Dataflow representation of data analyses: Towards a platform for collaborative data science

E. Patterson, R. McBurney, H. Schmidt, I. Baldini, A. Mojsilović, K. R. Varshney

Abstract

Data science plays an increasingly important role in solving today's scientific and social challenges. To promote progress towards a cure for multiple sclerosis, the Accelerated Cure Project (ACP) has created an open repository of biological and survey data on multiple sclerosis (MS) patients. Similar large-scale repositories are being created in other domains. As the open, data-driven model of science proliferates, the research community faces a growing need for a cloud platform for collaborative data science. Such a platform should facilitate collaboration between domain experts and data scientists and possess artificial intelligence capabilities for organizing, recommending, and manipulating data analyses. In this paper, we present some foundational technologies motivated by this vision. Our system automatically extracts a high-level dataflow graph from a data analysis. This graph describes how data flows through an analysis pipeline, including which statistical methods are used and how they fit together. The system requires no special annotations from the data analyst and consumes analyses written in Python using standard tools, such as Scikit-learn and StatsModels. In this paper, we explain how our system works and how it fits into our larger vision for a collaborative data science platform.

1. Introduction

Data science—the umbrella discipline encompassing machine learning, statistics, and computing on data—plays an increasingly important role in solving today's scientific and social challenges. A new brand of data-intensive science is enabled by greater availability of data, collected from high-throughput measurement devices and massive observational sources. Large-scale, open-access data repositories are being created in many scientific domains. For instance, the Accelerated Cure Project (ACP) has created an open repository of biological and survey data on people afflicted with multiple sclerosis (MS). Its mission is to drive progress towards a cure for the disease. Yet the success of data-intensive science is by no means assured. It will require not just data sharing but effective collaboration between domain experts and data scientists. Moreover, the products of collaborative research must be shared effectively with research scientists, clinicians, policymakers, and other stakeholders. All these activities are complicated by inefficiencies in the scientific process.

Although improvements in computing and information technology have enabled remarkable progress in the sciences, the scientific process itself has largely failed to keep pace with these developments. Scientific papers are published in electronic formats designed to closely emulate a traditional physical paper. These unstructured data formats, while comfortable to human readers, do not lend themselves to machine consumption. As a result, automated processing of the scientific literature is challenging. Searching and indexing are limited by the inherent difficulties of natural language processing. Organizing a body of scientific work or conducting a statistical meta-analysis requires painstaking human labor, as the original datasets may be unavailable or

the data analyses poorly documented. Scientific reproducibility is hindered for the same reasons. Meanwhile, according to a recent estimate [1], the scientific literature is growing exponentially at a rate of about 9% per year, which amounts to a doubling in size about every 8 years. A new paper is published in a scientific journal roughly every 20 seconds [2]. The growth of science will only exacerbate the challenge of organizing, assimilating, and utilizing new scientific knowledge.

As a step towards addressing these challenges, we envision a cloud platform for collaborative data science, a single space for domain experts, data scientists, and other stakeholders to share datasets and data analyses. It should be equipped with artificial intelligence capabilities for organizing, recommending, and manipulating data analyses, as well as suggesting relevant datasets or potential collaborators. To enable such features, the platform must possess a high-quality, machine-interpretable representation of its contents, particularly its data analyses. In this work, we propose and implement a foundational technology for automatically extracting a machine representation of a data analysis. The representation is a dataflow graph that captures important steps of the analysis, such as loading data and fitting statistical models, and shows how the steps compose to form an analysis pipeline.

The potential utility of our dataflow representation is illustrated by the following hypothetical scenario. Suppose a user Alice performs a clustering analysis on some gene expression data hosted on the platform. Because the dataset is very high-dimensional, she decides to reduce its dimensionality using sparse principle components analysis (sparse PCA) before applying k-means clustering. Internally, the platform constructs a dataflow representation of her analysis, which shows how the data is loaded, transformed using sparse PCA, and then clustered using k-means clustering. It happens that another user, Bob, has already attempted a similar analysis of this dataset, except that he uses (standard) PCA instead of sparse PCA. The platform identifies these two similar analyses, using its internal representations, and notifies Alice that Bob's preexisting work may be relevant. Alice follows up this suggestion, requesting more information. The platform extracts the clusterings from the respective analyses and displays a comparison using popular metrics like the Rand index. It turns out that the clusterings are quite different, perhaps owing to the high dimensionality of data. Intrigued, Alice investigates further and reaches out to Bob for discussion.

Such use cases reveal the importance of a rich, machine-interpretable representation of data analyses. A simpler representation that ignored data flow, capturing only an unordered list of statistical methods, would not be sufficient. For instance, without knowledge of data flow, the use of PCA as a preprocessing step in clustering, as in the above scenario, is indistinguishable from the use of PCA as a simple aid to visualization in exploratory data analysis. But these two roles are very different. Our dataflow representation enables fine-grained semantic comparisons and manipulations of data analyses.

This paper is an overview of our technical work in its broader context. Future publications will single out particular components of our system for more rigorous treatment. The paper is organized as follows. In the remainder of this section, we summarize our technical contribution. Section 2 is devoted to an extended motivation of this project. We outline a vision for a

collaborative data science platform, drawing on the experience of the Accelerated Cure Project, an early advocate of open, data-driven MS research. In Sections 3 and 4, we explain the methodology behind our system and survey the relevant literature. Finally, we look to the future in Section 5, describing related open problems in computer program analysis, machine learning, statistics, and knowledge representation.

Summary

Briefly, our system automatically extracts a dataflow representation of a data analysis, which is interpretable by machines.

In this section, we clarify the meaning of that sentence and illustrate it with an example. For our purposes, a *data analysis* is a computer program, in the form of either a source file or a Jupyter notebook, that executes a sequence of data analysis tasks. (A Jupyter notebook [3, 4] is an interactive document that interleaves computer code with natural text, mathematical equations, and visualizations.) A typical data analysis program might perform a clustering analysis, fit a sparse linear regression model, or test a statistical null hypothesis. At present we require that the program be written in Python, but this limitation is not fundamental; we hope to add support for R and Julia in the near future.

The representation that we extract is a *dataflow graph* (or simply *flow graph*) summarizing the execution of the analysis. The flow graph is a directed acyclic graph (DAG) whose structure will be described in detail later. Informally, it should capture the most semantically relevant steps of the analysis, such as: reading a data file, fitting a statistical model, making predictions, computing error metrics and p-values, saving transformed data to a new file, and so on. In any realistic analysis, there will also be some steps that the system cannot interpret. For example, data munging tasks like data cleaning, sorting, filtering, and feature extraction are generally not interpretable by our system. Although we hope to improve coverage of important areas like data munging in the future, it is unrealistic to ever expect complete coverage. By necessity, the dataflow graph will contain both steps that are interpretable, and steps that are not interpretable, by the machine.

The construction of the dataflow graph is *automatic* in the sense that it requires no additional annotations by the data analyst. Put differently, the only input to the system is the computer program constituting the data analysis. However, the system does rely on human annotations of another kind, namely for the underlying analysis tools. For example, if the analyst fits a support vector machine (SVM) using the Python library Scikit-learn [5], then the system's annotation database must include entries for the SVM library functions. This approach to functional annotation leads to a large gain in human efficiency, since library annotations written by a single individual can be leveraged by the whole community. It also supports one of our most important design goals: to be minimally intrusive to the workflows of experienced data analysts.

