JupySim: Jupyter Notebook Similarity Search System

Misato Horiuchi
Osaka University, Japan
horiuchi.misato@ist.osaka-u.ac.jp

Yuya Sasaki
Osaka University / JST PRESTO, Japan
sasaki@ist.osaka-u.ac.jp

Chuan Xiao
Osaka University, Japan
chuanx@ist.osaka-u.ac.jp

Makoto Onizuka
Osaka University, Japan
onizuka@ist.osaka-u.ac.jp

ABSTRACT

Computational notebooks such as Jupyter notebooks are popular for machine learning and data analytic tasks. Numerous computational notebooks are available on the Web and reusable; however, searching for computational notebooks manually is a tedious task and so far there are no tools to search for computational notebooks effectively and efficiently. In this paper, we develop JupySim, which is a system for similarity search on Jupyter notebooks. In JupySim, users specify contents (codes, tabular data, libraries, and formats of outputs) in Jupyter notebooks as a query, and then retrieve top-k Jupyter notebooks with the most similar contents to the given query. The characteristic of JupySim is that the queries and Jupyter notebooks are modeled by graphs for capturing the relationships between codes, data, and outputs. JupySim has intuitive user interfaces that the users can specify their targets of Jupyter notebooks easily. Our demonstration scenarios show that JupySim is effective to find Jupyter notebooks shared on Kaggle for data science.

1 INTRODUCTION

Many users currently use computational notebook software such as Google Colab and Amazon SageMaker for machine learning and data analytic tasks. They interactively conduct various data processing, for example, data cleaning, analysis, optimization, and visualization. Due to the increasing popularity of computational notebooks, numerous computational notebooks are available and reusable on the Web, such as GitHub and Kaggle [5].

Motivation. We often search for computational notebooks to reuse them for our own data science tasks and to learn programming skills from them. When searching for the computational notebooks, we want computational notebooks including *similar* contents to what we specify, i.e., codes, data, libraries, and/or output formats. For example, we look for computational notebooks analyzing similar data to our data and using libraries and functions that we would like to learn.

However, there are no effective and efficient solutions to searching for similar computational notebooks. We need to consider two characteristics of computational notebooks; (1) unclear running orders of cells (and cells are often skipped) and (2) including analytic datasets. As preliminary experiments, we evaluated the accuracy and efficiency of existing methods for Jaccard-based similar code search [3] and tabular data search [7]. Table 1 shows nDCG@15 (based on user experiments) and search time to find top-10 similar computational notebooks. We can see that existing works are inaccurate because they cannot consider the entire contents of computational notebooks, such as codes and tabular data. If we rank the computational notebooks according to code similarity evaluated by users, the nDCG is 0.763, which indicates the upperbound of accuracy on similar code search methods without considering tabular data. When we combine their results, the accuracy slightly increases but it takes more time, because we need to conduct two searches individually. From the preliminary experiments, we confirmed that existing works are not suitable for similarity search on computational notebooks.

Table 1: Preliminary results

Method	Code sim	Data sim	Combined	JupySim
nDCG	0.61	0.35	0.66	0.77
Search time [sec]	1.58	21.40	22.98	21.92

Contribution. In this paper, we develop JupySim, which is a system for similarity search on Jupyter notebooks[1]. JupySim aims to find the top-k Jupyter notebooks with the most similar contents (including codes, tabular data, libraries, and output formats) to the contents specified by a given query. It provides a Web interface through which users can interactively search for similar Jupyter notebooks by freely inputting/loading/saving queries.

Our work has two technical novelties. First, we define the similarity of computational notebooks that captures the relationships between contents (e.g., order of running cells and reading/updating datasets). We represent computational notebooks by graphs. Second, we reduce the problem of similar computational notebook search to subgraph matching, which leads efficient searches (see Table 1).

For demonstrating our system, we use Jupyter notebooks shared on Kaggle competitions. We prepare three demonstration scenarios to show that JupySim helps to learn how to analyze our own tabular data, how to use libraries, and how to implement data preprocessing, respectively. We provide the source code of our demonstration, *demonstration movies*, and detailed experimental results at our Github repository[2].

Related work. Similarity search methods for source codes [2, 4] and tabular data [6, 7] are actively studied. However, to the best of our knowledge, there are no methods that can be applied to computational notebooks directly. In particular, there are no existing works to define the similarity of computational notebooks and/or collect similarity scores given by people. This causes the difficulty to use machine learning-based methods (e.g., [4]) because they require the pre-defined similarities (e.g., scoring by people and similarity functions) to train machine learning models. We here note that our system is modular; it can use existing methods as similarity functions for contents.

