Machine Learning (2021) 110:779–799
https://doi.org/10.1007/s10994-021-05950-7

AUTOMAT[R]IX: learning simple matrix pipelines

Lidia Contreras-Ochando[1] · Cèsar Ferri[1] · José Hernández-Orallo[1]

Received: 15 May 2020 / Revised: 3 November 2020 / Accepted: 23 January 2021 / Published online: 13 April 2021

Abstract
Matrices are a very common way of representing and working with data in data science and artificial intelligence. Writing a small snippet of code to make a simple matrix transformation is frequently frustrating, especially for those people without an extensive programming expertise. We present AUTOMAT[R]IX, a system that is able to induce R program snippets from a single (and possibly partial) matrix transformation example provided by the user. Our learning algorithm is able to induce the correct matrix pipeline snippet by composing primitives from a library. Because of the intractable search space—exponential on the size of the library and the number of primitives to be combined in the snippet, we speed up the process with (1) a typed system that excludes all combinations of primitives with inconsistent mapping between input and output matrix dimensions, and (2) a probabilistic model to estimate the probability of each sequence of primitives from their frequency of use and a text hint provided by the user. We validate AUTOMAT[R]IX with a set of real programming queries involving matrices from Stack Overflow, showing that we can learn the transformations efficiently, from just one partial example.

Keywords Automating data science · Inductive programming · Program synthesis

1 Introduction

Many areas in artificial intelligence, from data pre-processing to optimisation algorithms, from image transformation to the visualisation of tables and results, require common operations using matrices. This is especially the case in machine learning and data science, where programming languages, such as R or Python, are used to manipulate data. While

Editors: Nikos Katzouris, Alexander Artikis, Luc De Raedt, Arturd'Avila Garcez, Ute Schmid, Sebastijan Dumančić, Jay Pujara.

Lidia Contreras-Ochando
liconoc@upv.es

Cèsar Ferri
cferri@dsic.upv.es

José Hernández-Orallo
jorallo@upv.es

[1] Valencian Research Institute for Artificial Intelligence (vrAIn), Universitat Politècnica de València, Valencia, Spain

NA	0.30	0.50	NA	NA	NA	NA
NA	NA	NA	0.90	NA	NA	0.40
NA	NA	NA	NA	NA	NA	NA
NA	NA	NA	NA	0.60	NA	NA
NA	NA	NA	NA	NA	NA	NA

(a) Matrix A with some NA values.

1	2
1	3
2	4
4	5
2	7

(b) Position (row, column) of non-NA values in A.

Fig. 1 Example of data transformation using matrices. The snippet must transform the matrix on the left into the matrix on the right. Can you code it?

libraries and hand-crafted algorithms usually capture the core data pipelines, there is an extensive use of glue code, small snippets that perform simple transformation between the output of one module to the input of the following one. Sometimes, these transformations must be done by experts without a profound programming knowledge, with frustrating results (Jenkins 2002).

Matrices are a very common way of working with data. Data-frames, tables, spreadsheets, bi-dimensional arrays and tensors are similar structures that can be assimilated to matrices. Matrix algebra can be applied to transform data or extract a variety of useful information. It is a common strategy for programmers, AI experts and data scientists to use an example of a problem and use it to solve the desired transformation or detect where it fails. A few examples are used to 'verify' whether the matrix snippet is doing the right processing, before applying the transformation to other examples.

For instance, consider an AI expert or a data scientist who wants to extract the positions of the non-empty values in the data matrix shown in Fig. 1a. She may just figure out the output she is expecting as a result of the desired transformation (represented in Table 1b), which may also be used to check the snippet once it is written. Interestingly, as happens with humans, having both the input and output matrices could be sufficient for *an automated system* to learn this transformation smoothly. This is what we do in this paper.

Common strategies to solve this problem by hand would be to program a snippet using a traditional loop, think of a more algebraic function that avoids the loop or simply check Stack Overflow[1] to find an elegant transformation. Instead, we could rely on a system that takes Table 1a, b as inputs, and generate an elegant code snippet in the programming language *R* such as `which(!is.na(A), arr.ind=TRUE)`? Note that in this example there is no similarity whatsoever between input and output. The input contains real numbers and `NA`s, and the output only has integer numbers, none of them in common with the input. Also, the dimension of the input matrix is 5×7, while the output matrix has dimension 5×2, where the same number of rows is just a coincidence. Is this problem solvable at all? And, to make it more challenging and realistic, what if we only give some of the rows (or even a few cells) of the solution?

Generating code from input-output pairs falls under the area of programming by example (PbE) in program synthesis, and more generally inductive programming (Gulwani et al. 2015). From the perspective of inductive programming, matrix transformation has several characteristics that make it more feasible than other problems: only one data structure is considered (matrices), we can use the matrix dimension as a constraint to restrict the search, and functional programming (with higher-order functions such as `apply`) is

[1] https://stackoverflow.com/.

particularly appropriate here. A first key insight to tackle our problem is that very interesting and complex transformations can be obtained by composing very few primitives. A second observation is that the primitives used in matrix pipelines usually have very low arity, with many of them having just one argument. A third consideration is that the input and output matrices represent just *one*—albeit rich—example: the information to exclude the infinitely-many alternative transformations must come from the (partial) content of the matrices and a strong simplicity prior. This makes other approaches that require significant amounts of data unfeasible and suggests a learning approach that is based on a compositional search.

In this paper, we present a system that is able to learn simple matrix pipelines from: (1) *one* input data matrix, (2) *one* partial output matrix filled by the user representing the desired transformation, and (3) *optionally*, a short description or hint in natural language of the desired transformation. The system works with a library of b primitives to combine into a snippet of at most d primitives. Because the combinatorics of primitives and operations is in the order of b^d, we use several strategies to reduce the search space. First, we use the characteristics of the input and output matrices, and the primitives themselves, in the form of constraints. Basically, our system checks that the sequence of compositions is consistent with the dimensions. Second, we estimate the a priori probabilities of the primitives according to how frequent they are used on Github. Third, when available, we can use small hints in natural language given by the user, such as a short text of the form: "positions of non-empty values" (see Fig. 1). This helps us estimate the conditional probabilities for primitive sequences. We use all this information for a tree-search procedure that re-estimates branching candidates dynamically.

The main contributions of this paper[2] are: (1) the definition of the matrix transformation problem from one single example, and (2) a learning algorithm guided by a tree-based search to learn these transformations efficiently. Besides, we also provide: (3) a collection of matrix pipeline transformations, matransf , some of them with textual hints, organised as a benchmark repository for the community, and (4) our system AUTOMAT`[R]`IX, validated with a battery of experiments on matransf . Code and data are publicly available for reproducibility.

The following section contains a summary of relevant work in programming by example and inductive programming, and how much this has impacted on the automation of some repetitive tasks in data manipulation and data science. Section 3 defines the problem that we address in this paper. Section 4 gives details of our approach and how it leverages novel and traditional ideas in artificial intelligence to solve this new problem effectively and efficiently. Section 5 includes experiments with artificial and real data. Finally, Sect. 6 closes the paper with the applicability of the system and the future work.

