Teaching machines to understand data science code by semantic enrichment of dataflow graphs

Evan Patterson
Stanford University & IBM Research AI
epatters@stanford.edu

Ioana Baldini
IBM Research AI
ioana@us.ibm.com

Aleksandra Mojsilović
IBM Research AI
aleksand@us.ibm.com

Kush R. Varshney
IBM Research AI
krvarshn@us.ibm.com

ABSTRACT

Your computer is continuously executing programs, but does it really understand them? Not in any meaningful sense. That burden falls upon human knowledge workers, who are increasingly asked to write and understand code. They would benefit greatly from intelligent tools that reveal the connections between their code and its subject matter. Towards this prospect, we develop an AI system that forms semantic representations of computer programs, using techniques from knowledge representation and program analysis. We focus on code written for data science, although our method is more generally applicable. The semantic representations are created through a novel algorithm for the semantic enrichment of dataflow graphs. This algorithm is undergirded by a new ontology language for modeling computer programs and a new ontology about data science, written in this language.

ACM Reference Format:
Evan Patterson, Ioana Baldini, Aleksandra Mojsilović, and Kush R. Varshney. 2018. Teaching machines to understand data science code by semantic enrichment of dataflow graphs. In *Proceedings of 2018 KDD Workshop on the Fragile Earth (FEED '18).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Your computer—through which you are, in all likelihood, reading this paper—is continuously, efficiently, and reliably executing computer programs, but does it really understand them? Artificial intelligence researchers have taken great strides towards teaching machines to understand images, speech, natural text, and other media. The same cannot be said of computer code, which has been relatively neglected as a target of machine intelligence in the last two decades. Yet the growth of computing's influence on society shows no signs of abating, with knowledge workers in all domains increasingly required to create, maintain, and extend computer programs. For all workers, but especially those outside software engineering roles, programming is a means to achieve real-world goals, not an end in itself. Programmers would benefit greatly from intelligent tools that reveal the connections between their code, their colleagues' code, and the subject-matter concepts to which the code implicitly refers and to which their real enthusiasm belongs. By teaching machines to comprehend code, we could create artificial agents that empower human knowledge workers or perhaps even generate useful programs of their own.

One computational domain undergoing particularly rapid growth is data science. Besides the usual problems facing the scientist-turned-programmer, the data scientist must contend with a proliferation of programming languages (like Python, R, and Julia) and frameworks (too numerous to recount). Data science therefore presents an especially compelling target for machine understanding of computer code. An AI agent that simultaneously comprehends the generic concepts of computing and the specialized concepts of data science could prove enormously useful, for example by automatically visualizing machine learning workflows or summarizing data analyses as natural text for human readers.

Towards this vision, we propose and implement an AI system that forms semantic representations of computer programs in a particular subject-matter domain. Our system is fully automated, inasmuch as it expects no input from the programmer besides the program itself. We will focus on applications to data science because we, the authors, are all data scientists of various stripes. Nevertheless, we think that our methodology could be fruitfully applied to other scientific domains with a heavy computational focus, such as bioinformatics or climate science.

We contribute several components that cohere as an AI system but also hold independent interest. First, we define a dataflow graph representation of a computer program, called the *raw flow graph*. We extract raw flow graphs from computer programs using static and dynamic program analysis. We define another program representation, called the *semantic flow graph*, combining dataflow information with domain-specific information about data science. This representation is supported by a new ontology language for modeling computer programs and a new ontology about data science, written in this language. Finally, we introduce a *semantic enrichment* algorithm for transforming the raw flow graph into the semantic flow graph.

The paper is organized as follows. In Section 2, we motivate our method through a pedagogical example. In Section 3, we explain the method itself, in a largely informal way. (An online supplement adds precision and rigor to this discussion.) In Section 4, we bring out connections to existing work in knowledge representation, program analysis, programming language theory, and category theory. Finally, in Section 5, we locate our work within the ongoing movement towards open, reproducible, and collaborative data science.

FEED '18, August 19, 2018, London

```python
import numpy as np
from scipy.cluster.vq import kmeans2

iris = np.genfromtxt('iris.csv',
    dtype='f8',
    delimiter=',',
    skip_header=1,
)
iris = np.delete(iris, 4, axis=1)

centroids, clusters = kmeans2(iris, 3)
```

Listing 1: k-means clustering in Python via NumPy and SciPy

```python
import pandas as pd
from sklearn.cluster import KMeans

iris = pd.read_csv('iris.csv')
iris = iris.drop('Species', 1)

kmeans = KMeans(n_clusters=3)
kmeans.fit(iris.values)
centroids = kmeans.cluster_centers_
clusters = kmeans.labels_
```

Listing 2: k-means clustering in Python via Pandas and Scikit-learn

We also demonstrate our method on a real-world data analysis drawn from a biomedical data science challenge.

2 AN EXAMPLE

We begin with a small, pedagogical example, to be revisited and elaborated later. Three versions of a toy data analysis are shown in Listings 1, 2 and 3. The first is written in Python using the scientific computing packages NumPy and SciPy; the second in Python using the data science packages Pandas and Scikit-learn; and the third in R using the R standard library. The three programs perform the same computation: they read the Iris dataset from a CSV file, drop the last column (labeling the flower species), fit a k-means clustering model with three clusters to the remaining columns, and return the cluster assignments and centroids. The programs are syntactically distinct but semantically equivalent.

