Wrattler: A platform for AI-assisted data science

Authors

1 Introduction

Data science is an iterative, exploratory process that requires a collaboration between a computer system and a human. A computer can provide advice based on statistical analysis of the data and discover hidden structures or corner cases, but only a human can decide what those mean and decide how to handle them in data science scripts. Data science is often cited as an expensive and time consuming task, especially due the costs of data cleaning and data wrangling. We propose four fundamental reasons why practical data science is expensive:

Big data is big, so the analyst doesn't understand it all.* Even if a data set is small enough fits on one computer, it's still too large to fit in an analyst's working memory. But this means every analysis is huanted by the spectre of lurking, potentially unknown data quality issues. This also makes it more difficult to do data fusion, because there may be corner cases that make it more difficult to join two disparate data sources than expected.

The double Anna Karenina principle. Not only is every dirty data set dirty in its own way, but *pace* Tolstoy, every clean data set is clean in its own way as well. "Data" is such an abstract concept that specific integrity conditions to characterize whether data is dirty, and potentially even the most appropriate formalism for integrity conditions, differ dramatically across the vast array of disparate use cases of data science, ranging form relational data describing the customers of a country, time series data describing sensor data in an internet of things platform, and huge datasets of satellite imagery of the earth over a multi-year time scale.

Death by a thousand cuts. Often data transformation and processing steps are individually very simple. But there may need to be lots of them, and because big data is big, an analyst never knows if she has found them all.

Feedback cycles everywhere. Data science is not a pipeline but a connected mess of epicycles. This is because every step in a data analysis actually teaches the analyst more about the data and the problem, which might require rehtinking the earlier steps. For example, there might be data quality issues that are not uncovered until the analyst investigates the output of a regression model.

To meet these challenges, we present Wrattler, a new type of system for data science that aims to transform the process of data analysis. Wrattler combines the interactive and literal programming paradigms of notebook systems such as Jupyter with new advances in AI systems for data wrangling and in provenance. The main design principles in Wrattler are:

Interactive. Interactivity is a necessary because "big data is big", so the analyst learns about the data set and the problem as she explores it. Wrattler enables an efficient interaction by bringing computation closer to the human. Notebooks run in the browser, cache partial results of computations and provide previews of script results on-the-fly during development.

Reproducible. Data analyses must be reproducible because of feedback cycles. As the analyst learns more about the problem, this may uncover data cleaning or preparation issues that require redesigning and rerunning the analysis. Wrattler separates the task of running scripts from the task of managing state. This is handled by a data store, which tracks the provenance and semantics of data, supports versioning and keeps the history, making the data analyses fully reproducible.

Polyglot. Modern data science naturally draws on many competing languages and libraries, such as R and Scipy. As a side effect of our interactive, reproducible design, we obtain nearly for free the ability to support polyglot data analyses. Multiple languages can be used in a single notebook and share data via the data store. Analysts can use R and Python, but also interactive languages for data exploration that run in the browser and provide live previews.

Smart. AI can examine and find patterns in big data where a human cannot, again aiming at the problem that big data is big. AI can be used to direct the analyst's attention and to generalize decisions about data transformation to new data that that the analyst hasn't seen. Wrattler serves as a platform for AI assistants that use machine learning to provide suggestions about data. Such AI assistants connect to the data store to infer types and meaning of data, provide help with data cleaning and joining, but also help data exploration by finding typical and atypical data points and automatically visualizing data.

Explainable. The hints provided by AI assistants are explainable. Rather than working as black boxes that transform one dataset into another, AI assistants generate scripts in simple domain-specific languages, that specify how the data should be transformed. Those scripts can be reviewed and modified by a human.

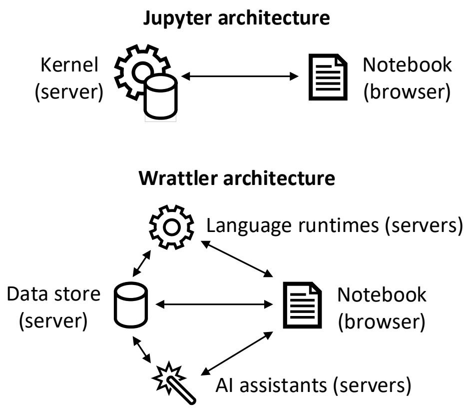

Figure 1. In notebook systems such as Jupyter, state and execution is managed by a kernel. In Wrattler, those functions are separated and enriched with AI assistants.

