrained. Like the number schemas case, one cannot hope to learn both finite and infinite sets of strings.

PROPOSITION 3.3.

(i) *Both* $\mathscr{S}_{min}$ *and* $\mathscr{S}^{pattern}_{min\text{-}max}$ *are learnable in the limit.*

(ii) $\mathscr{S}_{min} \cup \mathscr{S}^{pattern}_{min\text{-}max}$ *is not learnable in the limit.*

3.3. Objects

Objects (and arrays) are fundamentally different from numbers and strings because they involve nesting. Thus, our first task is to understand how any learning algorithm is supposed to work in the presence of nested documents. Let us motivate these issues by means of an example.

Example 3.1. *Consider all three documents from Figure 3.1. In trying to discover the best schema to deduce from these documents, let us look at the keys in these documents. First of all, the key* `"age"` *appears in all documents, and it is always an integer. It seems fair to assume that all documents must include* `age`*, in form of an integer. Next, both* J_2 *and* J_3 *include the key* `"address"`*, the value of this key is an object, but not the same object: in* J_2 *we see a* `"city"` *as a string, but in* J_3 *we see* `"street"`*, which is again an object with two key:value pairs.*

We want to be as general as possible, but at the same time compliant with Gold's framework. Figure 3.2 shows the schema that would be inferred by our algorithms. This algorithm is the result of some important choices we have made, and that constitute a path to define a class of schemas that can be learned in the limit. In particular, we note the following choices:

- *We close our schemas using the pair* `"additionalProperties":false`*. As we see below, this is the only way we can work with Gold's framework.*
- *We are assuming that if a key appears in all documents of a sample, then this key is required (see* `"age"`*).*

- *A more involved choice is to assume all values under* `address` *comply to the same schema, which yields the statement that* `address` *pairs may have a city, a street, or both. We could have imposed instead a better fit by stating that addresses may contain either a city or a street, but not both (since we haven't seen any document with both a city and a street). However, to express this we need to use boolean combinations of schemas, a feature that we later show to produce unlearnable classes of schemas.*
- *Our assumptions made it possible to learn schemas in a level-to-level fashion: to learn the schema under* `address` *we group all documents nested under the path leading to* `address`*, and learn the schema of those documents. This schema is then appended to the* `address` *key when learning the outer level schema.*

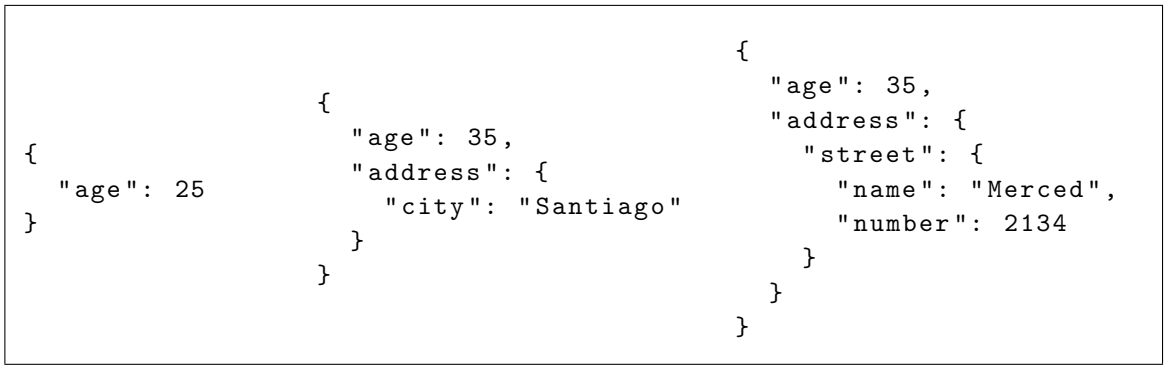


Figure 3.1. Documents that we want to infer a schema from.

In the remainder of this section we give a formal justification for these choices.

The first choice is to work with `"additionalProperties":false` for all object schemas we attempt to learn. To justify this election, define $\mathscr{O}^{+}_{\text{prop}}$ as the class of schemas specifying `"properties"` and `"additionalProperties":true`, and $\mathscr{O}_{\text{prop}}$ the class of schemas specifying `"properties"` and `"additionalProperties":false`. We prove:

PROPOSITION 3.4. *$\mathscr{O}^{+}_{prop}$ is not learnable in the limit. Further, any class of schemas including at least one schema from $\mathscr{O}^{+}_{prop}$ and every schema in $\mathscr{O}_{prop}$ is also not learnable in the limit.*

```
{
  "type": "object",
  "properties": {
    "age": {"type": "integer"},
    "address": {
      "type": "object",
      "properties": {
        "city": {"type": "string"},
        "street": {
          "type": "object",
          "properties": {
            "name": {"type": "string"},
            "number": {"type": "integer"}
          }
        }
      }
    }
  },
  "required": ["age"],
  "additionalProperties": false
}
```

Figure 3.2. A possible learned schema for documents in Figure 3.1. For readability we have left out the `"additionalProperties":false` keyword from all the inner object schemas.

The argument for this proof is similar to what we used for numeric schemas. Assuming that these classes are learnable, we look for the characteristic set of a particular schema, and then construct another schema that fools the algorithm into returning two different schemas. Let $\mathscr{O}^{\dagger}$ be a class of schemas including $\mathscr{O}_{\text{prop}}$ and one schema S from $\mathscr{O}^{+}_{\text{prop}}$. Assume that $\mathscr{O}^{\dagger}$ is learnable in the limit, say by an algorithm $\mathscr{A}$. then S has a characteristic set D_S. We construct an object schema $S' \in \mathscr{O}_{\text{prop}}$ that (1) is consistent with D_S and (2) $S' \subseteq S$; and let $D_{S'}$ be its characteristic set. It follows that, on input $D_S \cup D_{S'}$, the algorithm must learn both S and S', which is a contradiction.

Regarding positive results, we can learn schemas with the `"required"` without troubles. We can also add `"minProperties"` and/or `"maxProperties"`, but we do not present this result since it will not be used in our algorithm and, as we see in Section 4.1,

they are seldom used in practice. We remark that the choices we made also allows us to learn object schemas in a level-to-level fashion, which is highly desirable. To put this formally, let $\mathscr{O}_{\text{prop,req}}$ be the class of object schemas allowing `"properties"` and `"required"`, but such that any key:value pair `"a"`:S inside a `"properties"` clause uses a schema S that is learnable in the limit. We then have:

PROPOSITION 3.5. *$\mathscr{O}_{prop,req}$ is learnable in the limit.*

Another choice we did not mention in the example above is not work with `"patternProperties"` at all. Essentially, `"patternProperties"` allow to constrain the possible keys in an object with respect a certain regular expression. Regular expressions are themselves not identifiable in the limit from positive examples, and we can use this result to show that schemas with pattern properties are not learnable either.

3.4. Arrays

We continue working with JSON values involving nesting, this time with arrays. Similarly, as we did with objects, we want to deal with nesting in a level-to-level fashion; we show that this is doable as long as we learn schemas for each position of the array. The first result is that once again we cannot distinguish classes of array schemas where the number of items can be upper bounded or not. To that extent, let us define these classes of array schemas:

(i) $\mathscr{A}^{\text{min}}$ is the class of schemas that use the keyword `"minItems"`.

(ii) $\mathscr{A}^{\text{min-max}}$ is the class of schemas that use the keywords `"minItems"` and `"maxItems"`.

What those classes capture is that the length of the array is constrained to a finite or infinite interval. If we want to learn the union of both classes we arrive at the same problem we had for numeric and string types.

PROPOSITION 3.6. *$\mathscr{A}^{min} \cup \mathscr{A}^{min\text{-}max}$ is not identifiable in the limit from positive examples*

The rest of the features are easier to learn in arrays, as long as the nested schemas are also learnable. Formally, let us now define the following classes:

(i) $\mathscr{A}^{\text{min-max}}_{\text{items}}$ of schemas using keywords `"minItems"`, `"maxItems"`, `"items"` with an array of length equal to `"maxItems"` and `"uniqueItems"`, and such that all schemas under the `"items"` clause are learnable in the limit.

(ii) $\mathscr{A}^{\text{min}}_{\text{items}}$ of schemas using keywords `"minItems"` and `"additionalItems"`, and optionally the keywords `"items"` and `"uniqueItems"`. We require again that all schemas under the `"items"` and `"additionalItems"` clauses are learnable in the limit as well.

Both classes are indeed learnable in our framework.

PROPOSITION 3.7. *$\mathscr{A}^{min\text{-}max}_{items}$ and $\mathscr{A}^{min}_{items}$ are identifiable in the limit.*

3.5. Boolean combinations

JSON Schema also allows for union, intersection and complement of a schema. So are these boolean combinations learnable? It is not hard to suspect that the freedom of combining any number of schemas would directly lead to non-learnable schemas. For example, if we look at integers we can take the union of the interval $[0,\infty)$ with $(-\infty,0]$ to obtain the schema representing any possible number, which means that augmenting $\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\text{max}}$ with unions leads to a broader class of schemas that we know is not learnable (see Proposition 3.1). Similar examples can be found for the case of complement and intersection.

We do not analyze all possible boolean combinations of all learnable classes that we have identified, as it involves studying at least three times as many cases as we have studied so far. Instead, we do the following.

First we show a positive result about learning unions of schemas of different type (we rule out intersection because intersection of schemas of different type is always empty). Then, we defer further study on boolean combinations for the following section, where we choose a particular combination of learnable classes of different types that make up for a good overall class of schemas that can be learned. We show that adding unions leads to the intractability of even this class of schemas, but intersection can be added at no cost because the class we select is closed under intersection.

For now, let us formally state the following easy result about unions of different types.

PROPOSITION 3.8. *Let $\mathscr{C}_1, \ldots, \mathscr{C}_k$ be classes of schemas such that every schema in $\mathscr{C}_i$ only accepts documents of one type, and of the same type; and such that the type of the documents accepted by schemas in $\mathscr{C}_i$ and $\mathscr{C}_j$ is different whenever $i \neq j$. Then the class of schemas given by the union of at most one element in each of $\mathscr{C}_1, \ldots, \mathscr{C}_k$ is learnable in the limit.*

The proof is straightforward when considering that the characteristic set of a union $C_1 \cup \cdots \cup C_k$ is simply the union of the characteristic sets of each schema $C_i \in \mathscr{C}_i$.

4. A LEARNABLE CLASS OF SCHEMAS

We have identified several distinct classes of schemas for each type of JSON value that can be learned in the limit. But the union of these classes is not learnable, which means we have to chose amongst these possible options. We provide what we think is the best choice for real case scenarios, justifying our choice both from a formal point of view and also with a number of real-world examples.

4.1. Analyzing current schemas in the wild

Understanding how are JSON Schemas used in practice would give us a good input to decide the best class of schemas to learn. To this end we set up to analyse all 157 schemas from schemastore.org (Kristensen et al., 2018), one of the few repositories of schemas currently available online. Our goal is to understand which particular keywords are commonly used in practice, and which ones are not. We analyze all of the type constructs in each of these schemas as if they were a separate entity, so that instead of 157 schemas we count a total of 114807 entities, of which 54% were strings, 22% were objects, 21% were boolean combinations, and only 2% were arrays and less than 1% were numbers. Table 4.1 shows the number of entities in detail.

Table 4.1. Instances of each type of schema found in schemastore.org.

Schema type	Instances	% Instances	Documents	% Documents
Numeric	916	0.8	69	43.9
String	61769	53.8	132	84.1
Boolean	1490	1.3	90	57.3
Null	383	0.3	12	7.6
Object	24736	21.5	150	[1]95.5
Array	1672	1.5	115	73.2
Boolean combination	23841	20.8	99	63.1

[1]We found that some schemas fail to declare `"type"` keyword.

There was a little amount of pre-processing to be done: we corrected obvious encoding errors and rewrote schemas so that they comply with the compact syntax presented in this work. We next provide an analysis going through each type.

4.1.1. Numbers

For numbers, we realized that more than 90% of the schemas do not specify a maximum, but about 30% does for a minimum; when such bounds are present, they tend to be near-zero values. Also, no schema in the pool of the examined examples specified the keyword `"multipleOf"`. We can see these results in detail in the Table 4.2. Complementing the presented information, Table 4.3 exhibits the number of instances partitioned by groups of keywords, excluding `"multipleOf"`.

Table 4.2. Statistics for numeric schemas found in schemastore.org grouped by keyword use. Percentage columns are with respect to the total number of instances of the type, and the number of documents where this kind of data appears.

Keyword	**Instances**	**% Instances**	**Documents**	**% Documents**
`"minimum"`	258	28.2	18	26.1
`"maximum"`	75	8.2	10	14.5
`"multipleOf"`	0	0.0	0	0.0

Table 4.3. Statistics for numeric schemas found in schemastore.org partitioned by combinated keywords, excluding `"multipleOf"`. Percentage columns are with respect to the total number of instances of the type, and the number of documents where this kind of data appears.