2 Related work

Teaching a machine new behaviours based on examples or demonstrations falls under the field of Programming by Example (PbE) (Lieberman 2001; Raza et al. 2014). PbE is both a subfield of program synthesis and inductive programming, in which the system learns

[2] This paper fully develops some preliminary and exploratory ideas we presented in Contreras et al. (2020a). Here we present a new algorithm based on a probabilistic model and textual hints, as well as an extensive experimentation.

programs that match examples provided by the user. Normally, only a few examples should be required to induce the solution. PbE with Domain-Specific Languages (DSLs) has been successfully applied for data transformation (see Wu et al. 2012; Cropper et al. 2015; Parisotto et al. 2016; He et al. 2018). These systems are mostly focused on mapping one or more cells into a single *cell* using string transformations. For example, *FlashFill* (Gulwani 2011) is a tool included in Microsoft Excel that uses regular expressions, conditionals and loops on spreadsheet tables to make syntactic transformations of strings. `BK-ADAPT` (Contreras et al. 2020b, c) is a web-based tool for the automation of data format transformation that uses inductive programming with a dynamic background knowledge generated by a machine learning meta-model to select the domain and the primitives from several descriptive features of the data wrangling problem. The system allows users to provide a set of inputs and one or more examples of outputs, in such a way that the rest of examples are automatically transformed by the tool. Trifacta *Wrangler* (Kandel et al. 2011) generates a list of transformations (merge, move, pivot, split, etc.) also ranked and inferred automatically from the user's input. In Santolucito et al. (2019) the authors use CVC4's Syntax-guided synthesis (SyGuS) algorithm (Reynolds and Tinelli 2017) for Javascript live coding.The most related work in this area is TaCLe (Paramonov et al. 2017; Kolb et al. 2017). TaCLe reconstructs the formulae used in a data spreadsheet based on a comma-separated file. The system is able of detect the formulas applied to rows, columns or cells that have generated the values of another cells, based on constraint templates. However, TaCLe only works by looking cell by cell but not if the whole table has been transformed into a different table or value.

Despite everything, more and more transformations in AI require taking data represented as a matrix (e.g., an image, a dataset, a weight matrix, a result table, etc.) and apply a few primitives (e.g., a convolution, a pivoting, a thresholding, a column-wise mean, etc.). In Wang et al. (2017), the authors present an inductive framework called Blaze. The approach employs the abstract semantics of a DSL (the domain-specific language provided by a domain expert) and a set of examples to find a program whose abstract behaviour covers the examples. The authors used Blaze to build synthesisers for string and matrix transformations. Although the results of their experiments are positive, the number of functions included in the DSL is limited (less than 10 functions). This is motivated by the huge search space, but their very short number of functions dramatically reduces the number of real examples that can be solved by the system. DSLs do not have to be less expressive than generic languages, but for some applications it is reasonable to exclude everything that is not needed, making them usually less expressive.

In contrast, we do not introduce any DSL, but develop a system that induces transformations in a general programming language. A generic language can contain syntactic sugar and present many alternative ways to do the same thing, increasing the search space, if all functions and characteristics of the language are available. We have used the R language,[3] having in mind that the final goal is to use it to solve real problem examples in that language. In our case, we start with a reduced core version of the language, only allowing unary function compositions, and the user is the one that chooses the functions. Accordingly, redundant functions or syntactic sugar can be reduced by this core part of the language and an appropriate choice of the background knowledge. As an advantage, the snippets that are found by our algorithm

[3] https://www.r-project.org/.

will be directly applicable in the language and understood easily by anyone minimally familiar with R.

Finally, a different and interesting approach is Generalised Planning (Segovia-Aguas et al. 2019), which is able to generate a single solution using one or more examples. In this type of problems, a plan is a sequence of actions that induces a state sequence to reach a goal condition (for instance, a termination instruction `end`) departing from an initial state. In this sense, a matrix transformation could also be seen as a planning problem.

In this work: (1) We can solve a much wider range of problems than systems such as Wang et al. (2017); and (2) by increasing the number of primitives we need to reduce the search space by prioritising some over others. The ideas explained in this paper are easily extensible to other languages such as Python or Matlab. The automation of small but convoluted snippets in these languages could represent an important reduction of the time needed in many AI and data science projects.

3 Problem definition

We assume there is a set of operations that can be combined and applied to a matrix A in order to obtain a result S. The operations are primitives or functions in some specific language (in our case, R) and a human needs some assistance to generate the code from a single example. This is the setting that serves as problem formulation:

1. We are given an input matrix A: a finite real matrix of size $m \times n$ $(m, n > 0)$.
2. We are given a partially filled matrix B: a finite real matrix of size $m' \times n'$ $(m', n' > 0)$ where only some elements are filled and the rest are empty (we will use the notation '$\cdot$').
3. Optionally, we are given some textual hint or short description T in natural language: this is provided by the user, describing the problem to solve.
4. We look for a function $\hat{f}$ such that $\hat{f}(A) = S$, where S is a finite real matrix of size $m' \times n'$, such that for every non-empty $b_{ij} \in B$ the corresponding $s_{ij} \in S$ matches, i.e., $b_{ij} = s_{ij}$.
5. We produce the function $\hat{f}$, expressed as a composition of matrix operations in a given programming language.

As additional criterion we will consider that the representation of $\hat{f}$ in the programming language should be as short as possible in number of functions combined , and we will also allow for some precision error ϵ (so that we relax item 4 above with $|b_{ij} - s_{ij}| \le \epsilon$, instead of $b_{ij} = s_{ij}$). We use the notation $\hat{f}(A) \vDash_\epsilon B$ to represent this, and say that the transformation *covers B*.

As a basic example, consider the matrices A and B:

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 4 & 2 & 6 \\ 3 & 8 & 7 \end{bmatrix}$$

$$B = \begin{bmatrix} 8 & 13 & \cdot \end{bmatrix}$$

where A is the input matrix and B the partially-filled output. We try to find $\hat{f}$ such that $\hat{f}(A) \vDash_\epsilon B$. In this case the function `colSums` in R, which adds the values columnwise, gives the following matrix S that covers B.

$$S = \begin{bmatrix} 8 & 13 & 18 \end{bmatrix}$$

Note that we look for a system that: (1) works with only *one* input matrix and only *one* partially-filled output matrix, and nothing more (the textual hint is optional), (2) automatically synthesises the composition of primitives in the base programming language that solves the above problem, and (3) returns the complete transformed matrix and the synthesised code.

As far as we know from the related work seen in the previous section, no other approach is able to solve this problem using only one or few cells of the solution matrix.

4 Method

We emphasise a series of characteristics of this problem: (1) the combination of a few functional primitives can achieve very complex transformations, (2) the arity of the primitives is usually low (one in many cases), so the snippet becomes a pipeline, where the output of one primitive becomes the input of the next one, (3) we work from one example and (4) we want the shortest transformation with the given primitives, as built-in primitives usually lead to more efficient transformations. All these characteristics suggest that the problem can be addressed by exploring all combinations of primitive sequences, by using a strong simplicity bias (the number of primitives used). This strategy is common in other inductive programming scenarios (Mitchell et al. 1991; Katayama 2005; Menon et al. 2013; Mitchell et al. 2018) but it must always be coupled with some constraints (e.g., types, schemata, etc.) or strong heuristics. In our case, we will use the dimensions of the matrices as the main constraint for reducing the combinatorial explosion, as well as some priors about the frequency of each primitive and, optionally, some posteriors using text hints in order to guide a tree-based search where each combination of functions will be sorted and selected based on its assigned probability.

