Enhancing JSON Schema Discovery by Uncovering Hidden Data

Justin R. Namba
supervised by Michael J. Mior
Rochester Institute of Technology
Rochester, New York, USA
jrn1325@rit.edu

ABSTRACT

Schema discovery is finding the structure of data. It helps users understand the meaning of data and write queries to manipulate it. This is typically easy for relational databases, but complex for non-relational (NoSQL) databases with JavaScript Object Notation (JSON) documents. JSON is a representation of documents that contain objects stored in the form of nested key-value pairs. For relational databases, the schema is predefined because the data they contain is structured, but for NoSQL databases, data is usually unstructured or semi-structured. In a collection of JSON documents, the structure of one document can be completely different from another. Several algorithms were developed to discover schemas from JSON documents, but they provide the physical structure and semantic information that is insufficient for data understanding and analysis. In this paper, we enumerate the major techniques used to extract a schema from JSON documents and present the next challenge: uncovering hidden data disguised as metadata. This challenge needs to be addressed within the field of JSON schema discovery to enhance the quality of the discovered schemas.

1 INTRODUCTION

NoSQL is an approach to database design that can accommodate a wide variety of data models, including key-value, document, and graph formats [6]. It provides an alternative to relational databases in which data is placed into tables and the schema is designed before the database is populated [10]. Over the years, NoSQL databases have become popular because of their flexibility and performance. However, analyzing NoSQL data is challenging because of the lack of schema. We focus on data represented in JSON format, which is semi-structured and consists of documents stored in the form of nested key-value pairs. A key is a string that represents the name of an attribute, and the value is an instance of the attribute that can be a string, a number, a Boolean, an array, or a nested key-value pair. Different documents within the same database may have a different structure, therefore a query intended to retrieve one document may not work for another. Many researchers have designed algorithms to analyze JSON documents and extract schemas from them. However, their contributions mainly provide the structure of the schema, but lacks additional information on how the JSON documents are related that will enhance the quality of the discovered schemas. We aim to extract this additional information by addressing the following challenge: distinguishing data from metadata.

Proceedings of the VLDB 2021 PhD Workshop, August 16th, 2021. Copenhagen, Denmark. Copyright (C) 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In this challenge, we aim to uncover hidden data disguised as metadata by separating data from metadata. In relational databases, there is a clear line that separates data from metadata, but in JSON documents, that distinction is not evident. Within some sets of JSON documents, a portion of the keys is data, but is classified as metadata during schema extraction because the algorithms do not take the possibility of keys being part of the data into consideration. We call this misclassified data *dynamic* keys. We distinguish them from the other category of keys called *static*. *Static* keys are the metadata of the documents. Figure 1 shows two sampled JSON documents adapted from a dataset that contains Amazon products [12]. The key **related** is *static* or metadata because its nested keys are generally constant across the JSON documents and only communicate structural information. On the other hand, the key **salesRank** is *dynamic* and part of the data because its nested keys differ from one document to the next and represent more than just structure.

The distinction between these two categories of keys can be made by analyzing features that we extract from the JSON key-value pairs. We parse the JSON documents, extract numerous features, and use a classification algorithm to separate *static* from *dynamic* keys. Overall, making this distinction within nested JSON documents will enhance the quality of the discovered schemas by clearly identifying which keys are data or metadata, as in relational databases.

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 describes our approach to uncover the hidden data. Section 4 presents the results. Section 6 concludes the paper and enumerates potential future works.

2 RELATED WORK

A JSON schema is a structural representation of a collection of JSON documents that consists of nested key-value pairs in which the keys represent the metadata and the values represent the data. Several researchers developed different algorithms to extract the JSON schema. Their algorithms take a set of JSON documents as input, but provide different outputs. Wang et al. [13] present a framework that outputs a graph data structure to store the schema of each unique document. From this graph, a skeleton model can be formed to represent a summary of the smallest number of attributes that capture the core features of a document.

Klettke et al. [8] present a framework that produces a graph data structure called structure identification graph (SG) that stores the attributes' information such as data type and frequency of occurrence. From an SG, a JSON schema that consists of the attributes and their data types can be generated.

