TEACHING MACHINES TO UNDERSTAND DATA SCIENCE CODE BY SEMANTIC ENRICHMENT OF DATAFLOW GRAPHS

EVAN PATTERSON, IOANA BALDINI, ALEKSANDRA MOJSILOVIĆ, AND KUSH R. VARSHNEY

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 deserve to have intelligent tools that reveal the connections between 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. To create the representations, we introduce an algorithm for enriching dataflow graphs with semantic information. The semantic enrichment algorithm is undergirded by a new ontology language for modeling computer programs and a new ontology about data science, written in this language. Throughout the paper, we focus on code written by data scientists and we locate our work within a larger movement towards collaborative, open, and reproducible science.

1. Introduction

Your computer 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 problem of understanding computer code has received far less attention over 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 asked to create, maintain, and extend computer programs. For all workers, but especially those outside software engineering roles, programming is a means to achieve practical goals, not an end in itself. Programmers deserve 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, by, for example, automatically visualizing machine learning workflows or summarizing data analyses as natural text for human readers.

(Evan Patterson) Stanford Unversity, Statistics Department

(Ioana Baldini, Aleksandra Mojsilović, Kush R. Varshney) IBM Research AI

E-mail addresses: epatters@stanford.edu, ioana@us.ibm.com, aleksand@us.ibm.com, krvarshn@us.ibm.com.

Towards this prospect, we develop an AI system that forms semantic representations of computer programs. Our system is fully automated, inasmuch as it expects nothing from the programmer besides the program itself and the ability to run it. We have designed our system to handle scripts written by data scientists, which tend to be shorter, more linear, and better defined semantically than the large-scale codebases written by software engineers. Our methodology is not universally applicable. Nevertheless, we think it could be fruitfully extended to other scientific domains with a computational focus, such as bioinformatics or computational neuroscience, by integrating it with existing domain-specific ontologies.

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. To support the two representations, we introduce an *ontology language* for modeling computer programs, called Monocl, and an *ontology* written in this language, called the Data Science Ontology. Finally, we propose a *semantic enrichment* algorithm for transforming the raw flow graph into the semantic flow graph. The Data Science Ontology is available online[1] and our system's source code is available on GitHub under a permissive open source license (see Section 7).

Organization of paper. In the next section, we motivate our method through a pedagogical example (Section 2). We then explain the method itself, first informally and with a minimum of mathematics (Section 3) and then again with greater precision and rigor (Section 4). We divide the exposition in this way because the major ideas of the paper can be understood without the mathematical formalism, which may be unfamiliar to some readers. We then take a step back from technical matters to locate our work within the ongoing movement towards collaborative, open, and reproducible data science (Section 5). We also demonstrate our method on a realistic data analysis drawn from a biomedical data science challenge. In the penultimate section, we bring out connections to existing work in artificial intelligence, program analysis, programming language theory, and category theory (Section 6). We conclude with directions for future research and development (Section 7). For a non-technical overview of our work, emphasizing motivation and examples, we suggest reading Sections 1, 2, 5 and 7.

2. First examples

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 analysis: 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. To be more precise, the programs are written in different programming languages—Python and R—and the two Python programs invoke different sets of libraries. Moreover, the programs exemplify different programming paradigms. Listings 1 and 3 are written in functional style and

[1]To browse and search the Data Science Ontology, and to see additional documentation, please visit `https://www.datascienceontology.org`.

```
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