4.1 Dimensional constraints

We consider the background knowledge as a set of primitives G. This number of primitives $|G|$ taken into account for the search is known as the *breadth* (b) of the problem, while the minimum number of such primitives that have to be combined in one solution is known as *depth* (d). Clearly, both depth and breadth highly influence the hardness of the problem, in a way that is usually exponential, $O(b^d)$ (Ferri-Ramírez et al. (2001)) affecting the time and resources needed to find the right solution. This expression is exact if we consider unary primitives, so that solutions become matrix operation *pipelines*, i.e., a string of primitive $c = g_1 g_2 \cdots g_d$.

The first optimisation to this search comes from the constraints about the dimensions of the primitives and the input/output matrices. For each matrix primitive g we take into account the dimension of the input and output at any point of the composition, and also some other constraints about minimum dimension (for instance, calculating correlations, function `cor`, requires at least two rows, i.e., $m > 1$). More formally, for each primitive g we define a tuple $\langle m_{min}, n_{min}, \tau \rangle$ where m_{min} and n_{min} are the minimum number of rows and columns (respectively) for the input (by default $m_{min} = 1$ and $n_{min} = 1$), and $\tau : \mathbb{N}^2 \to \mathbb{N}^2$ is a *type function*, which maps the dimension of the input matrix to the dimension of the output matrix. For instance, for g = `colSums`, $m_{min} = 1$ and $n_{min} = 1$ as you need at least one column and row for the primitive to work. $\tau(m,n) = (1,n)$, because g takes a matrix of size $m \times n$ and returns a matrix of size $1 \times n$. Similarly, for g = `cor`, $m_{min} = 2$ and $n_{min} = 1$, as we need at least two rows to calculate a correlation. $\tau(m,n) = (n,n)$, because g takes a matrix of size $m \times n$ and returns a matrix of size $n \times n$.

4.2 Probabilistic model

Now, during exploration we can consider that not all primitives, and consistent sequences of primitives, are equally likely. Given our inputs: the hint text T, an input matrix A and partially filled output matrix B, we estimate the probability of a sequence of primitives, as follows:

$$p(g_1 g_2 \dots g_d | T, A, B) = \prod_{i=1}^{d} p(g_i | g_{i-1} g_{i-2} \dots g_1, T, A, B) \tag{1}$$

The expression on the right can be used partially as we include candidates for primitives during the search procedure.

In order to estimate this probability we consider the a priori probability $p(g)$ for each g, which we can derive from the frequency of use of the primitives in the library, as we will see in the following section. When T is available, we will use a frequency model that compares TF-IDF values (Salton and Buckley 1988) of the primitive in its R help documentation and the TF-IDF values in T. This model produces the conditional probability $p_0(g|T)$ $\forall g, T$. We combine these probabilities as follows:

$$p(g|T, A, B) = \gamma p(g) + (1 - \gamma) p_0(g|T) \tag{2}$$

with $\gamma \in [0, 1]$. Clearly, if T is not available $\gamma = 1$. Basically, γ gauges how much relevance we give to the primitive prior (valid for all problems) over the relevance of the hint given by the user.

Finally, we have the intuition that the probability of a primitive may depend on the previous primitives. In this paper, we explore a very simple model for sequential dependencies, by limiting the effect to trigrams and exploring whether the same primitive is repeated in any of the three previous operations. We use a parameter $\beta \in [0, 1]$, where high β values imply that repetitions are more penalised. More formally,

$$p(g_i | g_{i-1} \dots g_1, T, A, B) = \beta p(g|T, A, B) + (1 - \beta) p(g_i | g_{i-1} g_{i-2} g_{i-3}, T, A, B) \tag{3}$$

and the repetition part is simply:

$$\begin{aligned} p(g_i | g_{i-1} g_{i-2} g_{i-3}, T, A, B) &= 0 && \text{if } g_i \in \{g_{i-1}, g_{i-2}, g_{i-1}\} \\ &= p(g|T, A, B) && \text{otherwise} \end{aligned}$$

which means that if the primitive is repeated in the three previous operations, then the value is 0, becoming more relevant the lower β is. We will explore whether this repetition intuition has an important effect on the results.

4.3 Algorithm

With Eq. 1 using the expansion of Eq. 3, we can recalculate the probability after any primitive is introduced in a tree-based search. Note that every combination of primitives whose sizes do not match have probability 0 and are ruled out. However, for those that are valid, can we use extra heuristics to determine whether we are getting closer to the solution? One idea is to check whether we are approaching the final size of matrix. For instance, if the result has size $(m, 1)$ and an operation takes the dimension to exactly that, it may be more promising than another that leads to a size $(2m, n^3)$ (which would require further operations to be reduced, at least in size).

In particular, each node in the tree where functions $g_1 g_2 \dots g_d$ have been introduced will be assigned with the following priority[4]:

$$p^*(g_1 g_2 \dots g_d) = (1 + \alpha m) \prod_{i=1}^{d} p(g_i | g_{i-1} g_{i-2} \dots g_1, T, A, B) \tag{4}$$

where $m = 1$ if the final dimension match the size of the output matrix B, i.e., $\tau_d(\tau_{d-1}(\dots \tau_1(m_{input}, n_{input}) \dots)) = (m_{output}, n_{output})$, and $m = 0$ otherwise. For those ongoing transformations where the output size matches (even if the values are not yet equal) the priority will be higher than if the dimensions do not match. In other words, it is just an estimate of whether "we may already be there". The parameter $\alpha \in [0, 1]$ simply gives weight to this. If $\alpha = 1$ then the priority of a situation with the final size is doubled over another situation where the final size does not match. For $\alpha = 0$ the priority is not affected by the final size matching or not.

Now we can use Eq. 4 in the tree-based search. The search algorithm works as follows (see Algorithm 1):

1. The system can be configured to use a set of primitive functions (G), for each of them including the minimum values for the size of the input (m_{min},n_{min}) and the type function τ.
2. For each particular problem to solve, we take the input matrix A and the partially filled matrix B. Optionally, we take a text hint T describing the problem to solve.
3. Being d_{max} the maximum number of functions allowed in the solution, the procedure evaluates sequences of primitives $g_1 g_2 \dots g_d$, with $0 \le d \le d_{max}$ where each $g_i \in G$. The parameter s_{max} determines the maximum number of solutions (when reached, the algorithm stops).
4. We start with a set of candidate solutions $C = G$.
5. We extract $c = g_1 g_2 \dots g_d \in C$ such that $p^*(c)$ is highest. We use τ on A and all primitives in c to see if the combination is feasible according to the dimension constraints and, in that case, we calculate the output size. If the dimension of any composition in c does not match, we delete the node from C. If the dimension of the output matches the dimension of B, we effectively execute the combination on A, i.e., $c(A)$, and check whether the result covers S, as defined in the previous section. In the positive case, we add c as a solution, and we delete it from C. In any other case, if $d < d_{max}$ we expand c

[4] This is an unnormalised value of the expectation that a partial primitive sequence could lead to the final solution, as used in a tree-based search. Because of the α correction, it may even be greater than 1, so it is not a probability.

into $c \cdot g_{d+1}$ with all $g_{d+1} \in G$. We calculate p^* for each of them and add them to C. We remove c from C.