Keywords	**Instances**	**% Instances**	**Documents**	**% Documents**
None	656	71.6	63	91.3
`"minimum"`	185	20.2	10	14.5
`"maximum"`	2	0.2	1	1.4
`"minimum"` & `"maximum"`	73	8.0	9	13.0

4.1.2. Strings

Strings are the most prevalent type, but they are barely specified: around 10% of string schemas are an enumeration of strings (with 8 elements on average), and less than 1% of the string entities include more complex patterns. Details of these results are in Table 4.4.

Table 4.4. Statistics for string schemas found in schemastore.org grouped by degree of specification. Percentage columns are with respect to the total number of instances of the type, and the number of documents where this kind of data appears.

Specification	Instances	% Instances	Documents	% Documents
Pattern or bounds	272	0.4	55	41.7
Enumeration	7010	11.3	66	50.0

4.1.3. Objects

Regarding objects, almost no schema constraints the minimum or maximum amount of properties. Further, not all schemas specify `"properties"` directly, most of the cases this is because this entity is being used as part of a boolean combination. On the other hand, most schemas leave open the possibility of including additional properties, but the great majority of them does this by not including the `"adittionalProperties"` keyword at all (which by the standard is the same as `"adittionalProperties":true`). We speculate that some, if not most, of them actually intends to constraint additional properties, but miss this subtlety of the standard. Interestingly, 70% of schemas do include the `"required"` keyword. In Table 4.5 we can observe the number of instances with a given keyword.

4.1.4. Arrays

For arrays, most of them specify a single schema in `"items"` to describe all the elements, and most of the times, `"additionalItems"` is not present. This fact captures the idea that an array is a collection of elements of the same type. We found more schemas specifying a minimum amount of items than a maximum (20% versus 5%), although most

Table 4.5. Statistics for object schemas found in schemastore.org by keyword. In this table we abbreviated `"additionalProperties"` as `"addProps"`. Percentage columns are with respect to the total number of instances of the type, and the number of documents where this kind of data appears.

Keyword or pair	Instances	% Instances	Documents	% Documents
`"minProperties"`	42	0.2	11	7.3
`"maxProperties"`	7	0.0	4	2.7
`"properties"`	7439	30.1	140	93.3
`"patternProperties"`	136	0.5	20	13.3
`"required"`	17150	69.3	86	57.3
No `"addProps"`	23737	96.0	135	90.0
`"addProps":true`	164	0.7	27	18.0
`"addProps":`J	262	1.1	53	35.3
`"addProps":false`	573	2.3	36	24.0

of them do not explicitly bound the length in any way. With respect to `"uniqueItems"`, we observe that about 40% of the schemas defines it as `true`. In Table 4.6 we can observe the number of instances with a given keyword.

Table 4.6. Statistics for array schemas found in schemastore.org by keyword. In this table we abbreviated `"additionalItems"` as `"addItems"`. Percentage columns are with respect to the total number of instances of the type, and the number of documents where this kind of data appears.

Keyword or pair	Instances	% Instances	Documents	% Documents
`"minItems"`	317	19.0	25	21.7
`"maxItems"`	83	5.0	4	3.5
`"items":`J	1514	90.6	112	97.4
`"items":`$[J_1,\ldots,J_k]$	6	0.4	2	1.7
No `"addItems"`	1651	98.7	114	99.1
`"addItems":true`	2	0.1	1	0.9
`"addItems":`J	2	0.1	2	1.7
`"addItems":false`	17	1.0	2	1.7
`"uniqueness":true`	596	35.6	29	25.2

4.1.5. Boolean combinations

Finally, for boolean combinations, we observed as many unions of different types as unions of the same type. Intersection is somewhat used as well, but the most surprising fact is the wide presence of complement (70% of boolean combinations are complements). This poses additional challenges in trying to learn these schemas, but we leave those for future work.

Table 4.7 shows boolean combinations grouped by type. For unions, we decided to group them depending on how many elements of the same type involved without considering if they were `"anyOf"` or `"oneOf"`. We did not considered redundant operators, for instance, intersections or unions of one schema.

Table 4.7. Statistics for boolean combination schemas found in schemastore.org by type. We grouped union operators depending on the types they are joining, without regard to the type of the union itself. Percentage columns are with respect to the total number of instances of the type, and the number of documents where this kind of data appears.

Boolean combination	Instances	% Instances	Documents	% Documents
Unions only distinct types	1584	8.9	80	86.0
Unions same type	1483	8.3	37	39.8
Intersection	2666	14.9	26	28.0
Complement	12158	68.0	20	21.5

4.2. Class and learning algorithm

We now define the class $\mathscr{L}$ of schemas that is used by our algorithms. This class is the union of classes $\mathscr{N}^{\text{learn}}$ for numbers, $\mathscr{S}^{\text{learn}}$ for strings, $\mathscr{O}^{\text{learn}}$ for objects and $\mathscr{A}^{\text{learn}}$ for arrays, and all possible enumerations of `true`, `false`, and `null`. We define this class in a recursive fashion.

For numbers we can learn either closed intervals, intervals with only minimum or only maximum, or the union of these two. We believe that the algorithm for learning the

union of both classes of open intervals is too artificial to give good results, and that learning closed intervals could easily lead to overfitting when analyzing just a few samples. One the other hand, since almost a third of the instances analyzed online do use minimum, we go with this option. We also choose not to learn `"multipleOf"` either, because no one is using it in practice. Summing up, we choose to go with the class $\mathscr{N}^{\text{learn}}$ of number schemas where the only present keywords are `"type"`, with value `"number"` or `"integer"`, and `"minimum"`.

For strings, the situation is similar to the observed with numbers, because the vast majority of the schemas did not specify a maximum nor an explicit minimum. This means that for strings we are only learning the general class $\mathscr{S}^{\text{learn}}$ where the only present keyword is `"type"` with value `"string"`.

Next is objects. From the learnability results in Section 3.3, we already know that we must close our schemas using the `"additionalProperties":false` keywords and that we cannot use `"patternProperties"`. Minimum and maximum properties are not very important in practice, but required is a must. Summing up, we learn the class $\mathscr{O}^{\text{learn}}$ of schemas where only the following keywords or key:value pairs are present: `"type":"object"`, `"additionalProperties":false`, `"required"`, and `"properties"`, and such that each schema under the `"properties"` keyword belongs to $\mathscr{L}$.

For arrays, Section 3.4 gave us two options: learn arrays with a fixed length or learn arrays with a minimum number of items. We choose to go with the latter because it seems a more general class for some uses of arrays in JSON related to databases (where typically arrays are used as a form of relational data). However, in light of our analysis of practical schemas, we actually decided to further restrict this and just learn arrays in which all elements in an array must belong to the same schema. That is, we concentrate on the class $\mathscr{A}^{\text{learn}}$ of array schemas where the keyword `"items"` is specified with a single schema, the keyword `"minItems"` is specified with any valid value, the keyword `"uniqueItems"` is present when its value is `true`, and the schema under `"items"` belongs to $\mathscr{L}$.

Finally, we are left with the issue of boolean combinations. Our review of current JSON Schema hints that all of arbitrary unions, intersection and complement are desirable. However, as we argued in Section 3.5, neither arbitrary complement nor union is possible in our framework.

PROPOSITION 4.1.

(i) *The class of schemas in $\mathscr{L}$ or a complement of a schema in $\mathscr{L}$ is not learnable in the limit.*

(ii) *The class of unions of schemas in $\mathscr{L}$ is not learnable in the limit.*

SKETCH. This proof uses a different technique than most of the proofs presented before because these classes allow for infinite characteristic sets. For both statements we construct a schema, and show that any set of documents satisfying the requirements for a characteristic set of this schema must be infinite, and therefore cannot be a characteristic set as required in our framework. □

What we can do is to learn unions and intersections of different types of schemas within $\mathscr{L}$. Intersections we get for free because $\mathscr{L}$ is closed under intersection. For unions, let $\mathscr{L}^{\cup}$ be the closure of $\mathscr{L}$ under unions of schemas of different types. We show

PROPOSITION 4.2. *The class $\mathscr{L}^{\cup}$ is learnable in the limit.*

In the rest of this section we describe the algorithm for learning schemas in $\mathscr{L}^{\cup}$. Learning numbers and string schemas of this class is straightforward, so we only show the portion of the algorithm that deals with nested structures.

In particular, the Algorithm 1 deals with objects. The algorithm collects all the nested documents grouped by property name, then it learns a schema for each property. If a property appears in all the examples the algorithm infers that that property is required. In this case, `Learn` refers to the general algorithm to learn schemas (see Algorithm 3).

Algorithm 1: Algorithm to learn object schemas in $\mathscr{L}^{\cup}$. `Learn` refers to the general learning algorithm (Algorithm 3).

Data: A set D of object JSON documents
Result: A object schema in $\mathscr{L}^{\cup}$

properties $\leftarrow$ `map()`, schemas $\leftarrow$ `map()`;
required $\leftarrow$ [*key* | $\forall d \in D$ d.*key* exists];

foreach *document d in D* **do**
    foreach *key, value in d* **do**
        properties[key].add(value);

foreach *key, documents in properties* **do**
    schemas[key] = `Learn(`*documents*`)`

return {
    `"type":"object"`, `"properties"`: schemas,
    `"required"`: required, `"additionalProperties":false`
    };

The Algorithm 2 does it with arrays. The algorithm collects all the nested documents to generate a single schema for the keyword `"items"`, and it measures the minimum length.

Algorithm 2: Algorithm to learn array schemas in $\mathscr{L}^{\cup}$. `Learn` refers to the general learning algorithm (Algorithm 3).

Data: A set D of array JSON documents
Result: An array schema in $\mathscr{L}^{\cup}$

examples $\leftarrow$ `set()`, minLength $\leftarrow \infty$, uniqueItems $\leftarrow$ `true`;

foreach *example e in the set D* **do**
    minLength $\leftarrow$ min(minLength, e.length);
    uniqueItems $\leftarrow$ uniqueItems $\wedge$ `UniqueItems(`*example*`)`;
    examples.add_all(*e*);

items $\leftarrow$ `Learn(`*examples*`)`;

return {
    `"type":"array"`, `"items"`: items,
    `"minItems"`: minLength, `"uniqueItems"`: uniqueItems
    };

Finally, the Algorithm 3 is the point of entry of the general algorithm. The algorithm separes the documents by type, calls the corresponding subroutine for each one, and returns an `"anyOf"` combination of the learned schemas.

Algorithm 3: General algorithm to learn schemas in $\mathscr{L}^{\cup}$.

```
Data: A set D of JSON documents
Result: A schema in L^∪

examples_by_type ← SepareByType(D);
schemas ← map();

foreach type, examples in examples_by_type do
    switch type do
        case "object" do
            schemas[type] ← LearnObjects(examples);
        case "array" do
            schemas[type] ← LearnArrays(examples);
        case "number" do
            schemas[type] ← LearnNumbers(examples);
        case "string" do
            schemas[type] ← LearnStrings(examples);
        case "boolean" do
            schemas[type] ← LearnBooleans(examples);
        case "null" do
            schemas[type] ← LearnNulls(examples);

return {"anyOf": schemas.list()};
```

5. EXPERIMENTAL EVALUATION

We just defined a class of schemas identifiable in the limit that is suitable to be used in the real world, and an algorithm to learn that class. The next step is to check how our algorithms works with real world examples. With that purpose, we made requests to OpenWeather (`https://openweathermap.org/`), Github (`https://github.com`), and Twitter (`https://twitter.com`) endpoints and saved the responses in order to feed our algorithm.

We fed these three sets of examples to our algorithm, thus obtaining a schema for OpenWeather, another for GitHub and another for Twitter. None of these services currently provide JSON Schema, so we could not compare the results directly. Instead, we perform a manual inspection of the schemas, and end up very satisfied. For space reasons we cannot include the complete schemas, but we highlight some portions of the result.

5.1. OpenWeather

OpenWeather is an online service that provides current weather or weather forecasts for a given location. To collect data, we made 200 queries to 5 day forecast endpoint[1], using a different city for each request.

The main challenge in this API was the amount of optional data in the returns. An example is the fields keyword, that informs about the rain or snow volume for three hours intervals. The documentation does not specify what happens with those fields when the weather is sunny. It turns out that these fields are optional, but when present, they can also be empty (see Figure 5.1).

[1] `https://openweathermap.org/forecast5`

5.2. GitHub

GitHub is a hosting service for projects using the Git version control system. GitHub provides API endpoints to help developers improve their workflow and obtain data about their repository. For Github's API, we fetched examples from the contents API[2], a query that aims to describe the contents of a given repository. To generate the data we constructed a repository with a very complex structure, and then we made 75 queries for different paths in this repository.

The response on GitHub varies with the nature of the element that is being queried. For example, if the destination of the requested path is a folder then the API returns an array of documents, but if the destination is a file then the API returns a single object. There are other types of elements distinguished by the API with several edge cases, most of them, without an explicit example. Our schema does learn this, as it provides a union of an object schema with an array of objects. Interestingly, we can observe that the response includes either an object, of an array of them with slightly fewer details. The schema also reflects that folders must have at least one file (see Figure 5.2).

5.3. Twitter

Our third set of examples comes from Twitter's API for searching tweets. We made 200 queries using a set of 33 distinct search keywords and selecting at random the number of results we wanted to retrieve.