```
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

```
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

Listing 2 is written in object-oriented style. Thus, at the syntactic level, the programs appear to be very different, and conventional program analysis tools would regard them as being very different. However, as readers fluent in Python and R will recognize, the programs perform the same data analysis. They are semantically equivalent, up to possibly numerical error and minor differences in the implementation of the k-means clustering algorithm. (Implementations differ mainly in how the iterative algorithm is initialized.)

Identifying the 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 Data Science 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. At the highest level, two steps connect a computer program to its representations. First, *computer program analysis* distills the raw flow graph from the program. The *raw flow graph* is a dataflow graph that records the concrete function calls made during the execution of the program. This graph is programming language and library dependent. In the second step, a process of *semantic enrichment* transforms the raw flow graph into the semantic flow graph. The *semantic flow graph* describes the same program in terms of abstract concepts belonging to the Data Science Ontology. This graph is programming language and library independent. Thus, both dataflow graphs capture the execution of a

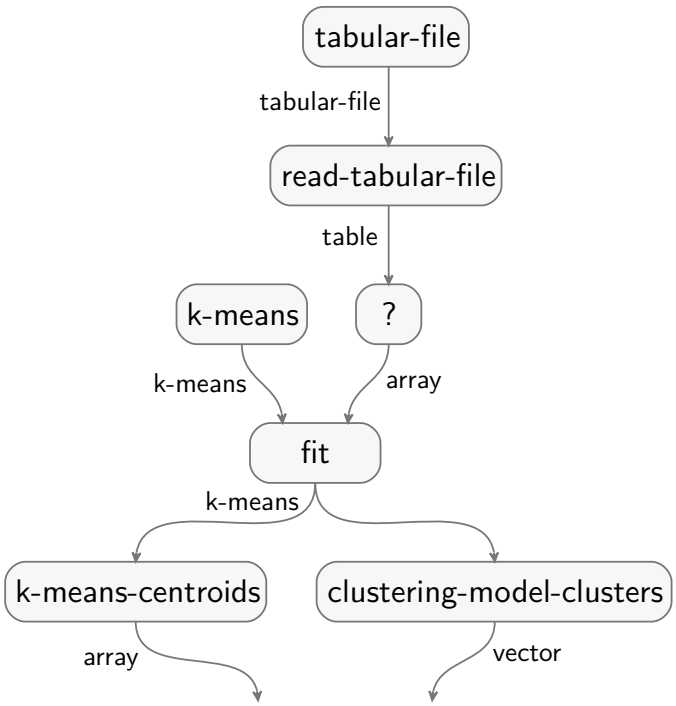


FIGURE 1. Semantic flow graph for three versions of k-means clustering analysis (Listings 1, 2 and 3)

computer program doing data analysis, but at different levels of abstraction. The architecture diagram in Figure 2 summarizes our method.

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 determine how concrete functions in the raw flow graph are translated into abstract functions in the semantic flow graph.

Such, in outline, is our method. Throughout the rest of this section we develop its elements in greater detail, beginning with the Data Science Ontology and the ontology language in which it is expressed.

3.1. **The Data Science Ontology.** We have begun writing an ontology—the Data Science Ontology—about statistics, machine learning, and data processing. It aims to support automated reasoning about data science software.

As we have said, the Data Science Ontology is comprised of concepts and annotations. *Concepts* catalog and formalize the abstract entities of data science, such as data tables and statistical models, as well as processes that manipulate them, like loading data from a file or fitting a model to data. Reflecting the intuitive distinction between "things" and "processes," concepts bifurcate into two kinds: types and functions. The terminology agrees with that of functional programming. Thus, a *type* represents a kind or species of thing in the domain of data science. A *function* is a functional relation or mapping from an input

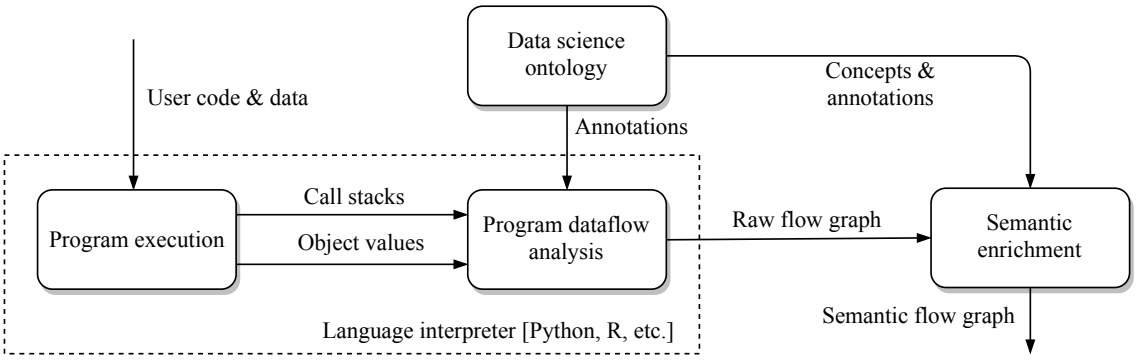


FIGURE 2. System architecture

TABLE 1. Example concepts and annotations from the Data Science Ontology

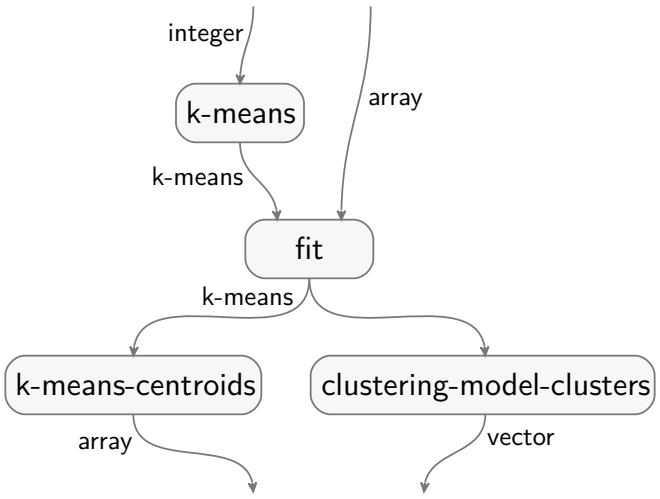
	Concept	**Annotation**
Type	data table	pandas data frame
	statistical model	scikit-learn estimator
Function	reading a tabular data file	`read_csv` function in pandas
	fitting a statistical model to data	`fit` method of scikit-learn estimators

type (the *domain*) to an output type (the *codomain*). In this terminology, the concepts of a data table and of a statistical model are types, whereas the concept of fitting a predictive model is a function that maps an unfitted predictive model, together with predictors and response data, to a fitted predictive model.

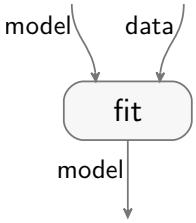
As a modeling assumption, we suppose that software packages for data science, such as Pandas and Scikit-learn, concretize the concepts. *Annotations* say how this concretization occurs by mapping types and functions in software packages onto type and function concepts in the ontology. To avoid confusion between levels of abstraction, we call the former "concrete" and the latter "abstract." Thus, 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.

The division of the ontology into concepts and annotations on the one hand, and into types and functions on the other, leads to a two-way classification. Table 1 lists basic examples of each of the four combinations, drawn from the Data Science Ontology.

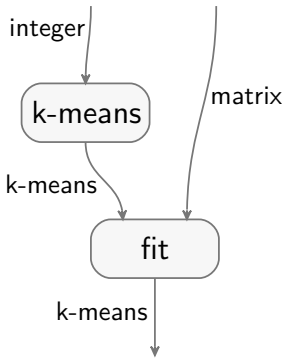
Significant modeling flexibility is needed to accurately translate the diverse APIs of statistical software into a single set of universal concepts. Section 2 shows, for example, that the concept of k-means clustering can be concretized in software in many different ways. To accommodate this diversity, we allow function annotations to map a single concrete function onto an arbitrary abstract "program" comprised of function concepts. In Figure 3, we display three function annotations relevant to the fitting of k-means clustering models in Listings 1, 2 and 3. By the end of this section, we will see how to interpret the three



(A) Annotation: `kmeans2` function in SciPy (cf. Listing 1)



(B) Annotation: `fit` method of `BaseEstimator` class in Scikit-learn (cf. Listing 2)



(C) Annotation: `kmeans` function in R's builtin `stats` package (cf. Listing 3)

FIGURE 3. Example function annotations from the Data Science Ontology

annotations and how the semantic enrichment algorithm uses them to generate the semantic flow graph in Figure 1.

We have not yet said what kind of abstract "program" is allowed to appear in a function annotation. Answering that question is the most important purpose of our ontology language, to which we now turn.

3.2. **The Monocl ontology language.** The Data Science Ontology is expressed in an *ontology language* that we call 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 but is imperfect because Monocl is simpler than any commonly used programming language, being designed for knowledge representation rather than actual computing.

The ontology language says how to construct new types and functions from old, for the purposes of defining concepts and annotations. 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. In this section, we emphasize the more

intuitive graphical syntax. We describe the constructors for types and functions and illustrate them using the graphical syntax. A more formal development is given in Section 4.2.

Monocl has a minimalistic type system, supporting product and unit types as well as a simple form of subtyping. A *basic type*, sometimes called a "primitive type," is a type that cannot be decomposed into simpler types. Basic types must be explicitly defined. All other types are *composite*. For instance, 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. Product types are similar to record types in conventional 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.

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 differs from inheritance in a typical object-oriented programming language. Instead, subtyping should be understood through *implicit conversion*, also known as *coercion* (Reynolds 1980; Pierce 1991). 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.

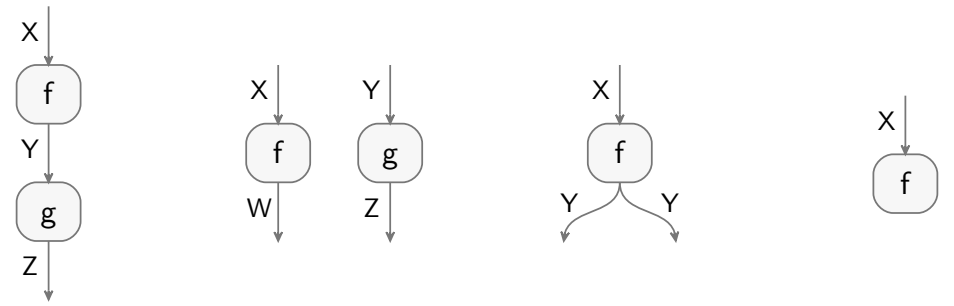
In the graphical syntax, types are represented by wires. A basic type X is drawn as a single wire labeled X. A product of n types is a bundle of n wires in parallel. The unit type is an empty bundle of wires (a blank space). This should become clearer as we discuss wiring diagrams for functions.

A function f in Monocl has an input type X, its *domain*, and an output type Y, its *codomain*. We express this in the usual mathematical notation as $f : X \to Y$. Like types, functions are either basic or composite. Note that a basic function may have composite domain or codomain. From the programming languages perspective, a program in the Monocl language is nothing more than a function.

Functions are represented graphically by *wiring diagrams* (also known as *string diagrams*). A basic function $f : X \to Y$ 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 general composite 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, 3, 5, 6, 7 and 8 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 two 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. 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. Figures 4a and 4b show the graphical syntax for composition and products.

The graphical syntax implicitly includes a number of special functions. For any type X, the *identity* function $1_X : X \to X$ maps every element of type X to itself. For each pair of



(A) Composition (B) Product (C) Copying data (D) Deleting data

FIGURE 4. Graphical syntax for operations on functions

types X and Y, the *braiding* or *transposition* function $\sigma_{X,Y} : X \times Y \to Y \times X$ exchanges its two inputs. Identities and braidings are drawn as straight wires and pairs of crossed wires, respectively. Any permutation function can be expressed by taking compositions and products of identities and braidings. Diagrammatically, this means that a bundle of wires may be criss-crossed in arbitrarily complex ways (provided that the wires do not bend backwards). For each type X, there is also a *copying* function $\Delta_X : X \to X \times X$, which duplicates its input, and a *deleting* function $\lozenge_X : X \to I$, which discards its input. In the graphical syntax, these functions allow a single output port to have multiple or zero outgoing wires. For instance, given a function $f : X \to Y$, Figures 4c and 4d display the compositions $f \cdot \Delta_Y : X \to Y \times Y$ and $f \cdot \lozenge_Y : X \to I$. The analogous situation is not permitted of input ports; in a well-formed wiring diagram, every input port has exactly one incoming wire.

Besides serving as the "is-a" relation ubiquitous in knowledge representation systems, the subtype relation for objects enables ad hoc polymorphism for functions. We extend the definition of function composition to include implicit conversion, namely, to compose a function $f : X \to Y$ with $g : Y' \to Z$, we require not necessarily that Y equals Y', but 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 connecting two boxes has valid types if and only if 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. The general definition of a subfunction is perfectly analogous.

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 are both wiring diagrams representing a data analysis. However, they exist 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 the 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 5, 6 and 7 show the raw flow graphs for Listings 1, 2 and 3, respectively. Note that the wire elements are not shown.

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. Figure 1 shows the semantic flow graph for Listings 1, 2 and 3. Another semantic flow graph is shown in Figure 8 below. Again, the wire elements are not shown.

3.4. **Program analysis.** We use computer program analysis to extract the raw flow graph from a data analysis. Program analysis therefore plays an essential role in our AI system. It plays an equally important role in our original publication on this topic (Patterson, McBurney, et al. 2017), reviewed below in Section 6. In the present work, we have extended our program analysis tools to support the R language, but our basic methodology has changed little. We review only the major points about our usage of computer program analysis, deferring to our original paper for details.

Program analysis can be static or dynamic or both. *Static analysis* consumes the source code but does not execute it. Much literature on program analysis is about static analysis because of its relevance to optimizing compilers (F. Nielson, H. R. Nielson, and Hankin 1999; Aho et al. 2006). *Dynamic analysis*, in contrast, executes the program without necessarily inspecting the code.

Our program analysis is mainly dynamic, for a couple of reasons. Static analysis, especially type inference, is challenging for the highly dynamic languages popular among data scientists. Moreover, we record values computed over the course of the program's execution, such as model parameters and hyperparameters. For this dynamic analysis is indispensable. Of course, a disadvantage of dynamic analysis is the necessity of running the program. Crucially, our system needs not just the code itself, but its input data and runtime environment. These are all requirements of scientific reproducibility (see Section 5), so in principle they ought to be satisfied. In practice they are often neglected.