[1] Jupyter notebook (formerly called IPython) is the most popular computational notebook, which is used on Amazon SageMaker and Google Colab.
[2] https://github.com/OnizukaLab/Similarity_Search_on_Computational_Notebooks

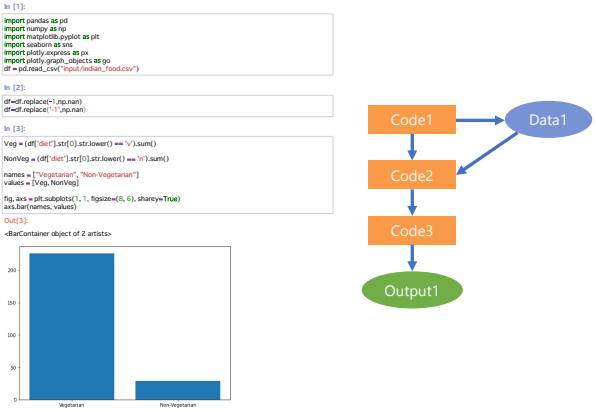


Figure 1: An example of Jupyter notebook (left side) and corresponding DAG (right side)

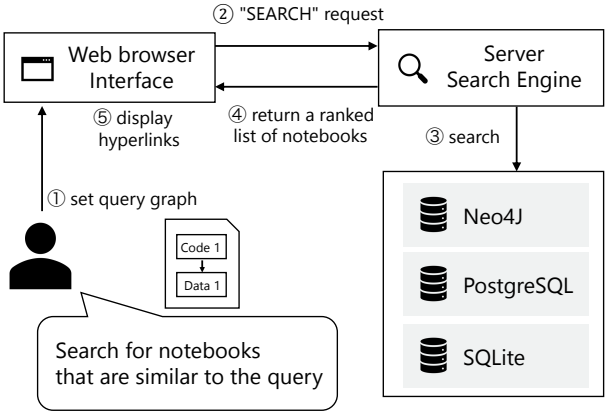


Figure 2: The architecture of JupySim and processing procedures

2 COMPUTATIONAL NOTEBOOK SEARCH

We formalize our problem after defining computational notebooks. Please see our extended report for technical details [1].

2.1 Preliminary

A computational notebook consists of *cells* with codes. The codes in each cell are executed to import libraries, read tabular data to DataFrame, process/analyze the data, and output analytic results.

Definition 1 (Computational notebook.). *A computational notebook N is a 4-pair $(C_N, \mathcal{D}_N, O_N, \mathcal{L}_N)$, where C_N is the set of cells (i.e., source codes), $\mathcal{D}_N$ is the set of tabular data, O_N is the multi-set of output formats, and $\mathcal{L}_N$ is the set of imported library names.*

The left figure in Figure 1 shows a Jupyter notebook. This notebook has three cells; the first one imports libraries and reads a CSV file, the second one conducts preprocessing for the data, and the third one analyzes the data and outputs analytic results.

2.2 Problem formulation

We formulate our problem in this paper. Our similarity search aims to find top-k computational notebooks with the most similar contents to a given query.

In our problem formulation, we define a query Q as sets of contents of computational notebooks; codes in cells, tabular data, output formats, and libraries. Q specifies contents that are expected to be included in the computational notebooks in the search result. The similarity between a given query and computational notebook is defined by a weighted sum of similarity functions of each content as follows:

Definition 2. Computational notebook similarity. *Given a query Q, a computational notebook N, we define similarity between Q and N, $Sim(Q, N)$ as follows:*

$$Sim(Q,N) = w_C sim_C(C_Q, C_N) + w_D sim_D(\mathcal{D}_Q, \mathcal{D}_N) + w_O sim_O(O_Q, O_N) + w_L sim_l(\mathcal{L}_Q, \mathcal{L}_N) \quad (1)$$

where sim is functions to evaluate the similarity between contents in queries and computational notebooks. w is a user-specified weight to control importance of contents.