In this case the challenge was to deal with nested data, for example, retweets which reference to their original source. The schema we obtained correctly suggests that nesting is not arbitrary and it is limited to one level, since the schema for tweets included in `"statuses"` can include a tweet in `"retweeted_status"`, but the latter cannot include that keyword. Another interesting point is that the number of obtained tweets can be zero in some circumstances, or may not have additional results available (see Figure 5.3).

[2] `https://developer.github.com/v3/repos/contents/#get-contents`

5.4. Shared findings

Fields that do not apply in all the situations (i.e., they are nullable or optional fields) were a common difficulty amongst the tree samples. The algorithm was capable of identifying those situations. We can see examples of optional fields in Figures 5.1 and 5.3, and examples of nullable properties in Figure 5.4.

Finally, it is interesting to see how distinct API describe geographic coordinates. According to Figure 5.5, OpenWeather defines coordinates with an object with `"lat"` and `"lon"` properties, describing the latitude and longitude of the point respectively. On the other hand, Twitter describes them with an object compliant with GeoJSON standard which uses a small array to emulate a tuple of coordinates. The latter is an example of when a constrained array length can be beneficial.

```
{
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "rain": {
        "type": "object",
        "properties": {
          "3h": {"minimum": 0.002, "type": "number"}
        },
        "required": [],
        "additionalProperties": false
      },
      "snow": {
        "type": "object",
        "properties": {
          "3h": {"minimum": 0.0005, "type": "number"}
        },
        "required": [],
        "additionalProperties": false
      },
      // ...
    },
    "required": ["clouds", "dt", "dt_txt", "main", "sys",
                 "weather", "wind"],
    "additionalProperties": false
  },
  "minItems": 38
}
```

Figure 5.1. Piece of OpenWeather schema. Rain and snow are not in "required", and therefore optional. They can also be empty.

```
"anyOf": [
  {
    "type": "array",
    "minItems": 1,
    "items": {
      "type": "object",
      "properties": {
        "_links": {...},
        "download_url": {...},
        "git_url": {...},
        "html_url": {...},
        "name": {...},
        "path": {...},
        "sha": {...},
        "size": {...},
        "type": {...},
        "url": {...}
      },
      "additionalProperties": false,
      "required": [...]
    }
  },
  {
    "type": "object",
    "properties": {
      "_links": {...},
      "content": {...},
      "download_url": {...},
      "encoding": {...},
      "git_url": {...},
      "html_url": {...},
      "name": {...},
      "path": {...},
      "sha": {...},
      "size": {...},
      "submodule_git_url": {...},
      "target": {...},
      "type": {...},
      "url": {...}
    },
    "additionalProperties": false,
    "required": [...]
  }
]
```

Figure 5.2. Outline of GitHub schema for repository contents

```
{
  "type": "object",
  "properties": {
    "search_metadata": {
      "type": "object",
      "properties": {"next_results": {"type": "string"}, ...},
      "required": ["completed_in",
                   "count",
                   "max_id",
                   "max_id_str",
                   "query",
                   "refresh_url",
                   "since_id",
                   "since_id_str"],
      "additionalProperties": false
    },
    "statuses": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {"retweeted_status": {...} ...}
      },
      "minItems": 0
    }
  },
  "required": ["search_metadata", "statuses"],
  "additionalProperties": false
}
```

Figure 5.3. Piece of Twitter schema showing that `"next_results"` keyword is optional. Retweets are under `"retweeted_status"`, which is similar to the schema inside the array but without the `"retweeted_status"` keyword.

```
"html": {
  "anyOf": [
    {
      "type": "string"
    },
    {
      "type": "null"
    }
  ]
}
```

```
"coordinates": {
  "anyOf": [
    {
      "type": "null"
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        // ...
      },
      "required": [
        "coordinates",
        "type"
      ]
    }
  ]
}
```

Figure 5.4. Examples of nullable fields on the generated schemas. (Left) is for GitHub data, and (Right) is for Twitter sample.

```
{
  "type": "object",
  "properties": {
    "lat": {
      "minimum": -36.906,
      "type": "number"
    },
    "lon": {
      "minimum": -156.9223,
      "type": "number"
    }
  },
  "required": ["lat", "lon"],
}
```

```
{
  "type": "object",
  "properties": {
    "coordinates": {
      "items": {
        "minimum": -122.419,
        "type": "number"
      },
      "minItems": 2,
      "type": "array"
    },
    "type": {"type": "string"}
  },
  "required": [
    "coordinates",
    "type"
  ]
}
```

Figure 5.5. Examples of coordinates schemas, OpenWeather (Left) and Twitter (Right). Additional properties key:value is left out for readability reasons.

6. CLUSTERING NESTED SCHEMAS

Our learning algorithms are very simple and appear to work well in practice. However, one particular limitation in our learning framework is that we cannot learn the `"patternProperties"` keyword. The pattern properties keyword allows one to write schemas such as the one in Figure 6.1 to indicate that any key:value pair whose key is either `"a"`, `"b"` or `"c"` must validate against a schema S. As such, this keyword is vital for producing human-readable code.

```
{
  "type": "object",
  "patternProperties": {"(a|b|c)": S, ...}
}
```

Figure 6.1. Schema using pattern properties

In the previous situation, the alternative is to write instead three triples under a properties keyword. To motivate the problem we address in this section, consider a situation where our learner returns a schema as shown in Figure 6.2, where S_a, S_b and S_c are schemas. If all three schemas are equivalent, then it is straightforward to merge them into a `"patternProperties"` clause. But often this will not be the case because of optional fields or slight variations on data. Thus, we need a way of learning that schemas S_a, S_b and S_c, while different, are close enough that they should be merged together. This interesting question takes us away from Gold's framework and more into applied learning.

```
{
    "type": "object",
    "properties": {"a": S_a, "b": S_b, "c": S_c, ...}
}
```

Figure 6.2. Example schema returned by our algorithm, where several properties can have similar schemas S_a, S_b, and S_c

We work with one solution to this problem, and show the good results it gave us in practice.

6.1. Cluster-and-join algorithm

Our improvement is based on a notion of distances for schemas, and we apply it to the schema already returned by our algorithm. Precisely, assume that our algorithm has already returned the schema with properties named above. Recall that this procedure is meant to be applied for object schemas. We describe the operations we make in the following subsections.

6.1.1. Distance

First we compute the pairwise distance between all the schemas. For example, if we are considering schemas S_a, S_b, and S_c, we obtain a symmetric 3×3 matrix. A natural approach is taking the Jaccard distance between the sets of JSON documents that the schemas accept, but this it is not immediate to define because sometimes this is too costly or the sets are infinite. Instead, we use a modified Jaccard score that is easier to compute, which allow us to get a distance. We refer to it as *similarity score* $SS(\mathbf{x},\mathbf{y})$, and the distance is defined as $D(\mathbf{x},\mathbf{y}) = 1 - SS(\mathbf{x},\mathbf{y})$.

The idea behind is as follows. For object schemas we compute a weighted Jaccard score between the sets of properties, where each key has a weight equal to the score of the associated sub-schemas. Additionally, we penalize keywords that are mandatory in precisely one schema putting an additional weight $\omega = 0.5$.

For example, consider two schemas S_a and S_b from the Figure 6.3. Let us compute the similarity score between those schemas. We realize that the union of the sets of keywords is $\{\texttt{a},\texttt{b},\texttt{c},\texttt{d},\texttt{e}\}$, and the intersection is $\{\texttt{a},\texttt{b},\texttt{c}\}$. We also know that, from the keys in the intersection, `b` and `c` are required precisely on one schema. Thus, the similarity score is computed as shown in Equation 6.1.

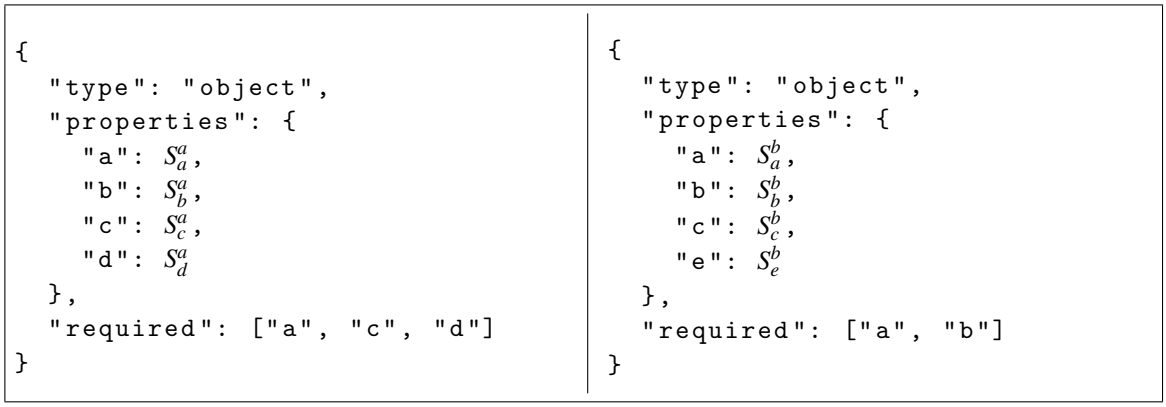


Figure 6.3. Object schemas S_a, S_b from which we compute the similarity score

$$SS(S_a, S_b) = \frac{SS(S_a^a, S_a^b) + SS(S_b^a, S_b^b) \times \omega + SS(S_c^a, S_c^b) \times \omega}{|\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}|} \tag{6.1}$$

When measuring two `"anyOf"` combinations, we compute the scores type-by-type and divide the sum of them by the number of different types considering both unions. The idea is to penalize the score if few or none of the types coincide. For example, think of the schemas S_a and S_b shown in Figure 6.4. We see that both unions have object and integer schemas, but we have four types in total (object, integer, string and null). Therefore, the similarity score is calculated as shown in Equation 6.2.

$$SS(S_a, S_b) = \frac{SS(S^a_{\text{object}}, S^b_{\text{object}}) + SS(S^a_{\text{integer}}, S^b_{\text{integer}})}{4} \tag{6.2}$$

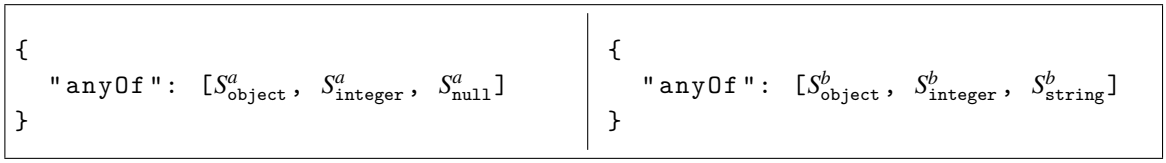


Figure 6.4. Union schemas S_a, S_b from which we compute the similarity score

We can apply the same idea to measure the distance between a union and a single schema: we consider the single schema as a union of one schema and use the before mentioned procedure. For arrays, we use the score for `"items"` sub-schemas. Finally, for

all the other types, we consider pairwise score as 1. Of course, measuring schemas with different types yields zero similarity score.

6.1.2. Clustering

Next, we need to identify clusters of *similar* schemas. We require an algorithm that can work with a matrix of distances, detects clusters with any arbitrary shape, and does not take the number of clusters to be detected as input.

In this work, we use DBSCAN (Ester, Kriegel, Sander, Xu, et al., 1996; Pedregosa et al., 2011) with parameters $\varepsilon = 0.3$ and $\mathsf{MinPts} = 5$. Admittedly, more algorithms comply with our requirements, but we left the benchmarking task as future work since, as we will see, the one we selected works reasonably well in our setup.

6.1.3. Join

Once we have identified cluster of schemas, we merge all schemas in a cluster together: If we recognize that S_a and S_b are in a cluster, then we need to produce a merged schema out of S_a and S_b. The merger is done by an algorithm similar in spirit to that of Baazizi, Ben Lahmar, et al. (2017): we compute the smallest schema in our class that is capable of representing the union of all the merged schemas. Based on Baazizi, Ben Lahmar, et al. (2017), we go over the main ideas of the merge process type by type.

First, we observe that merging string, null and boolean schemas of our class $\mathscr{L}^{\cup}$ is essentially trivial since none of them define constraints.

To represent the union of two numeric schemas in $\mathscr{N}^{\text{learn}}$, we use the schema with the least bound. For example, if we want to merge the schema {`"type":"integer", "minimum":33`} with {`"type":"integer", "minimum":-42`}, we should obtain {`"type":"integer", "minimum":-42`}. Note that, for numeric schemas, the merger is *exactly* the union of both schemas.

