Improving JSON Schema Inference by Incorporating User Inputs

Stijn Brian Broekhuis[1], Vadim Zaytsev[1,2]

[1] *Computer Science, EEMCS, University of Twente, The Netherlands*
[1] *Formal Methods & Tools, EEMCS, University of Twente, The Netherlands*

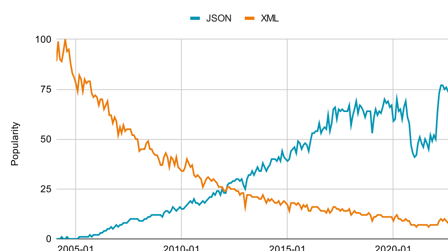
Abstract
JSON Schema schemata, as descriptive JSON files, define the expected structure of other JSON data, serving as a valuable resource for both developers and (meta)programs. They play a crucial role in data validation, testing, and maintaining data consistency. Since manually creating schemata for JSON can be challenging, it is common to derive them from sample data. In this paper, we focus on the introduction of user inputs during the inference process with the goal of reducing ambiguity and allow an algorithm to make, otherwise inconclusive, speculations from the sample data. We describe several strategies for utilising JSON Schema features based on sample JSON files and how they were implemented into a Kotlin program. We evaluate our tool on five distinct real world sample JSON datasets from which the results showed it is able to infer complex patterns.

Keywords
JSON, inference, JSON Schema, user input, interactivity

CEUR-WS.org/Vol-3941/BENEVOL2024_TECH_paper14.pdf

1. Introduction

The world needs formats for (semi)structured data that can be used very easily, without going through expertise-demanding and labour-intensive process of defining grammars, metamodels and schemata. XML (eXtensible Markup Language) [5] occupied this niche for a while, but JSON (JavaScript Object Notation) [10] certainly seems to be become more prominent, as can be observed on the Google Trends screenshot one can see on the right (click to follow to the dynamically updated source).



JSON Schema [20] offers a means to validate, test, and maintain the consistency of JSON data. It is meant for projects that mature beyond having purely self-descriptive data chunks, and can be introduced gradually for semi-structured data, restricting conformance only partially. However, its adoption has been rather slow [16]. One of the reasons for that is the time-consuming process of creating and maintaining such schemata.

The obvious solution is automated schema inference from sample data. However, existing approaches [3, 8, 11, 15, 21] cause overfitting and tend to produce structures that require further refinement. To address this issue, in this paper we introduce user inputs to be incorporated into the inference process. By doing so, we reduce ambiguity and enable algorithms to make informed speculations that would otherwise stay inconclusive. We assume that users have a deep understanding of the sample data, and their knowledge can be leveraged to extract more information and improve the accuracy of the schema.

In this paper, we present seven interactive strategies for harnessing the capabilities of JSON Schema schemata, implemented in a Kotlin program [6], openly available via GitHub [7] under the terms of the MIT license, as an extension of an existing non-interactive inferrer [21]. We evaluate our tool using five real world sample JSON datasets, highlighting its strengths and limitations.

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium
broekhuis.stijn@gmail.com (S. B. Broekhuis); vadim@grammarware.net (V. Zaytsev)
https://grammarware.net (V. Zaytsev)
0000-0001-7764-4224 (V. Zaytsev)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

2. Related Work

JSON is known for its structural simplicity, but becomes more complex when JSON Schema are involved because these schemata have a schema to follow themselves. Automated schema inference is performed by analysing sample data, identifying basic types (strings, numbers, Booleans, objects and arrays), patterns and constraints. In essence, it is akin to known and well-researched approaches of database schema synthesis [4], grammatical inference [18], generation by example [19] and, to some extent, process mining [1]. In some setups these inference algorithms work together or in pipeline with language/dialect/version identification [12, 13, 14].

Much of the existing research focuses on the uses of inference regarding databases, since "NoSQL" (standing for "not only SQL") databases also permit semi-structured data. All existing approaches we know of, work in a similar fashion: a large collection of JSON files is processed in parallel into a new format that the system uses; the collection is merged into one single specification (details vary per method); the combined specification is then transformed into a schema and serialised as such. In recent work, Čontoš and Svoboda [9] studied multiple current approaches for JSON inference and their limitations. They compared the works of Sevilla et al [17], Klettke et al [15], Baazizi et al [3], Cánovas et al [8] and Frozza et al [11]. With minimal repetition, we briefly describe how these approaches work.

Klette et al [15] use a *Structure Identification Graph* to combine all the JSON properties from a NoSQL database into a single schema. They are able to detect required and optional properties and union types, but not foreign keys ($stop_id \to id$). The algorithm of Baazizi et al [3] builds two versions of the schema: one that fuses all objects together, marking fields that lack anywhere as optional; and the other that only combines records if they share all the same fields. This results in a relatively small schema and a potentially large schema, and the user is left to pick and choose to construct the final result. Cánovas and Cabot [8] present an approach that generates class diagrams from JSON files, motivated by the need for a structure from services building or using APIs. This method traverses the input JSON data, systematically crafting multiple class diagrams, which are then reduced into a single class diagram.