In the rest of this document, we discuss limitations of current notebook systems and how Wrattler resolves them (Section 2). We discuss how the individual components of Wrattler work together (Section 3) and then focus on two of them in detail – we look at how Wrattler can be extended with AI assistants (Section 4) and how semantic information about data is managed by the data store (Section 5).

2 Wrattler and notebooks

Notebook systems such as Jupyter became a popular programming environment for data science, because they support gradual data exploration and provide a convenient way of interleaving code with comments and visualizations. However, notebooks suffer from a number of issues that hanper reproducibility and limit the possible interaction model.

Notebooks can be used in a way that breaks reproducibility. The state is maintained by a *kernel* and running a code in a cell overwrites the current state. There is no record of how the current state was obtained and no way to rollback to a previous state. The fact that the state is maintained by the kernel means that it is hard to combine multiple programming languages and other components such as AI assistants. Finally, notebooks provide a very limited interaction model. To see the effect of a code change, an entire cell and all subsequent cells need to be manually reevalutated.

The architecture of Wrattler allows us to address these issues, as well as to provide a platform for building novel AI assistants and interactive programming. The architecture is illustrated in Figure 1. The components of Wrattler are:

Data store. Imported external data, results of running scripts and of applying AI assistants are stored in the data store. It stores versioned data frames with metadata such as types, inferred semantics, data formats or provenance.

Language runtimes. Scripts are evaluated by one or more language runtimes. The runtimes read input data from and write results back to the data store.

AI assistants. When invoked from the notebook, AI assistants read data from data store and provide hints to the analyst. They help to write data cleaning scripts or annotate data with additional metadata such as inferred types.

Notebook. The notebook is displayed in a web browser and orchestrates all other components. The browser builds a dependency graph between cells or individual expressions in the cells. It calls language runtimes to evaluate code that has changed, AI assistants to provide hints and reads data from the data store to display results.

3 Wrattler components

Wrattler consists of a notebook user interface running in the web browser, which communicates with a number of server-side components including language runtimes, the data store and AI assistants. In this section, we discuss the components in more detail, starting with the dependency graph that is maintained on the client-side, by the web browser. AI assistants are discussed later in Section 4

3.1 Dependency graph

When opening a notebook, Wrattler parses the source of the notebook (consisting of text cells and code cells) and constructs a dependency graph, which serves as a runtime representation of the notebook.

As an example, the graph in Figure 3 represents the notebook in Figure 2. Top-level nodes (squares) represent individual notebook cells. Code (circles) is represented either as a single node (Python, JavaScript) or as a sub-graph with node for each operation (cleaning DSL understood by Wrattler). Any node of a cell can depend on a data frame (hexagons) exported by an earlier cell. When code in a cell changes, Wrattler updates the dependency graph, keeping existing nodes for entities that were unaffected by the change.

The dependency graph enables several features that are difficult for most notebook systems:

- When code changes, Wrattler only needs to recompute a small part of the graph. This makes it possible to provide live previews, especially for simple data analytical DSLs.
- Refactoring can extract parts of the notebook that are needed to compute a specified output.
- Refactoring can translate nodes of an analytical DSLs (generated by an AI assistant) into R or Python.
- The graph can be used for other analyses and refactorings, such as provenance tracking or code cleanup.

3.2 Data store

The data store provides a way for persistently stroing data and enables communication between individual Wrattler

We retrieve data from a government web site using Python:

```
url = open("http://data.gov.uk/.../bb2014.csv")
bbRaw = csv.reader(url)
```

In the next block, we used an AI assistant to assist with cleaning and created a script that marks #!NULL values as missing:

```
let bbClean = bbRaw.missing.as_missing("#!NULL")
```

Finally, we use JavaScript to create a histogram of upload speeds:

```
vega.hsit(bbClean, "Upload speed")
```

Figure 2. A small notebook consisting of three cells. The first Python cell exports bbRaw; the second Wrattler cell uses it and exports bbClean and the JavaScript cell visualizes bbClean.

