# What is an AI assistant?

Tomas Petricek

July 27, 2018

## 1 Introduction

The idea of an *AI assistant* has been discussed in many contexts in the AIDA project. The Wrattler notebook system aims to provide a platform through which AI assistants can be easily used; the datadiff project developed an AI assistant which automates the process of reconciling inconsistencies between pairs of tabular datasets. However, we never clearly defined what an AI assistant is. This document aims to collect some notes to help answer this question. In the current version, these are mostly based on the needs of Wrattler and datadiff integration in Wrattler – as such, it clearly does not give the final answer, but merely one (limited, but hopefully useful) perspective.

A simple way of trying to address the data wrangling problem would be to build a set of functions that take some dirty datasets and produce one or more clean datasets. The user could then call these functions on their dirty data and would get a clean result (conceivably, this could also be further automated). However, this simple approach has two problems. First, real-world data sets often contain numerous corner cases that require some human insight. Second, we want the data analysis to be transparent and be able to understand how have the data wrangling problems been solved. A simple function does not address these because it does not encourage human interaction[1] and it is not transparent – it produces a result without indicating how and why.

## 2 What makes AI assistants unique

AI assistants address the above issues as follows. First, AI assistants are interactive. Given some input data, they perform some analysis and provide the user with several options to choose from (those can be most likely solutions to some problem, possible ways of resolving ambiguous corner cases, or a choice of next steps the AI assistant can follow). The options can be a finite list (where the user has to choose one) or open-ended (where the user can specify some input). They should also be sorted by likelihood (if we always choose the most likely option, the AI assistant will behave as a fully automatic function).

Second, AI assistants do not directly transform data, but instead, generate scripts (or, more formally, expressions) in simple languages (domain-specific languages, or DSLs)

[1] At best, users can run the function again with different parameters, but they get no suggestion about suitable values of the parameters.

that define how data should be transformed. The AI assistant can still use sophisticated machine learning algorithms (that do not produce an explanation of how they work) behind the scenes, but the end result will always be a simple script that can be reviewed and understood by the user.

Note that each AI assistant can use its own domain-specific language – the language should be as small as possible, i.e. it should be able to express the data transformations that the AI assistant can recommend, but nothing more.

The combination of interactivity and domain-specific languages allows us to do a number of interesting things:

- Scripts generated by AI assistant can be translated to R or Python (and it is easy to add support for other languages), which means that one can use an AI assistant during data analysis and exploration, but then end up with code that can be used in production in any programming language.
- The interactive nature of AI assistants makes it possible to give immediate feedback to users and let them correct potential issues early. For example, in Wrattler, we display the options that AI assistant provides together with a preview of results (of applying the generated expression to a sample dataset).

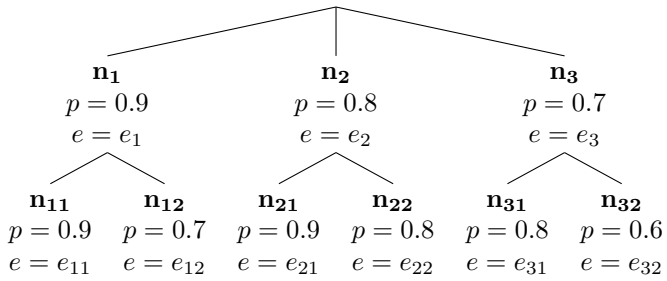
# 3 AI assistants as trees

We can think of an AI assistant as a function that takes some input data and generates a tree where each node offers a finite list of options that the user can choose from. (For simplicity, we ignore the possibility of open-ended options where the user specifies some input and we also ignore the fact that the AI assistant might take some initial parameters in addition to the input). Formally:

$$
\begin{array}{rcll}
a & = & \mathsf{Data} \to t & \text{(AI assistant)} \\
t & = & (n_1, p_1, e_k, t_1), \ldots (n_k, p_k, e_k, t_k) & \text{(Provided options)} \\
eval & = & \mathsf{Data} \times e \to \mathsf{Data} & \text{(Evaluator for the DSL)}
\end{array}
$$

