Automating Data Lineage and Pipeline Extraction

Sebastian Eggers
Supervised by Prof. Dr. Abedjan
BIFOLD & TU Berlin
Berlin, Germany
sebastian.eggers@campus.tu-berlin.de

ABSTRACT

Jupyter Notebooks are widely spread in modern data science environments. They allow data professionals to create models, analyze data, and build data pipelines. With an increasing focus on research areas such as explainability and fairness in machine learning, there is a need to understand the relationship between the data and the model in ad-hoc project setups. This doctoral research aims to automate the process of extracting pipelines from Jupyter Notebooks and deriving data lineage from those pipelines without executing the notebook. The goal is to develop a set of tools that identify all datasets, transformations, models, and columns that serve model training inside a notebook without the need for human intervention or execution of these pipelines.

VLDB Workshop Reference Format:
Sebastian Eggers. Automating Data Lineage and Pipeline Extraction. VLDB 2024 Workshop: VLDB Ph.D. Workshop.

1 INTRODUCTION

With the increasing adoption of artificial intelligence use cases in highly regulated sectors, such as financial or medical institutions, and the *AI Act* [7] of the European Union, there is a growing focus on the research areas of explainability, fairness, and interpretability in machine learning [1, 10, 22]. Understanding decision-making in models requires a deeper comprehension of the relationship between the data, the transformations it undergoes until it is used for model training and finally the model itself [13, 19]. For example, if the transformations result in bias within the training data, the resulting model will reflect this bias. Further, data transformations can induce data leakage [23], e.g. by splitting the dataset and applying different transformations to the training and test datasets, respectively. Data lineage is essential in understanding such relationships. Furthermore, the identification of transformation operations on datasets and their respective relationship to the model can support the reuse of existing pipelines [15, 28] and data scientists during data and model engineering [15, 17, 26, 28]. Namaki et al. find that 82% of their participants consider data lineage tracking useful for deploying and bringing models to production[19]. Yet, current systems for extracting pipelines are limited to a set of preselected libraries or knowledge bases [13, 15, 19, 21, 28]. Furthermore, many systems require notebooks to be executed to obtain data flow graphs, which is intractable in an environment with many notebooks because executing each notebook individually demands significant computational resources and time. These issues lead in practice to manual or semi-automatic documentation of data lineage by the data scientist. Further, semi-automatic tools are often unable to store experiment specific data. Therefore, the goal of this doctoral research is to develop pipeline extraction techniques for computational notebooks that overcome these limitations to enable a flexible approach that can easily incorporate new libraries and APIs as part of potential pipelines. Further, we aim to deduce data lineage from pipeline embeddings and to automatically generate new notebooks without the requirement to alter or execute the code. To extract pipelines in such a flexible way, we need a numerical representation of code and data that allows us to detect datasets and data flow between variables. To this end, I aim to answer the following research questions (RQs):

RQ1: How can computational notebooks be embedded into a vector space to encode code structure, and later co-embed data to represent pipeline structures and data transformations?
RQ2: How can pipeline information be extracted from computational notebooks based on code structure and method names without being limited to a closed set of libraries or relying on external knowledge bases?
RQ3: How can computational notebooks be automatically generated to create pipelines, based on previously encoded pipelines, using a dataset as input without human supervision?

2 RELATED WORK

This work is related to lines of research on data lineage, pipeline generation, representation learning.

2.1 Data Lineage

Data lineage or provenance refers to the process of tracking the origin, transformations, and final destination of datasets [14]. Lineage can be classified into two subcategories: The first category concerns the origin of datasets for a given output, known as where-lineage [5]. The second involves identifying the transformations applied to an input to produce specific outputs, called how-lineage. Both subcategories can be categorized further into schema- and instance-level. Schema-level lineage concentrates on identifying the datasets required to generate a given output, whereas instance-level lineage focuses on the individual tuples needed for the output [4]. This research will primarily explore schema-level lineage, aiming to find both where- and how-lineage of datasets in Jupyter Notebooks. Additionally, instance-level lineage could be deduced from the results of the system that will be created, provided the input datasets are available. Ikeda and Widom characterize the problem

Proceedings of the VLDB Endowment. ISSN 2150-8097.

of lineage as having k input datasets $I_1, \ldots, I_k$ that are fed into a directed acyclic graph of n transformations $T_1, \ldots, T_n$, to produce m output datasets $O_1, \ldots, O_m$ [14]. However, this framework does not adequately address the challenges this research aims to tackle. For purposes of model debugging, fairness and compliance, it is crucial to document the datasets fed into a model and the model itself.