To build the raw flow graph, our program analysis tools record interprocedural data flow during program execution. We begin with the empty wiring diagram and add boxes incrementally as the program unfolds. Besides recording function calls and their arguments and return values, the main challenge is to track the provenance of objects as they are passed between functions. When a new box is added to the diagram, the provenance record says how to connect the input ports of the new box to the output ports of existing boxes.

How all this is accomplished depends on the programming language in question. In Python, we register callbacks via `sys.settrace`, to be invoked whenever a function is called or returns. A table of object provenance is maintained using weak references. No modification of the abstract syntax tree (AST) is necessary. In R, we add callbacks by rewriting the AST, to be invoked whenever a term is (lazily) evaluated. Care must be taken to avoid breaking functions which use nonstandard evaluation, a kind of dynamic metaprogramming unique to R (Wickham 2014).

Our usage of program analysis involves conceptual as well as engineering difficulties, because the programming model of Monocl is simpler than that of typical programming

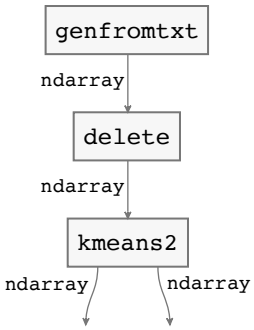


FIGURE 5. Raw flow graph for k-means clustering in Python via NumPy and SciPy (Listing 1)

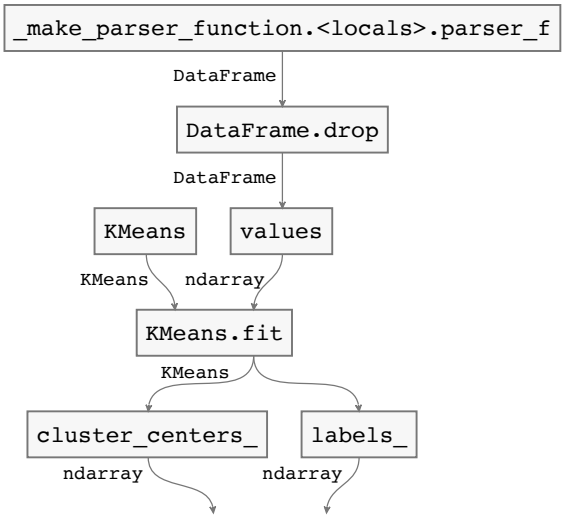


FIGURE 6. Raw flow graph for k-means clustering in Python via Pandas and Scikit-learn (Listing 2)

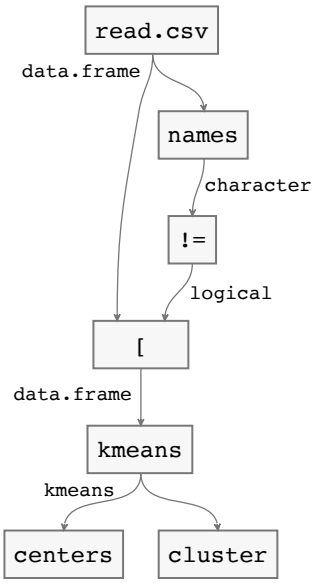


FIGURE 7. Raw flow graph for k-means clustering in R (Listing 3)

languages. We mention just one conceptual problem to give a sense of the issues that arise. Monocl is purely functional, whereas most practical languages allow mutation of objects.[2] Our program analysis tools have limited capabilities for detecting mutations. When a "function" mutates an object, the mutated object is represented in the raw flow graph as an extra output of the function. For instance, we interpret the `fit` method of a Scikit-learn estimator, which modifies the model in-place, as returning a new model (Figure 3b).

3.5. **Semantic enrichment.** The *semantic enrichment* algorithm transforms the raw flow graph into the semantic flow graph. It proceeds in two independent stages, 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 in 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 properly speaking it does not belong in the semantic flow graph. On the other hand, it usually cannot be deleted without altering the dataflow in 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.

[2]Mutation is more common in Python than in R because most R objects have copy-on-modify semantics.

Example revisited. To reprise our original example, semantic enrichment transforms the raw flow graphs of Figures 5, 6 and 7 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.

Expansions related to k-means clustering occur in all three programs. In the Python program based on SciPy (Figure 5), the `kmeans2` function expands into a program that creates a k-means clustering model, fits it to the data, and extracts its cluster assignments and centroids, as described by the annotation in Figure 3a. 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, the Python program based on Scikit-learn (Figure 6) is written in object-oriented style. The `KMeans` class expands to an abstract k-means clustering type. The `fit` method of the `KMeans` class is not annotated in the Data Science Ontology. However, the `fit` method of the superclass `BaseEstimator` *is* annotated (Figure 3b), so the expansion is performed using that annotation. As this case illustrates, subtyping and polymorphism are indispensable when annotating object-oriented code.

The R program (Figure 7) is intermediate between these two styles. The `kmeans` function, annotated in Figure 3c, directly takes the data and the number of clusters, but returns an object of class `kmeans`. The cluster assignments and centroids are slots of this object, annotated separately. This design pattern is typical in R, due to its informal type system.

Now consider the contractions. In the first program (Figure 5), the only unannotated box is NumPy's `delete` function. Contracting this box does not reduce the size of the wiring diagram. A contraction involving multiple boxes occurs in the second program (Figure 6). The subdiagram consisting of the pandas `NDFrame.drop` method composed with the `values` attribute accessor is encapsulated into a single unlabeled box. We have left these functions unannotated for the sake of illustration and because the section of the Data Science Ontology dedicated to data manipulation has not yet been developed. We expect this gap to close as the ontology grows.

4. Mathematical foundations

To put the foregoing ideas on a firmer footing, we formalize the ontology and the semantic enrichment algorithm in the language of category theory. We are not professional category theorists and we have tried to make this section accessible to other non-category theorists. Nevertheless, readers will find it helpful to have a working knowledge of basic category theory, as may be found in the introductory textbooks (Spivak 2014; Awodey 2010; Leinster 2014; Riehl 2016), and of monoidal category theory, as in the survey articles (Baez and Stay 2010; Coecke and Paquette 2010). For readers without this background, or who simply wish to understand our method informally, this section can be skipped without loss of continuity.

Here, in outline, is our program. We take the ontology's concepts to form a category, with type concepts corresponding to objects and function concepts corresponding to morphisms. Defining the ontology language amounts to fixing a categorical doctrine, which will turn out to be the doctrine of cartesian categories with implicit conversion. Up to this point, we conform to the general scheme of categorical knowledge representation, according to which ontologies are simply categories in a suitable doctrine (Spivak and Kent 2012; Patterson 2017). Having defined the ontology's concepts as a category $\mathcal{C}$, we then interpret the annotations as a partial functor from a category $\mathcal{L}$ modeling a software ecosystem to the concept category $\mathcal{C}$. Finally, we formalize the raw and semantic flow graphs as morphisms in categories of elements over $\mathcal{L}$ and $\mathcal{C}$.

4.1. **Why category theory?** Because category theory does not yet belong to the basic toolbox of knowledge representation, we pause to motivate the categorical approach before launching into the formal development. Why is category theory an appealing framework for representing knowledge, especially about computational processes? We offer several answers to this question.

First, there already exist whole branches of computer science, namely type theory and programming language theory, dedicated to the mathematical modeling of computer programs. To neglect them in knowledge representation would be unfortunate. Category theory serves as an algebraic bridge to these fields. 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 dwell in the syntactically and semantically flexible world of algebra but still draw on the highly developed theory of programming languages. In Section 4.2, we borrow specific notions of subtyping and ad hoc polymorphism from programming language theory (Goguen 1978; Reynolds 1980).

Category theory is also useful in its own right, beyond its connection to programming language theory. The essential innovation of category theory over the mainly syntactical theory of programming languages is that programs become *algebraic structures*, analogous to, albeit more complicated than, classical algebraic structures like groups and monoids. Like any algebraic structure, categories of programs are automatically endowed with an appropriate notion of structure-preserving map between them. In this case, the structure-preserving maps are a special kind of *functor*. In Section 4.3, we formulate the semantic enrichment algorithm as a functor between categories of programs. The structuralist philosophy underlying modern algebra is therefore central to our whole approach.

Another advantage of category theory is flexibility of syntax. Unlike the lambda calculus and other type theories, algebraic structures like categories exist independently of any particular system of syntax. Syntactic flexibility is mathematically convenient but also practically important. Monoidal categories admit a graphical syntax of *wiring diagrams*, also known as *string diagrams* (Baez and Stay 2010; Selinger 2010). We introduced the graphical syntax informally in Section 3.2. It offers an intuitive yet rigorous alternative to the typed lambda calculus's textual syntax (Selinger 2013), which beginners may find impenetrable. The family of graphical languages based on string diagrams is a jewel of category theory, with applications to such diverse fields as quantum mechanics (Coecke and Paquette 2010), control theory (Baez and Erbele 2015), and natural language semantics (Coecke, Grefenstette, and Sadrzadeh 2013).

Having arrived at the general categorical perspective, the next question to ask is: what kind of category shall we use to model computer programs? We begin our investigation with cartesian categories, which are perhaps the simplest possible model of typed, functional computing. As we recall more carefully in Section 4.2, *cartesian categories* are symmetric monoidal categories with natural operations for copying and deleting data. Morphisms in a cartesian category behave like mathematical functions.

As a model of computation, cartesian categories are very primitive. They do not allow for manipulating functions as data (via lambda abstraction) or for recursion (looping), hence they can only express terminating computations of fixed, finite length. Extensions of this computational model abound. *Cartesian closed categories* arise as cartesian categories with a *closed* structure, whereby the whole collection of morphisms $X \to Y$ is representable as an *exponential* object Y^X. Closed categories have function types, in programming jargon. According to a famous result, cartesian closed categories are equivalent to the typed lambda calculus (Lambek and Scott 1988). *Traced symmetric monoidal categories* model looping

and other forms of feedback. According to another classic result, a trace on a cartesian category is equivalent to a Conway fixed point operator (Hasegawa 1997; Hasegawa 2003). Fixed points are used in programming language theory to define the semantics of recursion. Combining these threads, we find in *traced cartesian closed categories* a Turing-complete model of functional computing, amounting to the typed lambda calculus with a fixed point operator.

Relaxing the cartesian or even the monoidal structure is another way to boost modeling flexibility. Starting with the cartesian structure, we interpret morphisms that are unnatural with respect to copying as performing non-deterministic computation, such as random number generation or Monte Carlo sampling. We interpret morphisms unnatural with respect to deleting as partial functions, because they raise errors or are undefined on certain inputs. In a symmetric monoidal category $\mathcal{C}$ with diagonals (not necessarily cartesian), the morphisms that *do* satisfy the naturality conditions for copying and deleting data form a cartesian subcategory of $\mathcal{C}$, called the *cartesian center* or *focus* of $\mathcal{C}$ (Selinger 1999). It is also possible to relax the monoidal product itself. *Symmetric premonoidal categories* model side effects and imperative programs, where evaluation order matters even for parallel statements, such as variable access and assignment. Any premonoidal category has a *center* that is a monoidal category (Power and Robinson 1997). Thus, classical computational processes form a three-level hierarchy: a symmetric premonoidal category has a center that is symmetric monoidal, which in turn has a cartesian center (Jeffrey 1997).