An AI assistant $a$ is a function that takes some input data and returns a root tree node $t$. A tree node contains a list of options $(n_i, p_i, e_i, t_i)$ where $n_i$ is a name of the node (shown to the user, but otherwise unimportant), $p_i$ is a number that gives the likelihood of the option (between 0 and 1), $e_i$ is an expression in the domain-specific language that represents the data transformation attached to the node and $t_i$ is a sub-tree that can be empty or can offer more follow-up options. Note that this sub-tree can be generated lazily – it is only needed if the user chooses the option with which it is associated and so we do not need to evaluate the full tree upfront.

The nature of expressions $e$ depends on the particular AI assistant and so we leave that abstract for now. However, we certainly need to be able to apply the transformation on sample dataset, which is captured by the $eval$ function, which takes data together with a DSL expression and returns transformed data. An (abstract) example tree might look as follows:




Here, the root offers three options represented by nodes $n_1, n_2, n_3$. The most likely one as recommended by the AI assistant is $n_1$. The expression attached to the node is $e_1$ and when the user selects the node, we can display a preview using sample data $sample$ by evaluating $eval(sample, e_1)$.

Assuming the user chooses the most likely option $n_1$, they will then be presented with two subsequent options, $n_{11}, n_{12}$. Even though the AI assistant marks the first node as more likely, the user may review the previews $eval(sample, e_{11})$, $eval(sample, e_{12})$ and decide that the latter is actually the correct data cleaning operation. They can then choose the node $n_{12}$ as the final one use the cleaning script represented by the DSL expression $e_{12}$.

# 4 Examples of AI assistants

What would a sample AI assistant look like when seen through the above model? We consider two examples. The first is hypothetical AI assistant for extracting data from CSV files. The second is the datadiff assistant. For datadiff, we discuss the current implementation in Wrattler, but also briefly sketch a possible improvement (that improves the interactivity aspect).

## 4.1 Cleaning CSV files

As an example, consider the following broken CSV file. The fle starts with two metadata lines (one specifying the name and another specifying copyright information). This is followed by headers and actual data.

```
Counts of things
Copyright (c) Poor data export company, London, UK
Name, City, Count
Joe, London, 3
Jane, Edinburgh, 16
Jim, Cardiff, na
```

A hypothetical minimalistic AI assistant for cleaning CSV files could generate tree with two levels of nodes. In the first level, it will aim to infer where the actual data is in the CSV file (and it will offer its best guesses to the user). In the second level, it will aim to infer the types of the columns.

The domain-specific language of data cleaning operations consists of two functions named range and types that specify the range of data in the file and the types of the columns: $\mathsf{range}(\mathsf{lines} = l_n \ldots l_m, \mathsf{cols} = k)$ denotes that the data starts on line $l_n$ and ends on line $l_m$ and there are $k$ columns; $\mathsf{types}(t_1, \ldots, t_k)$ specifies that the types of the $k$ columns are $t_1, \ldots, t_k$. The types are either string or int.

Given the above file as an input, the AI assistant might produce the following tree:

$$
\begin{aligned}
&p = 0.9,\ e = \mathsf{range}(\mathsf{lines} = 2 \ldots 6, \mathsf{cols} = 3) \\
&\quad \longrightarrow p = 0.9,\ e = \mathsf{range}(\mathsf{lines} = 2 \ldots 6, \mathsf{cols} = 3); \mathsf{types}(\mathsf{string}, \mathsf{string}, \mathsf{string}) \\
&\quad \longrightarrow p = 0.3,\ e = \mathsf{range}(\mathsf{lines} = 2 \ldots 6, \mathsf{cols} = 3); \mathsf{types}(\mathsf{string}, \mathsf{string}, \mathsf{int}) \\
&p = 0.7,\ e = \mathsf{range}(\mathsf{lines} = 3 \ldots 6, \mathsf{cols} = 3) \\
&\quad \longrightarrow p = 0.8,\ e = \mathsf{range}(\mathsf{lines} = 3 \ldots 6, \mathsf{cols} = 3); \mathsf{types}(\mathsf{string}, \mathsf{string}, \mathsf{int}) \\
&\quad \longrightarrow p = 0.7,\ e = \mathsf{range}(\mathsf{lines} = 3 \ldots 6, \mathsf{cols} = 3); \mathsf{types}(\mathsf{string}, \mathsf{string}, \mathsf{string}) \\
&p = 0.3,\ e = \mathsf{range}(\mathsf{lines} = 1 \ldots 6, \mathsf{cols} = 1) \\
&\quad \longrightarrow p = 0.9,\ e = \mathsf{range}(\mathsf{lines} = 1 \ldots 6, \mathsf{cols} = 1); \mathsf{types}(\mathsf{string})
\end{aligned}
$$

At the first level, the AI assistant gives us three options. The first, most likely, one skips just the first line and identifies that there are three columns (perhaps because the second line contains two commas and can thus be interpreted as data row). The second skips first two lines and recognises three columns. The AI assistant might find this less likely, but it is the option we want to choose as users. Finally, the third option takes all lines, but then it has to join all columns into one which is unlikely.

The user can choose the second option, at which point a user interface can evaluate the expression $\mathsf{range}(\mathsf{lines} = 3 \ldots 6, \mathsf{cols} = 3)$ and show a preview of the cleaning result. Next, the user has a choice of two ways of inferring types – the most likely one (and the right one) is that the third column has a type int (which means that the value na in the last row will be treated as missing).

The resulting expression $\mathsf{range}(\mathsf{lines} = 3 \ldots 6, \mathsf{cols} = 3); \mathsf{types}(\mathsf{string}, \mathsf{string}, \mathsf{int})$ can then be applied to the input data using the $eval$ function, which gives us columns Name of type string, City of type string and Count of type int with data:

$$
\begin{aligned}
&(\text{"Joe"}, \text{"London"}, 3) \\
&(\text{"Jane"}, \text{"Edinburgh"}, 16) \\
&(\text{"Jim"}, \text{"Cardiff"}, NaN)
\end{aligned}
$$

## 4.2 Current design of datadiff AI assistant

The datadiff project takes two tabular datasets that are in inconsistent formats, but represent two sets of observations about the same entity (such as broadband speed in two different years). It generates a list of patches that can be applied to one dataset in order to convert it to the format used by the second dataset. The patches form a domain-specific language (DSL) of the data transformation suggested by datadiff. A patch $p$ can be one of the following (the details are not important for this document, but it provides a nice example of a DSL):

$$
\begin{array}{rcll}
p & = & \mathsf{permute}(\pi) & \text{(Reorder columns using a given permutation)} \\
& | & \mathsf{recode}(i, [s_1 \mapsto s_2, \ldots]) & \text{(Recode categorical values using in column } i\text{)} \\
& | & \mathsf{linear}(i, a, b) & \text{(Apply linear transform } va + b \text{ to a numerical column } i\text{)} \\
& | & \mathsf{delete}(i) & \text{(Drop the column at index } i\text{)} \\
& | & \mathsf{insert}(i, d) & \text{(Insert a new column at index } i \text{ initialized to values } d\text{)}
\end{array}
$$

The current implementation of datadiff is a non-interactive R package that takes two datasets and produces a list of patches. It takes a couple of parameters that specify which patches should datadiff attempt to generate (together with various weights to tweak the optimization).