Besides white literature, there are also online tools available to infer a structure from a JSON sample or samples. QuickType [22] is a tool that is available as a website, program, library, and IDE extension written in TypeScript. It is able to infer a JSON Schema from JSON samples or even a single JSON file, but only includes descriptions (and not types) in the result and thus does not validate anything. The JSON Schema inferrer from Saasquach [21] is an advanced library written in Java. It is able to infer from multiple JSON samples or a single JSON file. The resulting schema can be configured for different drafts, policies, formats, etc. The library has API features to expand the complexity of the inferrer. Lastly, Liquid Technologies [23] & JsonSchema.net [24] are both online JSON Schema generator tools that infer a JSON Schema from a single JSON sample. They both have limited options and settings and are not open source. They are easy to use compared to the other tools mentioned, making them useful when one needs a simple schema quickly.

3. JSON and JSON Schema

Consider the following piece of data in JSON:

```
{
  "orderId" : "2022343-34AZEEF",
  "userId"  : 433,
  "reason" : 1
}
```

This JSON file is unclear and not self-describing. Questions may arise such as: What is `orderId`? Is `userId` required? Why is `reason` a number? A schema would be able to answer these questions! For example, a corresponding schema could look as follows:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "orderId": {
      "description": "Unique identifier of the order", "type": "string"
    },
    "userId": {
      "description": "Unique identifier of the user", "type": "integer"
    },
    "reason": {
      "description": "Reason for the return", "type": "string"
    }
  },
  "required": ["orderId","userId","reason"]
}
```

Before we proceed, let us focus on one specific feature that we will call **informational keys**. Normally, in a JSON file, it is intended that the key is used to uniquely identify and retrieve a specific value from the data. However, it is possible to use the key as an identifier, attaching data into the key itself. This often makes a file smaller, but results in inconsistent keys in the file structure. Consider the following example.

```
  "people" : {
      "Alis" : {
        "age" : 34, "email" : "alis@example.com"
      }
  }
  ...
  "people" : [
    {
      "name": "Alis", "age": 34, "email" : "alis@example.com"
    }
  ]
```

We see two ways to encode the same data: the first example uses an informational key `"Alis"` to “name” the entire object, and in the second example, a normal array of objects/tuples is formed. In practice, the situation might get even worse, introducing keys that follow some predefined structure themselves:

```
{
  "variants": {
    "powered=false": {
      "model": "minecraft:block/oak_pressure_plate"
    },
    "powered=true": {
      "model": "minecraft:block/oak_pressure_plate_down"
    }
  }
}
```

This is a complex real world example from a JSON configuration file for the "*blockstate*" specification for an oak pressure plate within the game Minecraft [25]. This pressure plate is a block that has a `state` called `powered`, which changes when stepped on. The key `powered=true` in this situation serves as a condition for what model to display in the game when stepped on. Note that in Minecraft, blocks can contain various states, such as directionality, waterlogging, or connections to neighbouring blocks. These states can be combined by separating them with a comma to create more complex conditions.

It is unbelievable that we started to use JSON to escape from complex data structures, and we ended up having to write a parser (regular, in this case) for textually encoded structures within key names! To be able to claim a full victory, now a schema inference algorithm should, in this example, need to be able to parse keys and detect the regular expression pattern that corresponds to possible structure.

Unfortunately, informational keys is one of the problems we do not solve in this paper.

4. Interactive Schema Inferrer

Existing algorithms of JSON Schema inference are facing challenges mentioned in previous sections. As a result, their output schemata tend to be relatively simple compared to the full range of capabilities of the JSON Schema specification. This limitation arises from assumptions these algorithms would be required to make. Sample data, while informative about what is allowed, cannot convey what is disallowed. Consequently, any algorithm venturing into schema inference inevitably makes assumptions.

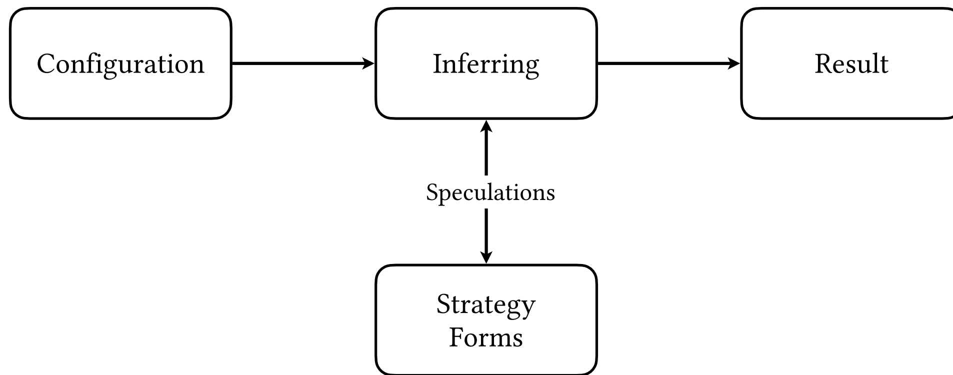
A prevalent assumption involves defining the type of a field. For instance, if a field such as *foo* is always a number, the system deduces it to be exclusively numeric. This deduction rests on the assumption that, because we have not received any other type for this field, only numeric are allowed for the *foo* key. While this assumptions is trivial, it is far from trivial for more complex situations.

What if our JSON snippet is `{"fruit-type": "apple"}`? It has a string type, but the number of allowable values for this field is unknown from this example. What if we analyse 1000 samples and witness only five unique values of *fruit-type*? The input JSON files may not encompass all possible options, but we can make an assumption to restrict the number of valid values for *fruit-type* to this minimum of five. This assumption works reasonably well on large data sets with little variability.

Ideally, for character names in a structure representing the plot of a story, it would be great to infer all names from available data and restrict the enumeration to them. However, for phone numbers we want to stick to the basic type and not impose any restrictions at all, since we know this to be a very flexible and extensible enumeration. For country codes or country names there are certified lists that cover “all” possible values and are updated occasionally when they are officially and lawfully extended. For postcodes, we could possibly produce such an enumeration, but that would be undesirable, since it would be overly long and much more complex than a pattern that says “four digits and two capital letters” (like postcodes in the Netherlands). For types of fruit, we cannot even make a statement generic enough for this paper, since in one application the list of allowed values will be closed, in another open, and in the third one (such as a game world) restricted to a predefined range, not necessarily fully covered by the dataset.

This leads us towards exploring alternative approaches, such as a user-input-based method. In this context, users could play an active role by offering supplementary information or clarifying ambiguous situations. In such instances, we can communicate to the user that, based on our observation of the given 1000 examples, we have encountered only 5 unique values, potentially suggesting an enumeration type. The user may then verify whether the value indeed conforms to an enum type and accept the valid values.

Our objective in this paper is to develop a JSON Schema inference program capable of handling such scenarios, utilising a balance between under-approximation and over-approximation to aim for true accuracy. We implement different strategies to handle specific scenarios for the user to respond to. These strategies will be described further in this section. We focus specifically on handling JSON files and producing JSON Schema files, leaving out related activities such as parsing YAML files or handling NoSQL databases.



The operation of the *Interactive Schema Inferrer* goes through three distinct steps. The initial step involves displaying the *configuration view*, where the user is prompted to specify the schema version and select the JSON files to be used as samples. Additionally, a checkbox is provided to indicate whether the input JSON files are structured as an array, where each value in the array should be considered a sample.

The second step encompasses the inference process, in which the inferrer is constructed, providing it with all the strategies. A *strategy* is a method of improving an inferred schema by detecting speculations from a sample set, and using user input to confirm or deny *speculations*. It is crucial to emphasise that the absence of user input to affirm or reject these speculations would result in the generation of schemata overly tailored to the sample data. A user can always deny any speculation. This is the underlying rationale why conventional inference systems are unable to incorporate such strategies. During the inference process, the strategies may replace the view with a form, enabling the user to response to a speculation. Upon completion, the *loading view* is reinstated and the response is processed.

Lastly, when the data has been processed and the inference has been completed, the *loading view* is replaced with the *result view*. This view presents the inferred JSON Schema as the outcome, along with a button for copying it to the clipboard for saving purposes.

The inference part of *Interactive Schema Inferrer* bears some resemblance to the corresponding component of Saasquatch [21], one of the online tools we mentioned above. Their inference system works by combining all the sample JSONs and traversing for each key all values provided, building up a schema from the bottom up. The library possessed the capability to build enum extractors and generic feature classes, which were essential components for implementing user interaction functionalities. Naturally, we made the decision to use the library rather than developing a new one from scratch.