Identifying this semantic equivalence, our system furnishes the same semantic flow graph for all three programs, shown in Figure 1. The labeled nodes and edges refer to concepts in the ontology. The node tagged with a question mark refers to code with unknown semantics.

3 IDEAS AND TECHNIQUES

We now explain our method of constructing semantic representations of computer programs. It is summarized diagrammatically in Figure 2.

The system outputs two program representations, the raw and semantic flow graphs. Both dataflow graphs capture the execution

```r
iris = read.csv('iris.csv', stringsAsFactors=FALSE)
iris = iris[, names(iris) != 'Species']

km = kmeans(iris, 3)
centroids = km$centers
clusters = km$cluster
```

Listing 3: k-means clustering in R

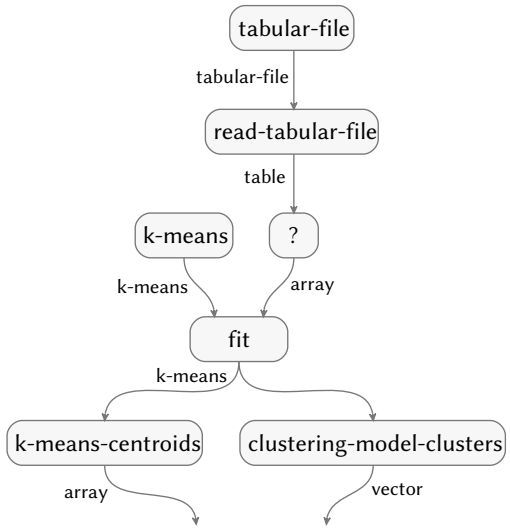


Figure 1: Semantic flow graph for three versions of k-means clustering analysis (Listings 1, 2 and 3)

of a computer program doing data analysis, but at different levels of abstraction. The *raw flow graph* records the concrete function calls made by the program. This graph is language and library dependent. The *semantic flow graph* describes the same program in terms of abstract concepts belonging to a formal ontology about data science. This graph is language and library independent.

Our method has two major steps, which connect the computer program and its representations (Figure 2). In the first step, *computer program analysis* distills the raw flow graph from the program. A process of *semantic enrichment* then transforms the raw flow graph into the semantic flow graph.

Semantic enrichment requires a few supporting actors. An *ontology* (or *knowledge base*), called the Data Science Ontology, underlies the semantic content. It contains two types of knowledge: concepts and annotations. *Concepts* formalize the abstract ideas of machine learning, statistics, and computing on data. The semantic flow graph has semantics, as its name suggests, because its nodes and edges are linked to concepts. *Annotations* map code from data science libraries, such as Pandas and Scikit-learn, onto concepts. During semantic enrichment, annotations say how to translate concrete functions in the raw flow graph into abstract functions in the semantic flow graph.

With the outline of our method now in view, we develop its elements in greater detail. We do so as fully as space permits but

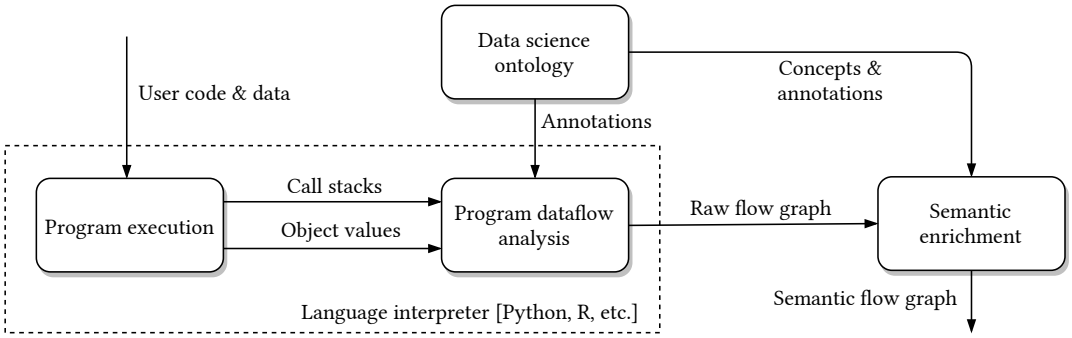


Figure 2: System architecture

with minimal mathematical formalism. An extended, online version of this paper adds technical detail for the interested reader.

3.1 The Monocl ontology language

The Data Science Ontology is written in a new *ontology language* called the MONoidal Ontology and Computing Language (Monocl). We find it helpful to think of Monocl as a minimalistic, typed, functional programming language. The analogy usually suggests the right intuitions. It is, however, imperfect because the ontology language differs from any real-world programming language, being designed for knowledge representation rather than actual computing.

Monocl is written in a point-free textual syntax or equivalently in a graphical syntax of interconnected boxes and wires. The two syntaxes are parallel though not quite isomorphic. For simplicity, we present only the graphical syntax.