This short survey hardly exhausts the categorical structures that have been used to model computer programs. However, our purpose here is not to define the most general model possible, but rather to adopt the *simplest* model that still captures useful information in practice. For us, that model is the cartesian category. The structures in this categorical doctrine agree with the features currently supported by our program analysis tools (Section 3.4). We expect that over time our software will acquire more features and achieve better fidelity, whereupon we will adopt a more expressive doctrine. The survey above shows that this transition can happen smoothly. In general, modularity is a key advantage of categorical knowledge representation: category theory provides a toolkit of mathematical structures that can be assembled in more or less complex ways to meet different modeling needs.

Notation. We compose our maps in diagrammatic (left-to-right) order. In particular, we write the composition of a morphism $f : X \to Y$ with another morphism $g : Y \to Z$ as $f \cdot g : X \to Z$ or simply $fg : X \to Z$. Small categories $\mathcal{C}, \mathcal{D}, \mathcal{E}, \ldots$ are written in script font and large categories in bold font. As standard examples of the latter, we write **Set** for the category of sets and functions and **Cat** for the category of (small) categories and functors. Other categories will be introduced as needed.

4.2. **Concepts as category.** We formalize the ontology as a category. The type and function concepts in the ontology are, respectively, the objects and morphisms that generate the category. Abstract programs expressed in terms of concepts correspond to general morphisms in the category, assembled from the object and morphism generators by operations like composition and monoidal products. In this subsection, we develop the categorical doctrine where the ontology category will reside, by augmenting cartesian categories, motivated in Section 4.1, with a form of subtyping based on implicit conversion. Ultimately, we define a Monocl ontology to be a finite presentation of a cartesian category with implicit conversion.

The definition of diagonals in a monoidal category is fundamental (Selinger 1999). In stating it, we take for granted the definition of a *symmetric monoidal category*; see the references at the beginning of this section for further reading.

Definition. A *monoidal category with diagonals* is a symmetric monoidal category $(\mathcal{C}, \times, 1)$ together with two families of morphisms,

$$\Delta_X : X \to X \times X \qquad \text{and} \qquad \lozenge_X : X \to 1,$$

indexed by objects $X \in \mathcal{C}$. The morphisms Δ_X and $\lozenge_X$, called *copying* and *deleting*, respectively, are required to make X into a cocommutative comonoid (the formal dual of a commutative monoid). Moreover, the families must be *coherent*, or *uniform*, in the sense that $\lozenge_1 = 1_1$ and for all objects $X, Y \in \mathcal{C}$, the diagrams commute:

$$\begin{array}{ccc} X \times Y & \xrightarrow{\Delta_X \times \Delta_Y} & X \times X \times Y \times Y \\ & \searrow^{\Delta_{X\times Y}} & \downarrow{\scriptstyle 1_X \times \sigma_{X,Y} \times 1_Y} \\ & & X \times Y \times X \times Y \end{array} \qquad \begin{array}{ccc} X \times Y & \xrightarrow{\lozenge_X \times \lozenge_Y} & 1 \times 1 \\ & \searrow^{\lozenge_{X\times Y}} & \downarrow{\scriptstyle \cong} \\ & & 1 \end{array}$$

As explained in graphical terms in Section 3.2, the copying and deleting morphisms allow data to be duplicated and discarded, a basic feature of classical (but not quantum) computation. Uniformity is a technical condition ensuring that copying and deleting are compatible with the symmetric monoidal structure. So, for example, a uniform copying operation has the property that copying data of type $X \times Y$ is equivalent to simultaneously copying data of type X and copying data of type Y, up to the ordering of the outputs.

A monoidal category with diagonals is a very general algebraic structure. Its morphisms need not resemble computational processes in any conventional sense. However, adding just one additional axiom yields the cartesian category, a classical notion in category theory and a primitive model of functional computing.

Definition. A *cartesian category* is a monoidal category with diagonals whose copying and deleting maps, Δ_X and $\lozenge_X$, are *natural* in X, meaning that for any morphism $f : X \to Y$, the diagrams commute:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle \Delta_X}\downarrow & & \downarrow{\scriptstyle \Delta_Y} \\ X \times X & \xrightarrow{f \times f} & Y \times Y \end{array} \qquad \begin{array}{ccc} X & \xrightarrow{f} & Y \\ & \searrow^{\lozenge_X} & \downarrow{\scriptstyle \lozenge_Y} \\ & & 1 \end{array}$$

We denote by **Cart** the category of (small) cartesian categories and cartesian functors (strong monoidal functors preserving the diagonals).

Remark. Although it is not obvious, this definition of cartesian category is equivalent to the standard definition via the universal property of finite products (Heunen and Vicary 2012). We prefer the alternative definition given here because it is phrased in the language of monoidal categories and string diagrams.

In a cartesian category, the naturality conditions on copying and deleting assert that computation is deterministic and total. In more detail, naturality of copying says that computing a function f, then copying the output is the same as copying the input, then computing f on both copies. This means that f always produces the same output on a given input, i.e., f is *deterministic*. Naturality of deleting says that computing the function f, then deleting the output is the same as simply deleting the input. This means that f is

well-defined on all its inputs, i.e., f is *total*. Together, the naturality conditions establish that the category's morphisms behave like mathematical functions.

Cartesian categories are perhaps the simplest model of typed, functional computing, as we argued in Section 4.1. We considered there several extensions and relaxations of the cartesian structure, all centered around morphisms. One can also entertain richer constructions on objects. In programming jargon, this amounts to adding a more elaborate type system. A cartesian category has a type system with product and unit types, introduced from the programming languages perspective in Section 3.2.

In our experience, augmenting the type system with some form of polymorphism is a practical necessity, for the sake of code annotation and also of knowledge representation. We will not try to summarize the large literature on polymorphism. In keeping with the spirit of this paper, our objective is to define the minimal practically useful system. The following definitions are adapted, with relatively minor modifications, from Joseph Goguen and John C. Reynolds (Goguen 1978; Goguen and Meseguer 1992; Reynolds 1980).

Definition. A *category with implicit conversion* is a category $\mathcal{C}$ with a distinguished wide subcategory $\mathcal{C}_0$ containing at most one morphism between any two objects. If there exists a morphism $X \to X'$ in $\mathcal{C}_0$, we write $X \leq X'$ and say that X is a *subtype* of X'. The morphism $X \to X'$ itself is called an *implicit conversion* or *coercion*.

Remark. To be consistent in our usage of categorical and programming terminology, we ought to say that X is a *subobject* of X'. However, the term "subobject" already has an established meaning in categorical logic, which is related to, but different than, our usage here.

We explained the informal interpretation of subtyping and implicit conversion in Section 3.2. One subtle point should be noted: even when types are interpreted as sets, implicit conversions are not necessarily interpreted as set inclusions. In the example from Section 3.2, matrices are a subtype of data tables, yet the set of matrices is *not* a subset of the set of data tables. (The implicit conversion function adds names to the columns of the matrix.) Hence the slogan that "types are not sets" (Morris 1973).

Mathematically speaking, the subtype relation defines a preorder on the objects of $\mathcal{C}$. Thus, every type X is a subtype of itself. If X is a subtype of X' and X' a subtype of X'', then X is a subtype of X''. The corresponding implicit conversions are given by identities and by composition, respectively. In what follows, there is no mathematical obstruction to allowing the conversions $\mathcal{C}_0$ to form an arbitrary category, not necessarily a preorder. That would, however, defeat the practical purpose: conversions would need to be disambiguated by *names* and hence would cease to be implicit.

When $\mathcal{C}$ is a monoidal category, we insist that implicit conversions be compatible with the monoidal structure.

Definition. A *cartesian category with implicit conversion* is a category $\mathcal{C}$ with implicit conversion that is also cartesian. Moreover, the implicit conversions $\mathcal{C}_0$ must form a *monoidal* subcategory of $\mathcal{C}$.

We denote by $\mathbf{Cart}_{\leq}$ the category whose objects are the (small) cartesian categories with implicit conversion and whose morphisms are the cartesian functors that preserve implicit conversions. For brevity, we call these morphisms simply "functors."

The definition requires that subtyping be compatible with product types. Specifically, if $X \leq X'$ and $Y \leq Y'$, then $X \times Y \leq X' \times Y'$, with the corresponding implicit conversion given by a product of morphisms. The subtype relation thus makes $\mathcal{C}$ into a *monoidal preorder*.

Remark. Asking $\mathcal{C}_0$ to inherit the cartesian or even the symmetric monoidal structure leads to undesirable consequences, such as unwanted implicit conversions and even strictification of the original category $\mathcal{C}$. Namely, if $\mathcal{C}_0$ is a symmetric monoidal subcategory of $\mathcal{C}$, then the braidings $\sigma_{X,Y} : X \times Y \to Y \times X$ in $\mathcal{C}$ must satisfy $\sigma_{X,X} = 1_{X \times X}$, which is false under the intended set-theoretic interpretation.

Because our notion of subtyping is operationalized by the implicit conversions, we can extend it from objects to morphisms through naturality squares.

Definition. Let $\mathcal{C}$ be a category with implicit conversion. A morphism f in $\mathcal{C}$ is a *submorphism* (or *subfunction*) of another morphism f', written $f \leq f'$, if in the arrow category $\mathcal{C}^{\to}$ there exists a (unique) morphism $f \to f'$ whose components are implicit conversions.

Explicitly, if $f : X \to Y$ and $f' : X' \to Y'$ are morphisms in $\mathcal{C}$, with $X \leq X'$ and $Y \leq Y'$, then $f \leq f'$ if and only if the diagram commutes:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle\leq}\downarrow & & \downarrow{\scriptstyle\leq} \\ X' & \xrightarrow{f'} & Y' \end{array}$$

Remark. In a closed category, subtypes of basic types, $X \leq X'$ and $Y \leq Y'$, canonically induce subtypes of function types, $Y^{X'} \leq (Y')^X$, by "restricting the domain" and "expanding the codomain." Be warned that this construction is *not* the same as a submorphism (it is contravariant in X, while a submorphism is covariant in both X and Y). Indeed, we do not treat cartesian closed categories at all in this paper.

Again, see Section 3.2 for informal interpretation and examples of this notion. Just as subtypes define a preorder on the objects of $\mathcal{C}$, submorphisms define a preorder on the morphisms of $\mathcal{C}$. Moreover, submorphisms respect the compositional structure of $\mathcal{C}$. They are closed under identities, i.e., $1_X \leq 1_{X'}$ whenever $X \leq X'$, and under composition, i.e., if $f \leq f'$ and $g \leq g'$ are composable, then $fg \leq f'g'$. All these statements are easy to prove. To illustrate, transitivity and closure under composition are proved by pasting commutative squares vertically and horizontally:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle\leq}\downarrow & & \downarrow{\scriptstyle\leq} \\ X' & \xrightarrow{f'} & Y' \\ {\scriptstyle\leq}\downarrow & & \downarrow{\scriptstyle\leq} \\ X'' & \xrightarrow{f''} & Y'' \end{array} \qquad \begin{array}{ccccc} X & \xrightarrow{f} & Y & \xrightarrow{g} & Z \\ {\scriptstyle\leq}\downarrow & & \downarrow{\scriptstyle\leq} & & \downarrow{\scriptstyle\leq} \\ X' & \xrightarrow{f'} & Y' & \xrightarrow{g'} & Z' \end{array}$$