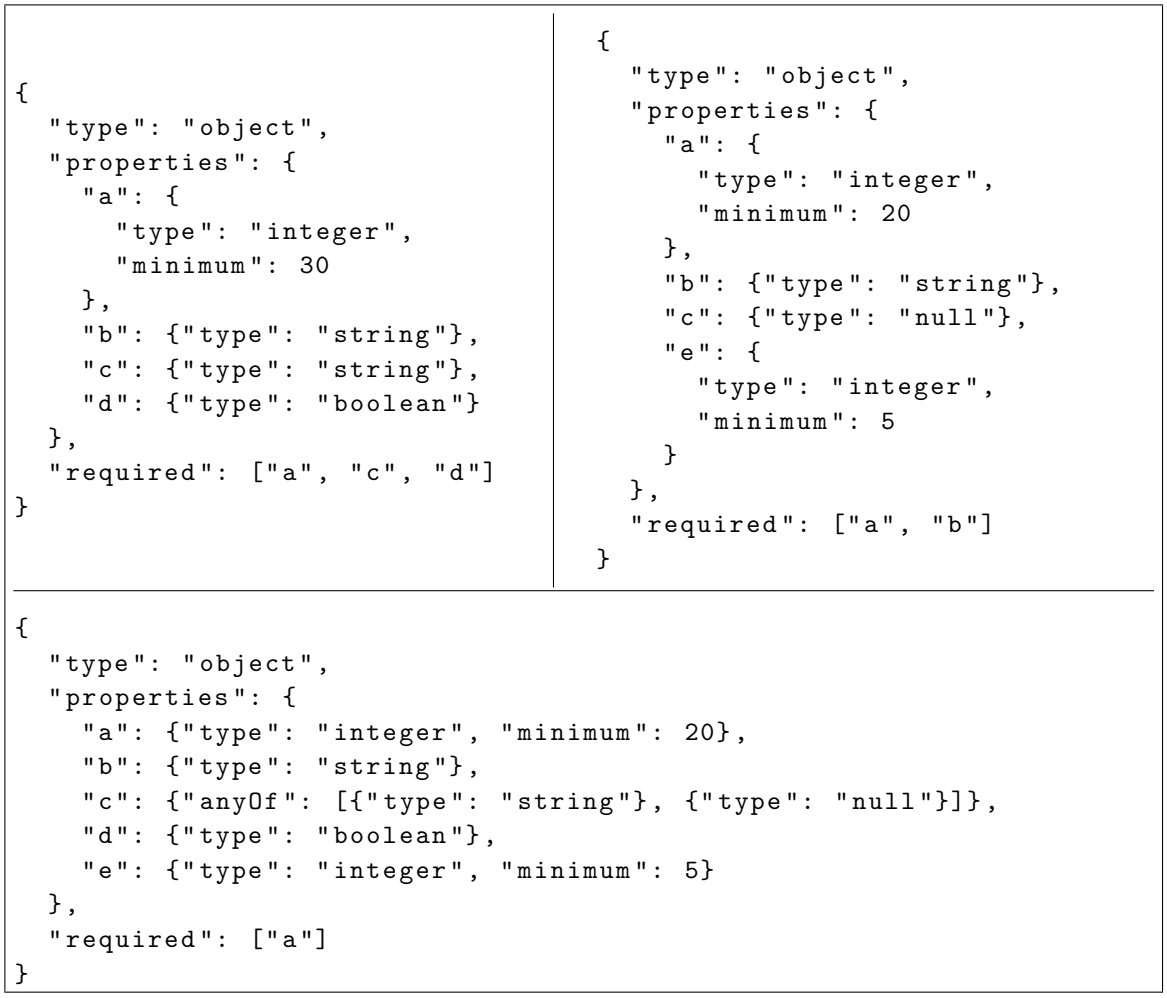
Joining two array schemas requires merging the sub-schemas under `"items"`, set the keyword `"minItems"` as the least value of both schemas, and `"uniqueItems"` to `true` if and only if both schemas have the pair `"uniqueItems":true`. For instance, if we try to merge the schemas {`"type":"array", "items":`S_1`, "minItems":3`} and {`"type":"array", "items":`S_2`, "minItems":2, "uniqueItems":true`} we obtain {`"type":"array", "items":`$merge(S_1, S_2)$`, "minItems":2`}, where $merge(\mathbf{x}, \mathbf{y})$ is the result of joining schemas **x** and **y**.

Next, we deal with boolean unions. Given two unions, we compute the merge type-wise, and then we return the union of those results. If one type is available only in one combination, we include it entirely in the union output. For example, consider the unions from Figure 6.4; the resulting merger is [$merge(S^a_{\text{object}}, S^b_{\text{object}})$, $merge(S^a_{\text{integer}}, S^b_{\text{integer}})$, S^a_{null}, S^b_{string}].

To join a boolean union with a single schema, we suppose that that a single schema is an `anyOf` combination of one schema. We apply the same principle to join two distinct types, yielding an `anyOf` of the given schemas. For example, if we try to merge an integer schema with a string schema, we get a boolean union between those schemas.

Merging object schemas is the backbone of this process. The idea is to compute the intersection between the required properties of the two schemas to merge. Then, we compute the merge recursively for each property that appears in both schemas, and we copy the corresponding sub-schema if the property appears in precisely one schema. We can see a brief example of how this is done in Figure 6.5: the top of this Figure contains two schemas that are to be merged, and the resulting schema is in the bottom.

An important point is that we must rerun the cluster-and-join procedure, because if not we would run with the same problems on a lower level of nesting. To illustrate a possible case, consider the schemas S_a and S_b from Figure 6.6. On those, we already have merged the schemas from the keys `"x"` and `"y"` in S_a, and the keys `"z"` and `"u"` in S_b.



```
{
  "type": "object",
  "properties": {
    "a": {
      "type": "integer",
      "minimum": 30
    },
    "b": {"type": "string"},
    "c": {"type": "string"},
    "d": {"type": "boolean"}
  },
  "required": ["a", "c", "d"]
}
```

```
{
  "type": "object",
  "properties": {
    "a": {
      "type": "integer",
      "minimum": 20
    },
    "b": {"type": "string"},
    "c": {"type": "null"},
    "e": {
      "type": "integer",
      "minimum": 5
    }
  },
  "required": ["a", "b"]
}
```

```
{
  "type": "object",
  "properties": {
    "a": {"type": "integer", "minimum": 20},
    "b": {"type": "string"},
    "c": {"anyOf": [{"type": "string"}, {"type": "null"}]},
    "d": {"type": "boolean"},
    "e": {"type": "integer", "minimum": 5}
  },
  "required": ["a"]
}
```

Figure 6.5. (Top) Schemas that we want to represent as one in $\mathscr{L}^{\cup}$. (Bottom) The smallest schema in $\mathscr{L}^{\cup}$ that contains the union. For `"required"` we use the intersection of the values, and for the keys under `"properties"` we use the union. If a key is present in both schemas, we represent it joining those subschemas recursively.

The merger at first glance should look like Figure 6.7 (Left). However, if we consider the case where $S^1_{merge(x,y)}$ and $S^2_{merge(z,u)}$ are similar enough to be clustered together, merging them would allow us to generate a more concise representation at the end of the process. Figure 6.7 (Right) shows the result rerunning the clustering.

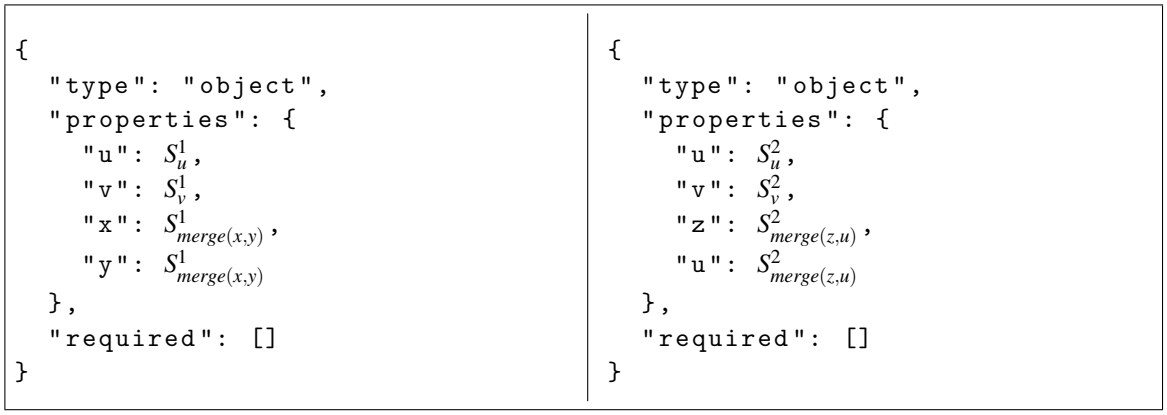


Figure 6.6. Object schemas S_a and S_b to be merged. If $S^1_{merge(x,y)}$ and $S^2_{merge(z,u)}$ are similar enough, we need to rerun the cluster-and-join process.

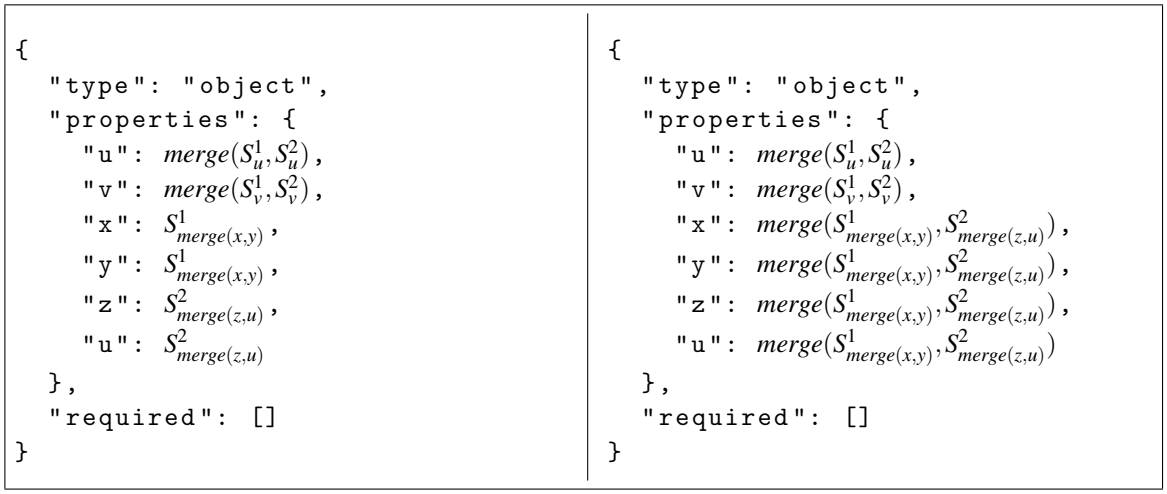


Figure 6.7. Merger schemas made out of the schemas in Figure 6.6. (Left) The resultant schema without rerunning cluster-and-join process. (Right) The resultant schema rerunning cluster-and-join process.

6.1.4. RegExp

Once the learning process ends, we attempt to represent the keys of the joined schemas as a single pattern property. Since we want to generalize, we use an algorithm capable of learning an expression from this set. Because of the use cases of JSON data, it makes sense to use the method XSYSTEM presented in (Ilyas et al., 2017). We do include a backdoor to avoid over-generalization: this backdoor involves computing a distance on intermediate

results during the XSYSTEM process; if the distance is higher than a threshold τ then we halt the process and just learn the enumeration of all examples[1].

To see how this last step is done, let us use the merger schema at the right of Figure 6.7. We already know that the schemas associated to the keys "u" and "v" are the same because they were merged. The same applies for the schemas under "x", "y", "z", and "u". Then, we try to learn a regular expression for those sets of keys, and use it as key in a pattern property with the merger schema as value, as it is possible to observe in Figure 6.8

```
{
  "type": "object",
  "properties": {},
  "patternProperties": {
    regex(u, v): merge(S^1_v, S^2_v),
    regex(x, y, z, u): merge(S^1_merge(x,y), S^2_merge(z,u))
  }
  "required": []
}
```

Figure 6.8. Merger schema from (Right) Figure 6.7, using pattern properties as a mean to make a compact representation. The function $regex(\mathbf{w})$ computes a regular expression from the set $\mathbf{w}$.

To be able to use this procedure recursively, we need to preserve the properties of the class $\mathscr{L}^{\cup}$ we defined. That is why we perform this stage only once all the learning process has finished.

6.1.5. Using the heuristic in the general algorithm

Using the before mentioned improvements requires to add the heuristic routines to the end of the algorithm that processes objects (Algorithm 1). Thus, the Algorithm 4 (which implements these changes) should be used in place of Algorithm 1 when using cluesting-and-join enhancement.

[1]Parameters for XSYSTEM: τ: 0.8, capture threshold: 1, p-value: 0.05, branching threshold: 0.5.

Algorithm 4: Algorithm to learn object schemas in $\mathscr{L}^{\cup}$, using the clustering-and-join improvement. `Learn` refers to the general learning algorithm (Algorithm 3).

```
Data: A set D of object JSON documents
Result: A object schema in L^∪ using clustering-and-join improvement

properties ← map(), schemas ← map();
required ← [key | ∀d ∈ D d.key exists];

foreach document d in D do
    foreach key, value in d do
        properties[key].add(value);

foreach key, documents in properties do
    schemas[key] = Learn(documents)

distances ← MatrixDistance(schemas);
clusters ← MakeClusters(schemas, distances);

foreach cluster in clusters do
    schema_cluster = cluster.schemas.first;
    foreach schema in cluster.schemas do
        schema_cluster = Merge(schema_cluster, schema);
    foreach key in cluster.keys do
        schemas[key] = schema_cluster;

return {
        "type":"object", "properties": schemas,
        "required": required, "additionalProperties":false
        };
```

Also, to produce the compact representation using pattern properties we need a new point of entry to ensure that the regex learning algorithm is applied only after all the learning process. For this, we create a new routine that first calls the original learning algorithm (Algorithm 3), and then, calls a procedure that traverses all the structure searching for clustered schemas and putting them together in a pattern property. Algorithm 5 roughly shows the new main function.