Every expression in the ontology language is either a type or a function. This terminology agrees with that of functional programming. Thus, a *type* represents a kind or species of thing in a subject-matter domain, here data science. A *function* is a functional relation or mapping from an input type (the *domain*) to an output type (the *codomain*). The main role of ontology language is to say how new types and functions can be constructed from existing types and functions. Let us see how using the graphical syntax.

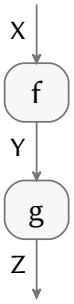
Types are represented graphically by wires. A basic type X is drawn as a single wire labeled X. The *product* of two types X and Y is another type $X \times Y$. It has the usual meaning: an element of type $X \times Y$ is an element of type X *and* an element of type Y, in that order. Products of three or more types are defined similarly. Diagrammatically, a product of n types is a bundle of n wires in parallel. Product types are similar to record types in real-world programming languages, such as `struct` types in C. There is also a *unit type* 1 inhabited by a single element. It is analogous to the `void` type in C and Java, the `NoneType` type in Python (whose sole inhabitant is `None`), and the `NULL` type in R. Diagrammatically, the unit type is an *empty* bundle of wires (a blank space).

Functions are represented graphically by *wiring diagrams* (also known as *string diagrams*). A basic function f with domain X and codomain Y, written $f : X \to Y$, is drawn as a box labeled f. The top of the box has input ports with incoming wires X and the bottom has output ports with outgoing wires Y. A wiring diagram defines a function by connecting boxes with wires according to certain rules. The diagram has an outer box with input ports, defining the function's domain, and output ports, defining the codomain. Figures 1, 4 and 5 are all examples of wiring diagrams.

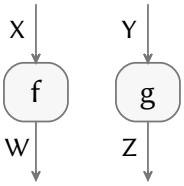
The rules for connecting boxes within a wiring diagram correspond to ways of creating new functions from old. The most fundamental ways are composing functions and taking products of functions. The *composition* of a function $f : X \to Y$ with $g : Y' \to Z$ is a new function $f \cdot g : X \to Z$, with the usual meaning. Algorithmically speaking, $f \cdot g$ computes *in sequence*: first f and then g. In order to have a valid composition, the intermediate types Y and Y' must be compatible, in a sense to be explained later. The *product* of functions $f : X \to W$ and $g : Y \to Z$ is another function $f \times g : X \times Y \to W \times Z$. Algorithmically, $f \times g$ computes f and g *in parallel*, taking the inputs, and returning the outputs, of both f and g. Figure 3 shows the graphical syntax for composition and products. Here we pass over several other constructions on functions, such as identities, duplication, and deletion.

A type can be declared a *subtype* of one or more other types. To a first approximation, subtyping establishes an "is-a" relationship between types. In the Data Science Ontology, matrices are a subtype of both arrays (being arrays of rank 2) and data tables (being tables whose columns all have the same data type). As this example illustrates, subtyping in Monocl is not like inheritance in a typical object-oriented programming language. Instead, subtyping should be understood through *implicit conversion*, also known as *coercion* [Pierce 1991; Reynolds 1980]. The idea is that if a type X is a subtype of X', then there is a canonical way to convert elements of type X into elements of type X'. Elaborating our example, a matrix simply *is* an array (of rank 2), hence can be trivially converted into an array. A matrix is *not* strictly speaking a data table but can be converted into one (of homogeneous data type) by assigning numerical names to the columns. Notice that there is no set-theoretic containment between matrices and data tables, hence the slogan that "types are not sets" [Morris 1973].

Besides serving as the "is-a" relation ubiquitous in knowledge representation systems, subtypes enable ad hoc polymorphism. To compose a function $f : X \to Y$ with $g : Y' \to Z$, we require only that Y be a subtype of Y'. Operationally, to compute $f \cdot g$, we first compute f, then coerce the result from type Y to Y', and finally compute g. Diagrammatically, a wire has valid types if and only if



(a) Composition



(b) Product

Figure 3: Graphical syntax for two fundamental operations on functions

	Concept	Annotation
Type	data table	pandas data frame
	statistical model	scikit-learn estimator
Function	reading a tabular data file	`read_csv` function in pandas
	fitting a statistical model to data	`fit` method of scikit-learn estimators

Table 1: Examples from the Data Science Ontology

the source port's type is a subtype of the target port's type. Thus implicit conversions really are implicit in the graphical syntax.

Monocl also supports "is-a" relations between functions, which we call *subfunctions* in analogy to subtypes. In the Data Science Ontology, reading a table from a tabular file (call it f) is a subfunction of reading data from a generic data source (call it f'). That sounds intuitively plausible but what does it mean? The domain of f, a tabular file, is a subtype of the domain of f', a generic data source. The codomain of f, a table, is a subtype of the codomain of f', generic data. Now consider two possible computational paths that take a tabular file and return generic data. We could apply f, then coerce the resulting table to generic data. Alternatively, we could coerce the tabular file to a generic data source, then apply f'. The subfunction relation asserts that these two computations are equivalent.

3.2 The Data Science Ontology

We have started to write, in the Monocl ontology language, an ontology about statistics, machine learning, and data processing. We call it the Data Science Ontology. It aims to support automated reasoning about data science software. As we have said, it consists of concepts and annotations, each of which is either a type or a function. This leads to a two-way classification of the ontology's contents. The four combinations are listed in Table 1 along with examples of each.