However, the library was missing a crucial component to regarding user interaction. It was unable to provide context about the current field (key), as it only provided information about the values. If the system wanted to use user interaction, providing context to the user about what field needed clarification is crucial. Luckily, since this project was open source, we implemented the missing functionality and opened a pull request to add the current JSON path to the API. After minor adjustments, it was accepted and by now is a part of release 0.2.1.

For the graphical user interface part, we have used *TornadoFX* [26], a Kotlin-based JavaFX framework. Compared to alternatives, it maintains the right balance between native integration and simplicity.

5. User Input Strategies

Each strategy is a class which implements a method called by the inferrer for each applicable field. Generally, it receives the following information to infer from; the preliminary schema for this field, the type of the current field (`array, number, object, ...`), the draft version provided, the samples of this field, and the JSON path of this field.

JSON Schema has multiple versions, called “drafts” [20]. These specify what keywords are available and how they should be used. Each strategy might be disabled or behave differently based on the version.

5.1. Constants

A `const` is a keyword that specifies that a field is always this specific value. This keyword is available since draft 6 and is part of the validation vocabulary. When samples of a field consists of only a single distinct value the system speculates that this field is a `const`. However, this approach proves inadequate when confronted with limited sample sizes or, even more disadvantageously, when the sample size is merely one. In the latter case, the system refrains from making any speculations altogether.

5.2. Enumerators

An `enum` is a keyword that specifies that a field is restricted to a specific set of values. This keyword is available since draft 4 and is part of the validation vocabulary. The system speculates similarly to the const. The system perform a division of the distinct sample size by the total size and examines whether this surpasses a predefined threshold. The determination of the threshold value emerged during testing, and led to value of 0.2. This threshold was selected to strike a balance between minimising false positives and maximising true negatives. It is essential to understand that the exact threshold value, is not a critical determinant in the scope of this project. The primary objective is to achieve reasonable coverage rather than pinpoint accuracy. Fine-tuning this threshold can be a topic for discussion and adjustment in future iterations.

5.3. Default

The `default` annotation keyword specifies that *"...if a value is missing, then the value is semantically the same as if the value was present with the default value"*. This keyword is available since draft 4 and is part of the meta-data vocabulary. This strategy is unique in the sense that it does not influence validation of a JSON file. Nevertheless, it remains feasible and beneficial to deduce a default value. Initially, the process of speculating whether a field possesses a default value requires an analysis of the frequency distribution of distinct values within the sampled data. The system would employ the empirical rule to identify potential outliers in these frequencies. If such outliers are present, the system postulates that the most substantial outlier represents the default value.

However, through experimentation, it became evident that the effectiveness of outlier detection was not as reasonable as initially presumed. To illustrate this point, consider a scenario in which one value occurs 800 times while another occurs only once. In such a case, traditional outlier detection methods fail to identify the latter value as an outlier, as they tend to assume an average frequency of around 400. Consequently, a more straightforward approach was proposed.

This approach involves assessing the frequency of each distinct value and determining if the most frequent value appears in more than 80% of the cases. The threshold of 80% was chosen somewhat arbitrarily, but it seemed suitable during testing. Similarly to the enum strategy, the exact threshold value, whether it is 75%, 80%, or 85%, is not of critical importance. It should be noted that if the frequency is 100%, we assume it to be a constant and do not process this value further.

5.4. Uniqueness

The `uniqueItems` keyword specifies whether an array field can or cannot contain the same value multiple times. This keyword is available since draft 4 and is part of the applicator vocabulary. By analysing the array values of an field, we can speculate if the field can be marked with `uniqueItems` when each sample of type array for a specific field does not contain the same value twice.

5.5. Contains/PrefixItems

The `contains` (draft 6+) and `prefixItems` (draft 4+) keywords specify that an array should contain the a specific set of values, where `prefixItems` also specifies the index. This approach is particularly

effective when applied in the context of post-order traversal, as it benefits from the prior inference of the schema beneath the current stage.

We use the preliminary schema to test whether the array always contains a specific condition. This strategy is exclusively used when the schema encompasses multiple conditions, typically in the form of an `anyOf` and/or `"type": [...]`. If a consistent pattern emerges where the same index consistently adheres to the same condition, the system designates it as a `prefixItems`. In the event that a user declines a `prefixItems` inference, the program will ask whether it should be considered as a `contains` instead. Here the the system provides options for `minContains` and `maxContains`.

5.6. MultipleOf

The `multipleOf` keyword specifies that an numerical value should be a multiple of a given positive number. This keyword is available since draft 4 and is part of the validation vocabulary. By finding the greatest common divider (GCD) of the samples, we can speculate if the field can be marked with `multipleOf`. This only happens if the sample size and the GCD are both larger than 1.

5.7. Length

The last strategy implements keywords regarding the size or length of values. JSON Schema can add these conditions for Numbers (Range), Arrays (Item Count), Strings (Length), and Objects (Property Count). These keywords are available since draft 4.

This strategy waits until the inference is complete before asking the user for input. By doing so, the system can present the user with a list of all options at once (disabled by default), rather than multiple screens. During the inference process, the system keeps track of the minimum and maximum values for each condition mentioned earlier. This information is used to ensure that the user cannot set an invalid minimum or maximum value that would invalidate the samples. Additionally, for numbers the system provides an option to specify if the range is exclusive or inclusive.