Frozza et al. generate a single schema from JSON documents [4]. The algorithm parses JSON documents, applies aggregation techniques to group and order documents with the same keys and

```
{"asin": "0309069963", "categories": [["Books"]],
 "salesRank": {"Books": 2174268},
 "related": {"also_bought": ["0465022227"], "buy_after_viewing": ["0465022227"],
             "also_viewed": ["0465022227","0309069963"], "bought_together": ["0309069963"]}}

{"asin": "B007M6IMQO", "title": "Adrienne Vittadini Footwear Women's Vida Flat...",
 "salesRank": {"Shoes": 139961, "Clothing":596278},
 "related": {"also_bought": ["B006WVESEK", "B007VMCFLC"], "buy_after_viewing": ["B006WVESEK"],
             "also_viewed": ["B006WVESEK","B00880CLHE"], "bought_together": ["B006WVESEK"]}}
```

Figure 1: Sample Amazon data

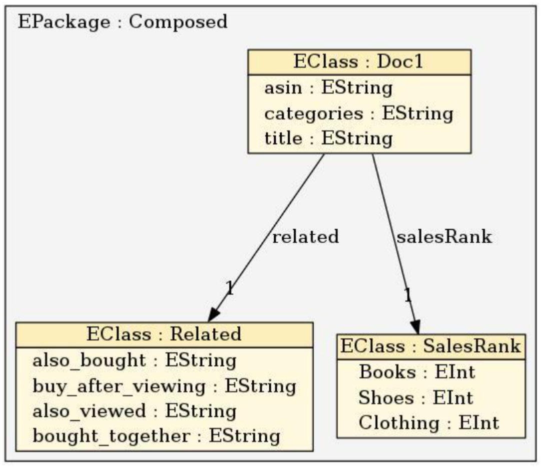

Figure 2: Schema extracted from JSON discoverer tool

remove duplicates, and stores all the information about the JSON documents in a tree-based data structure called Reduced Schema Unified Structure (RSUS). Here information refers to the objects' attributes, their datatypes, the elements within and arrays and their datatypes, and the count of attributes frequencies. From RSUS, a JSON schema can be generated for each attribute.

Cánovas Izquierdo and Cabot [5] develop the JSON discoverer tool that aims to discover and integrate the schemas of JSON documents. This tool has three main functionalities: (1) simple discovery, (2) advanced discovery, and (3) composer. Simple discovery finds the schema of a set of JSON documents and stores it in Unified Modeling Language (UML) format. Advanced discovery takes the output of a set of simple discoveries to infer a global schema. Finally, the composer functionality takes the inferred global schemas as input and produces a graph that is composed of the attributes the inferred global schemas have in common.

Overall, these existing algorithms and tools provide the structure of the JSON documents, but the semantic information they provide is insufficient to understand and analyze those documents. Figure 2 shows the result obtained when using the JSONDiscoverer over the two documents in Figure 1. There is no distinction made between attributes that are data (**Books, Shoes, Clothing**) or metadata (**also_bought**, **also_viewed**, **bought_together**, **buy_after_viewing**).

Spoth et al. [11] design an algorithm, Jxplain, to reduce ambiguity in JSON schemas. They propose a threshold-based model to distinguish collection-like objects from tuple-like objects, the equivalent of our *dynamic* and *static* keys, respectively. The authors calculate two features: datatype entropy and key entropy. The first feature is the entropy of a particular key's value datatype and the second feature is the entropy of the number of keys nested under a particular key. As a result, a key is considered collection-like if it has a datatype entropy of 0 (all nested values are the same type) or if its key entropy is greater than 1. Otherwise, the key is considered a tuple-like object. As we detail in our evaluation, this model is simpler, but does not outperform our feature-based classifier.

3 METHODOLOGY

In this section, we describe the methods we analyzed and will use to address the challenge mentioned above.