When $\mathcal{C}$ is a *cartesian* category with implicit conversion, submorphisms are also closed under products: if $f \leq f'$ and $g \leq g'$, then $f \times g \leq f' \times g'$, because, by functorality, monoidal products preserve commutative diagrams.

We now define an ontology to be nothing other than a *finitely presented* cartesian category with implicit conversion. More precisely:

Definition. An *ontology in the Monocl language* is a cartesian category with implicit conversion, given by a finite presentation. That is, it is the cartesian category with implicit conversion generated by finite sets of:

- *basic types*, or *object generators*, X
- *basic functions*, or *morphism generators*, $f : X \to Y$, where X and Y are objects

- *basic subtypes*, or *subtype generators*, $X \leq X'$, where X and X' are objects
- *basic subfunctions*, or *submorphism generators*, $f \leq f'$, where $f : X \to Y$ and $f' : X' \to Y'$ are morphisms satisfying $X \leq X'$ and $Y \leq Y'$
- *function equations*, or *morphism equations*, $f = g$, where $f, g : X \to Y$ are morphisms with equal domains and codomains.

If the set of morphism equations is empty, the category is called *free* or *freely generated*.

Strictly speaking, a finite presentation of a category is not the same as the category it presents. The former is a finitary object that can be represented on, and manipulated by, a machine. The Monocl language consists of a textual and graphical syntax for defining presentations on a computer. The latter is an algebraic structure of infinite size, convenient for mathematical reasoning. However, we will abuse terminology by calling both finitely presented categories, and particular presentations thereof, "ontologies."

At the time of this writing, the Data Science Ontology is freely generated. Inference in a freely generated ontology is straightforward. Deciding the subtype or subfunction relations amounts to computing a reflexive transitive closure. Deciding equality of objects is trivial. Deciding equality of morphisms is the *word problem* in a free cartesian category. The congruence closure algorithm for term graphs (Baader and Nipkow 1999, §4.4) can be adapted to solve this problem. In the future, the Data Science Ontology will likely include knowledge in the form of morphism equations, creating a need for new inference procedures. If arbitrary morphism equations are allowed, the word problem becomes undecidable.

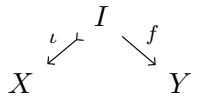
4.3. **Annotations as functor.** If the concepts form a category, then surely the annotations ought to assemble into a functor. Let the ontology be a cartesian category $\mathcal{C}$ with implicit conversion. Suppose we have another such category $\mathcal{L}$, modeling a programming language and a collection of modules written in that language. The annotations should define a functor $F : \mathcal{L} \to \mathcal{C}$, saying how to translate programs in $\mathcal{L}$ into programs in $\mathcal{C}$.

This tidy story does not quite survive contact with reality. We cannot expect a finite set of formal concepts to exhaust the supply of informal concepts found in real-world programs. Therefore any "functor" $F : \mathcal{L} \to \mathcal{C}$ annotating $\mathcal{L}$ must be *partial*, in a sense that we will make precise. There will be both objects and morphisms in $\mathcal{L}$ on which F cannot be defined, because the category $\mathcal{C}$ is not rich enough to fully interpret $\mathcal{L}$.

We approach partial functors indirectly, by way of partial functions. In accordance with mathematical custom, we reduce the pre-theoretical idea of "partial function" to the ubiquitous notion of total function. There are two standard ways to do this, the first based on pointed sets and the second on spans. They are equivalent as far as sets and functions are concerned but suggest different generalizations to categories and functors. Let us consider them in turn.

The category of pointed sets leads to one viewpoint on partiality, popular in programming language theory. Given a set X, let $X_\perp := X \sqcup \{\perp\}$ be the set X with a freely adjoined base point $\perp$. A *partial function* from X to Y is then a function $f : X_\perp \to Y_\perp$ preserving the base point ($f(\perp) = \perp$). The function f is regarded as "undefined" on the points $x \in X$ with $f(x) = \perp$. This notion of partiality can be transported from sets to categories using enriched category theory (Kelly 1982; Riehl 2014). Categories enriched in pointed sets, where each hom-set has a base morphism $\perp$, have been proposed as a qualitative model of incomplete information (Marsden 2016). Such categories make partiality an all-or-nothing affair, because their composition laws satisfy $\perp \cdot f = f \cdot \perp = \perp$ for all morphisms f. That is far too stringent. If we adopted this composition law, our semantic representations would rarely be anything but the trivial representation $\perp$.

Alternatively, a partial function can be defined as a special kind of span of total functions. Now let us say that a *partial function* from X to Y is a span in **Set**

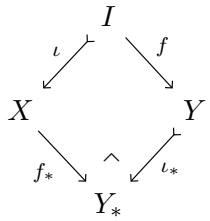


whose left leg $\iota : I \to X$ is monic (injective). The partial function's domain of definition is I, which we regard as a subset of X. Although we shall not need it here, we note that partial functions, and partial morphisms in general, can be composed by taking pullbacks (Borceux 1994, §5.5).

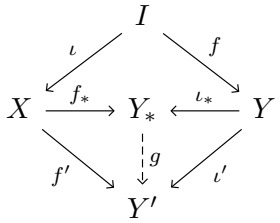
We interpret the span above as *partially* defining a function f on X, via a set of equations indexed by I:

$$f(x_i) := y_i, \qquad i \in I.$$

It is then natural to ask: what is the most general way to define a *total* function on X obeying these equations? The answer is given by the pushout in **Set**:

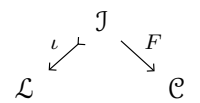


Because $\iota : I \to X$ is monic, so is $\iota_* : Y \to Y_*$, and we regard Y as a subset of Y_*. The commutativity of the diagram says that f_* satisfies the set of equations indexed by I. The *universal property* defining the pushout says that any other function $f' : X \to Y'$ satisfying the equations factors uniquely through f_*, meaning that there exists a unique function $g : Y_* \to Y'$ making the diagram commute:

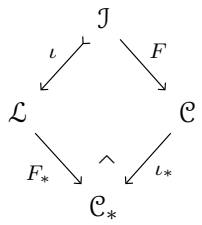


The codomain of the function $f_* : X \to Y_*$ consists of Y plus a "formal image" $f(x)$ for each element x on which f is undefined. Contrast this with the codomain of a function $X \to Y_\perp$, which consists of Y plus a single element $\perp$ representing *all* the undefined values.

This viewpoint on partiality generalizes effortlessly from **Set** to any category with pushouts. We partially define the annotations as a span in $\mathbf{Cart}_\leq$



whose left leg $\iota : \mathcal{I} \to \mathcal{L}$ is monic. We then form the pushout in $\mathbf{Cart}_\leq$:



Given a morphism f in $\mathcal{L}$, which represents a concrete program, its image $F_*(f)$ in $\mathcal{C}_*$ is a partial translation of the program into the language defined by the ontology's concepts.

The universal property of the pushout in $\mathbf{Cart}_\leq$, stated above in the case of **Set**, gives an appealing intuitive interpretation to program translation. The category $\mathcal{C}$ is not rich enough to fully translate $\mathcal{L}$ via a functor $\mathcal{L} \to \mathcal{C}$. As a modeling assumption, we suppose that $\mathcal{C}$ has some "completion" $\overline{\mathcal{C}}$ for which a full translation $\overline{F} : \mathcal{L} \to \overline{\mathcal{C}}$ *is* possible. We do not know $\overline{\mathcal{C}}$, or at the very least we cannot feasibly write it down. However, if we take the pushout functor $F_* : \mathcal{L} \to \mathcal{C}_*$, we can at least guarantee that, no matter what the complete translation $\overline{F}$ is, it will factor through F_*. Thus F_* defines the most general possible translation, given the available information.

The properties of partial functions largely carry over to partial functors, with one important exception: the "inclusion" functor $\iota_* : \mathcal{C} \to \mathcal{C}_*$ need not be monic, even though $\iota : \mathcal{I} \to \mathcal{L}$ is. Closely related is the fact that $\mathbf{Cart}_\leq$ (like its cousins **Cat** and **Cart**, but unlike **Set**) does not satisfy the *amalgamation property* (MacDonald and Scull 2009). To see how ι_* can fail to be monic, suppose that the equation $f_1 \cdot f_2 = f_3$ holds in $\mathcal{L}$ and that the defining equations include $F(f_i) := g_i$ for $i = 1, 2, 3$. Then, by the functorality of F_*, we must have $g_1 \cdot g_2 = g_3$ in $\mathcal{C}_*$, even if $g_1 \cdot g_2 \neq g_3$ in $\mathcal{C}$. Thus the existence of F_* can force equations between morphisms in $\mathcal{C}_*$ that do not hold in $\mathcal{C}$.

When the categories in question are finitely presented, the pushout functor also admits a finitary, equational presentation, suitable for computer algebra. Just as we define an ontology to be a finitely presented category, we define an ontology with annotations to be a finitely presented functor.

Definition. An *ontology with annotations in the Monocl language* is a functor between cartesian categories with implicit conversion, defined by a finite presentation. Explicitly, it is generated by:

- a finite presentation of a category $\mathcal{C}$ in $\mathbf{Cart}_\leq$, the ontology category;
- a finite presentation of a category $\mathcal{L}$ in $\mathbf{Cart}_\leq$, the programming language category; and
- a finite set of equations partially defining a functor F from $\mathcal{L}$ to $\mathcal{C}$.

The equations partially defining the functor F may be indexed by a category $\mathcal{I}$, in which case they take the form

$$F(X_i) := Y_i, \qquad X_i \in \mathcal{L}, \quad Y_i \in \mathcal{C},$$

for each $i \in \mathcal{I}$, and

$$F(f_k) := g_k, \qquad f_k \in \mathcal{L}(X_i, X_j), \quad g_k \in \mathcal{C}(Y_i, Y_j),$$

for each $i, j \in \mathcal{I}$ and $k \in \mathcal{I}(i, j)$. The equations present a span $\mathcal{L} \xleftarrow{\iota} \mathcal{I} \xrightarrow{F} \mathcal{C}$ whose left leg is monic and the functor generated by the equations is the pushout functor $F_* : \mathcal{L} \to \mathcal{C}_*$ described above.

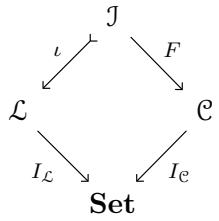
Remark. Our two definitions involving finite presentations are not completely rigorous, but can be made so using generalized algebraic theories (Cartmell 1978; Cartmell 1986). There is a generalized algebraic theory of cartesian categories with implicit conversion, whose category of models is $\mathbf{Cart}_{\leq}$, and a theory of functors between them, whose category of models is the arrow category $\mathbf{Cart}_{\leq}^{\to}$. Cartmell gives as simpler examples the theory of categories, with models $\mathbf{Cat}$, and the theory of functors, with models $\mathbf{Cat}^{\to}$ (Cartmell 1986). Any category of models of a generalized algebraic theory is cocomplete and admits free models defined by finite presentations.

Before closing this subsection, we should acknowledge what we have left unformalized. In construing the annotations as a functor, we model programming languages like Python and R as cartesian categories with implicit conversion. We do not attempt to do so rigorously. The formal semantics of Python and R are quite intricate and exist only in fragments (Guth 2013; Morandat et al. 2012). Our program analysis involves numerous simplifications, infidelities, and heuristics, as sketched in Section 3.4. Even if we could complete it, a formalization would probably be too complicated to illuminate anything about our method. We thus rest content with an informal understanding of the relationship between Monocl and full-fledged programming languages like Python and R.

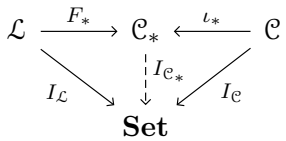
4.4. **Flow graphs and categories of elements.** To a first approximation, the raw and semantic flow graphs are morphisms in the categories $\mathcal{L}$ and $\mathcal{C}_*$, respectively. The expansion stage of the semantic enrichment algorithm simply applies the annotation functor $F_* : \mathcal{L} \to \mathcal{C}_*$ to a morphism in $\mathcal{L}$. The contraction stage, a purely syntactic operation, groups together morphisms in $\mathcal{C}_*$ that are not images of $\mathcal{C}$ under the inclusion functor $\iota_* : \mathcal{C} \to \mathcal{C}_*$.