Concepts formalize the abstract ideas of data science. They constitute the *basic types* and *basic functions* from which more complex types and functions are constructed, using the ontology language. According to our methodology, data analyses are modeled as functions written in the ontology language and composed of the ontology's concepts.

As a further modeling assumption, we suppose that software packages for data science, such as Pandas and Scikit-learn, instantiate concepts. *Annotations* say how this instantiation occurs by mapping the types and functions in software packages onto the types and functions of the ontology. To avoid confusion between levels of abstraction, we call the former "concrete" and the latter "abstract." So a type annotation maps a concrete type—a primitive type or user-defined class in a language like Python or R—onto an abstract type—a type concept. Likewise, a function annotation maps a concrete function onto an abstract function. We construe "concrete function" in the broadest possible sense to include any programming language construct that "does something": ordinary functions, methods of classes, attribute getters and setters, etc. A great deal of modeling flexibility is needed to accurately translate the diverse APIs of statistical software into a single set of universal concepts. To achieve this flexibility, an annotation's abstract definition may be an arbitrary Monocl program—any function expressible in the ontology language using the ontology's concepts.

3.3 Raw and semantic dataflow graphs

With this preparation, we can attain a more exact understanding of the raw and semantic flow graphs. The two dataflow graphs have in common that they are wiring diagrams representing a data analysis. However, they live at different levels of abstraction.

The *raw flow graph* describes the computer implementation of a data analysis. Its boxes are concrete functions or, more precisely, the function calls observed during execution of the program. Its wires are concrete types together with their observed elements. These "elements" are either literal values or object references, depending on the type. To illustrate, Figures 4, 5 and 6 show the raw flow graphs for Listings 1, 2 and 3, respectively. (The wire elements are not shown.)

The raw flow graph is extracted from a data analysis by computer program analysis [Nielson et al. 1999]. Unlike the usual application to optimizing compilers [Aho et al. 2006], our program analysis is dynamic, not static. It records interprocedural data flow during program execution. Since our original publication [Patterson et al. 2017], the mathematical formalism behind our program analysis has evolved but the algorithms and implementation have not. We defer to that paper for engineering aspects of the raw flow graph. The merits and limitations raised there remain in force.

The *semantic flow graph* describes a data analysis in terms of universal concepts, independent of the particular programming language and libraries used to implement the analysis. Its boxes are function concepts. Its wires are type concepts together with their observed elements. The semantic flow graph is thus an abstract function, composed of the ontology's concepts and written in the graphical syntax, but augmented with computed values.

3.4 Semantic enrichment

The *semantic enrichment* algorithm transforms the raw flow graph into the semantic flow graph. It proceeds in two independent stages,

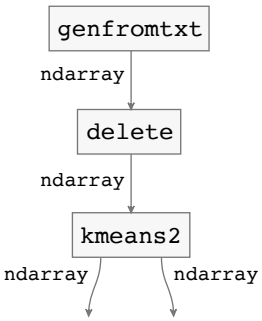


Figure 4: Raw flow graph for k-means clustering in Python via NumPy and SciPy (Listing 1)

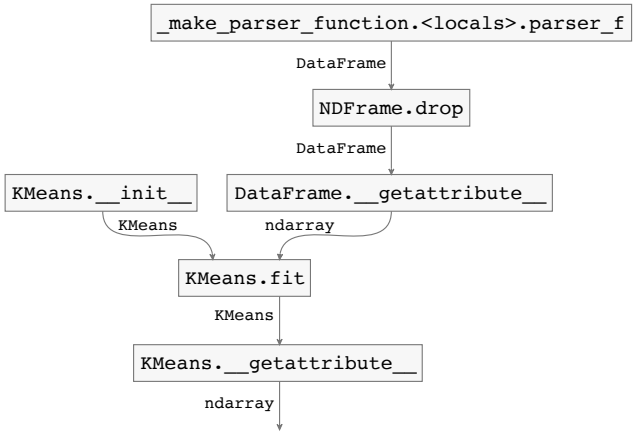


Figure 5: Raw flow graph for k-means clustering in Python via Pandas and Scikit-learn (Listing 2)

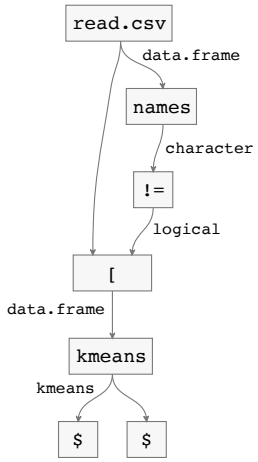


Figure 6: Raw flow graph for k-means clustering in R (Listing 3)

one of expansion and one of contraction. The expansion stage makes essential use of the ontology's annotations.

Expansion. In the *expansion* stage, the annotated parts of the raw flow graph are replaced by their abstract definitions. Each annotated box—that is, each box referring to a concrete function annotated by the ontology—is replaced by the corresponding abstract function. Likewise, the concrete type of each annotated wire is replaced by the corresponding abstract type. This stage of the algorithm is "expansionary" because, as we have seen, a function annotation's definition may be an arbitrary Monocl program. In other words, a single box in the raw flow graph may expand to an arbitrarily large subdiagram in the semantic flow graph.