2.2 Pipeline Extraction

Pipeline extraction is often referred to as data lineage identification within the scope of computational notebooks in the literature [13, 19, 28]. Nevertheless, most of the papers do pipeline extraction and deduce the data lineage afterward. Namaki et al. propose Vamsa, a system for deriving data lineage in Jupyter Notebooks with pipeline extraction. They analyze abstract syntax trees (ASTs) and workflow graphs and map concepts to a knowledge base for extracting and annotating pipelines [19]. They describe the lineage problem as identifying all triples $\langle M, D, C \rangle$ within a Jupyter Notebook, where M represents the model in the notebook, D denotes the datasets used for training these models, and C refers to the columns utilized for training. Yet, this definition falls short by ignoring all transformations (e.g. merging of datasets, normalization, or imputation) on a dataset, which are essential for debugging models and compliance. Additionally, their system depends on the extent of which the knowledge base contains information about libraries and library versions. Without having a specific data science library (e.g. pandas or scikit-learn) or library version available in the knowledge base, the system is unable to annotate the workflow graph and falls short in extracting the pipeline. This is problematic because libraries frequently change their APIs and new libraries are constantly emerging. Zhang and Ives use data lineage to recommend pipelines or related tables using data lakes while working in a notebook environment [28]. The authors present various table relatedness metrics, one of which is based on shared lineage. The system constructs a variable dependency graph $G = (V, E, F)$ using static code analysis to represent an extracted pipeline. Here, V represents all variables in the notebook, F the set of all functions applied to a variable in V, and E the dependency between two variables via a function in F. Afterward, a provenance graph $PG(v)$ is generated for every variable in the notebook that contains a dataframe. Although this addresses the problem of extraction and lineage, one would need to execute the entire Jupyter Notebook to determine which variable contains a dataframe. Furthermore, the authors' implementation is limited to pandas. Various other systems use similar approaches in extracting pipelines and data lineage compared to the preceding systems [13, 15, 21]. However, the problem of limiting the implementation to specific libraries and also to internal states of those libraries remains.

2.3 Pipeline Generation

Pipeline generation is closely related to the concept of extracting lineage information. Instead of understanding what happens within data science pipelines, the generation of pipelines focuses on creating them. For example, Mustafa et al. use a limited set of lineage information to generate reproducible pipeline code with the aid of user-defined metadata [18]. However, the limited set of operations constrains the system. Dorian is another system for a human-in-the-loop pipeline generation [20]. It uses previously executed pipelines, user queries, and a data science task to generate pipelines. Yan and He propose a system for recommending data preparation steps [25]. Their system uses a dataset of 4 million notebooks and various heuristics to either predict the next data preparation step based on all previous user-defined data preparation steps, or to configure a user-defined data preparation step. Both systems require a user to define either some steps of the pipeline themselves or to execute a great amount of pipelines [20, 25].

2.4 Representation Learning

To encode the notebook and its cells into a vector space, we rely on representation learning. Alon et al. proposed code2vec [3] and code2seq [2] to encode code snippets in the form of ASTs with a path-based attention model into a vector space. Their methods are not directly applicable to Jupyter Notebooks because the structure of a notebook is different from the structure of a code file. Especially the heavy use of library methods in notebooks and the resulting flat AST makes it difficult to encode the structure of a notebook solely based on its AST. Sui et al. add the concept of data flow in their flow2vec model to encode the value flow and reachability between variables [24]. However, flow2vec is also ignoring semantic information (e.g. standardized tokens of data science libraries). Feng et al. proposed CodeBERT [8] to leverage transformer-based architectures for learning bidirectional representations of source code. CodeBERT uses a tokenized form of the code snippet but ignores the information that can be extracted from the AST. Guo et al. propose GraphCodeBERT [11], a pre-trained model that is leveraging both the code snippet in a tokenized form and the data flow between variables. However, only encoding the data flow misses important semantic information like method names. Further, different lines of work try to pre-train or fine-tune large language models to solve table tasks [16, 27]. Extending these methods to code tasks can enable an embedding or model to directly encode transformations.

3 METHODOLOGY

This section introduces the fundamental problems I tackle in my thesis, explores potential solutions, and discusses methods for evaluating them.