6. We repeat the procedure in 5 above until s_{max} is reached or C is exhausted.

As mentioned in the problem formulation we allow for some small precision error ϵ between the cells in S (generated by $\hat{f}$) and the cells that are present in B (and are generated by $\hat{f}$).

Algorithm 1
AUTOMAT[R]IX : Selecting matrix operations by example

```
Require: $A[m \times n]$                       // Input matrix
Require: $B[m' \times n']$                     // Output matrix partially filled
Require: $G$                                   // Primitives, defined as tuples $\langle g, m_{min}, n_{min}, \tau \rangle$
Require: $d_{max}$                             // Max primitives allowed in each solution
Require: $s_{max}$                             // Number of solutions allowed
Ensure: Find a matrix $S \models_\epsilon B$
  $R \leftarrow \emptyset$                     // Initialise set of solutions
  $C \leftarrow G$                             // Initialise candidate set C with list of functions
  while $|R| \leq s_{max}$ and $C \neq \emptyset$ do
    $c \leftarrow argmax_{c \in C}\ p^*(c)$    // Select the element c with the highest priority $p^*$
    $d \leftarrow |c|$                         // Number of functions in c
    $valid \leftarrow$ True
    $\langle m_{input}, n_{input} \rangle \leftarrow \tau(m, n)$
    for $i \leftarrow 1, d$ do
      $\langle g, m_{min}, n_{min}, \tau \rangle \leftarrow c[i]$   // Extract the primitive and its type function
      if $m_{input} < m_{min}$ or $n_{input} < n_{min}$ then
        $valid \leftarrow$ False
        $C \leftarrow C \backslash c$          // Remove c from the set C
        $break$
      end if
      $\langle m_{input}, n_{input} \rangle \leftarrow \tau(m_{input}, n_{input})$   // Apply the type function $\tau$ to $m_{input}, n_{input}$
    end for
    if $valid$ and $m_{input} = m'$ and $n_{input} = n'$ then
      $S \leftarrow apply(c, A)$               // Run the sequence of functions in c over A
      if $S \models_\epsilon B$ then           // Solution found
        $R \leftarrow R \cup c$                // Add c to the set of solutions
      else
        if $d < d_{max}$ then                  // Expand C
          for $g \in G$ do
            $C \leftarrow c \circ g$           // Create new combination and add it to C
          end for
        end if
      end if
      $C \leftarrow C \backslash c$            // Remove c from the set C
    end if
  end while
  return $R$
```

4.4 Use of text hints

In some cases the user may provide a few words describing what she wants to do. This can be very helpful to give more relevance to those primitives that may be involved in the solution. For instance, if we consider a problem like "compute the correlation of a matrix", the primitive *cor* will probably appear in the solution. In our model, this is what we denoted $p_0(g|T)$. We now explain how we estimate this value.

First, we consider the set of primitives G and, for each of them, we download the text description from the corresponding R package *help* documentation. For instance `help("det")` gives the description for the function `det` as follows: "det calculates

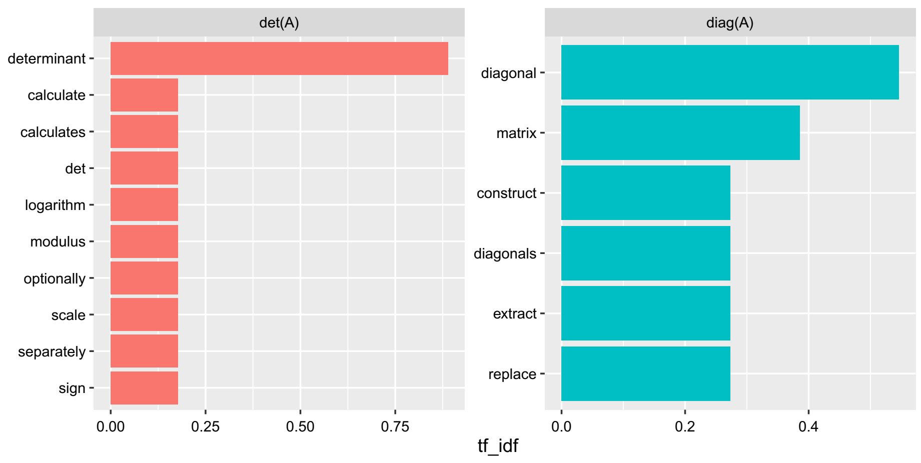

Fig. 2 TF-IDF values for two R primitives extracted from the R help documentation

the determinant of a matrix. determinant is a generic function that returns separately the modulus of the determinant, optionally on the logarithm scale, and the sign of the determinant". In the same way, the description of `diag` is: "Extract or replace the diagonal of a matrix, or construct a diagonal matrix". Each of these help texts H_g is converted into an array by applying a bag-of-words transformation, after removing the useless words (included in a list of stop words) and performing a stemming conversion (reducing inflected words to their word stem).

Secondly, given a short description T of the task we want to solve, we also apply the bag-of-words transformation, remove the stop words and do stemming. Now we have the processed text chunks H_g for each $g \in G$ and the processed text chunk T. We extract the vocabulary V from all these text chunks.

Thirdly, we apply the TF-IDF conversion (Salton and Buckley 1988) to all vectors H_g and T using the same vocabulary V. TF-IDF gives more relevance to more informative words. This leads to a word vector $\mathbf{h}_g$ for each $g \in G$ and a word vector $\mathbf{t}$. As an example, in Fig. 2 we can see the frequent terms for these two functions, as represented by their TF-IDF values. For instance, for the function `det`, it is clear that when the word *determinant* (and its steammed form *determin*) appears in a text hint, the function must have a higher probability of being required for the solution compared with other functions, such as the `diag` function.

Finally, for each g we calculate the cosine similarity $s(H_g, T)$ between $\mathbf{h}_g$ and $\mathbf{t}$. We normalise the $|G|$ similarities to sum up to one as follows:

$$p_0(g|T) = \frac{s(H_g, T)}{\sum_{g \in G} s(H_g, T)}$$

This estimate is used for Eq. 2.

Table 1 Top six R functions most used on GitHub

	Function	$p(g)$		Function	$p(g)$
1	`length`	0.327975648	4	`max`	0.055495595
2	`nrow`	0.088785612	5	`mean`	0.046439995
3	`is.na`	0.082437026	6	`cbind`	0.045872954

5 Experiments

We have implemented AUTOMAT[R]IX for R, a language and environment for statistical computing, data science and graphical representations. R operates on named data structures (vectors, matrices, data frames, etc.). In our case, we work with those functions such that input and output are data-related structures (matrices, vectors, etc.). These include primitives that extract characteristics of a matrix, such as number of rows or maximum value, or apply operations to the values, for instance bind columns, calculate the mean or compute a correlation matrix. More specifically, we take 34 R functions[5] related to matrices from the *base*,[6] *stats*[7] and *Matrix*[8] packages included in R. The experiments shown in this section aim to answer the following questions:

Q1 How much impact on success do the a priori probabilities have?
Q2 Are both α and β key to obtain better and faster results? How much?
Q3 Is the user text helpful and how much relevance (γ) should we give to it?