Algorithm 5: Main function of the learning algorithm when using clustering-and-join improvement. `Learn` refers to the Algorithm 3.

Data: A set D of JSON documents
Result: An schema using cluesting-and-join approach

schema $\leftarrow$ `Learn(`D`)`;
schema $\leftarrow$ `TraverseAndCreatePatternProperties(`*schema*`)`;

return *schema*;

6.2. Experiments

Our first task is to gauge the effectiveness of our heuristic-based improvement by attempting to learn the schema for the Wikidata database (`www.wikidata.org`), the central data storage for Wikimedia and one of the most important use cases of JSON databases with a size of over 400 GB. There is no current official JSON Schema for Wikidata, and learning its schema gives us an interesting use case for our cluster-and-join approach because we need to be able to generalise over keys itself. We do not provide a full description of the database, but rather concentrate on what is important for establishing our use case. We defer interested readers to Vrandečić and Krötzsch (2014). The idea of Wikidata is to store data in form of two different entities: items and properties. Properties link items together, and they have their own identifier in the database; all of them are of the form P[0-9]+. For example, P6 is the "head of goverment" property, and P123 is "publisher". The important thing for us is that item entities have a section where all properties linking this item with others are listed as keys, followed with information about the nature of the property (so that in the document for Donald Trump we find a key:value pair whose key is `"P6"`, the information here would be for example the year a president came to office). Now, if we are to learn a schema such as Wikidata's, it is important that we realise that documents following `"P6"` are under the same schema that documents following, say, `"P123"`, so that we can include a statement of the form `"^P[0-9]+$":"{S}"`.

To learn the schema of wikidata, we obtain a random sample of 0.1% of the entities in the dataset, and feed them to the learning process. Impressively, the recently presented cluster-and-join approach with pattern properties produces a more concise—yet expressive—schema for Wikidata entities, and in particular manages to capture an expression that succesfully deal with the issue described above (see Figure 6.9).

```
"claims": {
  "type": "object",
  "patternProperties": {
    "^(P6|P[1-4]\\d{3}|P[1-9]\\d{2}|P\\d{2})$": {
      "type": "array",
      "items": { ... },
      "minItems": 1
    }
  },
  "required": [],
  "additionalProperties": false
}
```

Figure 6.9. Claims subschema for Wikidata entities. It uses only pattern properties since all the appearing properties had a similar schema, and thus, were clustered together. Using the algorithm from Section 4.2, the generated schema had 2001 properties with redundant subschemas.

As a proof of concept, we also ran our cluster-and-join approach to learn the schemas for OpenWeather, Github and Twitter presented in Chapter 5. Here we observed that both algorithms generate schemas that are almost equivalent, but the cluster-and-join approach grouped equivalent subschemas, therefore making the representation more compact. For OpenWeather and Twitter datasets, the created schemas were not exactly equivalent because sometimes the regular expressions in `"patternProperties"` accepted more words than the ones we provided. In Table 6.1 we compare the size of the schemas using both algorithms.

Table 6.1. Comparison between size of schemas (characters, without whitespaces) using both the method presented in Section 4.2 (†) and Cluster-and-Join (◊).

Dataset	**Method 1†**	**Method 2◊**	**% var**
GitHub	1627	1459	-10.3
OpenWeather	2341	2139	-8.6
Twitter	54485	49164	-9.8
Wikidata	4400078	13390	-99.7

7. CONCLUDING REMARKS

We performed an analysis of JSON Schema learnability. We first introduced the identifiability in the limit from positive samples framework, according to previous work with JSON documents and other similar formats like XML.

Regarding learnability of JSON schema, it is immediate to prove that the full class is not identifiable in the limit. That is why this work concentrated on identifying fragments of the specification and check their learnability under Gold's framework. We studied fragments grouped by type (e.g., numbers, strings, objects, arrays, among others), presenting both positive and negative theoretical results.

For numeric data, we studied intervals depending on their bounds. In general, we discovered that one could not expect to learn intervals with a different number of bounds. For example, it is impossible to mix finite intervals with infinite ones. Similarly, we cannot mix unbounded intervals with intervals which have either an infimum or supremum.

About strings, we immediately stated that we could not learn the full class of string schemas due to the presence of regular expressions. That is why we focused on two learnable classes, one for infinite sets of strings, and other for finite ones; where the latter class could include regular expressions since they defined finite languages.

Next, with objects, we introduced a level-to-level procedure to reach desirable results. That means that each property value in the object has to be valid against precisely one schema, which could be of any identifiable type. We also argued why it is necessary to close the schemas by not including additional properties nor pattern properties. With those restrictions, we achieved a learnable class for objects with the remaining keywords.

Arrays were particularly interesting due to their similarity with objects, in the sense of storing more nested data, and with strings, in the sense of length limitations. In this case, we managed to create two learnable classes with almost all the relevant keywords, one for

length-bounded arrays and other for arrays of arbitrary size. Each class is identifiable in the limit by separate, but the union of them is not.

With boolean combinations, we chose a safe approach. Arbitrary unions of different schemas can potentially lead to unlearnability, and then each possible combination should be studied thoughtfully. Instead of doing so, we provided a positive result with the union of distinct types. We also highlighted that we could choose a class closed under intersection, and then, add intersections at no cost.

With theoretical results, we dived in one of few schema repositories in order to understand how are schemas used. The most remarkable findings are that arrays are supposed to have elements of the same type, objects do not define separate bounds for the number of properties, and practically nobody uses multiple constraints for numbers. Our conclusions helped to devise a learnable JSON Schema class useful in real-world situations and a simple algorithm to learn it under the proposed framework.

We showed how the algorithm works with three real samples. They confirmed that, in general, assuming level-by-level structure and uniformity in arrays is safe. In practical terms, the results proved to be reasonable. Nevertheless, there is room for improvement. For instance, we noted that in the same schemas there are repeated patterns; therefore, it would be interesting to identify them as the same subschema. Also, the generated schemas are verbose, and a method to generate a more compact representation is needed.

Finally, we addressed the verbosity of the output using a clustering-and-join approach inside object schemas, representing similar schemas like one. We are satisfied with the results that this little improvement can achieve. Moreover, we think that we are close to being ready to deploy this method and start working online to increase schema metadata.

7.1. Future work

As we said earlier, the algorithm proved to be useful in the tests we ran, but there was room for improvement. In particular, using JSON Schema for document APIs has the shortcoming that we do not specify the relation between the input and the output. Thus, if we query Twitter API with a user, then this is the same user that will be shown in the JSON output of the API. There are some proposals to address this issue, this is also part of the OpenAPI desiderata, and there is even an extension of JSON Schema called JSON Hyper-Schema. It remains to see how these specifications can be learned from examples, and whether the approach we have presented generalizes for JSON Hyper-Schema.

There is also room for improvement regarding making schemas more compact. For instance, we did not examine similar sub-schemas across all the levels of the schema, nor moved them to a `"definitions"` section to reference those with JSON Pointer.

Another potential future contribution is to create a framework to benchmark learner algorithms from positive examples, concerning efficiency in time, space, and the number of needed examples to give a good result.

We also should explore more algorithms out of the Gold's framework. The framework is intended to put a starting point and to generate an idea of what we want to learn, but if we are going to learn broader classes, we need to introduce PAC (probably approximately correct learning) algorithms as we briefly did in Chapter 6.

Finally, we think it is desirable to study more in deep closure properties for JSON Schema classes. For example, it would be interesting to see a class closed under complement, union and intersection.

REFERENCES

Baazizi, M. A., Ben Lahmar, H., Colazzo, D., Ghelli, G., & Sartiani, C. (2017). Schema inference for massive JSON datasets. In *Proceedings of the 20th international conference on extending database technology, EDBT 2017, venice, italy, march 21-24, 2017*. (pp. 222–233).

Baazizi, M. A., Colazzo, D., Ghelli, G., & Sartiani, C. (2017). Counting types for massive JSON datasets. In *Proceedings of the 16th international symposium on database programming languages, DBPL 2017, munich, germany, september 1, 2017* (pp. 9:1–9:12).

Bex, G. J., Gelade, W., Neven, F., & Vansummeren, S. (2010, September). Learning deterministic regular expressions for the inference of schemas from xml data. *ACM Trans. Web*, *4*(4), 14:1–14:32. Retrieved from `http://doi.acm.org/10.1145/1841909.1841911` doi: 10.1145/1841909.1841911

Bex, G. J., Neven, F., Schwentick, T., & Tuyls, K. (2006). Inference of concise dtds from xml data. In *Proceedings of the 32nd international conference on very large data bases* (pp. 115–126). VLDB Endowment. Retrieved from `http://dl.acm.org/citation.cfm?id=1182635.1164139`

Bex, G. J., Neven, F., Schwentick, T., & Vansummeren, S. (2010, May). Inference of concise regular expressions and dtds. *ACM Trans. Database Syst.*, *35*(2), 11:1–11:47. Retrieved from `http://doi.acm.org/10.1145/1735886.1735890` doi: 10.1145/1735886.1735890

Bex, G. J., Neven, F., & Vansummeren, S. (2007). Inferring xml schema definitions from xml data. In *Proceedings of the 33rd international conference on very large data bases* (pp. 998–1009). VLDB Endowment. Retrieved from `http://dl.acm.org/citation.cfm?id=1325851.1325964`

Bourhis, P., Reutter, J. L., Suárez, F., & Vrgoč, D. (2017). JSON: data model, query languages and schema specification. *CoRR*, *abs/1701.02221*. Retrieved from `http://arxiv.org/abs/1701.02221`

De La Higuera, C. (1997). Characteristic sets for polynomial grammatical inference. *Machine Learning*, *27*(2), 125–138.

Denis, F., Lemay, A., & Terlutte, A. (2002). Some classes of regular languages identifiable in the limit from positive data. In *International colloquium on grammatical inference* (pp. 63–76).

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd* (Vol. 96, pp. 226–231).

Fernau, H. (2009). Algorithms for learning regular expressions from positive data. *Information and Computation*, *207*(4), 521–541.

Freydenberger, D. D., & Kötzing, T. (2015). Fast learning of restricted regular expressions and dtds. *Theory of Computing Systems*, *57*(4), 1114–1158.

Frozza, A. A., dos Santos Mello, R., & da Costa, F. d. S. (2018). An approach for schema extraction of json and extended json document collections. In *2018 ieee international conference on information reuse and integration (iri)* (pp. 356–363).

Gold, E. M., et al. (1967). Language identification in the limit. *Information and control*, *10*(5), 447–474.

Google LLC. (2018). *API Discovery Service: Programmatically read metadata about Google APIs.* Retrieved 2018-10-16, from `https://developers.google.com/discovery/`

Ilyas, A., da Trinidade, J. M., Fernandez, R. C., & Madden, S. (2017). Extracting syntactic patterns from databases. *arXiv preprint arXiv:1710.11528*.

Izquierdo, J. L. C., & Cabot, J. (2013). Discovering implicit schemas in json data. In *International conference on web engineering* (pp. 68–83).

Jing, H., Haihong, E., Guan, L., & Jian, D. (2011, Oct). Survey on NoSQL database. In *2011 6th international conference on pervasive computing and applications* (p. 363-366). doi: 10.1109/ICPCA.2011.6106531

JSON-schema-org. (2018). *The home of JSON schema.* Retrieved 2018-10-16, from `http://json-schema.org`

Klettke, M., Störl, U., & Scherzinger, S. (2015). Schema extraction and structural outlier detection for json-based nosql data stores. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*.

Kristensen, M., et al. (2018). *Schema Store: a collection of JSON schema files including full API.* Retrieved 2018-10-14, from `http://schemastore.org/json/`

Mohri, M., Rostamizadeh, A., Talwalkar, A., & Bach, F. (2012). *Foundations of machine learning*. MIT Press. Retrieved from `https://books.google.cl/books?id=-ijiAgAAQBAJ`

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., & Vrgoč, D. (2016). Foundations of JSON schema. In *Proceedings of the 25th international conference on world wide web* (pp. 263–273). Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee. Retrieved from `https://doi.org/10.1145/2872427.2883029` doi: 10.1145/2872427.2883029

Snowplow Analytics Ltd. (2016). *Schema Guru.* Retrieved 2018-10-16, from `https://github.com/snowplow/schema-guru`

Suárez Barría, F. (2016). *Formal specification, expressiveness and complexity analysis for JSON schema* (Master's thesis, Pontificia Universidad Católica de Chile. Escuela de Ingeniería). Retrieved 2018-10-16, from https://repositorio.uc.cl/handle/11534/16908

The Linux Foundation. (2018). *OpenAPI Initiative.* Retrieved 2018-10-16, from https://www.openapis.org

Vrandečić, D., & Krötzsch, M. (2014, September). Wikidata: A free collaborative knowledgebase. *Commun. ACM*, *57*(10), 78–85. Retrieved from http://doi.acm.org/10.1145/2629489 doi: 10.1145/2629489

Wang, L., Zhang, S., Shi, J., Jiao, L., Hassanzadeh, O., Zou, J., & Wangz, C. (2015). Schema management for document stores. *Proceedings of the VLDB Endowment*, *8*(9), 922–933.