3.1 Problem Definition

To support pipeline reusability, debugging, and compliance, through lineage and pipeline extraction, I propose a library-flexible method for extracting pipelines from Jupyter Notebooks without requiring notebook execution. The problem I aim to solve is the automatic identification and extraction of pipeline components within notebooks to derive data lineage and generate new notebooks. Specifically, I will extend Namaki et al.'s problem definition of data lineage in notebooks by including transformations that can be applied to the set of datasets D. Therefore, given a set of Jupyter Notebooks N, the overall objective is to identify all tuples $\langle M, D, T, C \rangle$ within those notebooks. These tuples are a definition of extracted pipelines and can be used to track the lineage in the notebook. Here, M includes one or more models trained in a notebook $n \in N$. M can also

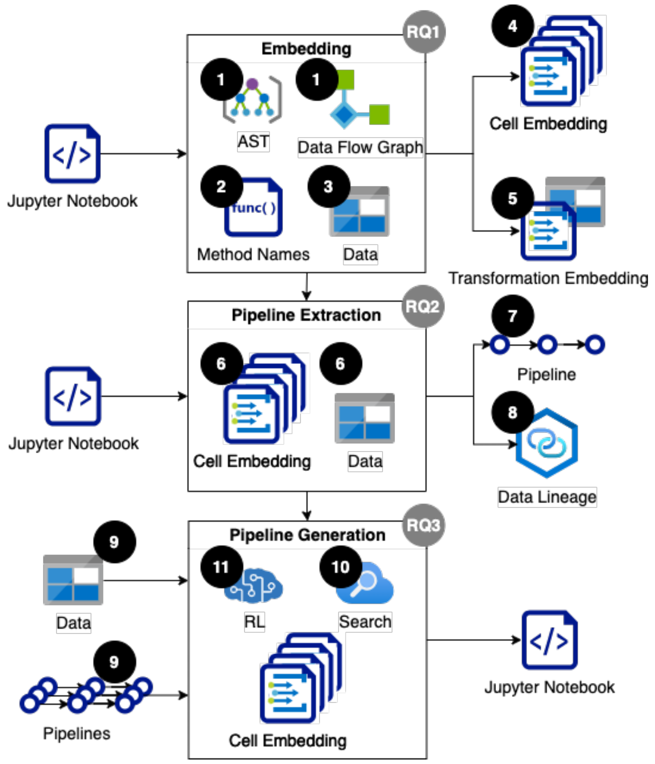

Figure 1: Solution overview for research questions (RQs) 1-3

be an empty set in the case that no model is trained in the notebook. D consists of one or more datasets used in the notebook n. The datasets are used to prepare or enrich data with transformations T. Consequently, transformations T are linked specifically to their datasets, hence represented as $\langle t, d\rangle$ where $d \in D$. The training columns C depend on both the model and the dataset, and are thus defined as $\langle c, m, d\rangle$ where $c \in C$, $m \in M$, and $d \subseteq D$. Every model in M can use a subset of columns in a dataset d. This set is also allowed to be empty in case no model is trained. To obtain the tuples without executing the notebook, it is necessary to encode the notebook and its cells into a vector space to detect patterns in the notebooks. Thus, another goal is to create an embedding function $f(C)$ that maps a cell C to a fixed-sized vector $v \in \mathbb{R}^n$, where v encodes both the structure and the semantic information of the code snippet.

3.2 Solution Approach

This subsection aims to introduce the solution approach of this work and its three main components: pipeline extraction, notebook embedding, and pipeline generation.

3.2.1 Pipeline Extraction (RQ2). Extracting pipelines presents the challenges of identifying the dataset, recording complex transformations, and ensuring the scalability of the approach. The main challenge in solving the problem of identifying and documenting all tuples $\langle M, D, T, C\rangle$ is detecting the dataset. ❼ Without limiting the implementation to a specific library or knowledge base, the system must infer this information based on the structure of the notebook and semantic information, such as method names. ❻ Theoretically, each variable defined in the notebook can contain a dataset. Further, datasets can be loaded in various ways, e.g. using libraries like pandas or via low-level Python APIs such as *open (filename.csv, ‘r‘)*. To address the problem of generalizing across arbitrary libraries and versions, the envisioned system will use the embedding that utilizes the code (as language feature), extracted ASTs, and data flow graphs. Figure 1 displays this workflow (RQ1 and RQ2). This work will evaluate a set of graph-based and natural language processing tools suchs as graph edit distance and fine-tuned language models for this task. Once the datasets are located within the notebook, the system can leverage static code analysis methods to identify the remaining sets M, T, and C. Following the data flow between variables enables the recording of complex transformations. ❼ By analyzing ASTs and code features, our approach is also applicable to custom scripts. Another problem is the scalability of the approach. While static analysis is fast, using a model the size of a large language model to encode cells in a large number of notebooks requires significant computational effort and time. Therefore, fine-tuning auto-encoding models [8, 11, 27] will be preferred to auto-regressive models [16] because they tend to be significantly smaller in terms of parameter size. Once the pipeline is extracted, the system can use this information to replay the pipeline on the dataset to extract the lineage (Figure 1 illustrates an overview of this process in RQ2). ❽ Further, previously extracted pipelines can be stored to generate new notebooks.