Finally, we clarify the sense in which our dataflow graph is *interpretable*. To achieve our goal of producing a machine-interpretable representation of a data analysis, it will not suffice to merely

attach labels to steps in the analysis; the system must also "know" how these labels are related to each other. For instance, suppose an analysis involves a logistic regression model, to which the system assigns the label "logistic regression". To put this step in context, the system should understand that logistic regression is a type of classification model, which is in turn a type of predictive model. Or, to take another example, suppose two analysts each perform a clustering analysis of the same dataset. The first uses k-means clustering, receiving the label "k-means", and the second uses hierarchical clustering, receiving the label "hierarchical". The system should be able to recognize two instances of clustering and compare them accordingly. To enable these features, our system aligns the dataflow graph with a *knowledge base* (or *ontology*) of data analysis concepts.

As an example, **Figure 1** shows the dataflow graph extracted from an exploratory data analysis of ACP's survey data on MS patients. The grey nodes represent *entities* (roughly, objects in the computer program) and the green nodes represent *actions* (roughly, function calls), terminology that will be clarified in Section 3. The edges represent inputs and outputs of actions and properties of entities. The objective of this data analysis is to understand how the symptoms of MS are distributed across the population of patients and how they are related to the four clinically recognized MS disease types. To provide context for Figure 1, we describe the major steps of the analysis here. The table of symptom indicator variables is loaded and studied using multiple correspondence analysis (MCA), a variant of principal components analysis (PCA) suitable for categorical data. K-means clustering with four clusters is then applied to the top four MCA factor scores. The resulting clusters are visually compared with the four disease types (loaded from a separate file) in a scatter plot of the top two factor scores. With some effort, a human viewer can find all this information in the dataflow graph of Figure 1. From the perspective of machine interpretability, the *size* of the dataflow graph, perhaps uncomfortable to human viewers, is much less important than its *meaning*, which is derived from the knowledge base.

2. Motivation

While it may present a worthy technical challenge, the formation of a dataflow graph representing a data analysis is ultimately a means, not an end. The methodology of dataflow representation, realized in a working prototype, belongs to a broader vision of collaborative data science, which has not been realized. Our system is designed to be a foundational technology for a cloud platform for collaborative science. In this extended motivation, we propose several requirements for a collaborative science platform, argue that no existing platform satisfies them, and explain how our technology could drive a new kind of data science platform. To focus the discussion, we consider a concrete use case in the domain of neurodegenerative disease.

Accelerated Cure Project

Multiple sclerosis (MS) is an inflammatory neurodegenerative disease of the central nervous system. It is thought to affect over 400,000 Americans and 2.5 million people worldwide. It is the leading non-traumatic cause of neurological disability in young adults. Yet, despite many

decades of research, including the recent use of powerful high-throughput techniques (genomics, proteomics, etc.), our understanding of the etiology (causes) of this disease remains extremely limited. Individual genetic and environmental risk factors have been identified, but it is not known how interactions among these factors cause the onset of the disease. The evidence suggests that MS is an etiologically heterogeneous and multifactorial disease, i.e., MS is not caused by any single factor, and the set of factors that interact to cause it varies between individuals. Likewise, the progression of the disease and the efficacy of existing treatments are highly variable and thought to have heterogeneous, multifactorial underpinnings.

Understanding these complex interactions requires the synthesis and analysis of patient data from diverse sources. To that end, the Accelerated Cure Project for Multiple Sclerosis (ACP) has curated a large-scale, high-quality repository of MS patient data, including clinical records, patient self-reports, and biomarker data (both genetic and proteonomic). ACP's data curation strategy has two major thrusts: first, to share its physical biosamples with laboratory researchers on an open-access basis to generate analytical data that can answer specific questions, and second, to aggregate this analytical data along with clinical and patient-reported data and share this collection with data scientists on an open-access basis to generate new insights about MS. This approach to "virtual collaboration" allows scientists to learn from and build upon each other's analyses without needing to enter into formal collaborations or legal contracts.

The efficiency of this model of collaborative science is currently limited by the absence of suitable collaboration technologies. It is essential to share not just the physical biosamples and the derived datasets, but also the statistical analyses of that data and any ensuing scientific conclusions. All these steps of the scientific process should be integrated within a unified, open-access knowledge sharing platform. IBM and ACP are working together to develop such a cloud platform, with the goal of supporting the continuing analysis of ACP datasets by scientists worldwide.

Platform requirements

The model of MS research promulgated by ACP is applicable more generally in the natural and social sciences. Complex biological systems and social institutions have resisted traditional mathematical modeling, while the efficiency of computing and data collection technology continually improves. These factors are driving a new kind of data-intensive science, characterized by large datasets and greater reliance on statistics and machine learning. The success of this paradigm depends not just on data sharing (important though it is), but on effective collaboration between domain experts and data scientists. In general, neither party—scientist or statistician—is equipped to solve the problem individually, but must rely on the other's expertise. At a higher level of organization, the research group must interact with other groups to disseminate their findings and track progress in the field. Information technology plays an increasingly important role in these knowledge sharing activities.

Our thesis is that a cloud platform for collaborative data science could lead to dramatic efficiency gains for the Accelerated Cure Project and for the larger scientific community. It may be objected that such platforms already exist, rendering a new one superfluous, or that existing

platforms offer limited benefits, having failed to impact the scientific process to the degree that online platforms have impacted open-source software development. In response, we propose two requirements for a collaboration platform, which we believe to be necessary for significant efficiency gains but which are not, to our knowledge, satisfied by any existing offering. First, the content hosted by the platform should not be restricted to datasets or error metrics, but should include what is probably of greatest scientific value, the data analyses themselves. Second, the content should be represented in a machine-interpretable form.

Since these requirements are not universally recognized, we offer a few points in their defense. Why is it essential to host complete data analyses, rather than summary statistics like the prediction error on held-out data? Under the *challenge* model of data science, organizers define a prediction problem and teams throughout the world compete to achieve the best error rate. The paradigmatic platform in this class is Kaggle, with variations offered by Driven Data [6] (focus on social good), Dream Challenges [7, 8] (focus on systems biology and translational medicine), and OpenML [9] (focus on open science). Naturally, this model is well-suited to questions that can be neatly formulated as supervised learning problems. Such questions occur infrequently in the sciences. More often, scientific questions are open-ended, multifaceted, and irreducible to a single real number captured by a loss function. On this view, data analysis is an open-ended activity that may involve (informal) exploratory data analysis and (formal) statistical inference beyond prediction. A successful platform for collaborative science will capture these activities. Consequently it must publish complete data analyses in addition to whatever error metrics are deemed appropriate.

Machine interpretability of the platform's content is important for several reasons. A first is the *scale* of science. The exponential growth of science, cited in Section 1, makes it ever more difficult for researchers to track the progress of their own field, much less other, related fields. Machines could conceivably help researchers filter and organize the body of scientific knowledge, but that is only possible insofar as the content is intelligible to machines. Machine interpretability is also related to another requirement for scientific publication: reproducibility. Our system requires that the analysis code be both available and executable, the same requirements posed by reproducibility. More generally, the rigidity of machine formats promotes detail and precision in the description of scientific work. This is good for reproducibility and hence for science as a whole, provided that the additional burden placed on human researchers is tolerable.