The current datadiff AI assistant as implemnented in Wrattler generates a tree that allows the user to select and apply individual patches. The user can review each patch to make sure that it is a reasonable transformation. For example, consider the following two datasets:

```
Name, City
Joe, London
Jane, Edinburgh
Jim, London
```

```
City, Name, Count
Cardiff, Alice, 1
Cardiff, Bob, na
Edinburgh, Bill, 2
```

If we ask datadiff for a list of patches to transform the dataset on the right-hand side to the format of the dataset on the left-hand side, we might get the following patches (assuming columns are indexed from 1):

$$\mathsf{delete}(3);\ \mathsf{permute}(2, 1);\ \mathsf{recode}(2, [\text{"Cardiff"} \mapsto \text{"London"}])$$

Datadiff correctly infers that we need to drop the newly added `Count` column and that the order of `Name` and `City` has been switched. It might also suggest that `City` is a categorical column where the encoding has been changed, which is not the case here (but this would be a reasonable guess if one dataset contained values "true" and "false" while the other contained "yes" and "no").

The datadiff AI assistant generates the following tree based on the three inferred patches (we omit the parameters of the patches):

$$
\begin{array}{l}
p = 0.9,\ e = \mathsf{delete}(\ldots) \\
\quad \longrightarrow p = 0.9,\ \mathsf{delete}(\ldots); \mathsf{permute}(\ldots) \\
\quad\quad \longrightarrow p = 0.6,\ \mathsf{delete}(\ldots); \mathsf{permute}(\ldots); \mathsf{recode}(\ldots) \\
\quad \longrightarrow p = 0.6,\ \mathsf{delete}(\ldots); \mathsf{recode}(\ldots) \\
\quad\quad \longrightarrow p = 0.9,\ \mathsf{delete}(\ldots); \mathsf{recode}(\ldots); \mathsf{permute}(\ldots) \\
p = 0.8,\ e = \mathsf{permute}(\ldots) \\
\quad \longrightarrow p = 0.9,\ \mathsf{permute}(\ldots); \mathsf{delete}(\ldots) \\
\quad\quad \longrightarrow p = 0.6,\ \mathsf{permute}(\ldots); \mathsf{delete}(\ldots); \mathsf{recode}(\ldots) \\
\quad \longrightarrow p = 0.6,\ \mathsf{permute}(\ldots); \mathsf{recode}(\ldots) \\
\quad\quad \longrightarrow p = 0.9,\ \mathsf{permute}(\ldots); \mathsf{recode}(\ldots); \mathsf{delete}(\ldots) \\
(\ldots)
\end{array}
$$

At the first level, the user can choose one of the three patches (we only show subtrees for the first two). The patches are sorted by their likelihood and so delete is offered before permute. In the second level, the user can choose one of the remaining patches. Assuming the user follows the first branch and chooses delete, they can now choose to apply permute or recode (the first option being more likely). In the last third level, the AI assistant offers the remaining patch. However, the user can stop before reaching the leaf and choose just $\mathsf{delete}(3)$; $\mathsf{permute}(2, 1)$ as the final data cleaning script. As before, Wrattler can use the *eval* function to give previews of the transformed datasets while the user is navigating through the tree.

## 4.3 Alternative design of datadiff AI assistant

The current design of the datadiff AI assistant is not optimal, because it performs a global optimization up-front and then merely allows the user to choose some of the patches. This is problematic, because the global optimization can be slow and it does not use the user input to improve the results.

For example, if the user could specify that a certain column should not be recoded, datadiff could run the global optimization taking this constraint into account and, say, match and reorder columns differently (this is currently not possible, because the current version runs the global optimization once up-front and then merely offers a choice of produced patches).

An alternative datadiff AI assistant would generate expressions in the same domain-specific language as the current one (list of patches $p$). However, the tree would be generated and structured differently. This could be done by maintaining a list of constraints and generating the best possible patch given certain constraints, together with a choice of most likely constraints the user might want to add. A constraint $c$ would be as follows:

| | | | |
|---|---|---|---|
| $c$ | $=$ | $\mathsf{norecode}(i)$ | (Values in a categorical column $i$ should not be recoded) |
| | $\mid$ | $\mathsf{notransform}(i)$ | (Values in a numerical column $i$ should not be transformed) |
| | $\mid$ | $\mathsf{nodelete}(i)$ | (Column $i$ should not be deleted) |
| | $\mid$ | $\mathsf{match}(i, j)$ | (Column $i$ should be mapped to sample column $j$) |

The first three constraints specify that certain patches should not be generated. The last one is more interesting as it allows the user to explicitly give a mapping for a column (but unless other constraints are given, datadiff can still attempt to transform or recode the column to get a better match). This is likely not a complete set of constraints, but it is sufficient to illustrate the idea.