3.1 Static vs. Dynamic Keys

Based on our knowledge, there are no existing algorithms that can distinguish *static* keys from *dynamic* ones, in other words, accurately delineate data from metadata in nested JSON documents. A JSON key is a field name and a JSON path is the route to a JSON key. For example, in Figure 3, a modified sample of a dataset containing metadata of games available on the Steam platform [9], **minimum** is a JSON key and **requirements** → **minimum** is the JSON path of that particular key. Our algorithm defines features that will help correctly separate the data from metadata. These features come from various domains such as intrinsic characteristics, central tendency, statistical dispersion, distribution shape, semantic/contextual similarity, and structural similarity. These domains were chosen after manually analyzing the JSON documents. We explain the features from each domain below.

3.1.1 Intrinsic Characteristics Domain. From our preliminary analysis, we observe that *dynamic* keys are generally less frequent and nested deeper than *static* keys. This leads us to examine two intrinsic features: percentage and nesting level. Percentage is the number of times a key appears in all documents relative to the number of documents in the dataset. For example, looking at Figure 3, we count the number of times the key **minimum**, belonging to the JSON path **requirements→minimum** appears. However, some documents may not have this specific key. Nesting level is the depth of each JSON key within a document. We call these features intrinsic because they only communicate each unique key's information that is independent to the presence of other keys within the JSON datasets. In Figure 3, the key **minimum**'s nesting level would be 2.

3.1.2 Central Tendency Domain.

All the features defined within this domain and the two following derive from the number of keys nested under each key as described below.

We count the number of keys nested under each particular key. We notice that *dynamic* keys have more nested keys than *static* keys. For our data, the mean represents the average of the total number of keys nested under a particular key across all the documents of a dataset. In Figure 3, we can see that the *dynamic* key **minimum** has three nested keys (**windows**, **linux**, **macOS**) whereas the *static* key **windows** has two nested keys (**processor** and **memory**).

3.1.3 Statistical Dispersion Domain.

In this domain, we examine the variation among the numbers of keys nested under each particular key. In Figure 1, across the two documents, the number of nested keys under the *dynamic* key **salesRank** varies, while the number of nested keys under the *static* key **related** remains constant. We examine this stability by calculating range, standard deviation, and entropy, the most common measures of dispersion. The range represents the difference between the largest number of keys and the smallest number of keys nested under a particular key. The standard deviation represents the variation within the number of keys nested under a particular key. Entropy is the amount of uncertainty the frequency of the number of keys nested under a particular key produces.

3.1.4 Distribution Shape Domain.

In this domain, we examine the distribution shape of the number of keys nested under *dynamic* keys and *static* keys across the documents and notice that *static* keys have a more normal distribution than *dynamic* keys. For example, in Figure 1, the *static* key **related** has four nested key in each of the two documents, while the *dynamic* key **salesRank** has one nested key in the first document and two in the second one. To examine the shape of this distribution, we calculate skewness and kurtosis. Whereas skewness measures the asymmetry between the frequency of the number of keys nested under a particular key, kurtosis measures the weight of the minimum and maximum of the frequency of the number of nested keys, relative to the mean.

3.1.5 Semantic and Contextual Similarity Domain.

To expand the distinction between of *static* and *dynamic* keys, we calculate three more features: distinct sub-keys, distinct sub-keys data types, and average sub-key contextual similarity. These three features are calculated across all the JSON documents. We choose them because our preliminary analysis shows that the keys nested under *dynamic* keys are usually more unique and related either structurally, semantically, or contextually than the keys nested under *static* keys.

The distinct sub-keys feature shows whether *dynamic* keys have more or fewer unique nested keys than *static* keys. The distinct sub-keys data types feature reveals whether the data types of the keys' values, nested under *dynamic* keys, are generally the same or not, compared to the data types of the keys' values nested under *static* keys. The average sub-key contextual similarity feature indicates whether the keys nested under *dynamic* keys are more or less contextually related than the keys nested under *static* keys.