Among existing platforms, challenge platforms like Kaggle store error metrics for the uploaded models. These metrics are obviously machine interpretable and are utilized to create public leaderboards of the best models. But as we have seen, the challenge platforms do not meet the first requirement. There is a different category of cloud platforms that treat data analyses as first-class citizens. These platforms are designed to make existing distributed computing frameworks, such as Hadoop and Spark, and data analysis environments, such as Jupyter Notebook and R Studio, conveniently available on the cloud. Some exemplars of this rapidly growing space are Domino Data Lab, IBM Data Science Experience, and Microsoft Azure Machine Learning. These platforms do not form useful machine representations of their content. We are not aware of any existing platform that satisfies both our requirements.

Platform features

Having motivated our hypothetical platform in the abstract, we now consider some specific features that could be enabled by a high-quality, machine-interpretable representation of a data analysis. An obvious first application is a sophisticated query engine. On a platform using our system, users could ask queries like "Find all analyses that take dataset *D* as input, perform clustering analysis, and output three clusters" or "Find all plots of variable *X* against variable *Y* (drawn from dataset *D*)". It is impossible to make such highly structured queries using existing indexes like Google Scholar.

More ambitiously, the machine representations could serve as input to artificial intelligence or machine learning algorithms that operate on data analyses. For instance, we envision a recommender system that automatically identifies relevant data analyses or potential collaborators on the basis of shared datasets, methodology, and social connections. Likewise, we could try to identify analyses with novel (but fruitful) methodology, a form of outlier detection. One might even try to develop an automated, personalized system for organizing and summarizing a body of scientific work. At present, such surveys must be conducted at great cost by human subject-matter experts.

All these features operate on existing data analyses created by human scientists. In another frontier of artificial intelligence, called *computational creativity* [10, 11], machines play an active role in the creation of new content. The platform could surgically modify existing analyses, replacing certain steps with semantically compatible ones (e.g., replacing k-means clustering with hierarchical clustering), or even generate entirely original analyses. The new analyses would be evaluated by some combination of novelty and quality metrics, where novelty is measured against existing analyses on the platform. We note that creative applications require a *bidirectional* representation: the high-level dataflow graph must be converted back into executable computer code. This capability, interesting in its own right, is not currently supported by our system.

3. Dataflow representation of data analyses

In this section, we elaborate on our technical contribution, a method for automatically creating dataflow representations of data analyses. We will focus on the architecture of our system, describing its major components and how they interact. Due to space limitations, we will treat the underlying algorithms only briefly, eschewing both mathematical formalism and implementation detail.

In the summary of Section 1, we spoke of "the" dataflow graph, but there are actually three different dataflow graphs in our system, which we call the *raw flow graph*, the *annotated flow graph*, and the *semantic flow graph*. In the rest of paper, we will generally use these more precise terms to refer to the dataflow graphs. The semantic flow graph is the final output of our system and is exemplified in Figure 1 above. The other two graphs are intermediate forms.

The relations between the three graphs are shown in **Figure 2**, which schematizes the

architecture of our system. A *runtime environment* for data analysis, such as the Python interpreter, supplies a stream of trace events to the system. Trace events are emitted for each function call and matching function return. The system processes these events in an online fashion, building up the raw dataflow graph as the analysis executes. The raw graph is a directed graph whose vertices represent function calls and whose edges represent objects passed between them. The "raw" graph is so called because it is low-level, language-dependent, and generally uninterpretable. For a typical data analysis encountered in practice, which involves not just statistical modeling but also data cleaning and preparation, the raw graph will contain hundreds or thousands of nodes. This representation is too large and detailed to be readily interpretable by humans, yet too unstructured to be interpretable by machines.

The subsequent stages of the pipeline transform the raw graph into more useful representations. These stages use auxiliary information from an *annotation database* for statistical software and a *knowledge base* of universal statistical concepts. First, the annotated graph is constructed from the raw graph by attaching annotations to functions and objects that have entries in the annotation database. In addition, subgraphs of unannotated vertices are collapsed into single vertices. The latter transformation tends to greatly reduce the size of the graph, as most function calls are unannotated. In the final stage, the semantic graph is created from the annotated graph by using information in the annotations to identify specific language and library constructs with universal concepts from the knowledge base. Unlike the raw and annotated graphs, the semantic flow graph is independent of the particular language (such as Python or R) and libraries (such as Scikit-learn or StatsModels) used to implement the data analysis.

From an architectural point of view, we emphasize that our system is *modular*. It is possible for the research community to build on some components of our system while ignoring or replacing others. For example, a researcher could retain our generic system for functional annotation without adopting our knowledge representation formalism. Consequently, although we have designated the raw and annotated graphs as "intermediate forms," we regard them as having independent interest.

In the remaining subsections, we describe the raw, annotated, and semantic dataflow graphs in greater detail. In Section 4, we situate our work within the relevant literature.

Raw flow graph

The *raw flow graph* is a nested directed acyclic graph (DAG) that captures the dataflow within a single run of a computer program. (Throughout the paper, the word "graph" should be understood as "multigraph", i.e., a graph that can have multiple edges and self-loops.) The vertices of the raw graph are in one-to-one correspondence with the function calls made during the run and are labeled by the name of the called function. Here the word "function" should be interpreted in the sense of programming languages, not mathematics. Thus, in a typical object-oriented programming language, our usage of "function" encompasses static methods, instance methods, object attribute accessors, container accessors (indexing), and built-in unary and binary operators. Of course, the precise scope of "function" depends on the target programming language, but roughly speaking our usage encompasses any reusable computational unit.

In the raw graph, there is a directed edge from vertex *u* to vertex *v* if the function call *v* takes as an input an object which is an output of the function call *u*. The edge is labeled with an ID that uniquely identifies the object being passed from *u* to *v*, as well as the name of the object's type. We insist that the object ID be unique across the entire program run, not just unique among all existing objects at the time of the function call. In particular, this means that object IDs cannot be memory addresses (since memory is reused). Nonetheless, in practice a mapping is maintained between object IDs and memory addresses.

To accommodate the hierarchical structure of computer programs, the raw graph is also equipped with hierarchical, or nested, structure. Specifically, each vertex of the graph may itself be a DAG, and so on recursively. (The notion of a *nested graph* is natural and appears frequently in computing practice, e.g., in the graph serialization protocol GraphML [12] and in the graph visualization software Graphviz [13]. Precise mathematical formulations are relatively rare, but see the recent work on *operads* [14, 15].) The hierarchical structure of the raw graph reflects the hierarchical structure of computer programs, in which functions can invoke other functions. Our system captures the internal structure of some functions, called *analyzable*, and ignores the internal structure of others, called *unanalyzable*. When an unanalyzable function is called, an ordinary vertex, with no nested structure, is created in the graph. When an analyzable function is called, a vertex containing a nested graph is created. The graph construction algorithm then operates on the nested graph until the original function returns. In this way, the nesting of the raw flow graph mirrors the call stack of the computer program.

Whether a function is analyzable is determined by both technical constraints and performance considerations. For instance, in Python it would be challenging to trace the execution of extension code written in C or Fortran. Such functions are therefore unanalyzable. Besides this constraint, we have chosen to designate user-defined functions as analyzable and library functions as unanalyzable. (By "user-defined" functions we mean functions defined in the data analysis script or notebook, and by "library" functions we mean functions defined in the standard library or in external libraries like Scikit-learn and Matplotlib.) In effect, this convention ensures that only user code is traced.

An example of a raw dataflow graph is shown in **Figure 3**. The raw graph associated with our exploratory analysis of the MS data (Figure 1) is too large and complex to have didactic value or fit comfortably onto the page. Instead, we show the raw graph derived from the Python program in **Listing 1**. In this toy example, a tabular dataset is read from a CSV file, a linear regression is fit, and the training error is computed in two different metrics, mean squared error and mean absolute error. The reader will notice that the raw graph is not readily interpretable, even for such a trivial program. In this example, the raw graph has no nested structure.