We can use any similarity functions for *sim*, such as Jaccard similarity.Based on the above definition of similarity, we formulate the problem that we solve in this paper as follows:

Problem Formulation. *Given a query Q, a set $\mathbb{N}$ of computational notebooks, an integer k, and weights, we find a ranked list $\mathbb{A}$ of k computational notebooks such that for $A \in \mathbb{A}$ and $N \in \mathbb{N} \backslash \mathbb{A}$, $Sim(Q, A) \geq Sim(Q, N)$.*

3 OUR SYSTEM JUPYSIM

We develop JupySim, which is a system for similarity search on Jupyter notebooks. JupySim converts Jupyter notebooks into graphs and stores the graphs into databases for efficiently and effectively finding similar Jupyter notebooks.

Figure 2 illustrates the architecture of JupySim and its processing procedures. JupySim has (1) interface, (2) search engine, and (3) database. In the interface, users input queries and see hyperlinks of Jupyter notebooks with top-10 similarity. The search engine accepts queries and returns search results after computing the similarity of Jupyter notebooks in the databases. The databases store Jupyter notebooks (e.g., their datasets), their corresponding graphs, and historical queries.

We explain a graph converting method, search engine, interface, functions, and implementation of JupySim.

3.1 Graph conversion

JupySim converts Jupyter notebooks to directed acyclic graphs (DAGs). Nodes on DAGs represent codes in cells, data, and outputs, and edges represent their flows. For example, if codes in cells read data, corresponding nodes of codes and data have edges. Each node is assigned with a type (i.e., code, data, and output) as a label and has a property corresponding to its label (e.g., a set of records for data label). Libraries are associated with DAGs instead of nodes.

Queries are also represented by graphs as the same style of Jupyter notebooks. Users set nodes and edges in queries as contents (i.e., codes, data, and outputs) that users want and their relationships, respectively. Each node in queries can include its property, for example, nodes representing codes can include names of functions that users want to learn.

The right-hand side figure in Figure 1 shows a DAG corresponding to the Jupyter notebook on the left-hand side. Each cell, data, and output is converted into Code1, Code2, Code3, Data1, and Output1, respectively. This DAG shows that Data1 is read by Code1 and processed by Code2, and Output1 is generated by Code3. Code1, Code2, and Code3 are run in the order.

3.2 Search engine

The search engine in JupySim uses two steps to find top-k Jupyter notebooks; subgraph matching and similarity computation. Each step effectively prunes candidates of Jupyter notebooks that are not obviously included in the results.

Subgraph matching finds Jupyter notebooks that are matched with contents and their relationships specified in queries. For example, if a query includes two nodes labeled with data and both nodes connect to the same node labeled with code, the results contain only Jupyter notebooks satisfying such requirements. Subgraph matching effectively prunes similarity computation of Jupyter notebooks. Since the sizes of queries are typically small, this step does not take a large cost.

Similarity computation computes $Sim(Q, N)$ for finding the ranked list A. For this purpose, we compute the similarity between a given query and Jupyter notebooks that are not pruned by subgraph matching. We compute the similarity between matched nodes according to sim in Equation(1). In the similarity computation, we compute the similarity of data (i.e., $sim_D(\mathcal{D}_Q, \mathcal{D}_N)$) after computing the similarity of code, output, and library. This is because computing the similarity of data is more expensive than computing that of other pairs because datasets are typically large for data analytic tasks. We can prune the Jupyter notebooks that exactly do not have the highest k $Sim(Q, N)$ before computing the similarity of data.

3.3 Web interface

Figure 3 shows a Web interface of JupySim. The four areas are used for visualizing a DAG of query, setting queries, saving/loading queries, and outputting the ranked list of search results, respectively.

By the visualization of a DAG of query, we can easily understand the structure of it, and see the contents of nodes when we hover the mouse over the nodes (see Figure 4). In the area of setting queries, we can input/edit queries (e.g., libraries and nodes) and weights for specifying targets of Jupyter notebooks. Users can try different weights settings to find their targets. For example, if users consider that code and data are more important than library and output, weights of code and data should be larger than those of library and output. We can add/delete nodes and edges, and also edit nodes' identifiers, types, and properties (see Figure 5). In the area of saving/loading queries, we can save the current query to the database and load a query in the database that stores the set of queries saved previously. We can delete the query stored in the database and reset the current query to empty. After clicking the "search" button, we can see the ranked list of hyperlinks of Jupyter notebooks with top-10 similarity to the query. When clicking the hyperlinks, we can see the notebooks on Jupyter Notebook software.