Distinct sub-keys represent the number of unique keys nested under a particular key. We also set an upper bound of 100 for this

```
{"pegi": {
    "pegi_url": "https://store.cloudflare",
    "pegi_tags": ["Blood", "and", "Gore"]},
 "requirements": {
    "minimum": {
        "windows": {
            "processor": "1 GHz Intel...",
            "memory": "1024 MB RAM",
        },
        "linux": {
            "processor": "1 GHz Intel...",
            "memory": "1024 MB RAM",
        },
        "macOS": {
            "processor": "SSE2 inst...",
            "memory": ""}}}}
```

Figure 3: Sample Steam game data

feature. A key with over 100 distinct sub-keys gives no further important information and significantly skews the distribution of the number of nested keys. Distinct sub-keys data types represent the number of unique keys' values data types under a particular key. The last feature we calculate in this domain is the average pairwise distance of embeddings. A key word embedding is a row of real-valued numbers in which each point represents a dimension of the key's linguistic meaning. We use fastText [2], a word embedding model to extract the vectors of each key from a set of nested keys and determine which ones are contextually related. After transforming the nested keys of a particular key into vectors, we compare each unique pair of vectors to measure their cosine distance and compute the average of these distances. We assume that nested keys part of the same contextual domain will have a smaller distance and be under a *dynamic* key. This phenomenon can be seen in Figure 3, a sample of Steam game dataset [9]. *Static* keys are in red and *dynamic* keys are in blue. The fastText model shows that the *dynamic* key **minimum** has nested keys (**windows**, **linux**, **macOS**) with an average cosine distance of 0.5655. This means that they are in a more similar contextual domain than the *static* key **windows** nested keys (**processor** and **memory**) where the cosine distance between their key embedding vectors is 0.9040.

3.1.6 Structural Similarity Domain.

To reduce the possibility that keys get misclassified, we decide to group keys with the same or similar nested structures within each JSON dataset. Grouping is beneficial because it will improve our classification results by categorizing a set of keys instead of all keys individually. We perform the grouping by using set similarity search [1]. It is an algorithm that measures the similarity of a collection of sets using Jaccard similarity, which gives a score between 0 and 1.

Given a set of sets of all the nested keys, the algorithm compares the sets among each other to find which sets have similarities (keys in common) greater than or equal to a user-defined threshold of 0.7. For example, the keys **windows**, **linux**, and **macOS** form a group because their similarity score is 1, which is greater or equal to the threshold. A score of 1 means that they have the same nested keys,

	Intrinsic Feat.	**Central Tend. Feat.**	**Dispersion Feat.**	**Dist. Shape Feat.**	**Add. Feat.**	**Grouping**
Classifier	**F1-score**	**F1-score**	**F1-score**	**F1-score**	**F1-score**	**F1-score**
Logistic regression	0.0897	0.0906	0.0921	0.0826	0.4875	0.4875
Random forest	0.1106	0.1198	0.1447	0.1272	0.5616	0.5016
SVMs	0.1110	0.1129	0.1029	0.0880	0.4218	0.4218

Table 1: Avg, F1-score Results

which are **processor** and **memory**. Once the groups are formed, we take the average of the percentage and nesting level of the keys that constitute them and replace all the features of the individual keys within a group with the values of that group.

3.2 Data Pre-processing

We collect the above information from various online sources including Kaggle and GitHub [3, 7], just to name a few. Kaggle is a web-based environment where data scientists can publish data sets and GitHub is a web platform for software development and version control. We store the extracted feature information and over-sample the *dynamic* keys because our data is unbalanced; there are disproportionately more *static* keys than *dynamic* keys. To balance our data, we randomly duplicate the records from the minority class (*dynamic* keys) to have as many records as the majority class (*static* keys). We also normalize our data by subtracting the mean and dividing by the standard deviation each row variable to obtain a normal distribution with a mean of zero and a standard deviation of one. The last step in this data preparation involves defining the ground truth by manually labelling *static* and *dynamic* keys.