To faithfully extract a dataflow representations of a computer program, one must confront not only practical software engineering issues, but also several fundamental challenges. These conceptual challenges arise because there are essential differences between the paradigms of imperative, object-oriented programming and dataflow programming. Space does not permit a thorough discussion, but we will outline two important challenges only partly addressed in our work.

The most glaring mismatch between a programming language for data analysis and our dataflow programming model is that the former is usually *imperative* while the latter is *purely functional*. In other words, objects are mutated in typical computer programs, while mutation does not belong to the specification of the raw flow graph. Our system handles mutation within a function by adding a new output to the function for each mutated object. That sounds simple enough, but complications arise because some mutations are *implicit*. For example, if a column in a data frame is mutated via a reference to that column, then the containing data frame should be regarded as mutated as well. At present, our system can detect only direct mutations. Furthermore, we rely on annotations to capture mutations from within unanalyzable function calls.

A second conceptual challenge is the representation of *control flow* in the dataflow graph. Most languages have control flow structures for conditional branching (`if` statements), looping (`for` and `while` statements), and recursion. Of these categories, only recursion is naturally represented in dataflow graphs, as a special case of hierarchical function composition. Traditional `for` and `while` loops are particularly awkward due to their inherently imperative nature [16]. Our system "solves" the problem of control flow by ignoring it: it captures only the function calls actually invoked during the execution of the program, not those that might have been invoked on different input data. Thus, given a branch statement, the system captures only the executed branch, and given a loop, the system captures the unrolled sequence of iterations.

This limitation, while important, is not as severe as might initially be supposed. A typical computer program, say a GUI application or a web server, is expected to run on many different user inputs and behave differently each time. A semantic representation of these programs that failed to capture control flow would be totally inadequate. In contrast, data analyses are usually created for and attached to specific datasets. As a requirement of scientific reproducibility, we expect that if the program is run repeatedly on the same data, it will produce the same result each time. It is therefore reasonable to ask what actually did happen on given data, rather than what might have happened on different data. Of course, we might prefer to ask *both* questions simultaneously. We leave that possibility to future work.

Annotated flow graph

The *annotated flow graph* is derived from the raw graph by attaching annotations to recognized functions and objects. In contrast to the raw graph, objects are represented not by edges but by a new type of vertex. The annotated graph is thus a *bipartite* DAG, whose two types of vertices we shall call *entities* and *actions*. We adopt this more abstract terminology because a single action in the annotated graph can correspond to multiple function calls in the raw graph, as we shall explain. In this subsection, we summarize our system's annotation database and the major steps in the construction of the annotated flow graph.

An entry in the annotation database is simply a JavaScript Object Notation (JSON) document identifying a library class or function by its full name. Examples of class and function annotations are shown in **Listings 2 and 3**. In the case of a class, individual attributes or slots of the class can be annotated; in the case of a function, arguments and return values can be

annotated. The annotation system is integrated with the object-oriented programming model. For instance, the annotation in Listing 3 applies to the fit method of *any* class inheriting both the base class `BaseEstimator` and the mixin class `RegressorMixin`. When multiple annotations match the same class or method, the most specific annotation is selected. Besides these features, the content of the annotation is basically arbitrary at this stage. In particular, the `type` field in the annotations is used to construct the semantic graph, but plays no role in the annotation process.

There are two major steps in the transformation of the raw graph to the annotated graph. First, annotations are attached to vertices (function calls) and edges (objects) in the raw graph by matching their full names to entries in the annotation database, as sketched above. Second, the topology of the graph undergoes several transformations to promote interpretability. A new class of *entity* vertices is created to represent objects. Sequences of unannotated function calls are merged into a single *action* vertices, provided that all ancestor-descendant relations between annotated function calls are preserved. Unannotated actions missing either annotated inputs or annotated outputs are removed entirely. In realistic data analyses most function calls are unannotated, so these transformations tend to dramatically simplify the structure of the graph.

Semantic flow graph

The *semantic flow graph* is derived from the annotated graph by aligning it with a knowledge base of data analysis concepts. It is the final output of our system. We have already seen an example semantic graph (under the name "dataflow graph") in Figure 1. A second example, corresponding to the Python program in Listing 1, is shown in **Figure 4**. In this subsection, we elaborate on our formalism for knowledge representation. Our exposition will be terse, as this component of our system is under active development. We hope to provide a more satisfactory treatment in a future publication.

The knowledge base utilized by our system is an *ontology log* (or *olog*) [17, 18]. The olog formalism reinterprets the classic idea of a semantic network [19] in the language of category theory. An olog contains *types*, which taxonomize the objects in the ontology; *aspects*, which define functional relationships between types; and *facts*, which represent equivalences between aspects. Mathematically, an olog is a category whose objects are types, morphisms are aspects, and commutative diagrams are facts. More information is available in the original paper [17], which is readable without prior knowledge of category theory.

We have begun to create an ontology log of the concepts of data analysis. Its types include both entities, such as data tables and statistical models, and actions, such as reading a data file or fitting a model. **Figure 5** shows a selection of the clustering methods in the olog. This excerpt of the olog models "is-a" relationships (agglomerative clustering is a type of a hierarchical clustering) and "has-a" relationships (a k-means clustering model has centers or centroids), among other kinds of relationships.

A salient feature of ologs is that they can be equipped with *instance data* (or *models*). Intuitively, a model of an olog is an instantiation its types, loosely analogous to how objects in an object-

oriented programming language instantiate their classes. In mathematical terms, a model of an olog is a functor from the olog to the category of sets. With this definition, the semantic flow graph is just a model of (instance data for) the data analysis olog. It is constructed from the annotated flow graph using the type information in the annotations.

4. Related work

Our system automatically constructs a dataflow representation of a data analysis by tracing the flow of objects at runtime. Thus, there are two essential elements in our work: (i) describing a data analysis as a flow graph, and (ii) tracing data flow in a computer program. Naturally, there is some prior art in both of these areas. However, to our knowledge, we are the first to combine these elements to produce a fully automated system for summarizing a data analysis. In this subsection, we review related work in these two areas. Prior work on data science platforms is reviewed in Section 2.

Regarding the first element, we should acknowledge the obvious fact that many data analysis environments *explicitly* represent analyses as dataflow graphs. Almost always, these environments are software applications with graphical interfaces, not programming languages; examples include open source software like Knime and Orange and commercial offerings like SPSS Modeler and RapidMiner. Graphical environments for scientific workflow management, such as Galaxy [20], Kepler [21], and Taverna [22], belong to a related category, as they often include data analysis functionality. In this paper, we focus exclusively on the programming language model of data analysis, in particular the Python language.

To some extent, our goal of recording the steps of a data analysis is shared by the field of *data provenance*. The provenance of a data resource includes its origin and the process of transformation by which it was derived [23]. Insofar as the survey [24] is representative, we would argue that the main difference between our system and the typical data provenance system is *granularity*. In data provenance, the finest granularity of data resource is usually files or database records, whereas our system operates on a single file and traces arbitrary programming language objects. For example, the StarFlow system [25] is, like our's, aimed at data analysts and built around Python, but operates at the level of scripts. There are some graphical environments for data provenance that offer finer granularity, such as VisTrails [26], but again our work targets text-based programming languages.