3.2.2 Notebook Embedding (RQ1). Embedding a notebook and its cells into a vector space (RQ1) has three additional challenges. ❹ Unlike code2vec [3] and flow2vec [24], the embedding cannot directly rely on the AST ❶ because the structure of a typical notebook is different from the structure of a typical code file: it often relies on heavily using library functions instead of defining these functions within the notebook. Further, the embedding must also encode the semantic information of the code snippet. Especially the method names are of interest here. ❷ For example, API functions for loading a dataset often contain the token *read* as in *pandas.read_csv*, *spark.read.csv*, or *polars.read_csv*. Code embeddings such as code2vec [3] and flow2vec [24] or pretrained models such as CodeBert [8] and GraphCodeBert [11] ignore this information in their representation of the code snippet and in data flow graphs. Additionally to encoding code structure and semantic information, the embedding will be extended in a second step to also include datasets. ❸ Adding data into the embedding enables a connection between the code, which includes the transformations on a dataset, and the data itself. This approach directly encodes instance-level data lineage. ❺ Both embeddings can employ a fine-tuning strategy on an auto-encoding model architecture to leverage already encoded knowledge [8, 27]. To include code and data into one fine-tuned model, we will develop a novel approach that will leverage tasks that includes code, tabular data, and transformations.

3.2.3 Pipeline Generation (RQ3). To solve the problem of pipeline generation, a search and ranking algorithm will be developed to get the most suitable pipeline given a dataset. ❾ The pipeline recommendation will be based on dataset similarity. ❿ Dataset similarity can be measured with table-relatedness metrics such as tuple overlap and n-gram similarity between cells. After finding the most suitable pipeline, the system will adjust the pipeline to achieve a user-defined goal, e.g. to minimize the runtime or maximize a metric like F1. This can be done by using reinforcement learning (RL) methods that adjust the pipeline to achieve the goal. 11

3.3 Evaluation

Next we will lay out our ideas on how to evaluate the proposed approach and framework.

3.3.1 Datasets. There is currently no dataset for data lineage and pipeline extraction. Code4ML contains 8K labelled cells and 20K Kaggle notebooks and is tailored to pipeline generation. However, it lacks non-model-training pipelines [6]. Therefore, I will create a new dataset for evaluating data lineage extraction systems. Data generators, curated datasets from Kaggle, and the augmentation of these will aid in generating such a dataset.

3.3.2 Assumptions and Exclusions. This work pragmatically follows a set of assumptions to obtain a feasible scope for our solution. We focus on Jupyter Notebooks with code written in Python. Only 10.8% of notebooks contained a different language than Python in a 2020 study by JetBrains [12]. Another important assumption is the order of the cells in a Jupyter Notebook. Notebook cells can be executed in an arbitrary order, but not every sequence of executions will work or make sense. Thus, we assume that each notebook is executable from top to bottom. Furthermore, we assume that every notebook can be executed without throwing exceptions.

3.3.3 Baselines. The baseline for the pipeline and lineage extraction will be Vamsa, as it is the only system specifically designed for extracting pipelines and lineage without executing the code [19]. Using the dataset created within this doctoral research, Vamsa and the envisioned system will be evaluated based on the metrics of precision, recall, and F1 score for each parameter of a pipeline $\langle M, D, T, C \rangle$. Additional to enabling the envisioned systems, the benefits of the notebook embedding can be tested against Auto-Suggest [25] by using the embedding and search to create a standalone system or by replacing the heuristic part of Auto-Suggest. Evaluation of the notebook generation system will be done through the quality of the resulting pipeline on three typical data science tasks: supervised classification, unsupervised clustering, and regression. Auto-Suggest [25] and AutoML tools such as Auto-Sklearn [9] will be the baseline for this experiment.

4 CONCLUSION

This proposal outlines the research plan for my PhD thesis on developing a notebook embedding that captures relationships between code structure, semantic information, and transformations on datasets. First, the notebook embedding will encode cells, code structure, and method names into a common vector space. We further expand this embedding to co-embed data to also retain transformation information. With the help of this embedding, we develop approaches for pipeline extraction and generation as well as lineage tracking.