The AI assistant would then implement a function that takes two datasets together with a set of constraints and generates a cleaning script together with a list of constraints (sorted by their likelihood) that the user might want to add. (If we allowed nodes with open-ended options, the user could write an arbitrary constraint, but for now, we just assume that the AI assistant can generate all sensible options.)

Using the above function, we can produce a tree as follows. We start with an empty set of constraints $C = \{\}$ and call the function to obtain a data cleaning DSL expression $e$ and potential constraints $c_1, \ldots, c_k$. For each constraint, we add it to the set $C$ and generate a sub-tree by calling the AI assistant function recursively. In other words,

following a path through the tree would add more and more constraints to the set $C$ and each node will come with the best DSL expression that datadiff can generate under the constraints determined by the path taken through the tree. The following illustrates (a small part of) the tree that the revised AI assistant for datadiff would generate for the above two files. (In addition to a DSL expression $e$ and likelihood $p$, we also include a set of constriants $C$ associated with each node):

$$
\begin{aligned}
C = [], p = 1,\ e = \mathsf{delete}(\ldots);\ \mathsf{permute}(\ldots);\ \mathsf{recode}(\ldots) \\
\quad \longrightarrow C = [\mathsf{norecode}(2)], p = 0.4, e = \mathsf{delete}(\ldots);\ \mathsf{permute}(\ldots); \\
\qquad \longrightarrow C = [\mathsf{norecode}(2); \mathsf{nodelete}(3)], p = 0.2, e = (\ldots) \\
\quad \longrightarrow C = [\mathsf{nodelete}(3)], p = 0.2, e = \mathsf{delete}(\ldots);\ \mathsf{permute}(\ldots);\ \mathsf{recode}(\ldots) \\
\quad \longrightarrow C = [\mathsf{match}(1,1)], p = 0.4, e = \mathsf{delete}(\ldots);\ \mathsf{permute}(\ldots);\ \mathsf{recode}(\ldots) \\
\quad \longrightarrow C = [\mathsf{match}(1,2)], p = 0.8, e = \mathsf{delete}(\ldots);\ \mathsf{permute}(\ldots);\ \mathsf{recode}(\ldots)
\end{aligned}
$$

The tree has only one root node which was generated without any constraints. The generated expression is the best guess by datadiff, which drops the `Count` column, rearranges columns and recodes (incorrectly) the `City` column. The nodes at the second level allow the user to add a number of constraints.

The first two sub-nodes prevent datadiff from generating two patches that it employed. The first one disallows recoding the column `City`. The resulting expression drops `Count` and rearranges the columns, which is the result we were hoping to obtain. The second sub-node disallows deleting the `Count` node. In this case, datadiff has to find other ways of reconcilling the data sets, perhaps by recoding `Code` and dropping another column. Finally, the last two sub-nodes allow us to explicitly specify a mapping between columns. The listing only shows the first two (map column 1 to column 1; map column 1 to column 2), but the AI assistant would likely generate the full range, sorted by likelihood.

# 5 Evaluating AI assistants

When building an AI assistant, we want to be able to evaluate how good it is. For an AI tool that is implemented as a data cleaning function, we can run it on sample inputs and asses how good the resulting outputs are. (Even this is not so simple, but we might know what the optimal results look like for some inputs, or we could generate inputs by introducing errors into clean datasets and see if those were eliminated.)

The situation with AI assistants is more complicated as they are interactive and aim to produce cleaning scripts that humans can understand. We could evaluate AI assistants using user studies (and this would be, no doubt, worthwhile), but there are a couple of things that we can say about AI assistants in a formal way too.