In order to answer these questions, in this section we describe how we obtain $p(g)$, the a priori distribution for the primitives, and how much impact has in the results; we describe the text models that lead to $p_0(g|T)$, we perform an ablation study to obtain the best values for parameters α, β and γ and then, we evaluate the algorithm with real problems taken from Stack Overflow.

For replicability and to encourage future research, all the matrix transformations used here (the matransf repository) and the code for AUTOMAT[R]IX are published on: https://github.com/liconoc/ProgramSynthesis-Matrix.

5.1 A priori Probabilities: *p*(*g*)

To answer question **Q1** and calculate the a priori distribution for the primitives we use the "Top 2000 most used R functions on GitHub" dataset, available for download on GitHub.[9] We reduce the 2000 functions to a subset containing only the functions included in our library G. When a function is duplicated in different packages we take one of them following this package order: *base*, *stats*, *Matrix*. Table 1 shows the six most used R functions from those functions included in G.

[5] See the complete list of functions in "Appendix 1".
[6] https://stat.ethz.ch/R-manual/R-devel/library/base.
[7] https://stat.ethz.ch/R-manual/R-devel/library/stats.
[8] https://stat.ethz.ch/R-manual/R-devel/library/Matrix.
[9] Top 2000 R functions: http://shorturl.at/pDFRZ.

Table 2 Results for the synthetic examples

Strategy	Accuracy	Generated	Explored	Time
Uniform	0.47 ± 0.51	5610 ± 3543	168 ± 104	59.54 ± 46.78
Prior	$\mathbf{0.65 \pm 0.49}$	$\mathbf{790 \pm 1266}$	$\mathbf{24 \pm 38}$	$\mathbf{3.69 \pm 7.01}$

Average of generated and explored nodes, accuracy (percentage of correct solutions) and time in seconds, for the uniform and prior strategies. Experiments are performed with $d_{max} = 4$ and $s_{max} = 1$. The timeout is set to $60s$. Best results are highlighted in bold

Being n_g the absolute frequency of use for the function g, we calculate the a priori probability as follows:

$$p(g) = \frac{n_g}{\sum_{g \in G} n_g}$$

Benchmark We will test whether the use of $p(g)$ as a very straightforward a priori probability is already useful in order to reduce the search time and space. For this, we first have tested the system with synthetic data. We have generated 10 random real matrices of different dimensions $m \times n$ where $m, n \in (2, 10)$. These matrices are filled with numeric values following a uniform distribution between 0 and 100. For each of these 10 matrices A we generate 10 transformations of depth $d = 1..4$ each using the functions from G according to their a priori probability (explained in Sect. 5). Finally, for each of the $10 \times 10 \times 4 = 400$ matrices S we generate a matrix B where we replace a percentage uniformly chosen between 60% and 80% of the cells by empty values. In total we have 400 pairs of matrices A, B to test the algorithm with different numbers of operations.

Results As we work with artificial examples with no text hints we cannot use $p_0(g|T)$ for these experiments. So, in this case we have tested the strategy using the a priori probabilities for $p(g)$ compared with a uniform baseline, assuming $p(g)$ uniform (so it is actually a breadth-first strategy). In both cases, we use $\gamma = 1$, $\alpha = \beta = 0$, $d_{max} = 4$, $s_{max} = 1$ (we only need to find one solution) and a timeout of 60 s.[10] Table 2 shows the results in accuracy (percentage of correct solutions found, i.e. $\hat{f}(A) = S$), average of the number of nodes generated and actually explored, and time in seconds. From these results we can clearly see that the use of probabilities following the real use of the functions from GitHub has a great impact on accuracy and even more on the size of the space explored, reducing drastically the time needed to find one solution.

5.1.1 Including text hints: $p_0(g|T)$

Benchmark Now, we want to test the algorithm also including $p_0(g|T)$, i.e., using text hints provided by the user. To do this we need to find real matrix transformation problems. In this case, we have used questions and answers collected with the Stack Overflow API using the following parameters: `tagged="R"`, `title="matrix"` and `is_accepted_answer=1`. With these parameters we guarantee that (1) the posts

[10] Note that as B is partial, each problem can be consistent with many transformations, so the first solution may not be what the user expects (in our case the one we used to generate the example), i.e. $\hat{f}(A) \models_\epsilon B$, while $\hat{f}(A) \neq S$. In those cases more non-empty cells would be required to disambiguate the right solution, which may need more primitives.

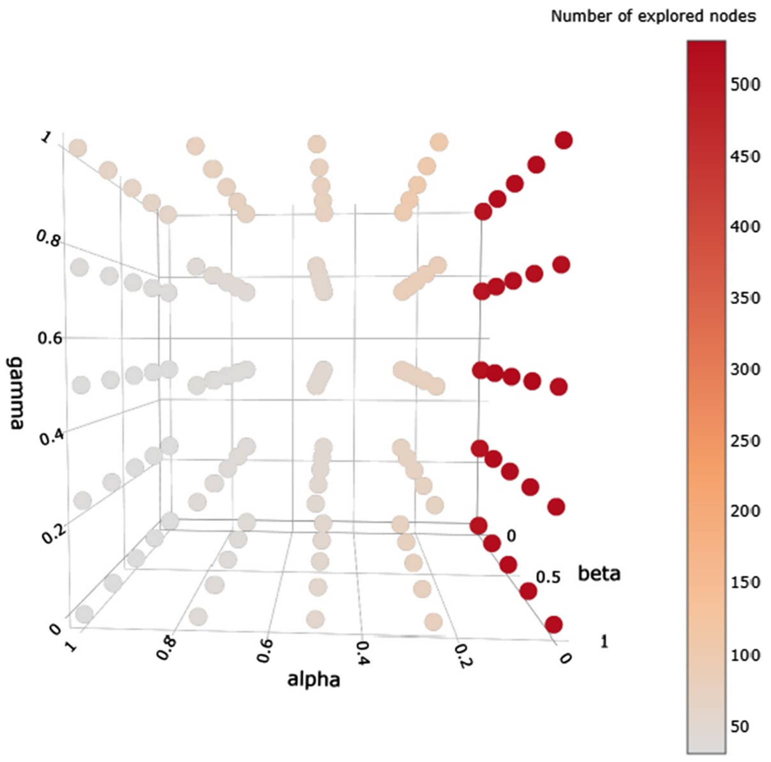

Fig. 3 Average of the number of nodes that need to be explored to find the solution, depending on the value of α, β and γ parameters. Results using the matransf data, $d_{max} = 4$ and $s_{max} = 1$

received are answered and the answer is accepted by the creator of the post, (2) they are related to R and (3) they contain the term “Matrix” in the title. In total we collected 20 questions and answers of R problems dealing with matrix transformations. Just as example, some of the questions used are: “How to reverse a matrix”, “Get positions for NA in a matrix”, “Rotate a Matrix in R” or “How to get the sum of each four rows of a matrix in R”. Each example includes an input matrix A, an output matrix B, a text T (the title) and a solution. The solution is not used in the process, but just to generate B.