The expansion procedure is *functorial*, to use the jargon of category theory. Informally, this means two things. First, notice that concrete types are effectively annotated twice, explicitly by type annotations and implicitly by the domain and codomain types of function annotations. Functorality requires that these abstract types be compatible, ensuring the logical consistency of type and function annotations. Second, expansion preserves the structure of the ontology language, including composition and products. Put differently, the expansion of a wiring diagram is completely determined by its action on individual boxes (basic functions). Functorality is a modeling decision that greatly simplifies the semantic enrichment algorithm, at the expense of imposing restrictions on how the raw flow graph may be transformed.

Contraction. It is practically infeasible to annotate every reusable unit of data science source code. Most real-world data analyses use concrete types and functions that are not annotated. This unannotated code has unknown semantics, so it does not belong in the semantic flow graph. On the other hand, it usually cannot be deleted from the wiring diagram without altering the connectivity of the rest of the diagram. Semantic enrichment must not corrupt the dataflow record.

As a compromise, in the *contraction* stage, the unannotated parts of the raw flow graph are simplified to the extent possible. All references to unannotated types and functions are removed, leaving behind unlabeled wires and boxes. Semantically, the unlabeled wires are interpreted as arbitrary "unknown" types and the unlabeled boxes as arbitrary "unknown" functions (which could have known domain and codomain types). The diagram is then simplified by *encapsulating* unlabeled boxes. Specifically, every maximal connected subdiagram of unlabeled boxes is encapsulated by a single unlabeled box. The interpretation is that any composition of unknown functions is just another unknown function. This stage is "contractionary" because it can only decrease the number of boxes in the diagram.

Example revisited. To reprise our original example, semantic enrichment transforms the raw flow graphs of Figures 4, 5 and 6 into the same semantic flow graph, shown in Figure 1. Let us take a closer look at a few of the expansions and contractions involved in the two Python programs.

In the first program (Listing 1), the annotated `kmeans2` function in SciPy expands to an abstract program that creates a k-means clustering model, fits it to the data, and extracts its clusters and centroids. The abstract k-means clustering model does *not* correspond

to any concrete object in the original program. We routinely use this modeling pattern to cope with functions that are not object-oriented with respect to models. By contrast, in the second program (Listing 2), the abstract k-means model corresponds to an instance of the `KMeans` class in Scikit-learn.

Now consider the contractions. In the first program, the only unannotated box is the NumPy `delete` function. Contracting this box does not reduce the size of the wiring diagram. Only the second program is subjected to a contraction involving multiple boxes. The subdiagram consisting of the pandas `NDFrame.drop` method composed with the `values` attribute access (via the special `DataFrame.__getattribute__` method) is encapsulated by a single unlabeled box.

4 RELATED WORK

This paper extends our previous work [Patterson et al. 2017] in several directions. We designed a new ontology language along with a new ontology written in this language. We replaced our original, ad hoc procedure for creating the semantic flow graph with the more flexible and principled semantic enrichment algorithm.

In this project, we have been inspired by a constellation of ideas at the intersection of knowledge representation, program analysis, programming language theory, and category theory. We now position our work in relation to these areas.

Knowledge representation and program analysis. The history of artificial intelligence is replete with interactions between knowledge representation and computer program analysis. In the late '80s and early '90s, automated planning and ruled-based expert systems featured in "knowledge-based program analysis" [Biggerstaff et al. 1994; Harandi and Ning 1990; Johnson and Soloway 1985]. Other early systems were based on description logic [Devanbu et al. 1991; Welty 2007] and graph parsing [Wills 1992]. Such projects are supposed to help software developers maintain large codebases (exceeding, say, a million lines of code) in specialized industrial domains like telecommunications.

Our research goals are less ambitious in scale but also, we hope, more tractable. We focus on knowledge workers who write short, semantically rich scripts, without the endless layers of abstraction found in large codebases. In data science, the code tends to be much shorter, the control flow more linear, and the underlying concepts better defined, than in large-scale industrial software. Our methodology is accordingly quite different from that of the older literature.

Ontologies for data science. There already exist several ontologies and schemas related to data science, such as STATO, an OWL ontology about basic statistics [Gonzalez-Beltran and Rocca-Serra 2016]; the Predictive Modeling Markup Language (PMML), an XML schema for data mining models [Guazzelli et al. 2009]; and ML Schema, a schema for data mining and machine learning workflows under development by a W3C community group [Lawrynowicz et al. 2017]. We have created a new ontology because these standards are not suitable for precisely describing data science code. The Data Science Ontology is designed for exactly this purpose.

Ontology languages and programming languages. We have also designed a new ontology language, Monocl, for modeling computer programs. Although it is the medium of the Data Science Ontology, the ontology language is conceptually independent of data science or any other computational domain.

The mathematical foundation of the ontology language is category theory, as hinted in Section 3 and developed in the extended version of this paper. We hope that our project will advance an emerging paradigm of knowledge representation based on category theory [Patterson 2017; Spivak and Kent 2012]. We find category theory appealing for several reasons.