To complete the formalization of semantic enrichment, we must account for the observed elements in the raw and semantic flow graphs. As noted in Section 3, flow graphs capture not only the types and functions comprising a program, but also the values computed by the program. In category theory, values can be bundled together with objects and morphisms using a device known as the *category of elements*. We formalize the raw and semantic flow graphs as morphisms in suitable categories of elements.

The objects and morphisms in the ontology category $\mathcal{C}$ can be, in principle, interpreted as sets and functions. A set-theoretic *interpretation* of $\mathcal{C}$ is a cartesian functor $I_{\mathcal{C}} : \mathcal{C} \to \mathbf{Set}$. In programming language terms, $I_{\mathcal{C}}$ is a *denotational semantics* for $\mathcal{C}$. Suppose the concrete language $\mathcal{L}$ also has an interpretation $I_{\mathcal{L}} : \mathcal{L} \to \mathbf{Set}$. Assuming the equations partially defining the annotation functor are true under the set-theoretic interpretations, the diagram below commutes:



By the universal property of the annotation functor F_*, there exists a unique interpretation $I_{\mathcal{C}_*} : \mathcal{C}_* \to \mathbf{Set}$ making the diagram commute:



Each of these three interpretations yields a category of elements, also known as a "Grothendieck construction" (Barr and Wells 1990, §12.2; Riehl 2016, §2.4).

Definition. The *category of elements* of a cartesian functor $I : \mathcal{C} \to \mathbf{Set}$ has as objects, the pairs (X, x), where $X \in \mathcal{C}$ and $x \in I(X)$, and as morphisms $(X, x) \to (Y, y)$, the morphisms $f : X \to Y$ in $\mathcal{C}$ satisfying $I(f)(x) = y$.

The category of elements of a cartesian functor $I : \mathcal{C} \to \mathbf{Set}$ is itself a cartesian category. Composition and identities are inherited from $\mathcal{C}$. Products are defined on objects by

$$(X, x) \times (Y, y) := (X \times Y, (x, y))$$

and on morphisms exactly as in $\mathcal{C}$, and the unit object is $(1, *)$, where $*$ is an arbitrary fixed element. The diagonals are also inherited from $\mathcal{C}$, taking the form

$$\Delta_{(X,x)} : (X, x) \to (X \times X, (x, x)), \qquad \Diamond_{(X,x)} : (X, x) \to (1, *).$$

We may at last define a *raw flow graph* to be a morphism in the category of elements of $I_{\mathcal{L}}$. Likewise, a *semantic flow graph* is a morphism in the category of elements of $I_{\mathcal{C}_*}$. Note that the interpretations of $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{C}_*$ are conceptual devices; we do not actually construct a denotational semantics for the language $\mathcal{L}$ or the ontology $\mathcal{C}$. Instead, the program analysis tools observe a *single* computation and produce a *single* morphism f in the category of elements of $I_{\mathcal{L}}$. By the definition of the interpretation $I_{\mathcal{C}_*}$, applying the annotation functor $F_* : \mathcal{L} \to \mathcal{C}_*$ to this morphism f yields a morphism $F_*(f)$ belonging to the category of elements of $I_{\mathcal{C}_*}$.

In summary, semantic enrichment amounts to applying the annotation functor in the category of elements. The expansion stage simply computes the functor. As an aid to human interpretation, the contraction stage computes a new syntactic expression for the expanded morphism, grouping together boxes that do not correspond to morphisms in the ontology category.

5. The view from data science

Like the code it analyzes, our AI system is a means, not an end. Its impetus is the transformation of science, currently under way, towards greater openness, transparency, reproducibility, and collaboration. As part of this transformation, data and machines will both come to play a more prominent role in science. In this section, we describe the major themes of this evolution of the scientific process and how we hope our work will contribute to it. We also demonstrate our system on a realistic data analysis from the open science community.

5.1. Towards networked science.

Although the World Wide Web has already radically changed the dissemination of scientific research, its potential as a universal medium for representing and sharing scientific knowledge is only just beginning to be realized. A vast library of scientific books and papers is now available online, accessible instantaneously and throughout the world. That is a remarkable achievement, accomplished in only a few decades. However, this endorsement must be qualified in many respects. Scientific articles are accessible—but only to certain people, due to the prevalence of academic paywalls. Even when articles are freely available, the associated datasets, data analysis code, and supporting software may not be. These research artifacts are, moreover, often not amenable to systematic machine processing. In short, most scientific research is now on the Web, but it may not be accessible, reproducible, or readily intelligible to humans or machines.

A confluence of social and technological forces is pushing the scientific community towards greater openness and interconnectivity. The open access movement is gradually

eroding paywalls (Piwowar et al. 2018). The replication crisis affecting several branches of science has prompted calls for stricter standards about research transparency, especially when reporting data analysis protocols (Pashler and Wagenmakers 2012; Munafò et al. 2017). A crucial standard of transparency is *reproducibility*: the ability of researchers to duplicate the complete data analysis of a previous study, from the raw data to the final statistical inferences (Goodman, Fanelli, and Ioannidis 2016). Reproducibility demands that all relevant datasets, analysis code, and supporting software be available—the same requirements imposed by our system.

Another driving force is the growing size and complexity of scientific data. Traditionally, the design, data collection, data analysis, and reporting for a scientific experiment has been conducted entirely within a single research group or laboratory. That is changing. Large-scale observational studies and high-throughput measurement devices are producing ever larger and richer datasets, making it more difficult for the people who collect the data to also analyze it. Creating richer datasets also increases the potential gains from data sharing and reuse. The FAIR Data Principles aim to simplify data reuse by making datasets more "FAIR": findable, accessible, interoperable, and reusable (Wilkinson et al. 2016). Organizations like the Accelerated Cure Project for Multiple Sclerosis and the Parkinson Progression Marker Initiative are creating integrated online repositories of clinical, biological, and imaging data (Marek et al. 2011). In a related development, online platforms like Kaggle, Driven Data, and DREAM Challenges are crowdsourcing data analysis through data science competitions.

Science, then, seems to be headed towards a world where all the products of scientific research, from datasets to code to published papers, are fully open, online, and accessible. In the end, we think this outcome is inevitable, even if it is delayed by incumbent interests and misaligned incentives. The consequences of this new "networked science" are difficult to predict, but they could be profound (Hey, Tansley, and Tolle 2009; Nielsen 2012). We and others conjecture that new forms of open, collaborative science, where humans and machines work together according to their respective strengths, will accelerate the pace of scientific discovery.

An obstacle to realizing this vision is the lack of standardization and interoperability in research artifacts. Researchers cannot efficiently share knowledge, data, or code, and machines cannot effectively process it, if it is not represented in formats that they readily understand. We aim to address one aspect of this challenge by creating semantic representations of data science code. We will say shortly what kind of networked science applications we hope our system will enable. But first we describe more concretely one particular model of networked science, the data science challenge, and a typical example of the analysis code it produces.

5.2. **An example from networked science.** As a more realistic example, in contrast to Section 2, we examine a data analysis conducted for a DREAM Challenge. DREAM Challenges address scientific questions in systems biology and translational medicine by crowdsourcing data analysis across the biomedical research community (Stolovitzky, Monroe, and Califano 2007; Saez-Rodriguez et al. 2016). Under the challenge model, teams compete to create the best statistical models according to metrics defined by the challenge organizers. Rewards may include prize money and publication opportunities. In some challenges, the initial competitive phase is followed by a cooperative phase where the best performing teams collaborate to create an improved model (see, for example, DREAM Challenges 2017; Sieberts 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). Of special interest is whether genetic biomarkers can serve as a viable substitute for more obviously relevant clinical diagnostics. To answer this question, each participant was asked to submit two models, one using only genetic covariates and the other using any combination of clinical and genetic covariates. After examining a wide range of models, the challenge organizers and participants jointly concluded that the genetic covariates do not meaningfully increase the predictive power beyond what is already contained in the clinical covariates.

We use our system to analyze the two models submitted by a top-ranking team (Kramer et al. 2014). The source code for the models, written in R, is shown in Listing 4. It has been lightly modified for portability. The corresponding semantic flow graph is shown in Figure 8. The reader need not try to understand the code in any great detail. Indeed, we hope that the semantic flow graph will be easier to comprehend than the code and hence will be serve as an aid to humans as well as to machines. We grant, however, that the current mode of presentation is far from ideal from the human perspective.[3]

The analysts fit two predictive models, the first including both genetic and clinical covariates and the second including only clinical covariates. The models correspond, respectively, to the first and second commented code blocks and to the left and right branches of the semantic flow graph. Both models use the Cubist regression algorithm (Kuhn and K. 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 first model is constructed using a subset of the genetic covariates, as determined by a variable selection algorithm called VIF regression (Lin, Foster, and Ungar 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 8, including the wide node at the top, refer to code for data preprocessing or transformation. It is a commonplace among data scientists that such "data munging" is a crucial aspect of data analysis. There is no fundamental obstacle to representing its semantics; it so happens that the relevant portion of the Data Science Ontology has not yet been developed. This situation illustrates another 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 partial results even in the face of missing annotations.

5.3. **Use cases and applications.** Our system is a first step towards an AI assistant for networked, data-driven science. We hope it will enable, or bring us closer to enabling, new technologies that boost the efficiency of data scientists. These technologies may operate at small scales, involving one or a small group of data scientists, or at large scales, spanning a broader scientific community.

At the scale of individuals, we imagine an integrated development environment (IDE) for data science that interacts with analysts at both syntactic and semantic levels. Suppose a participant in the rheumatoid arthritis DREAM Challenge fits a random forest regression, using the `randomForest` package in R. Indeed, the analysts from Section 5.2 report experimenting with random forests, among other popular methods (Kramer et al. 2014). By a simple inference within the Data Science Ontology, the IDE recognizes random forests as a tree-based ensemble method. It suggests the sister method Cubist and generates R code invoking the `Cubist` package. Depending on their expertise, the analysts may learn about

[3]We would prefer a web-based, interactive presentation, with the boxes and wires linked to descriptions from the ontology. That is, regrettably, outside the scope of this paper.