Computation time and parameter settings In order to answer questions **Q2** and **Q3**, we have performed an ablation study to determine the good ranges for the parameters α, β and γ. As said in the previous section, higher α increases the weight for those (partial) solutions that match the dimension of the output matrix, whereas higher β increases the penalisation for those solutions including repeated functions. Finally, higher values of γ give more weight to the prior probability over the text hint. In our ablation study, we consider all the possible permutations including values from 0 to 1 with increments of 0.25 each time.

Figure 3 shows the average number of explored nodes for the real problems when using different values for the parameters. Here, we can see that $\alpha = 0$ increases drastically the nodes needed to find the right solution. In this case, the number of nodes is reduced considerably when $\alpha \geq 0.75$ (i.e., we give more relevance to the dimensions of the output matrix when building a partial solution during the search) and for values of

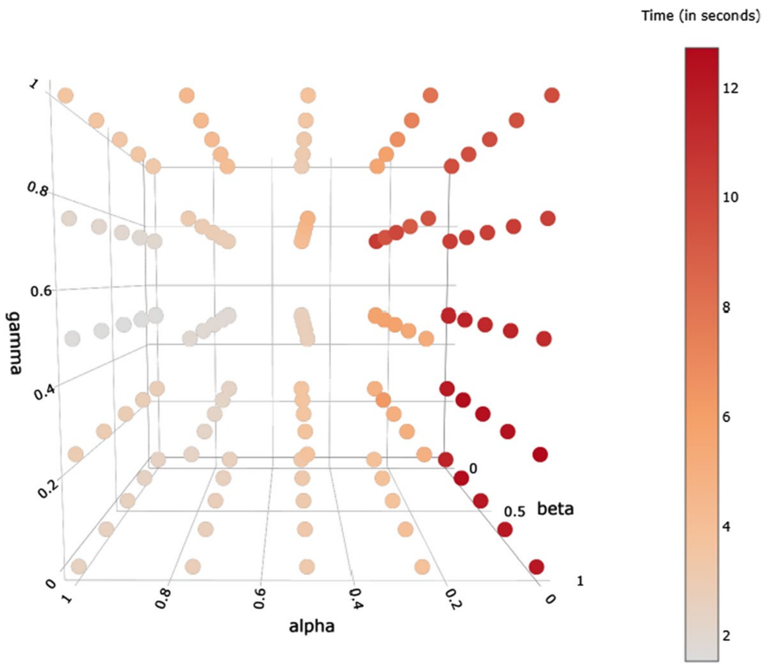

Fig. 4 Average of time (in seconds) needed to find the solution depending on the value of α, β and γ parameters. Results using the matransf data, $d_{max} = 4$ and $s_{max} = 1$

$\gamma \leq 0.75$ (not giving all the relevance to the prior probabilty). We can see that, in this case, β seems to have no relevance in the number of nodes explored.

If fewer nodes have to be explored, the time needed can be reduced. Figure 4 shows the average of time needed for the experiments to find the solution depending on different values of α, β and γ. We can see clearly again that α is very relevant to decrease the time needed to find the solution. The best times are obtained when $\gamma = 0.5$ and $\alpha = 1$. We can also see again that β seems to be not very useful in this study. However, although the β axis on the figures seems completely flat, there are values that do differ, yet slightly, with times and number of nodes in ranges which are insignificant in comparison with the other two parameters. It seems that penalising (or not) the use of repeated primitives in a matrix pipeline does not have a visible effect on the results. The reason for this is that the examples extracted from StackOverflow do not have repeated functions in their solutions (since the solutions proposed in this forum tend to be short and efficient), in such a way that there is not a big difference in the results using β. In general, elegant solutions rarely have repeated primitives, except when these primitives are constants (e.g., “the element in the third row and the third column”, using “3” in the solution as a repeated constant). However, we have not included constants in our list of functions because we would need to cherry pick a few small or frequent constants. This could be solved by a further study on how to include constants in the solutions without the use of explicit functions, by considering all constants in the algorithm itself but with an increasing cost for higher constants (lower probability). This would make the use and study of the beta parameter more insightful, with a controlled use of repetitions in the

Table 3 Results for the examples from StackOverflow

Strategy	Accuracy	Generated	Explored	Time
Uniform	0.60 ± 0.50	3414 ± 3511	867 ± 549	57.91 ± 52.36
Prior	0.65 ± 0.49	1065 ± 201	620 ± 362	39.84 ± 43.73
Prior + text	**1.00 ± 0.00**	**477 ± 429**	**14 ± 12**	**1.26 ± 1.18**

Average and standard deviation for accuracy, number of generated and explored nodes, and time in seconds, broken down by the three strategies and $s_{max} = 1$. The timeout is set to 120 s and $d_{max} = 4$. Best results are highlighted in bold

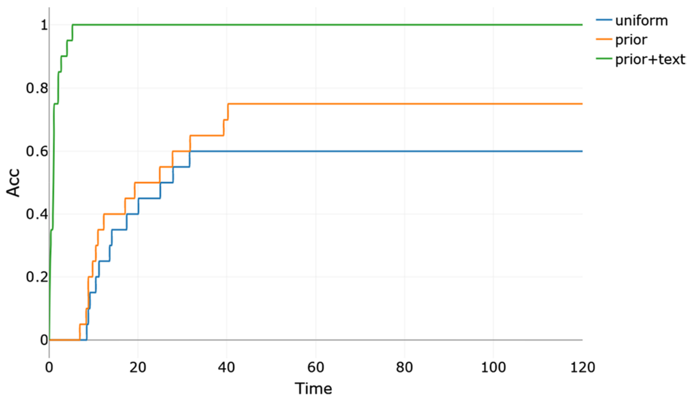

Fig. 5 Percentage of cases (y-axis) that are solved in less than the time (seconds) on the x-axis depending on the strategy used. Results using $d_{max} = 4$ and $s_{max} = 1$

solutions. For now, we are assuming that β is no longer needed so we remove it from the experiments (i.e. $\beta = 0$).

With the ablation study, we see that most ranges of the parameters are safe, but some of the refinements we introduced are relevant (except the primitive repetition). We can decide the parameters and run the experiments using the best settings. For the real-world datasets we have tested the strategy using the a priori probabilities including the text hint and the calculated parameters (Prior+Text). We have compared this strategy with two baselines: the uniform and the a priori (without text hint) probabilities as they are explained in Sect. 5. Concretely, we have tested the following strategies:

1. *Uniform* As said in Sect. 5, we consider $p(g)$ uniform with $\gamma = 1$ and $\alpha = 0$.
2. *Prior* $p(g)$ is estimated with the a priori probabilities with $\gamma = 1$ and $\alpha = 0$.
3. *Prior+Text* We use $p(g)$ as in the *Prior* strategy and $p(g|T)$ calculated using the text hint, with $\gamma = 0.5$ and $\alpha = 1$ (i.e., favouring solutions matching the dimension of B).