First, category theory serves as a bridge to programming language theory, whose relevance to modeling computer programs is obvious. Due to the close connection between category theory and type theory [Crole 1993; Jacobs 1999]—most famously, the correspondence between cartesian closed categories and simply typed lambda theories [Lambek and Scott 1988]—we may occupy the syntactically and semantically flexible world of algebra but still utilize the highly developed theory of programming languages. We borrow from programming language theory notions of subtyping and ad hoc polymorphism [Goguen 1978; Reynolds 1980].

Besides its connection to programming language theory, category theory is useful in its own right. We interpret the semantic enrichment algorithm as a functor. Also, the graphical syntax of string diagrams offers an intuitive yet rigorous alternative to the typed lambda calculus's conventional textual syntax [Selinger 2013], which beginners may find impenetrable. The family of graphical languages based on string diagrams is a jewel of category theory [Baez and Stay 2010; Selinger 2010], with applications to such diverse fields as quantum mechanics [Coecke and Paquette 2010], control theory [Baez and Erbele 2015], and natural language semantics [Coecke et al. 2013].

5 DISCUSSION

Like the code it analyzes, our AI system is not an end in itself but a means to achieve productive activity in our world. Teaching machines to comprehend code opens exciting possibilities, which are only just beginning to be explored.

The field of data science is ripe for transformation by artificial intelligence. Subcommunities are fragmented along lines of methodology, programming languages, and frameworks, hindering collaboration and knowledge sharing. A unified semantic representation of data analysis could help break down these barriers. The opportunity becomes only more apparent as the pressure builds for scientists to publish their code and data. Future science looks to be more open and collaborative than ever before [Nielsen 2012], due to a confluence of forces, from the birth of collaboration platforms like GitHub and Kaggle to the growing demand for reproducibility in science [Munafò et al. 2017]. This will be fertile ground for machine-assisted discovery and knowledge sharing, at both small and large scales.

At the scale of individuals, we imagine a data science IDE that interacts with the analyst at both syntactic and semantic levels. Suppose the analyst creates a logistic regression model using the `glm` function in R. By a simple inference within the ontology, the environment recognizes logistic regression as a classification model. It suggests a more flexible classification model to the analyst, say the logistic generalized additive model. It then generates code to

invoke the `gam` function from the R package `gam`. In this way, the environment encourages the analyst to discover both new code and new concepts.

As programmers, we are all prone to occasional lapses of discipline in commenting our code. To supplement the explanations written by fallible humans, an AI agent might translate our semantic representations of data analyses into written descriptions, via natural language generation [Gatt and Krahmer 2017]. The playful R package `explainr` does exactly that for a few R functions, in isolation [Parker et al. 2015]. A more comprehensive system based on our work would span multiple languages and libraries and would document both the individual steps and the high-level design of a data analysis.

New possibilities for AI emerge at the scale of online platforms for collaborative data science. Online platforms like Kaggle, Driven Data, and DREAM Challenges host thousands of data analyses, written in Python, R, and other languages. Whenever there are many data analyses at play, competing or complementary, we could try to automate certain kinds of statistical meta-analysis or model checking, usually conducted laboriously by hand. Or, taking the viewpoint of machine learning, our system might feed meta-learning algorithms—learning algorithms that take other models as input data. All such "meta-level" data science would benefit from a unified semantic representation.

For concreteness, let us see how our method behaves on a real-world data analysis, drawn from a DREAM Challenge. DREAM Challenges address questions in systems biology and translational medicine [Saez-Rodriguez et al. 2016]. The challenge we consider asks how well clinical and genetic covariates predict patient response to anti-TNF treatment for rheumatoid arthritis [Sieberts et al. 2016]. (The authors find that the genetic covariates do not meaningfully increase the predictive power beyond what is already contained in the clinical covariates.) We analyze two models submitted by a top-ranking contestant [Kramer 2014]. The code, written in R, has been lightly modified for portability.

The semantic flow graph for the two models is shown in Figure 7. We interpret the highlights of the analysis. Per the challenge requirements, the analyst builds two predictive models, one including only clinical covariates and the other including both genetic and clinical covariates. Both models use the Cubist regression algorithm [Kuhn and Johnson 2013, §8.7], a variant of random forests based on M5 regression model trees [Wang and Witten 1997]. Because the genetic data is high-dimensional, the second model is constructed using a subset of the genetic covariates, determined by a variable selection algorithm called VIF regression [Lin et al. 2011]. The linear regression model created by VIF regression is used only for variable selection, not for prediction.

Most of the unlabeled nodes in Figure 7, including the large node at the top, refer to code for data preprocessing or transformation. There is no fundamental obstacle to representing the semantics of such code; it so happens that the "data munging" portion of the Data Science Ontology has not yet been developed. However, this situation illustrates another, more important point. Our system does not need or expect the ontology to contain complete information about the program's types and functions. It is designed to degrade gracefully, producing useful results even in the face of missing information.

In future work, we plan to build on the suggestive examples presented in this paper. We will develop methods for automated meta-analysis based on semantic flow graphs and conduct a systematic empirical evaluation on a corpus of data analyses. We also have ambitions to more faithfully represent the mathematical and statistical structure of models. Our representation is currently focused on the computational aspects, but the most interesting applications require knowledge of both.