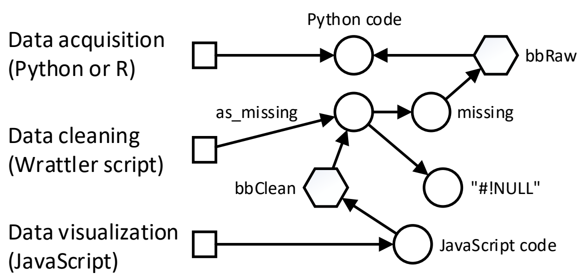

Figure 3. Dependency graph for the notebook in Figure 2. Data acquisition is represented as a single Python cell. Data cleaning is understood by Wrattler and is the graph has a node for each call and constant. Hexagons represent data frames shared by cell and stored in the data store.

components. Data store keeps external data files imported into Wrattler, as well as computed data frames (and, potentially, other data structures). It is immutable and keeps past versions of data to allow efficient state rollback. For persistency and versioning, data store also serializes and stores multiple versions of the dependency graph.

The data store supports a mechanism for annotating data frames with additional semantic information, such as:

- Columns can be annotated with (and stored as) primitive data type such as date or floating-point number.
- Columns (or a combination) can be annotated with semantic annotation such as geo-location or and address.
- Columns, rows and cells of the data frame can be annotated with other metadata such as provenance.

The format of the annotations as well as other features of the data store are discussed in more detail in Section 5.

3.3 Language runtimes

Language runtimes are responsible for evaluating code and assisting with code analysis during the construction of dependency graph. They can run as server-side components (e.g. for R and Python) or as client-side components (for JavaScript and small analytical DSLs used by AI assistants). Unlike Jupyter kernels, a language runtime is stateless. It reads data from and writes data to the data store. (Using a cache for efficiency.)

4 AI assistants

An AI assistant is a component of the system that guides data analysts through a single data collection, integration, preparation, analytics, or reporting task. For example, an assistant might specialize in identifying suspicious values, such as "-1" in a column labelled "heart rate", which might indicate that certain data was not collected. Assistants are interactive, in that they propose a transformation of the data to the analyst, based on combinations of statistical and symbolic AI analysis, and then refine the transformation based on feedback from the analyst. Typically analysts will interact with many separate assistants over the course of an analysis, each of which specialize in a single kind of task.

4.1 Architecture of AI assistants

There are two important design aspects of AI assistants, which may seem contradictory. First, they must be interactive, following the first principle behind Wrattler, but following the second, they must also be reproducible. We square this circle by using the notebook system as a reproducible interlingua. The output of an AI assistant is always code, rather than being a black-box that transforms data, and it is this code that is agreed between the AI and the analyst during the interaction. Thus the interaction is viewed as a "design phase" that is not reproduced on new data.

More specifically, an AI assistant defines a domain specific language (DSL) for the task at hand. When invoked from a notebook, it guides the user in creating a script in the DSL. Running the script performs the required operation (such as data cleaning). The assistant defines the semantics of those operations and, when the script is executed, stores the result into a data store. This new data frame can contain new data or additional information (such as their inferred probabilistic type). The interaction is illustrated in Figure 1. An AI assistant reads data from the data store and returns suggestions to the notebook, which the user can review and accept.

Assistants are therefore based on two different ideas:

Domain specific languages. Wrattler includes a scripting language with simple syntax that is used by individual AI assistants to define domain specific languages. The syntax is shared across multiple AI assistnts, but the operations are provided by individual assistants.

For example, an AI assistant based on datadiff guides the user in writing scripts that transform a malformed data frame into a correct format specified by a correct data frame. The resulting script might look as follows:

```
datadiff(broadband2014, broadband2015)
  .drop_column("WT_national")
  .drop_column("WT_ISP")
  .recode_column("URBAN", [1, 2], ["Urban", "Rural"])
```

The script specifies that two columns should be dropped from the badly structured data frame and one column needs to be recoded (turning 1 and 2 into strings Urban and Rural).

The code produced by an AI assistant can be used in two ways. First, the analyst writes the code interactively during an interaction with the AI assistant. During this phase, Wrattler keeps a fine-grained representation of the code in the dependency graph (second cell in Figure 3), which allows Wrattler to provie live previews of the result, giving the user an immediate feedback.