Wischenbart, M., Mitsch, S., Kapsammer, E., Kusel, A., Pröll, B., Retschitzegger, W., ... Lechner, S. (2012). User profile integration made easy: model-driven extraction and transformation of social network schemas. In *Proceedings of the 21st international conference on world wide web* (pp. 939–948).

Wootton, J. (2017). *JSONschema.net.* Retrieved 2018-10-16, from https://www.jsonschema.net

Wright, A., & Andrews, H. (2018). *JSON Schema: A Media Type for Describing JSON Documents.* Retrieved 2018-10-16, from http://json-schema.org/latest/json-schema-core.html

APPENDIX

A. PROOFS

A.1. Proof of Proposition 3.2

Algorithm 6 has the procedure to learn $\mathscr{N}_{\text{min-max}}$ and it is straightforward to check that this algorithm satisfies the desired properties. The other three classes $\mathscr{N}_{\_}$, $\mathscr{N}_{\text{min}}$, $\mathscr{N}_{\text{max}}$ can be learned with similar algorithms.

Algorithm 6: Algorithm to learn $\mathscr{N}_{\text{min-max}}$ schemas

Data: A set D of number JSON documents
Result: A schema in $\mathscr{N}_{\text{min-max}}$

minValue, maxValue $\leftarrow \infty, -\infty$;
multipleOf $\leftarrow \gcd\{x \in D, x \neq 0\})$;

foreach *e in the set D* **do**
    minValue $\leftarrow \min(\text{minValue}, e)$;
    maxValue $\leftarrow \max(\text{maxValue}, e)$;

return {
    `"type":"number"`, `"multipleOf"`: multipleOf
    `"minimum"`: minValue, `"maximum"`: maxValue
    };

To show that $(\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\text{max}})$ is learnable in the limit, we design an algorithm that complies with Gold's framework to detect that class of schemas. To do this, we are going to assume that the value of `"multipleOf"` is correctly detected from the provided examples. Let us say that this value is k. The core of this proof is making an injective mapping from $\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\text{max}}$ schemas to two elements that should appear in their respective characteristic sets. We introduced this mapping in a sketch, but we rewrite it here for completeness.

$$f(S) = \begin{cases} f_{\text{min}}(S) & S \in \mathscr{N}_{\text{min}} \\ f_{\text{max}}(S) & S \in \mathscr{N}_{\text{max}} \end{cases}$$

$$f_{\min}(S) = \begin{cases} bound(S), k \times (2 \times bound(S) + 1) & bound(S) \geq 0 \\ bound(S), -k \times (2 \times bound(S)) & bound(S) < 0 \end{cases}$$

$$f_{\max}(S) = \begin{cases} bound(S), -k \times (2 \times bound(S) + 1) & bound(S) \geq 0 \\ bound(S), k \times (2 \times bound(S)) & bound(S) < 0 \end{cases}$$

Every schema accepts the pair this mapping provides for it. For instance, if the schema has a lower bound, the generated pair includes the bound itself and a higher value. If the schema has an upper bound, the generated pair also includes the bound itself, but instead, it includes a lower value. Indeed, let us remember that it is possible to assume w.l.o.g that the bound is multiple of k. Assuming this, the gcd of the generated numbers is k. Hence, they are sufficient as a characteristic set.

In order to create an algorithm that takes advantage of this mapping, we need to create the inverse relationship between a pair of numbers and its associated schema:

$$f^{-1}(a,b) = \begin{cases} S \in \mathscr{N}_{\min},\ bound(S) = a & b = k \times (2 \times a + 1) \\ S \in \mathscr{N}_{\min},\ bound(S) = a & b = -k \times (2 \times a) \\ S \in \mathscr{N}_{\max},\ bound(S) = a & b = -k \times (2 \times a + 1) \\ S \in \mathscr{N}_{\max},\ bound(S) = a & b = k \times (2 \times a) \end{cases}$$

By construction is clear that, for each interval, exists a set of two elements which identifies it. Now, we show that it is possible to identify the interval given any superset of the characteristic pair of elements. Without loss of generality, let k be 1. Assume that S is the target interval, D_S is its characteristic set, and the sample is D'_S such that $D_S \subseteq D'_S \subset S$. Let a be the minimum element in D'_S and b the maximum. It is guaranteed that one of them is the bound of the interval.

(i) If a is the bound, then there exists an element $c \in D'_S$ such that $f^{-1}(a,c)$ is defined. That is, $c = 2 \times a + 1$ if $a \geq 0$, or $c = -2 \times a$ if $a < 0$.

(ii) If b is the bound, then there exists an element $c' \in D'_S$ such that $f^{-1}(b,c')$ is defined. That is, $c' = 2 \times b$ if $b < 0$, or $c' = -2 \times b + 1$ if $b > 0$.

One cannot hope to identify both situations. Consider these cases by contradiction:

(i) $a \geq 0$: then, $b \geq 0$. So, there exists $c = 2 \times a + 1$ and $c' = -2 \times b + 1$. Since $a \leq c'$, then this situation is impossible.

(ii) $b < 0$: then, $a < 0$. So, there exists $c = -2 \times a$ and $c' = 2 \times b$. Since $b \geq c$, then this situation is impossible.

(iii) $a < 0$ but $b \geq 0$. So, there exists $c = -2 \times a$ and $c' = -2 \times b + 1$. The following holds simultaneously:

$$\begin{aligned} c &\leq b \\ -2 \times a &\leq b \\ a &\geq \frac{-b}{2} \end{aligned} \qquad\qquad \begin{aligned} a &\leq c' \\ a &\leq -2 \times b + 1 \end{aligned}$$

Since $b \geq 0$ and $a \neq b$, the situation is impossible.

The previous proof gave us a procedure to identify a schema from a sample. Thus, we use the recently constructed relationship in the Algorithm 7. First, the algorithm computes the gcd of the provided examples; it is assumed to be correct. This algorithm checks all the possible pairs repect the extreme values in the provided examples, and if f^{-1} yields a schema, it is returned.

Finally, to accept every example currently provided (without regard to conforming a superset of a characteristic set or not), we return a $\mathscr{N}_{\text{min}}$ schema consistent with the sample if there is no match.

Algorithm 7: Algorithm to learn $\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\text{max}}$ schemas

Data: A set D of number JSON documents
Result: A schema in $\mathscr{N}_{\text{min}} \cup \mathscr{N}_{\text{max}}$

k $\leftarrow$ gcd($\{x \in D, x \neq 0\}$);
minValue $\leftarrow$ min(D);
maxValue $\leftarrow$ max(D);

foreach *x in the set D* **do**
    schema $\leftarrow f^{-1}(minValue, x)$;
    if *schema* $\neq$ `null` **then**
        return *schema*;
    schema $\leftarrow f^{-1}(maxValue, x)$;
    if *schema* $\neq$ `null` **then**
        return *schema*;

return {`"type":"integer", "minimum":` min(D)};

A.2. Proof of Proposition 3.3

It is easy to create an algorithm to learn a $\mathscr{S}_{\text{min}}$, since we just need to keep the shortest word and save its length. For $\mathscr{S}^{\text{pattern}}_{\text{min-max}}$ schemas, they are trivially learnable in the limit as the enumeration of the sample data, since these kind of schemas only define finite sets of strings.

Assume, by contradicition, that $\mathscr{S}_{\text{min}} \cup \mathscr{S}^{\text{pattern}}_{\text{min-max}}$ is identifiable in the limit. Let S be an schema in $\mathscr{S}_{\text{min}}$, and let D_S be its characteristic set. Now, let S' be an schema in $\mathscr{S}^{\text{pattern}}_{\text{min-max}}$ such that it is consistent with D_S, and let $D_{S'}$ be its characteristic set.

We can guarantee that S' exists since we only need to set the minimum length and maximum length according to D_S and choose the wildcard $*$ as the pattern. Also, we can assure that S is consistent with $D_{S'}$ since the language that S' describes is a subset of the language that S describes.

Now, let us check what happens if the learning algorithm receives $D' = D_S \cup D_{S'}$. The algorithm should return S since D' is a superset of D_S and S is consistent with D'.

However, the algorithm should return S' at the same time by similar arguments. This situation constitutes a contradiction.

A.2.1. Proof of Proposition 3.4

Let us assume that the class $\mathscr{O}^{+}_{\text{prop}}$ is learnable in the limit. In particular, in this class we can find the schema that accepts every document (S_{JSON}), since we just need to set the value of `"properties"` to an empty array and `"additionalProperties"` to `true`.

Since S_{JSON} is learnable in the limit, there is a characteristic set $D_{S_{\mathsf{JSON}}}$. We can construct another schema $S' \in \mathscr{O}^{+}_{\text{prop}}$ such that it is consistent with $D_{S_{\mathsf{JSON}}}$, for instance, the schema that names some properties seen in the examples (with their respective compliant subschema) and allows additional properties. Let $D_{S'}$ be the characteristic set of S'.

Let us check what happens if we have $D_{S_{\mathsf{JSON}}} \cup D_{S'}$ as input for the algorithm. On one side, the algorithm should give S' as a result, since the input is superset of its characteristic set, and consistent with that schema. On the other hand, the algorithm should yield S_{JSON} under the same arguments. This situation leads to a contradiction, then $\mathscr{O}^{+}_{\text{prop}}$ is not identifiable in the limit.

Now let us check the second part of the proposition. Let $\mathscr{C}$ be a class including all the schemas from $\mathscr{O}_{\text{prop}}$ and one schema $S^{+} \in \mathscr{O}^{+}_{\text{prop}}$. By contradicition, let us suppose the class $\mathscr{C}$ is identifiable in the limit.

Let $D_{S^{+}}$ be the characteristic set of S^{+}. One may create an schema $S' \in \mathscr{O}_{\text{prop}}$ that is consistent with $D_{S^{+}}$, leading to the same problem shown in the previous proof.

A.3. Proof of Proposition 3.6

Let us assume by contradicition that $\mathscr{A}^{\min} \cup \mathscr{A}^{\min-\max}$ is identifiable in the limit. Let S be a schema of $\mathscr{A}^{\min}$ class, and let D_S be its characteristic set. Let us construct a S' schema in $\mathscr{A}^{\min-\max}$ such that it is consistent with D_S. To doing so, simply check the

shortest and longest array in D_S and bound S' accordingly. Let $D_{S'}$ be the characteristic set of the constructed schema.

So, what does happen if we provide $D' = D_S \cup D_{S'}$ to the learner algorithm? For one part, the algorithm should return S since D' is superset of D_S and S is still consistent with D'. On the other hand, D' is superset of $D_{S'}$ and S' is consistent with D', then the algorithm should return S'. This constitutes a contradiction, therefore $\mathscr{A}^{\min} \cup \mathscr{A}^{\min-\max}$ is not identifiable in the limit.

A.4. Proof of Proposition 3.7

We provide the Algorithm 8 to learn $\mathscr{A}^{\text{min-max}}_{\text{items}}$ schemas. With minor modifications we can provide the Algorithm 9 to learn $\mathscr{A}^{\min}_{\text{items}}$ schemas.

Algorithm 8: Algorithm to learn $\mathscr{A}^{\text{min-max}}_{\text{items}}$ schemas

```
Data: A set D of array JSON documents
Result: A schema in A^min-max_items
examples_by_position ← ArrayOfSets();
minLength ← ∞, maxLength ← 0 items ← list(), uniqueItems ← true;
foreach e in the set D do
    minLength ← min(minLength, Length(e));
    maxLength ← min(maxLength, Length(e));
    uniquItems ← uniqueItems ∧ UniqueElements(e);
    for i = 0; i < Length(e); i ← i + 1 do
        example ← e[i];
        AddToSet(examples_by_position[i], example);
for i = 0; i < Length(examples_by_position); i ← i + 1 do
    items[i] ← Learn(examples_by_position[i]);
return {
        "type":"array", "items": items,
        "minItems": minLength, "maxItems": maxLength,
        "uniqueItems":uniqueItems
        };
```

Algorithm 9: Algorithm to learn $\mathscr{A}^{\text{min}}_{\text{items}}$ schemas

Data: A set D of array JSON documents
Result: A schema in $\mathscr{A}^{\text{min}}_{\text{items}}$

examples_by_position ← `ArrayOfSets()`;
minLength ← ∞ items ← `list()`, uniqueItems ← `true`;

foreach *e in the set D* **do**
    minLength ← min(minLength, `Length(`e`)`);
    uniqueItems ← uniqueItems ∧ `UniqueElements(`e`)`;
    for $i = 0$*;* $i <$ `Length(`e`)`*;* $i \leftarrow i + 1$ **do**
        example ← e[i];
        `AddToSet(`*examples_by_position[i], example*`)`;

for $i = 0$*;* $i <$ `Length(`*examples_by_position*`)`*;* $i \leftarrow i + 1$ **do**
    items[i] ← `Learn(`*examples_by_position[i]*`)`;

return {
    `"type":"array"`, `"items":` items,
    `"minItems":` minLength, `"additionalItems":` items.last(),
    `"uniqueItems"`:uniqueItems
    };

A priori, the characteristic set for a $\mathscr{A}^{\text{min-max}}_{\text{items}}$ schema is a collection of arrays such that each inner schema can be described via examples, at least one array of minimum length and at least one with the maximum permitted. When `"uniqueItems"` is `false`, we need only $1 + \max(\{D_S \mid S \text{ is an inner schema}\})$ examples, but when it is `true`, we might need more examples.