Turning to the second element, the analysis and instrumentation of computer programs is a large field of research unto itself. In the computer science community, dataflow analysis is usually a means to the end of building better compilers, debuggers, and verification tools. A dataflow analysis can involve static analysis or dynamic analysis (or both), but the literature emphasizes static analysis because of its relevance to optimizing compilers [27, 28]. Static dataflow analyses are classified as *intraprocedural* (within a single procedure) or *interprocedural* (between procedures across the entire program). Interprocedural analysis includes the construction of function call graphs [29], which we discuss below. On its own, static analysis is insufficient for our purposes because we require information that is *only* available at runtime, such as the column names of data tables and the parameters of statistical models.

To our knowledge, the most relevant work in the area of dynamic analysis is Adrian Lienhard's *dynamic object flow analysis*, described in a dissertation [30] and several papers [31, 32]. Like us, he proposes to track how objects are passed between functions at runtime. However, his approach differs in both purpose and implementation. His primary goal is to measure object aliasing in large object-oriented systems, as an aid to debugging and reverse engineering. His work does not explicitly address the problem of mapping the (imperative) object-oriented programming model onto the (functional) dataflow programming model. In our work, we offer only a partial solution to that problem, which is in general quite challenging.

Another related concept is a *dynamic call graph*, or a call graph constructed at runtime. In a (context-insensitive) *function call graph* [29], each vertex corresponds to a function and there is a directed edge from vertex *u* to vertex *v* if function *u* calls function *v*. The definition is reminiscent of the raw dataflow graph of Section 3, but there is a crucial difference: vertices in a call graph correspond to functions, whereas vertices in a dataflow graph correspond to function *calls*. This difference dramatically changes the structure of the graph. In a call graph, recursive functions create self-loops or cycles, while the dataflow graph is always acyclic, albeit possibly nested. In general, it is not possible to reconstruct the dataflow graph from the call graph because the latter does not contain enough temporal information. This problem remains even when the call graph is allowed to be *context-sensitive*, i.e., to have multiple vertices for a single function, corresponding to different calling contexts (call stacks).

5. Outlook

In this work, we have proposed and implemented a novel method to automatically extract a machine-interpretable representation of a data analysis. We argued that the scientific community could profit greatly by a new kind of collaborative data science platform, emphasizing open-ended data analysis and rich, machine-interpretable content. Our system could serve as a foundational technology for such a platform.

We hope that the reader shares our excitement about the prospect of more tightly integrating information technology, machine learning, and artificial intelligence with the scientific process. This development is in its nascent stage and open research and engineering problems abound. By way of conclusion, we will highlight several problems that have occurred to us in the course of this work.

There is considerable scope for future work even within the confines of our technical contribution. In Section 3, we mentioned limitations in detecting object mutations and representing control flow in the raw dataflow graph. To capture control flow, the existing dynamic analysis must be supplemented with static analysis. At the higher level of abstraction of the semantic dataflow graph, our knowledge representation system requires further development. The main challenge is to subsume all data analysis libraries, whose interfaces vary widely, under a universal ontology of data science concepts, while ensuring that the annotation system remains simple enough to be practically useful. More ambitiously still, one might hope to integrate domain-specific ontologies with the data science ontology, perhaps by attaching semantic information to specific data tables and columns in the dataflow graph. Domain-specific

ontologies are proliferating in biology [33, 34] and we expect their popularity to grow across the sciences.

Broadening the scope to a data science platform, how should we utilize the dataflow graph representation to perform the machine learning tasks sketched in Section 2, such as recommending or organizing data analyses? In experiments not reported here, we defined a kernel (similarity metric) between data analyses, drawing on the literature on graph kernels [35, 36]. We then visualized a small number of analyses for the Accelerated Cure Project using kernel PCA. Of course, kernel methods are but one possible approach to an open-ended problem, and we have not seriously attempted to solve, or even formulate precisely, any of the machine learning tasks mentioned above.

Many other worthy problems—in computer science, statistics, human-computer interaction, and social science—present themselves, but we must draw to a close. We see a bright future for the marriage of information technology and science, which, despite the impact of the internet and electronic publishing, is still in its infancy. We hope that the communities of natural and social science, computer science, and statistics will come together to realize this vision.

References

1. L. Bornmann and R. Mutz, "Growth rates of modern science: A bibliometric analysis based on the number of publications and cited references," *Journal of the Association for Information Science and Technology*, vol. 66, no. 11, pp. 2215–2222, 2015.

2. American Association for the Advancement of Science, "The rise of open access," *Science*, vol. 342, no. 6154, pp. 58–59, 2013.

3. F. Perez and B. Granger, "IPython: A system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 21–29, 2007.

4. M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonier, "The Jupyter/IPython architecture: A unified view of computational research, from interactive exploration to communication and publication," in *AGU Fall Meeting Abstracts*, vol. 1, 2014.

5. F. Pedregosa, G. Varoquaux, A. Gramfort, et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

6. P. Bull, I. Slavitt, and G. Lipstein, "Harnessing the power of the crowd to increase capacity for data science in the social sector," in *Proceedings of the ICML Workshop on #Data4Good: Machine Learning in Social Good Applications*, 2016, pp. 31–35.

7. G. Stolovitzky, D. Monroe, and A. Califano, "Dialogue on reverse-engineering assessment and methods," *Annals of the New York Academy of Sciences*, vol. 1115, no. 1, pp. 1–22, 2007.

8. D. Marbach, J. C. Costello, R. Küffner, N. M. Vega, R. J. Prill, D. M. Camacho, K. R. Allison, The DREAM5 Consortium, M. Kellis, J. J. Collins, and G. Stolovitzky, "Wisdom of crowds for

robust gene network inference," *Nature Methods*, vol. 9, no. 8, pp. 796–804, 2012.

9. J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "OpenML: Networked science in machine learning," *ACM SIGKDD Explorations Newsletter*, vol. 15, no. 2, pp. 49– 60, 2014.

10. S. Colton, R. L´opez de Mantaras, and O. Stock, "Computational creativity: Coming of age," *A.I. Magazine*, vol. 30, no. 3, pp. 11–14, 2009.

11. S. Colton and G. A. Wiggins, "Computational creativity: The final frontier?" In *ECAI 2012: 20th European Conference on Artificial Intelligence*, vol. 12, 2012, pp. 21–26.

12. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, "GraphML progress report: Structural layer proposal," in *International Symposium on Graph Drawing*, 2001, pp. 501–512.

13. E. Gansner and S. North, "An open graph visualization system and its applications to software engineering," *Software Practice and Experience*, vol. 30, no. 11, pp. 1203– 1233, 2000.

14. D. Spivak, "The operad of wiring diagrams: Formalizing a graphical language for databases, recursion, and plug-and-play circuits," 2013. arXiv: 1305.0297.

15. D. Rupel and D. Spivak, "The operad of temporal wiring diagrams: Formalizing a graphical language for discrete-time processes," 2013. arXiv: 1307.6894.

16. W. M. Johnston, J. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, 2004.

17. D. Spivak and R. Kent, "Ologs: A categorical framework for knowledge representation," *PLoS One*, vol. 7, no. 1, 2012.

18. D. Spivak, *Category theory for the sciences*. MIT Press, 2014.

19. R. Brachman and H. Levesque, *Knowledge representation and reasoning*. Elsevier, 2004.

20. J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team, "Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biology*, vol. 11, no. 8, p. 1, 2010.

21. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: An extensible system for design and execution of scientific workflows," in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, IEEE, 2004, pp. 423–424.