6. Evaluation

We continue with evaluating the tool we have developed, by executing it on specific datasets and examining the results. The evaluation procedure is as follows. The tool is executed on a designated dataset, and the resulting schema is manually reviewed. During the inference, noticeable speculations or lack thereof, are documented. Certain datasets originate from sources that already provide a JSON Schema, and in such instances, a comparison is conducted between the derived schema and the source schema. Our *Interactive Schema Inferrer* [7] serves as an extension of an established library *Saasquatch* [21], albeit with a distinct configuration where certain pre-existing features remain disabled intentionally. The deliberate omission of these features allows for the focus on newly added functionalities. Resulting schemata will, for example, not contain any attempts to infer "format" — strings with specific format rules, such as emails.

In the previous sections we have mentioned the use of specific sample files for experimentation and system testing purposes. In the interest of preserving the impartiality of the evaluation process, it is important to abstain from including these sample files during the evaluation, as the software's performance has likely been optimised to align with them.

The following datasets are used during the evaluation:

- Minecraft Biomes [25]
- Earthquakes data [27]
- NPM packages configurations, extracted from public GitHub repositories
- IMDb movies example dataset [28]
- OSI Licences [29]

The resulting schemata are available on the GitHub page [7] as they are too large to even demonstrate on the pages of this paper.

The selection of these five specific JSON datasets for the study was guided by several considerations:

- **Real-World Examples:** The datasets chosen are grounded in real-world scenarios, providing a practical foundation for the study. This decision was motivated by the intention to ensure that the schemata inferred are relevant and applicable in genuine operational contexts. The authenticity of these datasets contributes to the robustness of the study outcomes.
- **Diverse Use Cases:** One key criterion for selection was the diversity in the utilisation of the datasets. The chosen datasets represent a spectrum of applications, ranging from configurations files to scientific research data and database information. This deliberate variation in use cases aims to expose the inference algorithms to a wide array of JSON structures.
- **Variety in Data Types:** The datasets exhibit significant differences not only in their use cases but also in the types of data they encapsulate. This intentional diversity encompasses various data structures, field types, and nesting levels. This breadth in data types serves to challenge the inference algorithms and ensures that the resulting schemata are capable of accommodating a broad range of JSON structures.
- **Study Scope and Manageability:** The decision to limit the study to five datasets was deliberate, stemming from a balance between comprehensiveness and practicality. A more extensive dataset collection might not necessarily yield significantly different insights and could potentially overlap with the characteristics of other samples. By constraining the dataset count, the study aims to reduce the work while still ensuring a meaningful and focused exploration of JSON schema inference.

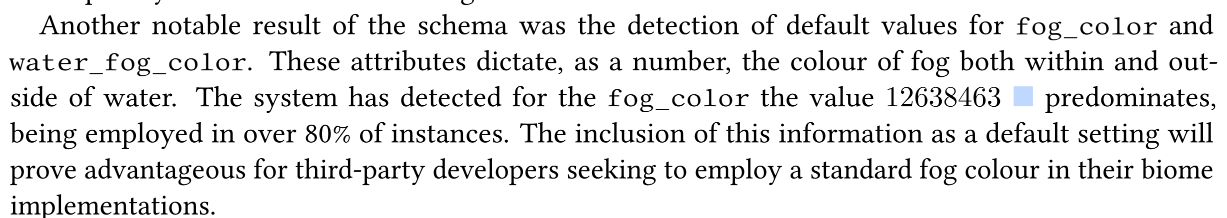
6.1. Sample 1: Minecraft Biomes

The first sample data that will be used is data from the game Minecraft. Minecraft is a video game made set in a world of cubes. A **biome** is a region in that world with its own geographical features and properties. A biome can have different grass, foliage, sky, water colours. Such information is stored as JSON files within the games files.

Notes & Comparisons

The unofficial Minecraft Wiki [30] describes the structure for custom biomes. This documentation is used to compare the resulting schema.

The initial point of distinction lies in the lack of fields within the `particle.options` object. Within the context of the game, certain biomes feature ambient particles that traverse the screen. In the case of sample biomes, these particles are defined through an `id` and a `probability` parameter. However, it is important to note that the game provides more intricate customisation options for biomes created by third-party developers. As these customised options are not utilised in the provided samples, they are consequently absent from the resulting schema.

Another notable result of the schema was the detection of default values for `fog_color` and `water_fog_color`. These attributes dictate, as a number, the colour of fog both within and outside of water. The system has detected for the `fog_color` the value 12638463  predominates, being employed in over 80% of instances. The inclusion of this information as a default setting will prove advantageous for third-party developers seeking to employ a standard fog colour in their biome implementations.

The complexity increases for the `temperature_modifier` field, which is an optional key. This particular field can assume one of two values: *none* or *frozen*, with *none* being the default in cases where it is omitted. Ideally, this field should be categorised as an enumeration encompassing these specific values. However, a challenge arises due to its optional nature. Since no JSON file would explicitly denote *none* in this context, the samples featuring this field consistently exhibit the only other option, *frozen*. Consequently, the system has mistakenly identified it as a constant value.

Lastly, we turn our attention to the `spawners` field, which delineates the entities that can potentially spawn within the confines of the biome. Each *mob category* field has the same structure, where the category is *monster*, *creature*, *ambient*, *water_creature*, *underground_water_creature*, *water_ambient*, *misc*, or *axolotls*. Ideally, the `propertyNames` keyword, in conjunction with `additionalProperties`, should be used to establish a consistent structure encompassing all mob categories without having to repeat the structure in the schema. However, due to the system inferring each field independently without considering other related fields, it fails to recognise the shared structure among these fields. This gives rise to two primary issues: first, the resulting schema redundantly represents the structure multiple times, and second, users are required to provide repetitive responses to identical speculations, which provides opportunity for inconsistencies in user input.

6.2. Sample 2: Earthquakes