Example. *Consider the schema in the Figure A.1. Which is its characteristic set? First, we have to include an empty array in order to detect the minimum length. Second, we have to provide enough examples for each position (1, 2, 3, 4, 5 and beyond) in order to help the algorithm to identify those subschemas. So far, our characteristic set has four elements: the empty array and three examples that are enough to make room for the largest characteristic set of the involved subschemas, with nearly no restrictions on how to construct the example arrays.*

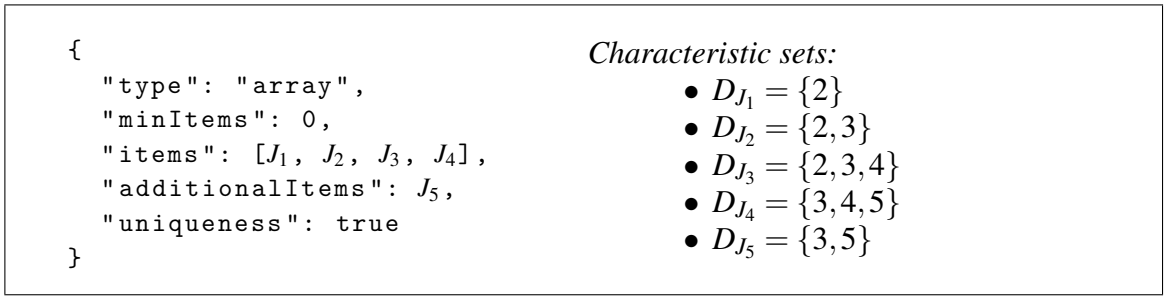


Figure A.1. (Left) an array schema of $\mathscr{A}^{\min}_{items}$ class. (Right) characteristic sets for the subschemas.

Things become more complicated when we deal with `"uniqueness":true`*. Immediately, we see that is impossible to construct a single example using only elements in the characteristic sets of the subschemas. Let us introduce* β*, such that* J_5 *is consistent with this number. Additionally, let us assume that* $\beta \notin \{2,3,4,5\}$*. Then, we can construct the example* $[2,3,4,5,\beta]$*. Now we have exhausted all the elements of* D_{J_1}*, so we have to use another element consistent with* J_1*, let* δ *be this additional element. Assuming that* $\delta \notin \{2,3,4,5\}$*, we can provide the examples* $[\delta,2,3,4,5]$*,* $[\delta,3,2,4,5]$*,* $[\delta,2,4,3,5]$ *and* $[\delta,2,4,5,3]$*. As observed, we made strong assumptions on* J_1 *and* J_5 *in order to make a small characteristic set for the schema in Figure A.1, but the impossibility of fulfilling those assumptions could lead to a sizeable characteristic set.*

We can follow a similar reasoning to provide a characteristic set for $\mathscr{A}^{\min}_{\text{items}}$ schemas.

B. EXTRA RESULTS

B.1. Keeping track of max/min values is mandatory

PROPOSITION.

- *The characteristic set of any $\mathscr{N}_{min}$ schema should include the smallest value.*
- *The characteristic set of any $\mathscr{N}_{max}$ schema should include the greatest value.*

PROOF. We will prove the first part of the proposition since the second is entirely analogous.

Let S_n be a schema with `"minimum":`n and `"multipleOf": 1`. Also, let S_{n+1} be a schema with `"maximum":`n and `"multipleOf": 1`. Let D_{S_n} and $D_{S_{n+1}}$ be their respective characteristic sets.

The first fact is that D_{S_n} should not be consistent with S_{n+1}. If it were, the learning algorithm with input $D = D_{S_n} \cup D_{S_{n+1}}$ should have to return S_n, since D is superset of D_{S_n} and subset of S_n; however, the algorithm should have to return S_{n+1}, since D is superset of $D_{S_{n+1}}$ and subset of S_{n+1}.

So, one might wonder how to avoid that situation. The number n is the only in S_n that is not in S_{n+1}, and hence, n should be in D_{S_n} in order to avoid the previously presented situation.

For simplicity reasons, in this proof we deliberately avoided working with `"multipleOf"`. Let us suppose that `"multipleOf"` is k, then our proof have to compare S_n and S_{n+k}, where n is multiple of k. This is so because we realized that it is not necessary to work with cases where the minimum is not multiple of the value presented in `"multipleOf"`, given that the smallest element is the first that is multiple of that number and greater than the minimum bound. □

C. SOME OF THE EXTRACTED SCHEMAS

C.1. OpenWeather schema extracted in Section 5.1


```
{
  "additionalProperties": false,
  "properties": {
    "city": {
      "additionalProperties": false,
      "properties": {
        "coord": {
          "additionalProperties": false,
          "properties": {
            "lat": {
              "minimum": -36.906,
              "type": "number"
            },
            "lon": {
              "minimum": -156.9223,
              "type": "number"
            }
          },
          "required": [ "lat", "lon" ],
          "type": "object"
        },
        "country": {
          "type": "string"
        },
        "id": {
          "minimum": 207570,
          "type": "integer"
        },
        "name": {
          "type": "string"
        }
      },
      "required": [ "coord", "country", "id", "name" ],
      "type": "object"
    },
    "cnt": {
      "minimum": 38,
      "type": "integer"
    },
    "cod": {
      "type": "string"
    },
    "list": {
      "items": {
        "additionalProperties": false,
        "properties": {
          "clouds": {
            "additionalProperties": false,
            "properties": {
              "all": {
                "minimum": 0,
                "type": "integer"
              }
            },
            "required": [ "all" ],
            "type": "object"
          },
          "dt": {
            "minimum": 1540188000,
            "type": "integer"
          },
          "dt_txt": {
            "type": "string"
          },
          "main": {
            "additionalProperties": false,
            "properties": {
              "grnd_level": {
                "minimum": 544.55,
                "type": "number"
              },
              "humidity": {
                "minimum": 23,
                "type": "integer"
              },
              "pressure": {
                "minimum": 544.55,
                "type": "number"
              },
```

```
            "sea_level": {
              "minimum": 990.86,
              "type": "number"
            },
            "temp": {
              "minimum": 249.085,
              "type": "number"
            },
            "temp_kf": {
              "minimum": -4.13,
              "type": "number"
            },
            "temp_max": {
              "minimum": 249.085,
              "type": "number"
            },
            "temp_min": {
              "minimum": 249.085,
              "type": "number"
            }
          },
          "required": [
            "grnd_level",
            "humidity",
            "pressure",
            "sea_level",
            "temp",
            "temp_kf",
            "temp_max",
            "temp_min"
          ],
          "type": "object"
        },
        "rain": {
          "additionalProperties": false,
          "properties": {
            "3h": {
              "minimum": 0.002,
              "type": "number"
            }
          },
          "required": [],
          "type": "object"
        },
        "snow": {
          "additionalProperties": false,
          "properties": {
            "3h": {
              "minimum": 0.0005,
              "type": "number"
            }
          },
          "required": [],
          "type": "object"
        },
        "sys": {
          "additionalProperties": false,
          "properties": {
            "pod": {
              "type": "string"
            }
          },
          "required": [ "pod" ],
          "type": "object"
        },
        "weather": {
          "items": {
            "additionalProperties": false,
            "properties": {
              "description": {
                "type": "string"
              },
              "icon": {
                "type": "string"
              },
              "id": {
                "minimum": 500,
                "type": "integer"
              },
              "main": {
                "type": "string"
              }
            },
            "required": [ "description", "icon", "id", "main" ],
            "type": "object"
          },
          "minItems": 1,
          "type": "array",
```

```
            "uniqueItems": true
          },
          "wind": {
            "additionalProperties": false,
            "properties": {
              "deg": {
                "minimum": 0.000274658,
                "type": "number"
              },
              "speed": {
                "minimum": 0,
                "type": "number"
              }
            },
            "required": [ "deg", "speed" ],
            "type": "object"
          }
        },
        "required": [
          "clouds",
          "dt",
          "dt_txt",
          "main",
          "sys",
          "weather",
          "wind"
        ],
        "type": "object"
      },
      "minItems": 38,
      "type": "array"
    },
    "message": {
      "minimum": 0.0018,
      "type": "number"
    }
  },
  "required": [ "city", "cnt", "cod", "list", "message" ],
  "type": "object"
}
```

C.2. OpenWeather schema extracted in Section 6.2


```
{
  "additionalProperties": false,
  "properties": {
    "city": {
      "additionalProperties": false,
      "properties": {
        "coord": {
          "additionalProperties": false,
          "properties": {
            "lat": {
              "minimum": -36.906,
              "type": "number"
            },
            "lon": {
              "minimum": -156.9223,
              "type": "number"
            }
          },
          "required": [ "lat", "lon" ],
          "type": "object"
        },
        "country": {
          "type": "string"
        },
        "id": {
          "minimum": 207570,
          "type": "integer"
        },
        "name": {
          "type": "string"
        }
      },
      "required": [ "coord", "country", "id", "name" ],
      "type": "object"
    },
    "cnt": {
      "minimum": 38,
      "type": "integer"
    },
    "cod": {
      "type": "string"
    },
    "list": {
      "items": {
        "additionalProperties": false,
        "properties": {
          "clouds": {
            "additionalProperties": false,
            "properties": {
              "all": {
                "minimum": 0,
                "type": "integer"
              }
            },
            "required": [ "all" ],
            "type": "object"
          },
          "dt": {
            "minimum": 1540188000,
            "type": "integer"
          },
          "dt_txt": {
            "type": "string"
          },
          "main": {
            "additionalProperties": false,
            "patternProperties": {
              "^(temp|temp_kf|temp_m[ai][nx]|pressure|sea_level|grnd_level)$": {
                "minimum": -4.13,
                "type": "number"
              }
            },
            "properties": {
              "humidity": {
                "minimum": 23,
                "type": "integer"
              }
            },
            "required": [
              "grnd_level",
              "humidity",
              "pressure",
              "sea_level",
              "temp",
              "temp_kf",
```

```
              "temp_max",
              "temp_min"
            ],
            "type": "object"
          },
          "rain": {
            "additionalProperties": false,
            "properties": {
              "3h": {
                "minimum": 0.002,
                "type": "number"
              }
            },
            "required": [],
            "type": "object"
          },
          "snow": {
            "additionalProperties": false,
            "properties": {
              "3h": {
                "minimum": 0.0005,
                "type": "number"
              }
            },
            "required": [],
            "type": "object"
          },
          "sys": {
            "additionalProperties": false,
            "properties": {
              "pod": {
                "type": "string"
              }
            },
            "required": [ "pod" ],
            "type": "object"
          },
          "weather": {
            "items": {
              "additionalProperties": false,
              "properties": {
                "description": {
                  "type": "string"
                },
                "icon": {
                  "type": "string"
                },
                "id": {
                  "minimum": 500,
                  "type": "integer"
                },
                "main": {
                  "type": "string"
                }
              },
              "required": [ "description", "icon", "id", "main" ],
              "type": "object"
            },
            "minItems": 1,
            "type": "array",
            "uniqueItems": true
          },
          "wind": {
            "additionalProperties": false,
            "properties": {
              "deg": {
                "minimum": 0.000274658,
                "type": "number"
              },
              "speed": {
                "minimum": 0,
                "type": "number"
              }
            },
            "required": [ "deg", "speed" ],
            "type": "object"
          }
        },
        "required": [
          "clouds",
          "dt",
          "dt_txt",
          "main",
          "sys",
          "weather",
          "wind"
        ],
        "type": "object"
```

```
      },
      "minItems": 38,
      "type": "array"
    },
    "message": {
      "minimum": 0.0018,
      "type": "number"
    }
  },
  "required": [ "city", "cnt", "cod", "list", "message" ],
  "type": "object"
}
```