22. T. Oinn, M. Addis, J. Ferris, et al., "Taverna: A tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

23. Y. Simmhan, B. Plale, and D. Gannon, “A survey of data provenance in e-science,” *ACM Sigmod Record*, vol. 34, no. 3, pp. 31–36, 2005.

24. ——, “A survey of data provenance techniques,” Indiana University, Computer Science Department, Tech. Rep., 2005. [Online]. Available: https://www.cs.indiana.edu/ftp/techreports/TR618.pdf.

25. E. Angelino, D. Yamins, and M. Seltzer, “StarFlow: A script-centric data analysis environment,” in *International Provenance and Annotation Workshop*, 2010, pp. 236– 250.

26. S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo, “VisTrails: Visualization meets data management,” in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ACM, 2006, pp. 745–747.

27. A. V. Aho, M. S. Lam, R. Sethi, and J. D. *Ullman, Compilers: Principles, techniques, and tools*, 2nd ed. Addison-Wesley, 2006.

28. H. Seidl, R. Wilhelm, and S. Hack, *Compiler design: Analysis and transformation*. Springer, 2012.

29. D. Grove, G. DeFouw, J. Dean, C. Chambers, “Call graph construction in object-oriented languages,” in *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA ‘97)*, ACM, 1997, pp. 108–124.

30. A. Lienhard, “Dynamic object flow analysis,” PhD thesis, University of Bern, 2008. 31. A. Lienhard, S. Ducasse, and T. Girba, “Taking an object-centric view on dynamic information with object flow analysis,” *Computer Languages, Systems & Structures*, vol. 35, no. 1, pp. 63–79, 2009.

32. A. Lienhard, T. Girba, and O. Nierstrasz, “Practical object-oriented back-in-time debugging,” in *European Conference on Object-Oriented Programming*, 2008, pp. 592– 615.

33. M. Ashburner, C. A. Ball, J. A. Blake, et al., “Gene Ontology: Tool for the unification of biology,” *Nature Genetics*, vol. 25, no. 1, pp. 25–29, 2000.

34. N. F. Noy, N. H. Shah, P. L. Whetzel, et al., “BioPortal: Ontologies and integrated data resources at the click of a mouse,” *Nucleic Acids Research*, 2009.

35. S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, “Graph kernels,” *Journal of Machine Learning Research*, vol. 11, pp. 1201–1242, 2010.

36. U. Losch, S. Bloehdorn, and A. Rettinger, “Graph kernels for RDF data,” in *Extended Semantic Web Conference*, Springer, 2012, pp. 134–148.

Bios

Evan Patterson *Stanford University, Stanford, CA 94305 (epatters@stanford.edu).* Mr. Patterson is a Ph.D. candidate in the Statistics department at Stanford University. He received the B.Sc. degree in Mathematics and Physics from the California Institute of Technology in 2014. He enjoys exploring the broad contours of the mathematical sciences. His current research concentrates on statistical theory for adaptive data analysis (aka post-selection inference) and knowledge representation for scientific research, particularly data analysis. He has previously worked as a software engineer and is fond of research with interplay between practical software engineering and mathematical theory.

Robert McBurney *Accelerated Cure Project, Waltham, MA 02451 USA (rmcburney@acceleratedcure.org).* Dr. McBurney is the CEO of the Accelerated Cure Project for Multiple Sclerosis and the Co-Principal Investigator for the iConquerMS™ Patient-Powered Research Network. In a career spanning more than 35 years, he has conducted basic and clinical research and managed research groups for drug discovery, personalized medicine, and clinical decision support systems at medical schools, research institutes, biopharmaceutical companies and non-profit organizations in Australia, the U.K. and the USA. Dr. McBurney is currently a member of the American Academy of Neurology, the Society for Neuroscience, the Pharmaceutical Research and Manufacturers of America (PhRMA) Foundation Informatics Advisory Committee and the International Society for Pharmacoeconomics and Outcomes Research. He is also a Trustee Emeritus of the F.W. Olin College of Engineering. He received B.Sc. and Ph.D. degrees from the University of New South Wales, Australia.

Hollie Schmidt *Accelerated Cure Project, Waltham, MA 02451 USA (hollie@acceleratedcure.org).* Ms. Schmidt is the Vice President of Scientific Operations at Accelerated Cure Project for Multiple Sclerosis. Her role includes top-level direction of ACP's MS Repository, which provides highly-characterized serum, plasma, DNA, RNA, and white blood cells from people with demyelinating diseases and controls to scientists worldwide. She has guided the formation of the ACP Clinical Research Network (CRN) and is working with CRN investigators and vendors to design and implement the Optimizing Treatment – Understanding Progression (OPT-UP) study. She co-chairs the Research Committee of iConquerMS™, a virtual patient-powered research network for MS launched by ACP in late 2014. She is currently leading a PCORI (Patient-Centered Outcomes Research Institute) Engagement Award project aimed at increasing racial and ethnic diversity in MS research studies. Before joining Accelerated Cure Project, Ms. Schmidt's endeavors included co-founding the management consulting firm Lifting Mind as well as co-founding two software companies, Midnight Networks and NorthStar Internetworking. Ms. Schmidt has an MS in Management, as well as a BS and MS in Materials Science and Engineering, all from the Massachusetts Institute of Technology.

Ioana Baldini *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (ioana@us.ibm.com).* Dr. Baldini is a research staff member at IBM Research in the Cloud Programming Technology group. She is currently working on serverless computing

infrastructure and data for social good. In the past, she worked on various projects spanning domains such as compute architecture, heterogeneous runtime systems, and system performance analysis. Dr. Baldini holds Masters and PhD degrees in Electrical and Computer Engineering from University of Toronto. She is the recipient of the NSERC (Natural Sciences and Engineering Research Council of Canada) Canada Graduate Scholarship, the Google Canada Anita Borg Scholarship, and the IBM PhD Fellowship.

Aleksandra Mojsilović *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (aleksand@us.ibm.com).* Dr. Mojsilović is an IBM Fellow and scientist at the IBM T. J. Watson Research Center. She received her Ph.D. degree in electrical engineering from the University of Belgrade, Belgrade, Serbia in 1997. She was a Member of Technical Staff at the Bell Laboratories, Murray Hill, New Jersey (1998-2000), and then joined IBM Research, where she currently leads the Data Science group. Dr. Mojsilović is a founder and co-director of the IBM Social Good Fellowship program. Her research interests include multidimensional signal processing, predictive modeling and pattern recognition. She has applied her skills to problems in computer vision, healthcare, multimedia, finance, human resources, public affairs and economics. She is one of the pioneers of business analytics at IBM and in the industry; throughout her career she championed innovative uses of analytics for business decision support. For her technical contributions and the business impact of her work Dr. Mojsilović was appointed an IBM Fellow, the company's highest technical honor. She is the author of over 100 publications and holds 16 patents. Her work has been recognized with several awards including IEEE Signal Processing Society Young Author Best Paper Award, Institute for Operations Research and the Management Sciences (INFORMS) Wagner Prize, IBM Extraordinary Accomplishment Award, IBM Gerstner Prize, and Best Paper awards at the European Conference on Computer Vision (ECCV) and the Conference on Service Operations and Logistics, and Informatics (SOLI). She is an IEEE Fellow and a member of INFORMS and Society of Women Engineers (SWE).