Only by a concerted community effort will the vision of machine-assisted data science be realized. To that end, we will release as open source software our Python and R program analysis tools, written in their respective languages, and our semantic enrichment algorithm, written in Julia. We will also crowdsource the further development of the Data Science Ontology. We entreat the reader to join us in the effort of bringing artificial intelligence to the practice of data science.

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2 ed.). Addison-Wesley.

John Baez and Jason Erbele. 2015. Categories in control. *Theory and Applications of Categories* 30, 24 (2015), 836–881.

John Baez and Mike Stay. 2010. Physics, topology, logic and computation: a Rosetta Stone. In *New Structures for Physics*. Springer, 95–172.

Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1994. Program understanding and the concept assignment problem. *Commun. ACM* 37, 5 (1994), 72–82.

Bob Coecke, Edward Grefenstette, and Mehrnoosh Sadrzadeh. 2013. Lambek vs. Lambek: Functorial vector space semantics and string diagrams for Lambek calculus. *Annals of Pure and Applied Logic* 164, 11 (2013), 1079–1100.

Bob Coecke and Eric Oliver Paquette. 2010. Categories for the practising physicist. In *New Structures for Physics*. Springer, 173–286.

Roy L. Crole. 1993. *Categories for Types*. Cambridge University Press.

Prem Devanbu, Ron Brachman, Peter G. Selfridge, and Bruce W. Ballard. 1991. LaSSIE: A knowledge-based software information system. *Commun. ACM* 34, 5 (1991), 34–49.

Albert Gatt and Emiel Krahmer. 2017. Survey of the state of the art in natural language generation: Core tasks, applications and evaluation. arXiv:1703.09902.

Joseph Goguen. 1978. *Ordered sorted algebra*. Technical Report 14, Semantics and Theory of Computation Series. UCLA, Computer Science Department.

Alejandra Gonzalez-Beltran and Philippe Rocca-Serra. 2016. Statistics Ontology (STATO). http://stato-ontology.org/ [Online].

Alex Guazzelli, Michael Zeller, Wen-Ching Lin, and Graham Williams. 2009. PMML: An open standard for sharing models. *The R Journal* 1, 1 (2009), 60–65.

Mehdi T. Harandi and Jim Q. Ning. 1990. Knowledge-based program analysis. *IEEE Software* 7, 1 (1990), 74–81.

Bart Jacobs. 1999. *Categorical logic and type theory*. Studies in Logic and the Foundations of Mathematics, Vol. 141. Elsevier.

W. Lewis Johnson and Elliot Soloway. 1985. PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering* 3 (1985), 267–275.

Eric Kramer. 2014. Rheumatoid arthritis final predictions. https://doi.org/10.7303/syn2491171

Max Kuhn and Kjell Johnson. 2013. *Applied Predictive Modeling*. Springer.

Joachim Lambek and Philip J. Scott. 1988. *Introduction to higher-order categorical logic*. Cambridge University Press.

Agnieszka Lawrynowicz, Joaquin Vanschoren, Diego Esteves, Panče Panov, et al. 2017. Machine Learning Schema (ML Schema). https://www.w3.org/community/ml-schema/ [Online].

Dongyu Lin, Dean P. Foster, and Lyle H. Ungar. 2011. VIF regression: a fast regression algorithm for large data. *J. Amer. Statist. Assoc.* 106, 493 (2011), 232–247.

James H. Morris. 1973. Types are not sets. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 120–124.

Marcus R. Munafò, Brian A. Nosek, Dorothy V. M. Bishop, Katherine S. Button, Christopher D. Chambers, Nathalie Percie du Sert, Uri Simonsohn, Eric-Jan Wagenmakers, Jennifer J. Ware, and John P. A. Ioannidis. 2017. A manifesto for reproducible science. *Nature Human Behaviour* 1, 0021 (2017).

Michael Nielsen. 2012. *Reinventing Discovery: The New Era of Networked Science*. Princeton University Press.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag.