ACKNOWLEDGMENTS

This research has been supported under the grant BIFOLD24B.

REFERENCES

[1] S. Abiteboul and J. Stoyanovich. 2019. Transparency, Fairness, Data Protection, Neutrality: Data Management Challenges in the Face of New Regulation. *JDIQ* 11, 3 (2019).

[2] U. Alon, S. Brody, O. Levy, and E. Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR*.

[3] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. 2019. code2vec: Learning distributed representations of code. In *POPL*.

[4] P. Buneman, A. Chapman., and J. Cheney. 2006. Provenance management in curated databases. In *SIGMOD*. 539–550.

[5] P. Buneman, S. Khanna., and T. Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *ICDT*. 316–330.

[6] A. Drozdova, E. Trofimova, P. Guseva, A. Scherbakova, and A. Ustyuzhanin. 2023. Code4ML: a large-scale dataset of annotated Machine Learning code. *PeerJ Computer Science* 9 (2023).

[7] European Commission. 2023. Regulatory Framework for AI. https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai. Accessed on July 16, 2024.

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*.

[9] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter. 2020. Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning. (2020).

[10] S. Grafberger, P. Groth, J. Stoyanovich, and S. Schelter. 2022. Data distribution debugging in machine learning pipelines. *VLDBJ* 31, 5 (2022), 1103–1126.

[11] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. Kun Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.

[12] A. Guzharina. 2020. We Downloaded 10,000,000 Jupyter Notebooks From Github – This Is What We Learned. https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned/. Accessed on July 16, 2024.

[13] G. Harrison, K. Bryson, A. E. Bamba, L. Dovichi, A. H. Binion, A. Borem, and B. Ur. 2024. JupyterLab in Retrograde: Contextual Notifications That Highlight Fairness and Bias Issues for Data Scientists. CHI.

[14] R. Ikeda and J. Widom. 2009. Data lineage: A survey. *Stanford University Publications. http://ilpubs. stanford. edu* 8090, 918 (2009), 1.

[15] Z. G Ives and Y. Zhang. 2019. Dataset relationship management. In *CIDR*.

[16] P. Li, Y. He, D. Yashar, W. Cui, S. Ge, H. Zhang, D. Rifinski Fainman, D. Zhang, and S. Chaudhuri. 2024. Table-GPT: Table Fine-tuned GPT for Diverse Table Tasks. *Proc. ACM Manag. Data* 2, 3 (2024).

[17] Y. Lin, Y. He, and S. Chaudhuri. 2023. Auto-BI: Automatically Build BI-Models Leveraging Local Join Prediction and Global Schema Graph. *PVLDB* 16, 10 (2023), 2578–2590.

[18] T. Al Mustafa, B. König-Ries, and S. Samuel. 2023. MLProvCodeGen: A Tool for Provenance Data Input and Capture of Customizable Machine Learning Scripts. In *BTW 2023*. 1059–1067.

[19] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer. 2020. Vamsa: Automated provenance tracking in data science scripts. In *SIGKDD*. 1542–1551.

[20] S. Redyuk, Z. Kaoudi, S. Schelter, and V. Markl. 2024. Assisted design of data science pipelines. *VLDBJ* (2024), 1–25.

[21] S. Schelter, J. Böse, J. Kirschnick, T. Klein, and S. Seufert. 2017. Automatically tracking metadata and provenance of machine learning experiments. MLS@NeurIPS 2017.

[22] N. Shahbazi, Y. Lin, A. Asudeh, and H. V. Jagadish. 2023. Representation Bias in Data: A Survey on Identification and Resolution Techniques. *ACM Comput. Surv.* 55, 13s (2023).

[23] P. Subotić, U. Bojanić, and M. Stojić. 2022. Statically detecting data leakages in data science code. In *SIGPLAN*. 16–22.

[24] Y. Sui, X. Cheng, G. Zhang, and H. Wang. 2020. Flow2vec: Value-flow-based precise code embedding. *OOPSLA* 4 (2020).

[25] C. Yan and Y. He. 2020. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *SIGMOD*.

[26] J. Yang, Y. He, and S. Chaudhuri. 2021. Auto-pipeline: synthesizing complex data pipelines by-target using reinforcement learning and search. *PVLDB* 14, 11 (2021), 2563–2575.

[27] P. Yin, G. Neubig, W. Yih, and S. Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. *CoRR* (2020).

[28] Y. Zhang and Z. G Ives. 2020. Finding related tables in data lakes for interactive data science. In *SIGMOD*. 1951–1966.