Table 4 Number of solutions found for each example using $d_{max} = 4$ and $s_{max} = \infty$, with a $timeout = 120\,\text{s}$

Example	n_{sols}	$n_{explored}$	$n_{created}$
1	1	267	9078
2	4	273	9180
3	16	285	9145
4	1	272	9282
5	2	273	9214
6	3	275	9235
7	3	268	8987
8	6	273	9050
9	1	270	9173
10	3	275	9224
11	2	268	9078
12	2	262	8874
13	2	265	8976
14	8	267	8806
15	32	292	8874
16	36	295	8806
17	28	286	8783
18	33	290	8747
19	2	272	9202
20	3	268	9078

n_{sols} represents the number of different solutions found; $n_{explored}$ is the total number of explored nodes; $n_{created}$ represents the total number of created nodes

For each strategy we run the experiments with $d_{max} = 4$ and $s_{max} = 1$. We give a time-out of 120 s.[11]

Results We now analyse whether the new strategy is able to reduce the search space significantly. Table 3 shows the nodes that are generated and explored. We see that the search space is considerably reduced when the text hint is considered and α is higher. The use of this strategy not only increases the accuracy (percentage of correct solutions found) to 100% but it also reduces significantly the search space and, consequently, the computation time. We can see the results in a different way in Fig. 5. This plot shows on the y-axis the percentage of cases that are solved in less than the time expressed on the x-axis.

Note that with 34 primitives there are $34^{d_{max}}$ nodes to explore as a maximum. For example, with $d_{max} = 4$ this is 1,363,336 nodes. How can we get only an average of 867 explored with the uniform strategy, which is breadth-first? The answer is given by the combinations that are pruned because the type function does not match, and all the further reduction is given by the use of probabilities in the other strategies. Table 4 shows that the number of solutions over the pruned hypothesis given by the type function is a huge improvement over the initial 34^4 space. We can see that the average number of solutions found when running the experiments using $d_{max} = 4$ and $s_{max} = \infty$ (trying to

[11] A complete list of events produced during the execution of the algorithm for a simple example can be seen in "Appendix 2".

find all the possible solutions), with a time-out of 120 s is 9.4, even when the average on the number of explored nodes is 275. In conclusion, the pruning by the type function is the main reason for the reduction of the search space.

6 Conclusions and future work

The process of generating code automatically can help data scientists when dealing with matrices (or data frames), the most common representation of information in data science, artificial intelligence and many other areas. Users can easily produce an example of the input matrix and a few cells of the output matrix, and the system will generate the code for them.

We have presented AUTOMAT[R]IX, a new system that is able to solve this very common problem of matrix transformation successfully. The system is based on a breadth-search approach pruned by the consistency of the types given by the dimensions of the matrices and the intermediate results. Besides, the system is guided by a strategy based on dynamic probabilities from a prior value depending on the frequency of the use of primitives on Github and, optionally, the relevance (TF-IDF) values of the terms in the text hints provided by the user compared with the terms in the R help documentation of the primitives.

We have tested the two baseline approaches with a synthetic dataset of 400 matrices and 34 different transformations in R. We have also tried the baselines compared with our algorithm using real examples of 20 problems extracted from Stack Overflow. The results show that the AUTOMAT[R]IX system is able to give the correct result for all of them in a very short time. In this case, using the strategy *Prior+Text* , which uses the text hints giving more importance to those solutions that match the dimension of the output matrix, AUTOMAT[R]IX can achieve 100% accuracy, reducing significantly the time spent to find the solution. Both datasets, with synthetic pipelines and real examples from Stack Overflow are available as matransf .

As future work we plan to add new characteristics (constraints) over the types (consider if data is numerical, text, boolean, etc.) or over the values (consider if numbers are negative or positive, consider the data distribution, etc.). We would like to include more primitives, new data structures and a deeper study of the β parameter also including more complex problems (i.e., needing more than 4 functions) and repeated functions (for instance, including constants). We also plan to create a visual interface or R package for ease of use. We would also like to explore and compare our algorithm with other optimisation approaches, such as planning or solvers. We encourage the approach to be replicated to synthesise functions for other languages such as Python. In general, the key ideas and the procedure to reduce the search space can be used for other similar problems in inductive programming, and other areas of artificial intelligence.

Appendix 1: List of functions included in the library

See Table 5.

Table 5 List of functions from the R language included in the background knowledge of AUTOMAT`[R]`IX

Function	m_{input}	n_{input}	m_{output}	n_{output}	m_{min}	n_{min}
`rowSums(A)`	m	n	1	m	2	0
`colSums(A)`	m	n	1	n	2	0
`rowMeans(A)`	m	n	m	1	0	0
`colMeans(A)`	m	n	1	n	2	0
`nrow(A)`	m	n	1	1	2	0
`ncol(A)`	m	n	1	1	2	0
`cor(A)`	m	n	n	n	2	0
`det(A)`	m	m	1	1	2	0
`diag(A)`	m	n	1	n	2	0
`t(A)`	m	n	n	m	0	0
`is.0(A)`	m	n	m	n	0	0
`!is.0(A)`	m	n	m	n	0	0
`A==0`	m	n	m	n	0	0
`A!=0`	m	n	m	n	0	0
`A<=0`	m	n	m	n	0	0
`A>=0`	m	n	m	n	0	0
`A<0`	m	n	m	n	0	0
`A>0`	m	n	m	n	0	0
`A==1`	m	n	m	n	0	0
`A!=1`	m	n	m	n	0	0
`A<=1`	m	n	m	n	0	0
`A>=1`	m	n	m	n	0	0
`A<1`	m	n	m	n	0	0
`A>1`	m	n	m	n	0	0
`A*A`	m	n	m	n	0	0
`apply(A,1,rev)`	m	n	m	n	2	0
`apply(A,2,rev)`	m	n	m	n	2	0
`as.table(A)`	m	n	m	m	0	0
`cbind(A,A)`	m	n	m	$n \times 2$	0	0
`rbind(A,A)`	m	n	$m \times 2$	n	0	0
`length(A)`	m	n	1	1	0	0
`max(A)`	m	n	1	1	0	0
`min(A)`	m	n	1	1	0	0
`mean(A)`	m	n	1	1	0	0

$m_{input} \times n_{input}$ represents the size of the input, while $m_{output} \times n_{output}$ represents the size of the output once the function is applied. m_min and n_{min} are the minimum number of rows and columns respectively that the input matrix requires for the function to be applicable

Appendix 2: Pipeline of events for a simple example

Figure 6 shows an example of a pipeline of events when running the algorithm. In this example, the goal is to learn a function that given an input matrix, returns an output matrix where each cell contains a Boolean value indicating if the corresponding position in the input matrix is `NA`. In the top of the figure, we include: matrix A (input), matrix B (partial

NA	1	NA	1
1	NA	NA	2
NA	2	NA	3

(a) Matrix A with some `NA` values.

TRUE	FALSE	·	·
·	·	·	·
·	·	·	·

(b) Matrix B with Boolean values representing whether a cell contains a `NA` or not. Only two cells are completed.

TRUE	FALSE	TRUE	FALSE
FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	TRUE	FALSE

(c) Matrix S, the completed version of B.