Kush R. Varshney *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (krvarshn@us.ibm.com).* Dr. Varshney is a research staff member and manager in the Data Science Group at the IBM T. J. Watson Research Center in Yorktown Heights, NY. He received his Ph.D. degree in electrical engineering and computer science from the Massachusetts Institute of Technology in 2010. He applies data science and predictive analytics to human capital management, healthcare, olfaction, public affairs, and international development. He conducts academic research on the theory and methods of statistical signal processing and machine learning. His work has been recognized through best paper awards at the 2009 International Conference on Information Fusion; 2013 IEEE Conference on Service Operations and Logistics, and Informatics (SOLI); 2014 ACM SIGKDD (Association for Computing Machinery's Special Interest Group on Knowledge Discovery and Data Mining); and 2015 SIAM (Society for Industrial and Applied Mathematics) Conference on Data Mining (SDM). Dr. Varshney is codirector of the IBM Social Good Fellowship program.

Figures

Figure 1. Example data flow graph: exploratory data analysis for ACP

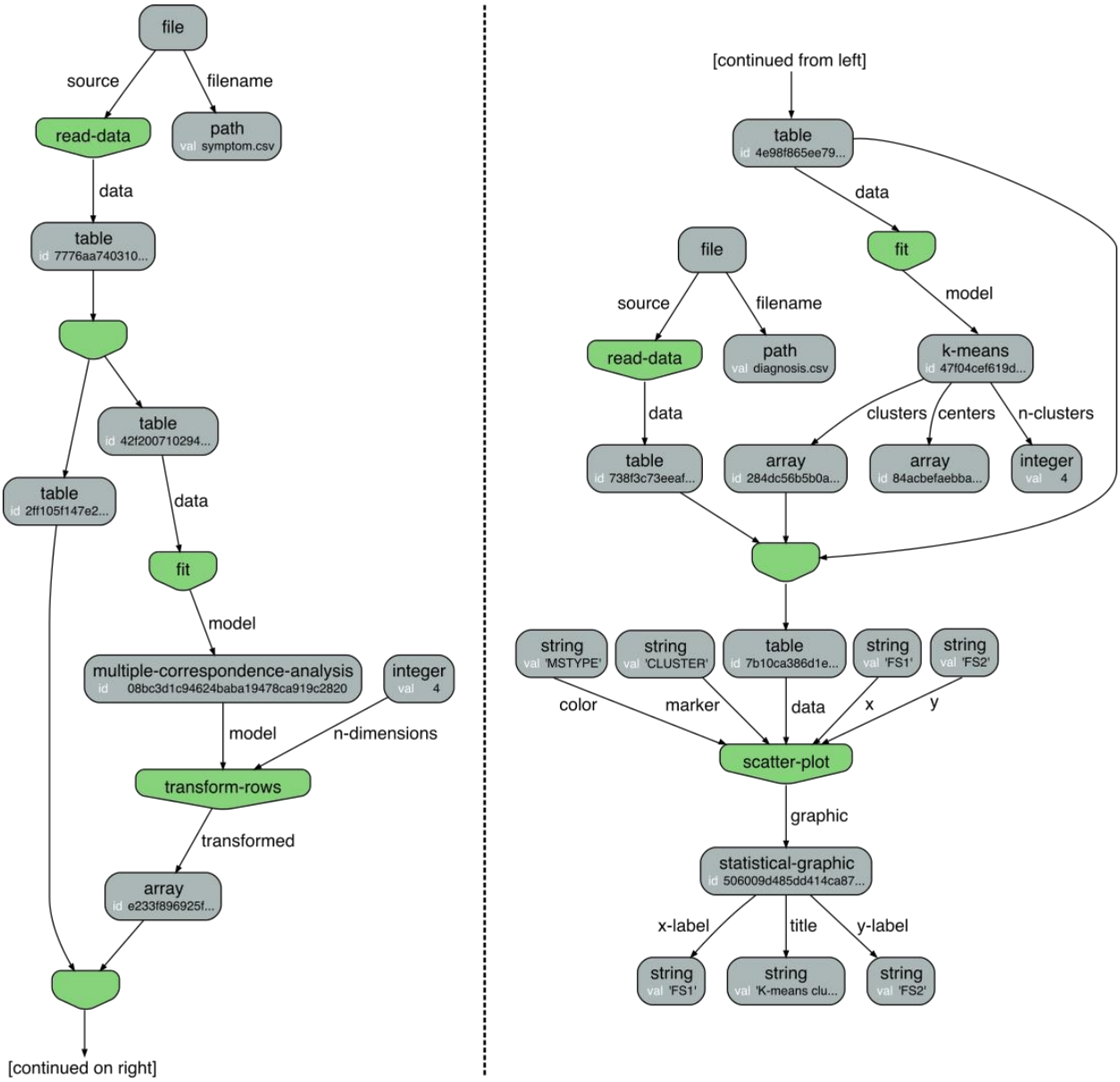


Figure 2. Schematic of dataflow graph pipeline

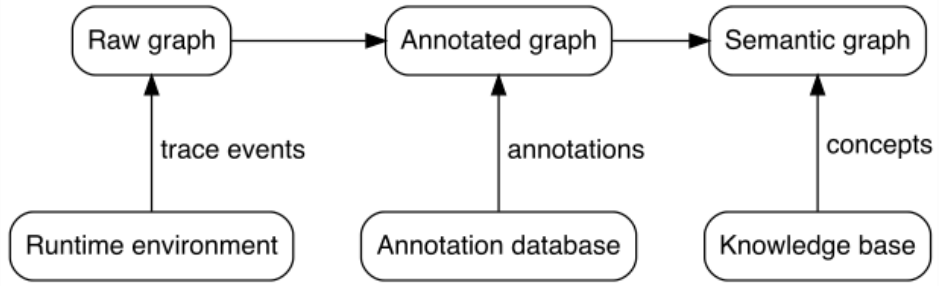


Figure 3. Example raw flow graph: fitting and evaluating a linear model for supervised learning

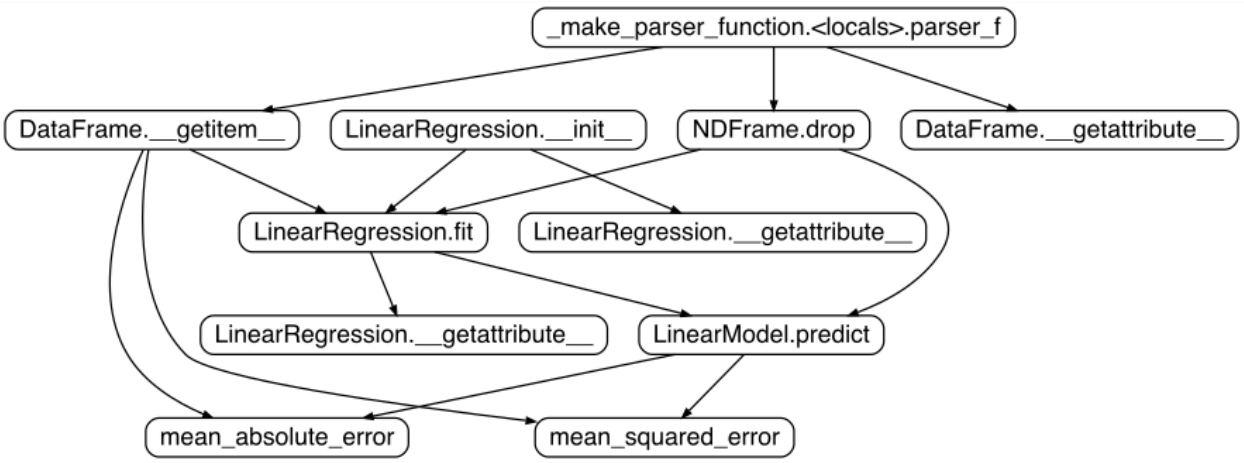


Figure 4. Example semantic flow graph: fitting and evaluating a linear model for supervised learning