The second dataset in this study comprises GeoJSON features representing earthquake locations from the past 30 days, sourced from the United States Geological Survey. GeoJSON is a format specifically designed for representing geographical locations in JSON. This dataset, initially presented as a GeoJSON `FeatureCollection`, has been streamlined to exclusively include the individual *Features* arranged within an array structure. One might realise that this will change the resulting structure.

The proposed schema's architecture will be compared against the official documentation provided by the United States Geological Survey, as published on their website.

Notes & Comparisons

The resulting schema was of good quality, as it was able to detect all conditions accurately. All properties were detected, and marked as required. Because the resulting samples only contained *Features*, the `type` field was detected as an constant. In cases where data was missing, the samples provided a `null` value. This resulted in the schema allowing both for these properties. Nonetheless, it is worth noting that certain values were consistently featured in the data, and as such, the schema did not add the option for `null` to allowed. It is unclear from the documentation which values are or are not allowed to be `null`.

Delving into the specifics of the properties, we encounter noteworthy detection for enums:

- The `status` property astutely discerns whether an event has undergone human review, signifying this via the `automatic` or `reviewed` options. Notably, the `deleted` alternative, while mentioned in documentation, is understandably absent from the samples (and thus also the resulting schema).
- The `alert` property informs the alert level according to the PAGER earthquake impact scale, and was detected as an enumeration of `green`, `yellow`, and `orange`. The absence of the `red` sample came from the apparent lack of `red` cases within the last 30 days in the sample data — perhaps a fortunate twist of fate.
- The `tsunami` property, denoting whether an event occurred in an oceanic region, was correctly identified as an enumeration of either `1` or `0`. It raises the question of why a boolean data type was not employed for this purpose. Possibly, it was the result of how booleans are stored in their database.
- The `type` property, categorising the seismic event, was detected as an enumeration of `earthquake`, `quarry blast`, `explosion`, `ice quake`, and `other event`. However, the official documentation does not specify this property as an enumeration. This detection leaves us uncertain whether this detection should be interpreted as a positive or negative result, as `other event` implies that the given options would suffice as an enum type.

Finally, the schema's length strategy allowed the addition of `minItems` and `maxItems` for the `coordinates` array. This would require the array to be comprised of three values (longitude, latitude, depth).

6.3. Sample 3: NPM Packages

The next dataset contained samples for the JavaScript package manager NPM. Information about a package is stored in the *package.json* file present in each project. This file provides information about the name of the project, mark what dependencies that are used, macros to run scripts, and other configurations. The gathering of this sample data was done with the use of the GitHub API. Using this API, *package.json* files from public repositories were extracted and aggregated into a single JSON file. Due to the presence of potentially sensitive or personal information within this document, despite its publicly accessible nature, we shall refrain from providing it.

Notes & Comparisons

The NPM *package.json* file presents a formidable challenge for schema inferences. As mentioned above, when a JSON files use informational keys, inference becomes difficult. Ideally, a single definition would be presented in the `additionalProperties`. Unfortunately, the current inference system is not implemented to detect such usage of keys, treating each field independently. As a consequence, the system produced an exceedingly extensive JSON Schema, where each field, be it a library, dependency, script, or configuration choice, is specified. Comparing this to the version available on SchemaStore.org, resulting schema is appalling.

However, it does reveal numerous licenses to be an enum, which the other schema also specifies. The SchemaStore.org variant adopts, however, the elegant approach by utilising the enumeration an suggestion, permitting any string value while still documenting the most prevalent licenses through the use of the `anyOf` keyword.

6.4. Sample 4: IMDb Movies

The Internet Movie Database (IMDb) is an online repository dedicated to entertainment media. Its API documentation includes a curated dataset comprising JSON responses spanning a range of queries as an example responses. Among these queries, 'title with parameters' movie responses were specifically extracted and utilised as the primary sample data.

Notes

The sample dataset appears to be curated, as they predominantly featuring highly-rated films. This was apparent in the program's repeated speculation for ratings (such as IMDb and Rotten Tomatoes ratings) to marked as an enum. Interestingly, when dealing with ratings, as they are stored as strings, the inference system can therefore not infer a potential `multipleOf` constraint for ratings. Moreover, a substantial portion of the movies in the dataset are English, which suggests a bias in the samples. Consequently, the program erroneously assumed that `language` was a constant, a speculation we declined.

The program was able to detect Language, Genre, and Country as enums. A deeper understanding of the back-end infrastructure could enable more informed judgements regarding the enumeration of these attributes. For instance, if IMDb merely stores languages as key/string pairs. Noticeably, the language data is stored as an object with two fields: *key* and *value*, where the two fields were always the same. My assumption is that value would be different in other languages. If this were the case, an enum would be rather complex to implement. While it was chosen to designate them as enums, this choice notably inflates the schema's size.

The program identified a comical repetition in the lists of people, particularly in the cast and crew context. In the *fullCast* section, the program noticed a pattern regarding job descriptions that were listed alongside individuals. This field was also observed in specific job sections, such as directors, where all directors were specified as *director*. This keen observation caused the system to detect that field as a possible constant.

6.5. Sample 5: OSI licenses

The Open Source Initiative (OSI) is a organisation dedicated to promoting and safeguarding the rules of open-source software development. It maintains a comprehensive dataset of open-source licenses that developers can use for their software. This dataset is in the form of a JSON file, which will be used as the last sample to test the system on. Unfortunately, we were unable to locate a schema for this file to use for comparison.

Notes