At a later stage, when the code is used in production, the code can be translated to a language like R or Python. This translation uses the fine-grained representation of the code in the dependency graph and maps individual nodes to corresponding Python or R functions.

Interactivity. Assistants should usually contain user interaction in their design, and adapt rapidly to feedback from the analyst. For example, a data cleaning assistant would probably want to ask the user to confirm potential changes, and if the user says no, to reconfigure the cleaning module significantly so that the user does not need to keep rejecting similar changes.

In the above example, the interaction is provided through the "." operator of the scripting language. When the analyst types the first line to invoke the AI assistant and types "." the datadiff assistant offers the most likely patches and the user can choose which ones to accept. Scripts constructed in such way are readable, allowing the data analyst to look at them and verify that the transformations accomplish the intended task (such verification further benefits from live previews that Wrattler provides when the analyst navigates through the individual transformations).

At the same time, AI assistants can be used in a fully automatic "batch mode" by analysts who already know what they want to achieve and know that the assistant produces the result they require. In this case, Wrattler will simply automatically accept the default recommendations made by an AI assistant. This allows packaging the assistant as an equivalent of scikit-learn for data preparation.

The Wrattler architecture supports the development of AI assistants in a number of ways. First, AI assistants have access to the data store and can use it to provide guidance about both data and code. Second, Wrattler includes an extensible framework for defining domain-specific languages (DSLs) that can describe, for example, data extraction, cleaning, transformation or visualization. The data store provides AI assistants with two sources of information that they can exploit to determine what guidance to provide the analyst:

– AI assistants can access all data stored in the data store. This includes imported external data files, such as raw text or raw HTML content from which the analyst wants to extract data, as well as structured data frames imported from CSV files or produced as a result of previous computations. Moreover, an AI assistant can also access multiple versions of a file.
– AI assistants can access the dependency graph created for a notebook. This allows the them to access the transformation history, which is the code that has already been run during the analysis. The gives assistants information about the context such as transformations that have already been performed.

The assistants should be self-contained in the sense that they interact with each other only through the data store. This restriction is necessary in order for provenance tracking and reproducibility.

4.2 The Assistant Landscape

AI assistants can be imagined for every step of the data analytics process. To give an idea of the breadth of this framework, we present a broad list of the data analytics tasks which have potential for assistant automation. This is an intentionally broad list, more so than could be fully explored in even a large research project, but gives a sense of the breadth of potential use cases of our framework.

Data collection. Assistants can search the data store, as well as external repositories to recommend data sets that are relevant to the task at hand and offer to import these. For example, when analysing data about different countries, the assistant could suggest joining in public demographic information from DBPedia.

Data integration is a problem as common as it is difficult. We envisage assistants that combine logical and statistical reasoning to assist with data integration. A simple but useful example of a tool in this space is to identifying pairs of variables across multiple data sources that are likely to be daily temperatures. More sophisticated examples could include and extend the most cutting edge techniques in the data fusion and semantic technologies literatures.

Schema inference. A sad fact of data science is that data sets are often not as well documented as we might like, even to the minimal level of documentation of the types of each column. There is an interesting opportunity to develop machine learning methods that inferring types of variables from their values, e.g., by noticing that integer values between 0 and 30 in the United Kingdom might describe an outdoor temperature, especially if other weather-related variables are detected in the data set.

Data parsing. A common task in data analysis is to convert data that is "almost" in structured form, such as HTML tables, into a relational table. Assistants in this space include tools specifying a scraper for web pages, related to wrapper induction [5], for inferring format of CSV files (while simple conceptually, they have no standard format), and for extracting tables from html and pdf files [7].

Data cleaning is a well studied problem in the databases and data mining literature [1, 4], but there are still a huge number of issues that have been unexplored. For example, we envisage assistants for Identifying values that seem to be out of range, e.g. missing value indicators; Identifying possible inconsistent coding of strings, such as different representations of acrynoms; indentifying rows that seem "jointly" unusual, i.e., that seem not to conform to statistical dependencies among the columns; and identifying distinct records that may refer to the same entity (record linkage).