```r
library("caret")
library("VIF")
library("Cubist")

merge.p.with.template <- function(p){
  template = read.csv("RAchallenge_Q1_final_template.csv")
  template$row = 1:nrow(template)
  template = template[,c(1,3)]

  ids = data.resp$IID[is.na(y)]
  p = data.frame(ID=ids, Response.deltaDAS=p)
  p = merge(template, p)
  p = p[order(p$row), ]
  p[,c(1,3)]
}

data = readRDS("pred.rds")
resp = readRDS("resp.rds")

# non-clinical model
data.resp = merge(data, resp[c("FID", "IID", "Response.deltaDAS")])
y = data.resp$Response.deltaDAS
y.training = y[!is.na(y)]

data.resp2 = data.resp[!(names(data.resp) %in% c("Response.deltaDAS", "FID", "IID"))]
dummy = predict(dummyVars(~., data=data.resp2), newdata=data.resp2)

dummy.training = dummy[!is.na(y),]
dummy.testing = dummy[is.na(y),]

v = vif(y.training, dummy.training, dw=5, w0=5, trace=F)
dummy.training.selected = as.data.frame(dummy.training[,v$select])
dummy.testing.selected = as.data.frame(dummy.testing[,v$select])

m1 = cubist(dummy.training.selected, y.training, committees=100)
p1 = predict(m1, newdata=dummy.testing.selected)

# clinical model
dummy = data.resp[c("baselineDAS", "Drug", "Age", "Gender", "Mtx")]
dummy = predict(dummyVars(~., data=dummy), newdata=dummy)
dummy.training = dummy[!is.na(y),]
dummy.testing = dummy[is.na(y), ]

m2 = cubist(dummy.training, y.training, committees=100)
p2 = predict(m2, newdata=dummy.testing)