Each licence has an array of identifiers that display the identifier of the licence in at most 3 different formats (SPDX, Trove, or DEP5). The system was able to detect the 3 types of format were as a possible enum. In the `text` field of the samples, the JSON file specifies the link to the licence and the type of the file. In this field the `media_type` property was correctly categorised into three distinct enumerations: *text/html*, *text/plain*, and *application/pdf*. Similarly, the `title` property was categorised into three enumerations as well: *HTML*, *Plain Text* and *PDF*.

However, it is noteworthy that the program did not inherently establish a direct correlation between the `media_type` and `title` properties, even though a clear correlation exists. As said before, the system processes each field independently, and is therefore lacking functionality in detecting correlations between fields. For instance, when `media_type` is identified as *text/html*, the corresponding `title` is *HTML*. This, and other previously mentioned, lack of correlation recognition highlights a potential area for potential enhancement in the program's functionality, as it could improve the accuracy of the resulting schema.

7. Concluding Remarks

We have provided some background information about JSON and JSON Schema in section 3 and gave an overview of existing algorithms in section 2. From the existing JSON Schema inference algorithms we found that most focus on generating a structure from NoSQL databases [2, 3, 9, 11, 17]. Generally, these algorithms infer one schema for each file, and merge them afterwards. Unfortunately, these algorithms often do not produce a JSON Schema directly, and produce guidelines, descriptions, class diagrams, or even their own defined structure definitions. We have also described several tools, and ended up using and extending one of them — namely, *Saasquatch* [21].

Users of *Interactive Schema Inferrer* are prompted by the tool when it requires clarification, this halts the schema synthesis process until the tool receives an answer. This can happen during the inference or after the inference has completed. During the inference, the inference system creates a basic schema from the primitive types of each file. It then calls strategies for each field with relevant information to improve the resulting schema. If the strategy thinks it has found an improvement, it asks and waits for the user to respond. To improve the user experience, the design of the UI for each strategy focuses on making it simple for the user to decline any speculation. Additionally, for strategies that would otherwise always require user input, they are instead combined and asked at the end of the inference process. We have explained in section 4 and section 5 how the tool is designed and which strategies it employs to combine inference of speculations with user input in confirming or denying them. Our strategies were formulated by examining all the keywords in the JSON Schema and contemplating how a program could identify situations in a set of JSON files where a keyword would be appropriate. As JSON Schema can be expanded with custom vocabularies, there is no limit to the potential for other strategies.

As seen in section 6, where we have evaluated the created program on five distinct JSON datasets, there were still some limitations in the inference process. The evaluation of the five samples revealed a spectrum of quality, ranging from schemata deemed highly favourable to those considered significantly unfavourable from my subjective standpoint, reflecting the varying degrees of refinement that would be needed. We saw that the enumeration strategy was the most successful, improving the structure for

almost all samples. In the remainder of this section we will delve into the limitations and future work of this study in more depth. Nevertheless, the implemented strategies have demonstrated how they can assist a user in enhancing the resulting schema of an inference algorithm.

In the course of this study, it became apparent that JSON Schema possesses a far greater degree of complexity than foreseen. This complexity allowed for many strategies to be created, although the list is naturally open-ended. In particular, strategies that would organise and structure parts of the schema look very promising as future work. However, as the system's ability to detect structure improves, the task of organising specific schema components with similar structures becomes progressively more intricate.

Besides validation rules, a JSON Schema provides tools for documentations. The current system does not make use of these tools as it is hard to infer documentation from samples. Given the recent successes in applying generative artificial intelligence for documentation inference, we foresee that as another possible avenue for future research. The user involvement paradigm does not have to be broken in this case, since the tool can produce an end screen where all fields could be provided with a description, some of them already inferred from the dataset yet still editable.

A important part of this study is the challenge of striking a balance between user interaction and automated inference. As a system that is supposed to make the creation of a JSON Schema easier, excessive user involvement counteracts this. The design attempted to minimise the user interaction, where most of the interaction is required when *confirm* any speculation. This aspect has not been validated on real end users beside the authors.

When we focus us on specific strategies, there are improvements to be made. For instance, in the strategy for detecting enumerations, instead of separating the constant and enumeration strategies, they could be merged. Since an enum with a single value is equivalent to a constant, the system could easily replace it. An additional improvement would be to allow the user to specify if these values are suggestions. If so, the schema would wrap the result around an `anyOf` with the primitive type. This allows all values to be valid but still provides suggestions for autocompletion.

One might have noticed in the evaluation that the strategy for speculating prefix items and contains keywords was not detected in any of the samples. This might indicate that this strategy is not working as well as expected. The strategy works by testing the already made schema for the current field, but this sub-schema might be to complex to detect any consistent patterns. To avoid any influence of the sample data on the code, the study was conducted in a one-time manner. While this approach aligns with the objective of unbiased analysis, it does introduce a challenge when identifying potential issues or errors in the code postfactum. Future work should evaluate the implementation of this strategy.

Ultimately, samples do not always provide a complete depiction of a structure. We observed scenarios, such as Minecraft biomes, where programs or systems receiving JSON files would use a default value when a field is not present. This reveals that even the most advanced systems cannot infer all nuances, and thus need to remain flexible.