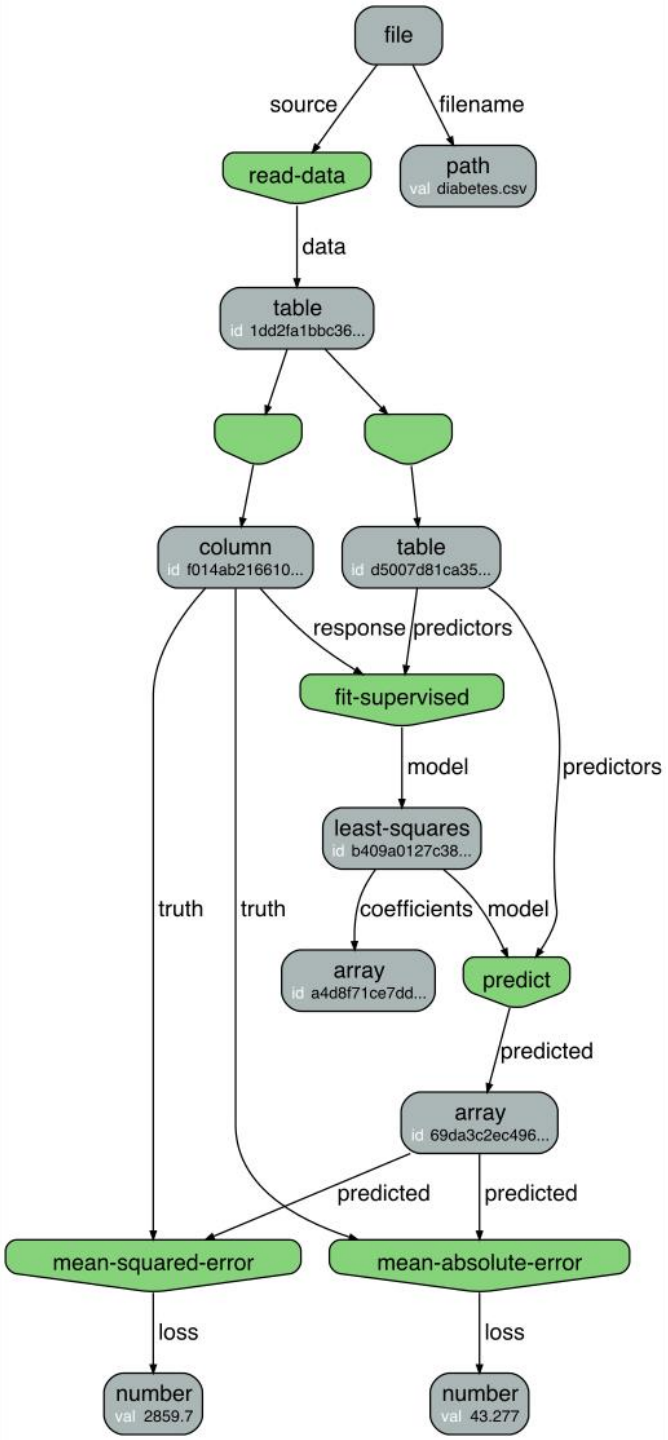
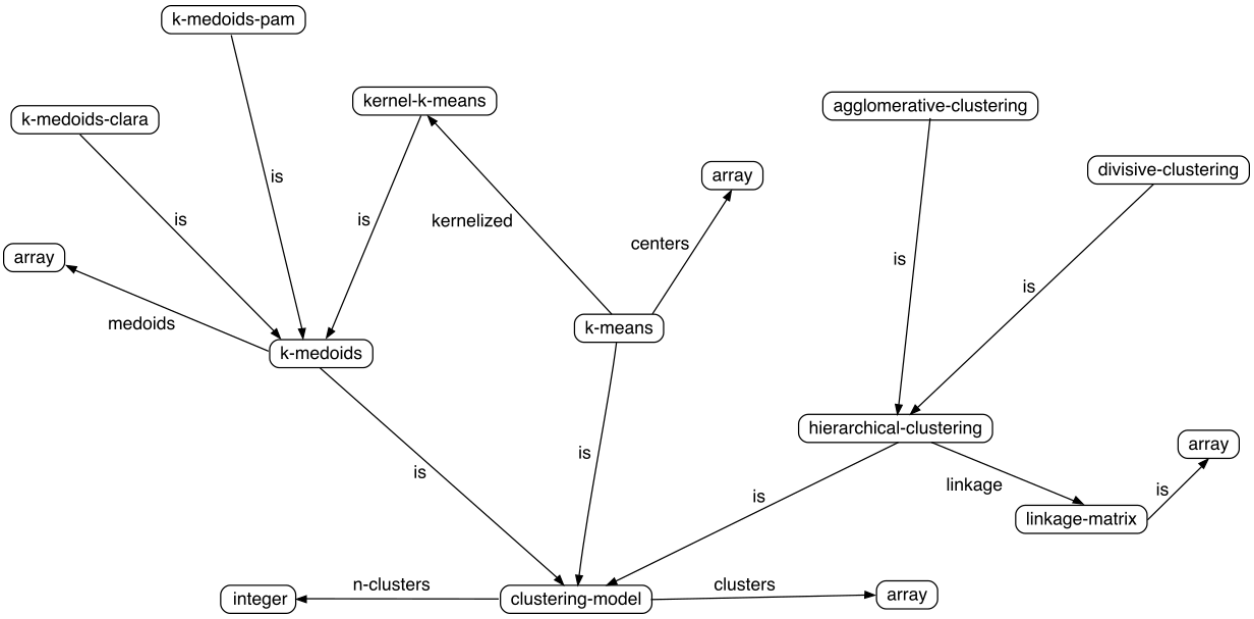


Figure 5. Excerpt of clustering methods from the data analysis olog



Listings

Listing 1. Python code: fitting and evaluating a linear model for supervised learning

```
import pandas as pd
from sklearn import linear_model, metrics

# Load the diabetes data
diabetes = pd.read_csv('diabetes.csv')

# OLS using all predictors
X = diabetes.drop('y', 1)
y = diabetes['y']
lm = linear_model.LinearRegression()
lm.fit(X, y)

# Compute error metrics
y_hat = lm.predict(X)
l1_err = metrics.mean_absolute_error(y, y_hat)
l2_err = metrics.mean_squared_error(y, y_hat)
```

Listing 2. Example annotation: K-means clustering class in Scikit-learn

```
{
  "language": "python",
  "package": "sklearn",
  "id": "k-means",
  "class":
    "sklearn.cluster.k_means_.KMeans",
  "slots": {
    "n-clusters": "get_params.n_clusters",
    "clusters": "labels_",
    "centers": "cluster_centers_"
  },
  "type": "k-means"
}
```

Listing 3. Example annotation: Fit method for regression models in Scikit-learn

```
{
  "language": "python",
  "package": "sklearn",
  "id": "fit-regression",
  "class": [
    "sklearn.base.BaseEstimator",
    "sklearn.base.RegressorMixin"
  ],
  "method": "fit",
  "inputs": {
    "model": "self",
    "predictors": 1,
    "response": 2
  },
  "outputs": {
    "model": "self"
  },
  "type": "fit-supervised"
}
```