## create csv files
p1.df = merge.p.with.template(p1)
p2.df = merge.p.with.template(p2)

write.csv(p1.df, quote=F, row.names=F, file="clinical_and_genetic.csv")
write.csv(p2.df, quote=F, row.names=F, file="clinical_only.csv")
```

LISTING 4. R source code for two models from the Rheumatoid Arthritis DREAM Challenge

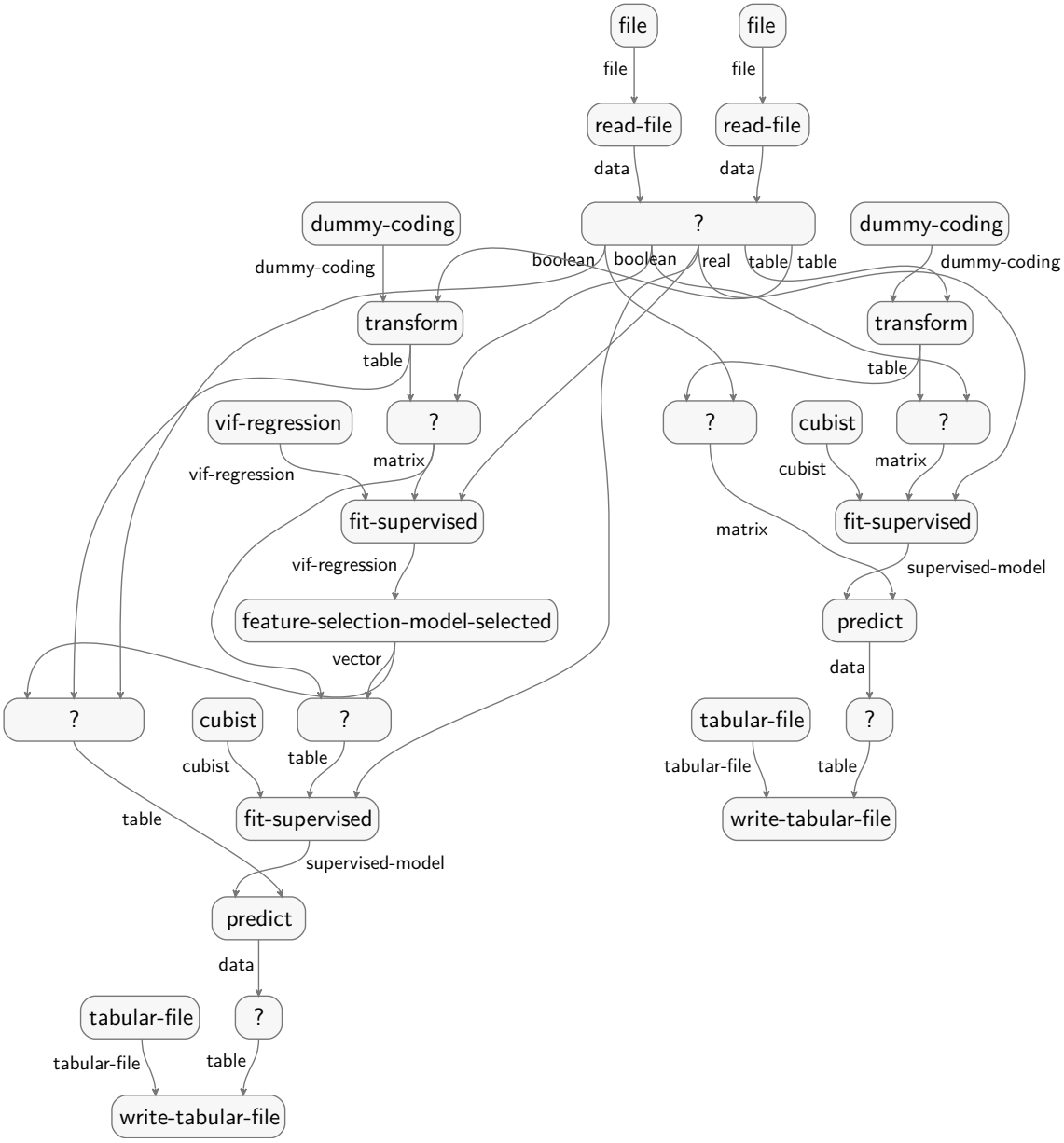


FIGURE 8. Semantic flow graph for two models from the Rheumatoid Arthritis DREAM Challenge (Listing 4)

new statistical methods or software packages implementing them. Even expert users should benefit from the possibility of more efficient experimentation.

As programmers, we are all prone to lapses of discipline in commenting our code. Poorly documented code is difficult to comprehend at a glance. To supplement the explanations written by fallible humans, an AI agent might translate our semantic flow graphs into written descriptions, via natural language generation (Gatt and Krahmer 2018). 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 emerge at the scale of online platforms for collaborative data science. Online platforms can host coordinated efforts to solve specific scientific problems, under variations of the challenge model. They may also host products of independent scientific experiments, serving as centralized repositories of papers, data, and code. In both situations, statistical meta-analysis is needed to aggregate the results of individual analyses or studies (Gurevitch et al. 2018). Today meta-analysis is a laborious and painstaking process, conducted largely by hand. We hope to lay the groundwork for more automated forms of meta-analysis.

Consider the challenge model again. Organizers typically want a panoramic view of the participants' activity. We could straightforwardly generate a summary report, given a corpus of semantic flow graphs corresponding to the submitted analyses. In challenges of scientific interest, organizers tend to be interested in more than simply what is the most predictive model. A recent DREAM Challenge, for example, aims to determine which factors affect the progression of amyotrophic lateral sclerosis (ALS), a fatal neurodegenerative disease with heterogeneous progression timelines (Kueffner et al. 2018). The organizers stipulate that submitted predictive models may use only a limited number of clinical features. Using consensus clustering (Monti et al. 2003), the organizers then aggregate feature sets across the submitted models to stratify the patients into clinically meaningful subgroups. This and other forms of meta-analysis could conceivably be simplified, or even automated, given sufficiently expressive semantic representations of the models.

6. Related work

In this paper, we extend and refine our previous work on semantic representations of data analyses (Patterson, McBurney, et al. 2017). Compared to the original work, we designed a new ontology language for modeling computer programs, along with a new ontology about data science written in this language. We also replaced our original, ad hoc procedure for creating the semantic flow graph with the semantic enrichment algorithm, which is more flexible and rests on a firmer mathematical foundation. We presented our current system as a demonstration at IJCAI 2018 (Patterson, Baldini, et al. 2018).

We have been inspired by a constellation of ideas at the intersection of artificial intelligence, program analysis, programming language theory, and category theory. We now position our work in relation to these areas.

6.1. **Knowledge representation and program analysis.**

The history of artificial intelligence is replete with interactions between knowledge representation and computer program analysis. In the late 1980s and early 1990s, automated planning and ruled-based expert systems featured in "knowledge-based program analysis" (W. L. Johnson and Soloway 1985; Harandi and Ning 1990; Biggerstaff, Mitbander, and Webster 1994). 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.

6.2. **Machine learning and program analysis.** Efforts are now underway to marry program analysis with machine learning. Inspired by an analogy between natural languages and programming languages, AI researchers are transporting successful techniques from natural language processing (NLP), such as Markov models and recurrent neural networks, to program analysis (Allamanis et al. 2018). Most program models are based on sequences of syntactic tokens, akin to sequences of words in natural language. Some models use graphical program representations, bringing them closer to our work. For example, a recent method called `inst2vec` ("instructions to vectors"), inspired by `word2vec`, fits a skip-gram embedding of program statements, using a notion of statement context which combines data flow and control flow (Ben-Nun, Jakobovits, and Hoefler 2018).

The logical and statistical paradigms of AI tend to exhibit different performance characteristics and are therefore complementary, not competitive. In the case of program analysis, our method delivers rich, precise, and human-interpretable semantics, at the expense of significant human knowledge engineering. Statistical methods scale better in terms of human effort and degrade more gracefully in the face of incomplete information, but yield semantics that are less precise and harder to interpret. In particular, embedding methods like `inst2vec`[4] create dense vector representations of statements, whose interpretations are defined only implicitly by their relation to other vectors. The vectors are useful for downstream prediction tasks but are difficult to interpret directly. Moreover, logical and statistical methods tend to understand the slippery notion of "semantics" in fundamentally different ways. Vector representations capture distributional information about how concepts are used in practice, whereas ontologies express logical constraints on how concepts are related to each other. Both kinds of information are useful and important. In the future, we hope to investigate ways of integrating logical and statistical information in semantic representations of data science code.

6.3. **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). What does the Data Science Ontology add to the landscape? While we can certainly point to differences in content—STATO focuses on classical statistics, especially hypothesis testing, whereas we are equally interested in machine learning—we prefer to make a more general point, applicable to all the ontologies that we know of.

[4] `inst2vec` also differs from our system by operating on LLVM's intermediate representation (IR), not the original code. This choice seems not to be viable for data science because Python and R do not have stable LLVM frontends, among other possible difficulties.

Every ontology is, implicitly or explicitly, designed for some purpose. The purpose of the Data Science Ontology is to define a universal language for representing data science code. Previous ontologies were designed for different purposes, and we cannot see any straightforward way to adapt them to ours. In STATO, concepts representing statistical methods can have inputs and outputs, but they are too imprecisely specified to map onto actual code, among other difficulties. PMML is a purely static format, designed for serializing fitted models. To successfully model computer programs, one must pay attention to the special structure of programs. That is what we have tried to do with the Data Science Ontology. This aspiration also drives our choice of ontology language.

6.4. **Ontology languages and programming languages.** We have designed an ontology language, Monocl, to model 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. Mathematically, it is founded on category theory and programming language theory. References are given in Section 4, where we develop the theory. We hope that our project will advance an emerging paradigm of knowledge representation based on category theory (Spivak and Kent 2012; Patterson 2017).

Due in part to the influence of the Semantic Web, the most popular paradigm for knowledge representation today is description logic, a family of computationally tractable subsystems of first-order logic (Baader, Calvanese, et al. 2007). Why have we not written the Data Science Ontology in a description logic, like the Semantic Web's OWL? We do not claim this would be impossible. The Basic Formal Ontology, expressible in OWL and underlying many biomedical ontologies, divides the world into *continuants* (persistent objects) and *occurrents* (events and processes) (Arp, Smith, and Spear 2015). We might follow STATO in modeling data analyses, or computational processes generally, as occurrents. This leads to some awkward consequences, as occurrents are ascribed a spatiotemporal structure which computer programs lack.

A more fundamental objection, independent of the Basic Formal Ontology, is that there already exists a long mathematical tradition of modeling programs, beginning nearly one hundred years ago with Alonzo Church's invention of the lambda calculus. We follow a few threads in this tradition in Section 4.1. Our work very much belongs to it. To instead ignore it, reinventing a programming model inside description logic, would be, at best, an unnecessary duplication of effort. That said, we understand the value of interoperability with existing systems. We are investigating ways to encode the Data Science Ontology in OWL, possibly with some loss of fidelity.

7. Conclusion

We have introduced an algorithm, supported by the Monocl ontology language and the Data Science Ontology, for creating semantic representations of data science code. We demonstrated the semantic enrichment algorithm on several examples, pedagogical and practical, and we supplied it with a category-theoretic mathematical foundation. We situated our project within a broader trend towards more open, networked, and machine-driven science. We also suggested possible applications to collaborative data science, at the small and large scales.

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 data science models. Our representation emphasizes computational structure, but the most

interesting applications require more. In a similar vein, scientific applications require that statistical models be connected with scientific concepts. Workers across the sciences, and especially in biomedicine, are building large ontologies of scientific concepts. Interoperation with existing domain-specific ontologies is therefore an important research direction.

Only by a concerted community effort will the vision of machine-assisted data science be realized. To that end, we have released as open source software our Python and R program analysis tools, written in their respective languages, and our semantic enrichment algorithm, written in Julia.[5] We are also crowdsourcing the further development of the Data Science Ontology.[6] We entreat the reader to join us in the effort of bringing artificial intelligence to the practice of data science.

References

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

Allamanis, Miltiadis, Earl T Barr, Premkumar Devanbu, and Charles Sutton (2018). "A survey of machine learning for big code and naturalness". *ACM Computing Surveys (CSUR)* 51.4, p. 81.

Arp, Robert, Barry Smith, and Andrew D Spear (2015). *Building ontologies with Basic Formal Ontology*. MIT Press.

Awodey, Steve (2010). *Category Theory*. 2nd ed. Oxford University Press.

Baader, Franz, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, eds. (2007). *The Description Logic Handbook: Theory, Implementation and Applications*. 2nd ed. Cambridge University Press.

Baader, Franz and Tobias Nipkow (1999). *Term Rewriting and All That*. Cambridge University Press.

Baez, John and Jason Erbele (2015). "Categories in control". *Theory and Applications of Categories* 30.24, pp. 836–881.

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

Barr, Michael and Charles Wells (1990). *Category theory for computing science*. Prentice Hall.

Ben-Nun, Tal, Alice Shoshana Jakobovits, and Torsten Hoefler (2018). *Neural Code Comprehension: A Learnable Representation of Code Semantics*. arXiv: 1806.07336.

Biggerstaff, Ted J., Bharat G. Mitbander, and Dallas E. Webster (1994). "Program understanding and the concept assignment problem". *Communications of the ACM* 37.5, pp. 72–82.

Borceux, Francis (1994). *Handbook of Categorical Algebra 3: Categories of Sheaves*. Cambridge University Press.

Cartmell, John (1978). "Generalised algebraic theories and contextual categories". PhD thesis. Oxford University.

– (1986). "Generalised algebraic theories and contextual categories". *Annals of Pure and Applied Logic* 32, pp. 209–243.

[5]All source code is available on GitHub under the Apache 2.0 license (Patterson and contributors 2018b; Patterson and contributors 2018c; Patterson and contributors 2018d).

[6]The Data Science Ontology is available on GitHub under the Creative Commons Attribution 4.0 license (Patterson and contributors 2018a).

Coecke, Bob, 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, pp. 1079–1100.

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

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

Devanbu, Prem, Ron Brachman, Peter G. Selfridge, and Bruce W. Ballard (1991). "LaSSIE: A knowledge-based software information system". *Communications of the ACM* 34.5, pp. 34–49.

DREAM Challenges (2017). *The Digital Mammography DREAM Challenge*. URL: `https://www.synapse.org/Digital_Mammography_DREAM_challenge`.

Gatt, Albert and Emiel Krahmer (2018). "Survey of the State of the Art in Natural Language Generation: Core tasks, applications and evaluation". *Journal of Artificial Intelligence Research* 61, pp. 65–170.

Goguen, Joseph (1978). *Ordered sorted algebra*. Tech. rep. 14, Semantics and Theory of Computation Series. UCLA, Computer Science Department.

Goguen, Joseph and José Meseguer (1992). "Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations". *Theoretical Computer Science* 105.2, pp. 217–273.

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

Goodman, Steven N, Daniele Fanelli, and John PA Ioannidis (2016). "What does research reproducibility mean?" *Science translational medicine* 8.341, 341ps12.

Guazzelli, Alex, Michael Zeller, Wen-Ching Lin, and Graham Williams (2009). "PMML: An open standard for sharing models". *The R Journal* 1.1, pp. 60–65.

Gurevitch, Jessica, Julia Koricheva, Shinichi Nakagawa, and Gavin Stewart (2018). "Meta-analysis and the science of research synthesis". *Nature* 555, pp. 175–182.

Guth, Dwight (2013). "A formal semantics of Python 3.3". MA thesis. University of Illinois at Urbana-Champaign.

Harandi, Mehdi T. and Jim Q. Ning (1990). "Knowledge-based program analysis". *IEEE Software* 7.1, pp. 74–81.

Hasegawa, Masahito (1997). "Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi". *International Conference on Typed Lambda Calculi and Applications*, pp. 196–213.

– (2003). "The uniformity principle on traced monoidal categories". *Electronic Notes in Theoretical Computer Science* 69, pp. 137–155.

Heunen, Chris and Jamie Vicary (2012). *Lectures on categorical quantum mechanics*. Computer Science Department, Oxford University.

Hey, Tony, Stewart Tansley, and Kristin Tolle, eds. (2009). *The fourth paradigm: Data-intensive scientific discovery*. Microsoft Research.

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

Jeffrey, Alan (1997). *Premonoidal categories and a graphical view of programs*. Tech. rep. University of Sussex, School of Cognitive and Computing Sciences.

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

Kelly, Max (1982). *Basic Concepts of Enriched Category Theory*. Vol. 64. Lecture Notes in Mathematics. Cambridge University Press.

Kramer, Eric, Bhuvan Molparia, Nathan Wineinger, and Ali Torkamani (2014). *Rheumatoid arthritis final predictions*. DOI: 10.7303/syn2491171. URL: https://www.synapse.org/#!Synapse:syn2491171.

Kueffner, Robert et al. (2018). "Stratification of amyotrophic lateral sclerosis patients: a crowdsourcing approach". DOI: 10.1101/294231. bioRxiv: 294231.

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

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

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

Leinster, Tom (2014). *Basic category theory*. Vol. 143. Cambridge University Press.

Lin, Dongyu, Dean P. Foster, and Lyle H. Ungar (2011). "VIF regression: a fast regression algorithm for large data". *Journal of the American Statistical Association* 106.493, pp. 232–247.

MacDonald, John and Laura Scull (2009). "Amalgamations of categories". *Canadian Mathematical Bulletin* 52.2, p. 273.

Marek, Kenneth et al. (2011). "The Parkinson Progression Marker Initiative (PPMI)". *Progress in Neurobiology* 95.4, pp. 629–635.

Marsden, Dan (2016). "Ambiguity and incomplete information in categorical models of language". *Quantum Physics and Logic (QPL 2016)*, pp. 95–107.

Monti, Stefano, Pablo Tamayo, Jill Mesirov, and Todd Golub (2003). "Consensus clustering: a resampling-based method for class discovery and visualization of gene expression microarray data". *Machine Learning* 52.1, pp. 91–118.

Morandat, Floréal, Brandon Hill, Leo Osvald, and Jan Vitek (2012). "Evaluating the design of the R language: Objects and functions for data analysis". *European Conference on Object-Oriented Programming*, pp. 104–131.

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

Munafò, Marcus R. et al. (2017). "A manifesto for reproducible science". *Nature Human Behaviour* 1.0021.

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

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

Parker, Hilary, David Robinson, Stephanie Hicks, and Roger Peng (2015). *explainr package*. GitHub. URL: https://github.com/hilaryparker/explainr.

Pashler, Harold and Eric-Jan Wagenmakers (2012). "Editors' introduction to the special section on replicability in psychological science: A crisis of confidence?" *Perspectives on Psychological Science* 7.6, pp. 528–530.

Patterson, Evan (2017). *Knowledge Representation in Bicategories of Relations*. arXiv: 1706.00526.

Patterson, Evan, Ioana Baldini, Aleksandra Mojsilović, and Kush R. Varshney (2018). "Semantic representation of data science programs". *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, pp. 5847–5849. DOI: 10.24963/ijcai.2018/858.

Patterson, Evan and contributors (2018a). *Data Science Ontology*. DOI: 10.5281/zenodo.1401677. URL: https://github.com/ibm/datascienceontology.

– (2018b). *Flow graphs for Python*. DOI: 10.5281/zenodo.1401683. URL: https://github.com/ibm/pyflowgraph.
– (2018c). *Flow graphs for R*. DOI: 10.5281/zenodo.1401685. URL: https://github.com/ibm/rflowgraph.
– (2018d). *Semantic flow graphs*. DOI: 10.5281/zenodo.1401687. URL: https://github.com/ibm/semanticflowgraph.
Patterson, Evan, Robert McBurney, et al. (2017). "Dataflow representation of data analyses: Towards a platform for collaborative data science". *IBM Journal of Research and Development* 61.6, 9:1–9:13. DOI: 10.1147/JRD.2017.2736278.
Pierce, Benjamin C. (1991). *Basic Category Theory for Computer Scientists*. MIT Press.
Piwowar, Heather et al. (2018). "The State of OA: A large-scale analysis of the prevalence and impact of Open Access articles". *PeerJ* 6, e4375.
Power, John and Edmund Robinson (1997). "Premonoidal categories and notions of computation". *Mathematical Structures in Computer Science* 7.5, pp. 453–468.
Reynolds, John C. (1980). "Using category theory to design implicit conversions and generic operators". *International Workshop on Semantics-Directed Compiler Generation*, pp. 211–258.
Riehl, Emily (2014). *Categorical Homotopy Theory*. Cambridge University Press.
– (2016). *Category Theory in Context*. Dover.
Saez-Rodriguez, Julio et al. (2016). "Crowdsourcing biomedical research: leveraging communities as innovation engines". *Nature Reviews Genetics* 17.8, pp. 470–486.
Selinger, Peter (1999). "Categorical structure of asynchrony". *Electronic Notes in Theoretical Computer Science* 20, pp. 158–181.
– (2010). "A survey of graphical languages for monoidal categories". *New Structures for Physics*. Springer, pp. 289–355.
– (2013). *Lecture notes on the lambda calculus*. arXiv: 0804.3434.
Sieberts, Solveig K. et al. (2016). "Crowdsourced assessment of common genetic contribution to predicting anti-TNF treatment response in rheumatoid arthritis". *Nature Communications* 7.
Spivak, David (2014). *Category Theory for the Sciences*. MIT Press.
Spivak, David and Robert Kent (2012). "Ologs: a categorical framework for knowledge representation". *PLoS One* 7.1.
Stolovitzky, Gustavo, Don Monroe, and Andrea Califano (2007). "Dialogue on Reverse-Engineering Assessment and Methods". *Annals of the New York Academy of Sciences* 1115.1, pp. 1–22.
Wang, Yong and Ian H. Witten (1997). "Inducing model trees for continuous classes". *Proceedings of the Ninth European Conference on Machine Learning*, pp. 128–137.
Welty, Christopher A. (2007). "Software Engineering". *The Description Logic Handbook: Theory, Implementation and Applications*. Ed. by Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. 2nd ed. Cambridge University Press. Chap. 11, pp. 402–416.
Wickham, Hadley (2014). *Advanced R*. CRC Press.
Wilkinson, Mark D. et al. (2016). "The FAIR Guiding Principles for scientific data management and stewardship". *Scientific data* 3.
Wills, Linda M. (1992). *Automated program recognition by graph parsing*. Tech. rep. MIT Artificial Intelligence Laboratory.