3.4 Functions

JupySim has three functions to improve usability:

- Reachability of nodes: We can set "reachability" as a type of nodes. If we set the reachability, our subgraph matching allows existing (both multiple and single) any nodes between two nodes before/after the node with reachability.
- Library extraction: We can put a fragment of source codes of importing libraries as the target libraries, and then our system automatically extracts the name of libraries. For example, when we input "from sklearn import neural_network, svm" , our system extracts "neural_network" and "svm".
- Load/save queries from/to JSON files: Our system transforms a query into a JSON file to download the query on local systems. We can upload the JSON file into our system. This helps users to easily edit the queries, for example, replacing words with other words.

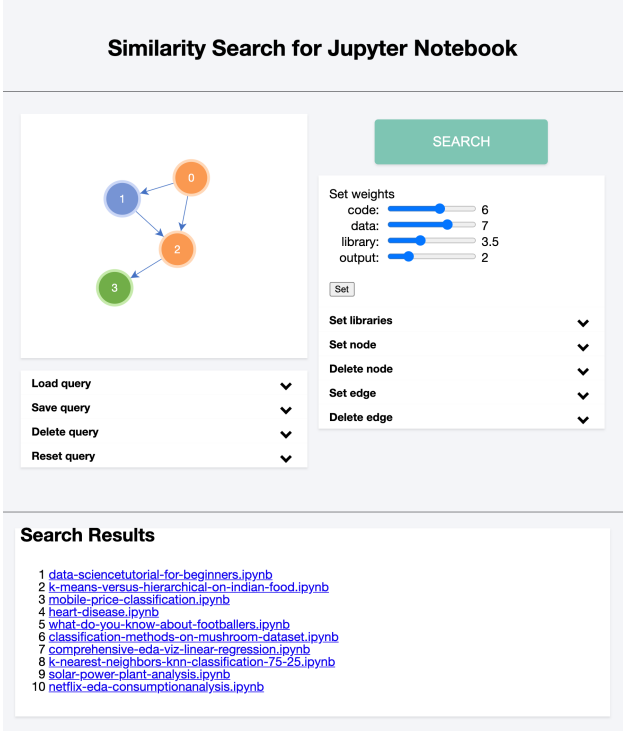


Figure 3: A Web interface of JupySim: This interface has four areas; visualizing a graph of query (left top), editing queries (right top), saving/loading queries (left middle), and the ranked list of search results (bottom).

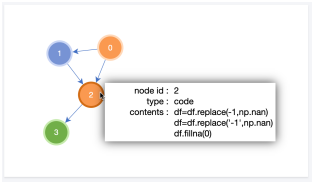


Figure 4: Behavior of mouseover on graphs

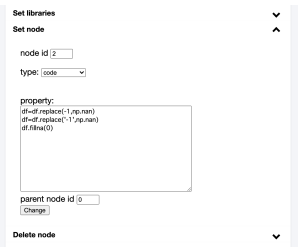


Figure 5: Editing nodes

3.5 Implementation

We use PostgreSQL, Neo4j, and SQLite as database management systems for storing contents (e.g., tabular data) of Jupyter notebooks, DAGs of Jupyter notebooks, and queries, respectively. The search engine is implemented by Python, and the Web interface is implemented by Javascript and Django.

4 DEMONSTRATION PLAN

Our demonstration shows how JupySim helps users to reuse Jupyter notebooks. JupySim supports quickly finding Jupyter notebooks to learn how to analyze data, use libraries, and implement operations. In the demonstration, we use 111 Jupyter notebooks shared on Kaggle competitions and stored them in the databases.

In this demonstration, we first provide a basic flow of similarity search: load a query, modify the query, and find similar Jupyter notebooks to the query. Users can compare the search results to the query in order to evaluate the similarity between them. Then, users can interactively modify the query for finding other Jupyter notebooks. Since JupySim can save and load queries, the users easily modify the queries to find Jupyter notebooks that they want.

We then demonstrate three scenarios to show the usefulness of JupySim for many purposes.

Scenario 1. Analysing datasets: Suppose that we have tabular data and do not have any good ideas for analysing the data. In such case, we would like to refer to other Jupyter notebooks that analyse similar data. JupySim supports finding Jupyter notebooks that read and analyze data similar to our data.

In this scenario, we pose queries that include matplotlib and seaborn as libraries, our data as data, a visualize function as codes, and a figure as output. Cells after reading data and before visualization functions include codes for analysing data. Users can see that JupySim effectively finds Jupyter notebooks including such data analysis.

Scenario 2. Learning libraries and functions: Suppose that we would like to learn how to use libraries and functions. JupySim can efficiently find Jupyter notebooks that import specific libraries and use functions.