## 5.1 Evaluating AI assistant in an automatic way

First of all, we can treat an AI assistant as a fully automatic tool. To do this, we automatically choose the most likely option provided by the AI assistant until we reach a leaf node (or, perhaps, until we reach a case where the sub-node has a likelihood $p \leq 0.5$).

More formally, given a dataset $d$ and an AI assistant $a$ (as defined on page 2), we run the AI assistant $a(d)$ to obtain a list of options $(n_1, p_1, e_k, t_1), \ldots (n_k, p_k, e_k, t_k)$. We choose the most likely node $(n_i, p_i, e_i, t_i)$ (i.e. $p_i \geq p_j$ for all $j$). If $t_i$ is empty or the likelihoods of all options inside $t_i$ are less than $0.5$, we return an expression $e_i$; otherwise, we recursively follow the sub-tree $t_i$. Finally, we evaluate the resulting data cleaning expression using $eval(d, e_i)$ and asses how good the cleaned data set is using whatever methods we would use for an automatic cleaning function.

This way, we can guarantee that the interactive AI assistant is as good as an automatic cleaning function would be. This is a reasonable baseline test, but it does not tell us much about how good the interactive aspects are.

## 5.2 Evaluating interactivity for annomaly handling

One of the motivations for the idea of AI assistants was that real-world data cleaning tasks often have ad-hoc issues that require some form of manual intervention. In other words, the file might have an annomaly that a human can spot, but that an automatic data cleaning algorithm would miss. For example, consider the following two CSV files:

```
1 Name, Count, ID
2 Jim, 45, 1
3 Jane, 52, 2
4 Joe, 67, 3
```

```
1 Name, Code, Value
2 Alice, 35, 4
3 Bill, 38, 5
4 Bob, 42, 6
```

As humans, we can see that the column `ID` should be aligned with `Code` and `Count` should likely match `Value`. However, unless the AI assistant considers possible meanings of the file headers, it has no way of discovering this.

We might call such annomalies *adversary annomaly*. There is very likely no way of characterising adversary annomalies in general for all AI assistants, but it should be possible to characterise them for concrete data wrangling problems. Examples might include rows that are not outliers in the statistical sense, textual values that have the right format, but wrong meaning such as "Refugee Olympic Athletes" among the names of countries[2], or a suspicious numerical value such as $123.45678$ in a numerical column with values $190.52136, 58.68297, 123.45678, 67.47067, 47.69846$.

An important quality of an AI assistant is tht it makes it possible to eliminate adversary annomalies with a small number of interactions, ideally just by choosing an appropriate non-default offered option at one point when traversing the tree. For example, given the above example, the alternative datadiff AI assistant would allow us to choose a tree node that adds a constraint $\texttt{match}(2, 3)$ and we would get the correct result with just one manual intervention (the other column would be reordered automatically).

To summarise, we want to be able to claim that an AI assistant makes it easy to handle ad-hoc corner cases that are not discovered by the machine learning algorithm behind it. To do this, we introduce adversary annomalies into the data – those are plausible real-world data annomalies that are not discovered by the algorithm emplyed by the AI assistant. A good AI assistant allows the user to eliminate these with a minimal number of interactions, i.e. choices of other than the most likely nodes in the tree.

[2]This is a name for an actual team of 10 athletes competing in the 2016 Rio Olympic games.

### 5.3 Easy discoverability of best results

Finally, the current integration for AI assistants in Wrattler exposes the tree through a user interface. The user calls the AI assistant and then chooses a sub-tree to explore (by typing '.' and choosing one of the options in a drop-down menu). This means that it is desirable to make the most likely result (domain-specific language expression) closer to the root rather than somewhere deep in the tree.

As an example, consider the two designs of the datadiff assistant. In the first design, the user has to manually accept all the desired patches. If the optimal patch needs to drop 15 extra columns from a dataset, the user will have to make 15 choices explicitly. The second design of the datadiff assistant is better because the most likely set of patches is available immediately in the root node (and users only need to continue exploring a sub-tree if they need to address an adversary annomaly).