C.3. Github schema extracted in Section 5.2

```
{
  "anyOf": [
    {
      "items": {
        "additionalProperties": false,
        "properties": {
          "_links": {
            "additionalProperties": false,
            "properties": {
              "git": {
                "anyOf": [ { "type": "string" }, { "type": null } ]
              },
              "html": {
                "anyOf": [ { "type": "string" }, { "type": null } ]
              },
              "self": {
                "type": "string"
              }
            },
            "required": [ "git", "html", "self" ],
            "type": "object"
          },
          "download_url": {
            "anyOf": [ { "type": "string" }, { "type": null } ]
          },
          "git_url": {
            "anyOf": [ { "type": "string" }, { "type": null } ]
          },
          "html_url": {
            "anyOf": [ { "type": "string" }, { "type": null } ]
          },
          "name": {
            "type": "string"
          },
          "path": {
            "type": "string"
          },
          "sha": {
            "type": "string"
          },
          "size": {
            "minimum": 0,
            "type": "integer"
          },
          "type": {
            "type": "string"
          },
          "url": {
            "type": "string"
          }
        },
        "required": [
          "_links",
          "download_url",
          "git_url",
          "html_url",
          "name",
          "path",
          "sha",
          "size",
          "type",
          "url"
        ],
        "type": "object"
      },
      "minItems": 1,
      "type": "array"
    },
    {
      "additionalProperties": false,
      "properties": {
        "_links": {
          "additionalProperties": false,
          "properties": {
            "git": {
              "anyOf": [ { "type": "string" }, { "type": null } ]
            },
            "html": {
              "anyOf": [ { "type": "string" }, { "type": null } ]
            },
            "self": {
              "type": "string"
            }
          },
          "required": [ "git", "html", "self" ],
```

```
          "type": "object"
        },
        "content": {
          "type": "string"
        },
        "download_url": {
          "anyOf": [ { "type": "string" }, { "type": null } ]
        },
        "encoding": {
          "type": "string"
        },
        "git_url": {
          "anyOf": [ { "type": "string" }, { "type": null } ]
        },
        "html_url": {
          "anyOf": [ { "type": "string" }, { "type": null } ]
        },
        "name": {
          "type": "string"
        },
        "path": {
          "type": "string"
        },
        "sha": {
          "type": "string"
        },
        "size": {
          "minimum": 0,
          "type": "integer"
        },
        "submodule_git_url": {
          "type": null
        },
        "target": {
          "type": "string"
        },
        "type": {
          "type": "string"
        },
        "url": {
          "type": "string"
        }
      },
      "required": [
        "_links",
        "download_url",
        "git_url",
        "html_url",
        "name",
        "path",
        "sha",
        "size",
        "type",
        "url"
      ],
      "type": "object"
    }
  ]
}
```

C.4. Github schema extracted in Section 6.2

```
{
  "anyOf": [
    {
      "items": {
        "additionalProperties": false,
        "patternProperties": {
          "^(url|sha|path|name|type)$": {
            "type": "string"
          }
        },
        "properties": {
          "_links": {
            "additionalProperties": false,
            "properties": {
              "git": {
                "anyOf": [ { "type": "string" }, { "type": "null" } ]
              },
              "html": {
                "anyOf": [ { "type": "string" }, { "type": "null" } ]
              },
              "self": {
                "type": "string"
              }
            },
            "required": [ "git", "html", "self" ],
            "type": "object"
          },
          "download_url": {
            "anyOf": [ { "type": "string" }, { "type": "null" } ]
          },
          "git_url": {
            "anyOf": [ { "type": "string" }, { "type": "null" } ]
          },
          "html_url": {
            "anyOf": [ { "type": "string" }, { "type": "null" } ]
          },
          "size": {
            "minimum": 0,
            "type": "integer"
          }
        },
        "required": [
          "_links",
          "download_url",
          "git_url",
          "html_url",
          "name",
          "path",
          "sha",
          "size",
          "type",
          "url"
        ],
        "type": "object"
      },
      "minItems": 1,
      "type": "array"
    },
    {
      "additionalProperties": false,
      "patternProperties": {
        "^(sha|url|type|name|path|target|content|encoding)$": {
          "type": "string"
        }
      },
      "properties": {
        "_links": {
          "additionalProperties": false,
          "properties": {
            "git": {
              "anyOf": [ { "type": "string" }, { "type": "null" } ]
            },
            "html": {
              "anyOf": [ { "type": "string" }, { "type": "null" } ]
            },
            "self": {
              "type": "string"
            }
          },
          "required": [ "git", "html", "self" ],
          "type": "object"
        },
        "download_url": {
          "anyOf": [ { "type": "string" }, { "type": "null" } ]
        },
```

```
        "git_url": {
          "anyOf": [ { "type": "string" }, { "type": "null" } ]
        },
        "html_url": {
          "anyOf": [ { "type": "string" }, { "type": "null" } ]
        },
        "size": {
          "minimum": 0,
          "type": "integer"
        },
        "submodule_git_url": {
          "type": "null"
        }
      },
      "required": [
        "_links",
        "download_url",
        "git_url",
        "html_url",
        "name",
        "path",
        "sha",
        "size",
        "type",
        "url"
      ],
      "type": "object"
    }
  ]
}
```

C.5. Wikidata schema extracted in Section 6.2

```
{
  "additionalProperties": false,
  "properties": {
    "aliases": {
      "additionalProperties": false,
      "patternProperties": {
        "^(aa|ab|ace|ady|ady-cyrl|aeb|aeb-arab|af|ak|aln|am|an|ang|anp|ar|arc|arn|arq|ary|...": {
          "items": {
            "properties": {
              "language": {
                "type": "string"
              },
              "value": {
                "type": "string"
              }
            },
            "required": [ "language", "value" ],
            "type": "object"
          },
          "minItems": 1,
          "type": "array"
        }
      },
      "properties": {},
      "required": [],
      "type": "object"
    },
    "claims": {
      "additionalProperties": false,
      "patternProperties": {
        "^(P6|P[1-4]\\d{3}|P[1-9]\\d{2}|P\\d{2})$": {
          "items": {
            "properties": {
              "id": {
                "type": "string"
              },
              "mainsnak": {
                "properties": {
                  "datatype": {
                    "type": "string"
                  },
                  "datavalue": {
                    "properties": {
                      "type": {
                        "type": "string"
                      },
                      "value": {
                        "anyOf": [
                          {
                            "type": "string"
                          },
                          {
                            "patternProperties": {
                              "^(amount|calendarmodel|entity-type|globe|id|language|lowerBound|text|time|unit|upperBound)$": {
                                "type": "string"
                              }
                            },
                            "properties": {
                              "after": {
                                "minimum": 0,
                                "type": "integer"
                              },
                              "altitude": {
                                "type": "null"
                              },
                              "before": {
                                "minimum": 0,
                                "type": "integer"
                              },
                              "latitude": {
                                "minimum": -85.2,
                                "type": "number"
                              },
                              "longitude": {
                                "minimum": -179.08568,
                                "type": "number"
                              },
                              "numeric-id": {
                                "minimum": 0,
                                "type": "integer"
                              },
                              "precision": {
                                "anyOf": [
```

```
                                {
                                  "minimum": 0,
                                  "type": "integer"
                                },
                                {
                                  "minimum": 0,
                                  "type": "number"
                                },
                                {
                                  "type": "null"
                                }
                              ]
                            },
                            "timezone": {
                              "minimum": 0,
                              "type": "integer"
                            }
                          },
                          "required": [],
                          "type": "object"
                        }
                      ]
                    }
                  },
                  "required": [ "type", "value" ],
                  "type": "object"
                },
                "property": {
                  "type": "string"
                },
                "snaktype": {
                  "type": "string"
                }
              },
              "required": [ "datatype", "property", "snaktype" ],
              "type": "object"
            },
            "qualifiers": {
              "patternProperties": {
                "^(P[1-4][0-9]\\d{2}|P[1-9]\\d{2}|P\\d{2})$": {
                  "items": {
                    "properties": {
                      "datatype": {
                        "type": "string"
                      },
                      "datavalue": {
                        "properties": {
                          "type": {
                            "type": "string"
                          },
                          "value": {
                            "anyOf": [
                              {
                                "patternProperties": {
                                  "^([lu][o-p][pw]erBound|[t-u][ein][imx][et]|amount
                                  ↪  |calendarmodel|entity\\-type|globe|id|language)$":
                                  ↪  {
                                    "type": "string"
                                  },
                                  "^(after|before|numeric\\-id|precision|timezone)$": {
                                    "anyOf": [
                                      {
                                        "minimum": 2.7777777777778e-8,
                                        "type": "number"
                                      },
                                      {
                                        "minimum": 0,
                                        "type": "integer"
                                      }
                                    ]
                                  }
                                },
                                "properties": {
                                  "altitude": {
                                    "type": "null"
                                  },
                                  "latitude": {
                                    "minimum": -34.584161111111,
                                    "type": "number"
                                  },
                                  "longitude": {
                                    "minimum": -118.847422,
                                    "type": "number"
                                  }
                                },
                                "required": [],
                                "type": "object"
                              },
```

```
                            {
                              "type": "string"
                            }
                          ]
                        }
                      },
                      "required": [ "type", "value" ],
                      "type": "object"
                    },
                    "hash": {
                      "type": "string"
                    },
                    "property": {
                      "type": "string"
                    },
                    "snaktype": {
                      "type": "string"
                    }
                  },
                  "required": [
                    "datatype",
                    "hash",
                    "property",
                    "snaktype"
                  ],
                  "type": "object"
                },
                "minItems": 1,
                "type": "array"
              }
            },
            "properties": {},
            "required": [],
            "type": "object"
          },
          "qualifiers-order": {
            "items": {
              "type": "string"
            },
            "minItems": 1,
            "type": "array"
          },
          "rank": {
            "type": "string"
          },
          "references": {
            "items": {
              "properties": {
                "hash": {
                  "type": "string"
                },
                "snaks": {
                  "patternProperties": {
                    "^(P[1-4]\\d{3}|P[1-9]\\d{2}|P\\d[018])$": {
                      "items": {
                        "properties": {
                          "datatype": {
                            "type": "string"
                          },
                          "datavalue": {
                            "properties": {
                              "type": {
                                "type": "string"
                              },
                              "value": {
                                "anyOf": [
                                  {
                                    "patternProperties": {
                                      "^(after|before|numeric\\-id|precision|timezone)$": {
                                        "anyOf": [
                                          {
                                            "minimum": 0.000001,
                                            "type": "number"
                                          },
                                          {
                                            "minimum": 0,
                                            "type": "integer"
                                          }
                                        ]
                                      },
                                      "^(amount|calendarmodel|entity\\-type|id|language|text
                                      ⇉ |time|unit)$":
                                      ⇉ {
                                        "type": "string"
                                      }
                                    },
                                    "properties": {
                                      "altitude": {
```

```
                                        "type": "null"
                                      },
                                      "globe": {
                                        "type": "string"
                                      },
                                      "latitude": {
                                        "minimum": 39.760178,
                                        "type": "number"
                                      },
                                      "longitude": {
                                        "minimum": -104.987096,
                                        "type": "number"
                                      }
                                    },
                                    "required": [],
                                    "type": "object"
                                  },
                                  {
                                    "type": "string"
                                  }
                                ]
                              }
                            },
                            "required": [ "type", "value" ],
                            "type": "object"
                          },
                          "property": {
                            "type": "string"
                          },
                          "snaktype": {
                            "type": "string"
                          }
                        },
                        "required": [
                          "datatype",
                          "datavalue",
                          "property",
                          "snaktype"
                        ],
                        "type": "object"
                      },
                      "minItems": 1,
                      "type": "array"
                    }
                  },
                  "properties": {},
                  "required": [],
                  "type": "object"
                },
                "snaks-order": {
                  "items": {
                    "type": "string"
                  },
                  "minItems": 1,
                  "type": "array"
                }
              },
              "required": [ "hash", "snaks", "snaks-order" ],
              "type": "object"
            },
            "minItems": 1,
            "type": "array"
          },
          "type": {
            "type": "string"
          }
        },
        "required": [ "id", "mainsnak", "rank", "type" ],
        "type": "object"
      },
      "minItems": 1,
      "type": "array"
    }
  },
  "properties": {},
  "required": [],
  "type": "object"
},
"datatype": {
  "type": "string"
},
"descriptions": {
  "additionalProperties": false,
  "patternProperties": {
    "^(ab|ace|aeb-arab|af|ak|am|an|ang|ar|arc|arz|as|ast|av|ay|az|ba|bar|bcl|be|...": {
      "properties": {
        "language": {
```

```
              "type": "string"
            },
            "value": {
              "type": "string"
            }
          },
          "required": [ "language", "value" ],
          "type": "object"
        }
      },
      "properties": {},
      "required": [],
      "type": "object"
    },
    "id": {
      "type": "string"
    },
    "labels": {
      "additionalProperties": false,
      "patternProperties": {
        "^(aa|ace|ady|aeb-arab|aeb-latn|af|ak|aln|am|an|ang|anp|ar|arc|arn|arz|as|ast|atj|...": {
          "properties": {
            "language": {
              "type": "string"
            },
            "value": {
              "type": "string"
            }
          },
          "required": [ "language", "value" ],
          "type": "object"
        }
      },
      "properties": {},
      "required": [],
      "type": "object"
    },
    "sitelinks": {
      "additionalProperties": false,
      "patternProperties": {
        "^(abwiki|acewiki|adywiki|afwiki|akwiki|alswiki|amwiki|angwiki|anwiki|arcwiki|...": {
          "properties": {
            "badges": {
              "items": {
                "anyOf": [
                  {
                    "minimum": -1.7976931348623157e+308,
                    "type": "number"
                  },
                  {
                    "type": "null"
                  },
                  {
                    "type": "boolean"
                  },
                  {
                    "type": "string"
                  }
                ]
              },
              "minItems": 0,
              "type": "array"
            },
            "site": {
              "type": "string"
            },
            "title": {
              "type": "string"
            }
          },
          "required": [ "badges", "site", "title" ],
          "type": "object"
        }
      },
      "properties": {},
      "required": [],
      "type": "object"
    },
    "type": {
      "type": "string"
    }
  },
  "required": [ "aliases", "claims", "descriptions", "id", "labels", "type" ],
  "type": "object"
}
```