As we have discussed, informational keys are difficult to infer, but perhaps not impossible. Many JSON files we encountered, prioritised human readability over adhering to the expected structure of a JSON file. For example, dependencies inside an NPM package could have been an array of objects with the same structure, where the name of the package would be value of a field. Instead, it was designed to use an object, with the key the name of the package. Since keys are unique, this makes it clear when there exists a duplicate dependency. It is important to recognise that the complexity of informational keys can vary widely. It can be as basic as an enum of keys or as sophisticated as Minecraft Blockstate.

Besides all these limitations, we claim this analysis a success, and welcome the community to inspect the tool [7] to replicate the results or use it as inspiration for other application domains. We have demonstrated how a complex JSON Schema can be created with the help of a user, in particular, for the context of enumerations. This research project has led us to insights about JSON Schema inference that can hopefully be useful for future research in this domain.

References

[1] W. van der Aalst, Process Mining: Overview and Opportunities, ACM Transactions on Management Information Systems (TMIS) 3 (2012). doi:10.1145/2229156.2229157.

[2] M.-A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, C. Sartiani, Schema Inference for Massive JSON Datasets, in: Proceedings of the 20th International Conference on Extending Database Technology (EDBT), OpenProceedings.org, 2017, pp. 222–233. URL: https://openproceedings.org/2017/conf/edbt/paper-62.pdf. doi:10.5441/002/EDBT.2017.21.

[3] M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, Parametric Schema Inference for Massive JSON Datasets, The VLDB Journal 28 (2019-08-01) 497–521. doi:10.1007/s00778-018-0532-7.

[4] J. Biskup, U. Dayal, P. A. Bernstein, Synthesizing Independent Database Schemas, in: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, SIGMOD '79, ACM, New York, NY, USA, 1979, p. 143–151. doi:10.1145/582095.582118.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, Extensible Markup Language (XML) 1.0, Technical W3CREC-xml-19980210, w3c, 1998. URL: https://www.w3.org/TR/1998/REC-xml-19980210.html.

[6] S. B. Broekhuis, Incorporating User Inputs for Improved JSON Schema Inference, Master's thesis, Universiteit Twente, Enschede, The Netherlands, 2023. URL: http://purl.utwente.nl/essays/97755.

[7] S. B. Broekhuis, Interactive Schema Inferrer, GitHub, 2023. URL: https://github.com/sbroekhuis/InteractiveSchemaInferrer.

[8] J. L. Cánovas Izquierdo, J. Cabot, Discovering implicit schemas in JSON data, in: F. Daniel, P. Dolog, Q. Li (Eds.), Web Engineering, LNCS, Springer, 2013, pp. 68–83. doi:10.1007/978-3-642-39200-9_8.

[9] P. Čontoš, M. Svoboda, JSON schema inference approaches, in: G. Grossmann, S. Ram (Eds.), Advances in Conceptual Modeling, LNCS, Springer International Publishing, 2020, pp. 173–183. doi:10.1007/978-3-030-65847-2_16.

[10] D. Crockford, JSON: JavaScript Object Notation, 2001. URL: https://www.json.org/json-en.html.

[11] A. A. Frozza, R. dos Santos Mello, F. de Souza da Costa, An Approach for Schema Extraction of JSON and Extended JSON Document Collections, in: 2018 IEEE International Conference on Information Reuse and Integration (IRI), 2018-07, pp. 356–363. doi:10.1109/IRI.2018.00060.

[12] M. Gerhold, L. Solovyeva, V. Zaytsev, Leveraging Deep Learning for Python Version Identification, in: F. Madeiral, A. Rastogi (Eds.), Proceedings of the 22nd Belgium-Netherlands Software Evolution Workshop (BENEVOL), volume 3567 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023, pp. 33–40. URL: http://ceur-ws.org/Vol-3567/paper5.pdf.

[13] M. Gerhold, L. Solovyeva, V. Zaytsev, The Limits of the Identifiable: Challenges in Python Version Identification with Deep Learning, in: V. Lenarduzzi, D. Taibi, G. H. Travassos, S. Vegas (Eds.), Proceedings of the 31st IEEE International Conference on Software Analysis, Evolution and Reengineering, Reproducibility Studies and Negative Results (SANER RENE), 2024, pp. 137–146. doi:10.1109/SANER60148.2024.00022.

[14] J. Kennedy van Dam, V. Zaytsev, Software Language Identification with Natural Language Classifiers, in: K. Inoue, Y. Kamei, M. Lanza, N. Yoshida (Eds.), Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering: the Early Research Achievements track (SANER ERA), IEEE, 2016, pp. 624–628. doi:10.1109/SANER.2016.92.

[15] M. Klettke, U. Störl, S. Scherzinger, Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores, Datenbanksysteme für Business, Technologie und Web (BTW 2015) (2015). Publisher: Gesellschaft für Informatik eV.

[16] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, D. Vrgoč, Foundations of JSON Schema, in: Proceedings of the 25th International Conference on World Wide Web, WWW '16, WWW Steering Committee, 2016-04-11, pp. 263–273. doi:10.1145/2872427.2883029.

[17] D. Sevilla Ruiz, S. F. Morales, J. García Molina, Inferring Versioned Schemas from NoSQL Databases and Its Applications, in: P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, O. Pastor Lopez (Eds.), Conceptual Modeling, LNCS, Springer, 2015, pp. 467–480. doi:10.1007/978-3-319-25264-3_35.

[18] A. Stevenson, J. R. Cordy, A Survey of Grammatical Inference in Software Engineering, Science of Computer Programming 96 (2014) 444–459. doi:10.1016/j.scico.2014.05.008, Selected Papers from the Fifth International Conference on Software Language Engineering (SLE 2012).
[19] V. Zaytsev, Parser Generation by Example for Legacy Pattern Languages, in: Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, ACM, 2017-10-23, pp. 212–218. doi:10.1145/3136040.3136058.
[20] JSON Schema Working Group, JSON schema, 2009. URL: https://json-schema.org/.
[21] slisaasquatch, et al., GitHub - saasquatch/json-schema-inferrer: Java library for inferring JSON schema from sample JSONs, 2023. URL: https://github.com/saasquatch/json-schema-inferrer.
[22] QuickType, quicktype/quicktype, 2023-05-23. URL: https://app.quicktype.io/#l=schema, original-date: 2017-07-13T00:22:50Z.
[23] Liquid Technologies Limited, Free online JSON to JSON schema converter, 2023. URL: https://www.liquid-technologies.com/online-json-to-schema-converter.
[24] JSONschema.net, JSON schema generator, 2022. URL: https://jsonschema.net/.
[25] Mojang Studios, Minecraft JSON biome data, 2021.
[26] E. Syse, TornadoFX, 2023. URL: https://tornadofx.io/.
[27] USGS, GeoJSON summary format, 2023-09-05. URL: https://earthquake.usgs.gov/earthquakes/feed/v1.0/geojson.php.
[28] IMDb, IMDb sample data, 2023-09-05. URL: https://imdb-api.com/API.
[29] open source initiative, Opensource.org licenses, 2023-09-05. URL: https://api.opensource.org/licenses/.
[30] Minecraft Wiki, Custom biome, 2023-09-13. URL: https://minecraft.wiki/w/Custom_biome.