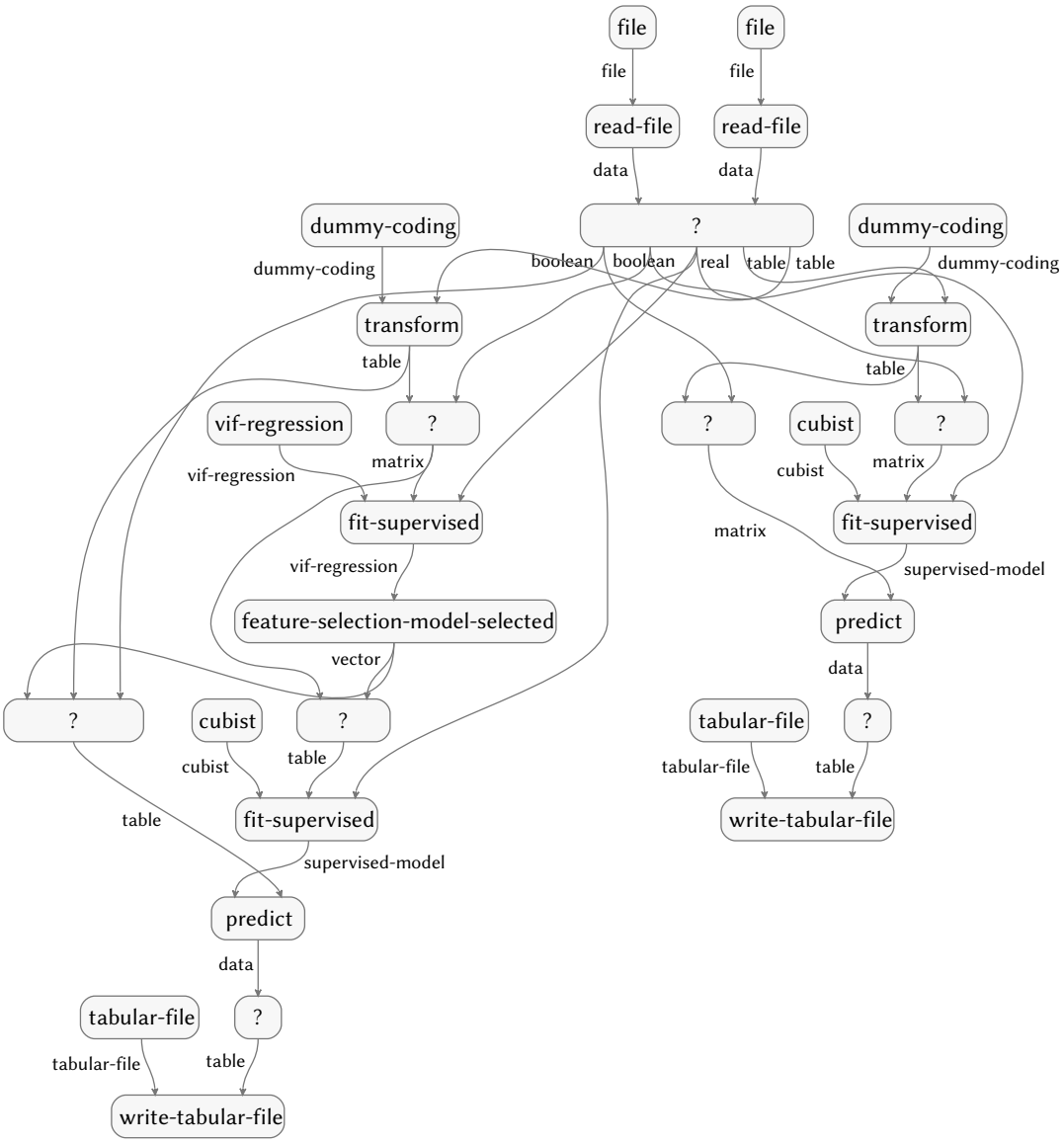


Figure 7: Semantic flow graph for two models from the Rheumatoid Arthritis DREAM Challenge

Hilary Parker, David Robinson, Stephanie Hicks, and Roger Peng. 2015. explainr package. GitHub repository.

Evan Patterson. 2017. Knowledge Representation in Bicategories of Relations. arXiv:1706.00526.

Evan Patterson, Robert McBurney, Holly Schmidt, Ioana Baldini, Aleksandra Mojsilović, and Kush R. Varshney. 2017. Dataflow representation of data analyses: Towards a platform for collaborative data science. *IBM Journal of Research and Development* 61, 6 (2017), 9:1–9:13.

Benjamin C. Pierce. 1991. *Basic Category Theory for Computer Scientists.* MIT Press.

John C. Reynolds. 1980. Using category theory to design implicit conversions and generic operators. In *International Workshop on Semantics-Directed Compiler Generation.* 211–258.

Julio Saez-Rodriguez, James C Costello, Stephen H Friend, Michael R Kellen, Lara Mangravite, Pablo Meyer, Thea Norman, and Gustavo Stolovitzky. 2016. Crowdsourcing biomedical research: leveraging communities as innovation engines. *Nature Reviews Genetics* 17, 8 (2016), 470–486.

Peter Selinger. 2010. A survey of graphical languages for monoidal categories. In *New Structures for Physics.* Springer, 289–355.

Peter Selinger. 2013. Lecture notes on the lambda calculus. arXiv:0804.3434.

Solveig K. Sieberts, Fan Zhu, Javier García-García, Eli Stahl, Abhishek Pratap, Gaurav Pandey, Dimitrios Pappas, Daniel Aguilar, Bernat Anton, Jaume Bonet, et al. 2016. Crowdsourced assessment of common genetic contribution to predicting anti-TNF treatment response in rheumatoid arthritis. *Nature Communications* 7 (2016).

David Spivak and Robert Kent. 2012. Ologs: a categorical framework for knowledge representation. *PLoS One* 7, 1 (2012).

Yong Wang and Ian H. Witten. 1997. Inducing model trees for continuous classes. In *Proceedings of the Ninth European Conference on Machine Learning.* 128–137.

Christopher A. Welty. 2007. Software Engineering. In *The Description Logic Handbook: Theory, Implementation and Applications* (2 ed.), Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (Eds.). Cambridge University Press, Chapter 11, 402–416.

Linda M. Wills. 1992. *Automated program recognition by graph parsing.* Technical Report. MIT Artificial Intelligence Laboratory.