Iteration: 1
BK (ordered by the probabilities):

1	**length(A)**	**0.239125**
2	!is.na(A)	0.1178071
3	is.na(A)	0.1178071
4	apply(A,1,rev)	0.05579061
5	apply(A,2,rev)	0.05579061

Variables:
$g \leftarrow$ **length(A)**
$found \leftarrow$ FALSE
$explored \leftarrow$ FALSE
Number of explored nodes: 1
Number of created nodes: 68

Iteration: 2
BK (ordered by the probabilities):

1	**!is.na(A)**	**0.1178071**
2	is.na(A)	0.1178071
3	length(length(A))	0.05718077
4	apply(A,1,rev)	0.05579061
5	apply(A,2,rev)	0.05579061

Variables:
$g \leftarrow$ **!is.na(A)**
$found \leftarrow$ FALSE
$explored \leftarrow$ FALSE
Number of explored nodes: 2
Number of created nodes: 34

Iteration: 3
BK (ordered by the probabilities):

1	**is.na(A)**	**0.1178071**
2	length(length(A))	0.05718077
3	apply(A,1,rev)	0.05579061
4	apply(A,2,rev)	0.05579061
5	nrow(A)	0.05569636

Variables:
$g \leftarrow$ **is.na(A)**
$found \leftarrow$ TRUE
$explored \leftarrow$ FALSE
Number of explored nodes: 3
Number of created nodes: 0

Time: $0.033s$

Fig. 6 Short example of the pipeline of events for a simple problem: marking the positions with `NA`

output) and matrix *S* (complete output). The text used as a hint is: "Get positions for NA". In the figure we show the three iterations of the algorithm. *BK* represents the five functions with highest probability for each iteration. *Variables* shows the function selected, whether the solution has been found or the three has been completely explored, the total number of nodes explored at the moment and the number of nodes created in this iteration. Note that in *Iteration 2* we explore the branch of the tree that falls under `length(A)`. One of the compositions in that branch is precisely `length(length(A))`, i.e., apply the function again, which, because the a priori probability of length alone is so high, still gets sufficient probability to appear in the top 5 in the figure.

Acknowledgements We thank the anonymous reviewers for their comments, which have improved the paper significantly. This research was supported by the EU (FEDER) and the Spanish MINECO RTI2018-094403-B-C32 and the Generalitat Valenciana PROMETEO/2019/098. L. Contreras-Ochando was also supported by the Spanish MECD Grant (FPU15/03219). J. Hernández-Orallo is also funded by FLI (RFP2-152).

References

Contreras-Ochando, L., Ferri, C., & Hernández-Orallo, J. (2020a). Automating common data science matrix transformations. In *Machine learning and knowledge discovery in databases (ECMLPKDD workshop on automating data science)* (pp. 17–27). Springer, ECML-PKDD '19.

Contreras-Ochando, L., Ferri, C., Hernández-Orallo, J., Martínez-Plumed, F., Ramírez-Quintana, M. J., & Katayama, S. (2020b). Automated data transformation with inductive programming and dynamic background knowledge. In *Machine learning and knowledge discovery in databases* (pp. 735–751). Springer, ECML-PKDD '19.

Contreras-Ochando, L., Ferri, C., Hernández-Orallo, J., Martínez-Plumed, F., Ramírez-Quintana, M. J., & Katayama, S. (2020c). BK-ADAPT: Dynamic background knowledge for automating data transformation. In *Machine learning and knowledge discovery in databases (ECMLPKDD demo track)* (pp. 755–759). Springer, ECML-PKDD '19.

Cropper, A., Tamaddoni, A., & Muggleton, S. H. (2015). Meta-interpretive learning of data transformation programs. In *Inductive logic programming* (pp. 46–59).

Ferri-Ramírez, C., Hernández-Orallo, J., & Ramírez-Quintana, M. J. (2001). Incremental learning of functional logic programs. In *FLOPS* (pp. 233–247). Springer.

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *Proceedings 38th principles of programming languages* (pp. 317–330).

Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S., Schmid, U., & Zorn, B. (2015). Inductive programming meets the real world. *Communications of the ACM, 58*(11), 90–99.

He, Y., Chu, X., Ganjam, K., Zheng, Y., Narasayya, V., & Chaudhuri, S. (2018). Transform-data-by-example (TDE): An extensible search engine for data transformations. *Proceedings of the VLDB Endowment, 11*(10), 1165–1177.

Jenkins, T. (2002). On the difficulty of learning to program. In *Proceedings of the 3rd annual conference of the LTSN Centre for information and computer sciences, Citeseer* (Vol. 4, pp. 53–58).

Kandel, S., Paepcke, A., Hellerstein, J., & Heer, J. (2011). Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 3363–3372). ACM.

Katayama, S. (2005). Systematic search for lambda expressions. *Trends in Functional Programming, 6,* 111–126.

Kolb, S., Paramonov, S., Guns, T., & De Raedt, L. (2017). Learning constraints in spreadsheets and tabular data. *Machine Learning, 106*(9–10), 1441–1468.

Lieberman, H. (2001). *Your wish is my command: Programming by example*. Burlington: Morgan Kaufmann.

Menon, A., Tamuz, O., Gulwani, S., Lampson, B., & Kalai, A. (2013). A machine learning framework for programming by example. In *ICML* (pp. 187–195).

Mitchell, T., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., & Schlimmer, J. (1991). Theo: A framework for self-improving systems. In *Architectures for intelligence* (pp. 323–355).

Mitchell, T., Cohen, W., Hruschka, E., Talukdar, P., Yang, B., Betteridge, J., et al. (2018). Never-ending learning. *Communications of the ACM, 61*(5), 103–115.

Paramonov, S., Kolb, S., Guns, T., & De Raedt, L. (2017). Tacle: Learning constraints in tabular data. In *Proceedings of the 2017 ACM on conference on information and knowledge management, ACM, New York, NY, USA, CIKM '17* (pp. 2511–2514).

Parisotto, E., Mohamed, Ar., Singh, R., Li, L., Zhou, D., & Kohli, P. (2016). Neuro-symbolic program synthesis. arXiv preprint arXiv:161101855

Raza, M., Gulwani, S., & Milic-Frayling, N. (2014). Programming by example using least general generalizations. In *Twenty-eighth AAAI conference on artificial intelligence*.

Reynolds, A., & Tinelli, C. (2017). Sygus techniques in the core of an SMT solver. arXiv preprint arXiv:171110641

Salton, G., & Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Inf Process Manag, 24*(5), 513–523.

Santolucito, M., Hallahan, W. T., & Piskac, R. (2019). Live programming by example. In *Extended abstracts of the 2019 CHI conference on human factors in computing systems* (p. INT020). ACM.

Segovia-Aguas, J., Jiménez, S., & Jonsson, A. (2019). Computing programs for generalized planning using a classical planner. *Artificial Intelligence, 272,* 52–85.

Wang, X., Dillig, I., & Singh, R. (2017). Program synthesis using abstraction refinement. In *Proceedings of the ACM on programming languages* 2(POPL):63.

Wu, B., Szekely, P., & Knoblock, C. A. (2012). Learning data transformation rules through examples: Preliminary results. In *Information integration on the web* (p. 8).