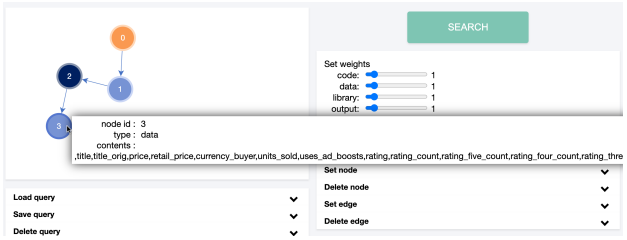
In this scenario, we pose queries that include lightgbm and xgboost as libraries, and then we find Jupyter notebooks importing these libraries. We can learn how to use these libraries from codes included in the Jupyter notebooks. We can additionally include specific functions in queries if we would like to learn the functions. We show that JupySim can support beginners in learning the implementation for data science.

Scenario 3. Implementing data preprocessing: We may conduct data preprocessing such as data cleaning manually for small datasets. Of course, it is difficult to conduct such preprocessing for large datasets manually. We would like to implement these preprocessing but we may not know how to implement them.

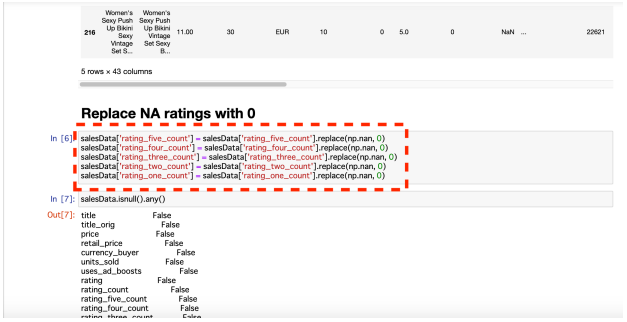
In this scenario, we show that JupySim can find Jupyter notebooks with data preprocessing. We input just data before/after data cleaning and do not input any codes for data preprocessing. Figure 6 illustrates a given query with data (light blue nodes) and the top-1and top-3 search results with preprocessing codes. The query has four nodes; 1. code for reading a CSV file, 2. data before preprocessing, 3. reachability (navy blue node), and 4. data after preprocessing. Top-1 and Top-3 results have different data preprocessing. The former replaces Null to zero and the later deletes records including Null. This result shows that JupySim can support to effectively find Jupyter notebooks for data preprocessing.

5 CONCLUSION

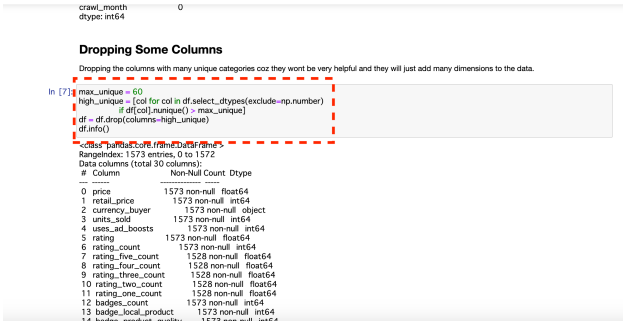
In this paper, we developed JupySim, which realizes similarity search on Jupyter notebooks based on the definition of similarity of computational notebooks. Our search engine converted Jupyter notebooks into DAGs for effectively pruning the candidates of search results by utilizing subgraph matching. We demonstrated that JupySim effectively supports finding Jupyter notebooks what users want. We hope that JupySim helps users to reuse Jupyter notebooks and accelerate data science tasks.



(a) Query



(b) Top-1 result



(c) Top-3 result

Figure 6: Scenario 3: Searching for Jupyter notebooks with data preprocessing

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number JP20H00583.

REFERENCES

[1] M. Horiuchi, Y. Sasaki, C. Xiao, and M. Onizuka. Similarity search on computational notebooks. *arXiv preprint arXiv:2201.12786*, 2022.
[2] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the WCRE*, 2001.
[3] J. Krinke and C. Ragkhitwetsagul. Code similarity in clone detection. In *Code Clone Analysis*. Springer, 2021.
[4] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the IEEE/ACM ASE*, 2016.
[5] C. Yan and Y. He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the ACM SIGMOD*, 2020.
[6] Y. Zhang and Z. G. Ives. Juneau: data lake management for jupyter. *Proceedings of the VLDB Endowment*, 12(12), 2019.
[7] Y. Zhang and Z. G. Ives. Finding related tables in data lakes for interactive data science. In *Proceedings of the ACM SIGMOD*, 2020.