3.3 Classification Algorithms

After feature extraction, we train a binary classifier to determine whether a key is *static* or *dynamic* across all documents. *Static* and *dynamic* are our two categories. For this purpose, we use cross-validation to split that data into testing and training sets. We group keys from the same dataset together. That means that for the different sets that resulted, each dataset had the opportunity to be the testing set. We then test three algorithms: 1) logistic regression, 2) random forest, and 3) support vector machines (SVMs), on the different training and testing sets.

4 EVALUATION

To analyze our models, we calculate the average F1-score of the *dynamic* keys. This value represents the mean of all the F1-scores obtained from the classification of the *dynamic* keys from each test set. We focus on the average F1-score of the *dynamic* keys because our datasets are highly skewed toward *static* keys and we want to know if our model learns to correctly predict and classify *dynamic* keys with unseen data.

The experimental results can be seen in Table 1. It shows the F1-score as the features from each domain are added progressively (the final column displays results using all of the described features). We obtain low average F1-scores across all three algorithms, but random forest outperforms the other algorithms at distinguishing data from metadata. We observe that the average F1-score sometimes decreases as we add more features from different domains. Currently, we do not know which specific features are detrimental to the average F1-score. Overall, it is challenging to obtain high F1-score values because we have disproportionately more *static* keys than *dynamic* keys.

We also tested JXPLAIN against our datasets and obtained an average F1-score of 0.1304, which is significantly lower than the best results we obtained from the other three algorithms we tested.

5 CONCLUSION AND FUTURE WORK

Our goal is to enhance the quality of the discovered schemas of the JSON documents. For this purpose, we presented a major challenge that needs to be addressed related to JSON schema discovery: uncovering hidden data disguised as metadata by distinguishing *static* from *dynamic* keys. We implemented a new algorithm that classifies the JSON keys as data or metadata by extracting and computing various features. Our next steps involve gathering more datasets with *dynamic* keys, identifying which features are impeding the average F1-score and reducing the number of misclassified keys.

REFERENCES

[1] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up All Pairs Similarity Search. In *WWW '07* (Banff, Alberta, Canada). Association for Computing Machinery, New York, NY, USA, 131–140.

[2] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[3] Justin Dorfman. 2020. GitHub Datasets. https://github.com/jdorfman/awesome-json-datasets.

[4] A. A. Frozza, R. d. S. Mello, and F. d. S. d. Costa. 2018. An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In *IRI 2018* (Salt Lake City, UT, USA). IEEE, New York, NY, USA, 356–363.

[5] Javier Luis [Cánovas Izquierdo] and Jordi Cabot. 2016. JSONDiscoverer: Visualizing the schema lurking behind JSON documents. *Knowledge-Based Systems* 103 (2016), 52 – 55.

[6] Jing Han, Haihong E, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *2011 6th International Conference on PCA*. IEEE, Port Elizabeth, South Africa, 363–366.

[7] Kaggle. 2020. Kaggle Datasets. https://www.kaggle.com/datasets.

[8] Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2015. Schema extraction and structural outlier detection for JSON-based nosql data stores. In *BTW 2015*. Gesellschaft für Informatik e.V., Bonn, 425–444.

[9] Deepan Moorthy. 2020. Steam Games. https://www.kaggle.com/deepann/80000-steam-games-dataset/version/2.

[10] Zachary Parker, Scott Poe, and Susan V. Vrbsky. 2013. Comparing NoSQL MongoDB to an SQL DB. In *ACMSE '13* (Savannah, Georgia). ACM, New York, NY, USA, Article 5, 6 pages.

[11] William Spoth et al. 2021. Reducing Ambiguity in Json Schema Discovery. In *SIGMOD/PODS '21* (Virtual Event, China). ACM, New York, NY, USA, 1732–1744.

[12] Mengting Wan and Julian McAuley. 2016. Modeling Ambiguity, Subjectivity, and Diverging Viewpoints in Opinion Question Answering Systems. arXiv:1610.08095 [cs.IR]

[13] Lanjun Wang, Shuo Zhang, Juwei Shi, Limei Jiao, Oktie Hassanzadeh, Jia Zou, and Chen Wangz. 2015. Schema Management for Document Stores. *Proc. VLDB Endow.* 8, 9 (May 2015), 922–933.