Data monitoring. Once an analysis is completed, it may result in a machine learning model which will then be deployed on a data stream. Once a classifier is deployed, it must be monitored to detect shifts in its input distribution, such as change points, covariate shift, and concept drift. This includes the problem to suggesting when/how often new data points should be labelled for monitoring. Even for data analyses that do not involve deploying a "production model", there is often a need to run an analysis periodically on updated data, such as re-running last quarter's analysis on this quarter's data. In such cases, we might believe that the two data sets come from approximately the same distribution and format, so we can exploit divergences from this assumption to indicate potential issues with data quality. As part of the Wrattler effort, we have introduced the *data diff* problem [8], which aims to improve the process of data wrangling by returning a report of the differences between two data sets.

Exploratory visualization to summarize data sets, both the target data of the analysis, and the results of transformations that the analyst creates. Potential assistants here include ones to summarize most representative rows of a data table, and automatically choosing appropriate interactive visualizations, such as scatterplots and bar charts, with styling decisions such as axes automatically chosen to be most informative.

Performing analytics. An increasing amount of attention is being given to the AutoML challenge [3], which focuses on automatically choosing a classification methods, feature representations and hyperparameters for a given classification task, without the need for human intervention.

Debugging and interpreting predictive analytics. Every time a new machine learning model is trained, the first question an analyst has is often why its accuracy is not better. But debugging predictive analytics is notoriously difficult, and the remedies indirect and expensive, e.g., labelling more data, adding more features, using a more complex model. Debugging and improving models is a rich area for potential assistants. There is potentially low hanging fruit available in how to aid the tasks of error analysis, exploring the predictions of the model on validation data. Some early work in this direction is [**?**]. There is a rich and growing literature on interpreting and explaining the predictions of a model [2, 6, 9**?**]. Two potential directions from data are debugging by using "training set blame": Given an error in the validation data, what examples from the training set caused me to make that error? This might build on the work of [**?**]. Another potential avenue is explaining differences between predictions: why was instance 42 treated differently than instance 24601, even though they appear highly similar to a human?

5 Data store

The data store links together individual components of Wrattler. It stores the raw input data for data analysis such as downloaded web pages, raw data frames and also data frames produced as the result of running a script or invoking an AI assistant. In addition to data, the data store also stores multiple versions of the dependency graph created by Wrattler.

In the initial version of Wrattler, we restrict the data storage to raw input files and data frames, however we note that it would be desirable to include other data structures such graph or network data.

In this section, we focus on the initial functionality that is required from the data store. We focus on the public interface exposed by the data store, but We do not discuss the technolgy used for data storage – this can be decided during the development.

Data frames stored in the data store can be annotated with metadata. Such metadata can communicate information between multiple AI assistants. For example, first AI assistant can infer that a type of a pair of columns represents GPS coordinate and produce a data frame with annotations specifying this information. A second AI assistant can then use this information and automatically provide a data visualization in the form of map. In the rest of this section, we discuss the structure of those annotations.

5.1 Semantic annotations

As noted in Section 3.2, annotations can be attached to individual cells, entire columns and rows, but also combinations of multiple columns or rows. As an example, consider the following small table that records data on broadband speed (and illustrates a number of real-world problems):

ISP	Postcode	Download
Virgin	CB43BE	46.34
BT	CB43PT	N/A
Virgin	NW53HL	41.2
BT	Tottenham	16.4

If we run an AI asistant that performs probabilistic type inference (ptype) on this data set, it will annotate the table with a number of information:

– aa

Data store provides a way of annotating cells, columns, combinations of columns and rows. For example, a column may be annotated with a type that indicates that its values are a mix of postcodes and city names. A cell can be annotated with an information saying that this value is more likely a postcode than a city name. We also need to support annotaitons on a combination of columns, because three columns can jointly represent an address. A row can be annotated with a provenance – for example, when merging rows from two data sets with different sources.

5.2 Annotation format

We want a lightweight annotation format so that implementing an AI assistant or a language runtime that needs to communicate with the data store is not too hard. Our annotation format will follow something like http://schema.org, which defines a hierarchy of entity types and has a simple way for adding annotations to things. We will need to define our own hierarchy for schemas for data science.

5.3 Schemas for data science

This hierarchy will include some basic things that the data store understands and can convert between (integers, floats, unbounded size integers, ISO dates, etc.)

In addition, we will define some purely semantic annotations that are commonly useful and we will define a hierarchy for those. This represents things like addresses, postcodes, countries, cities, etc.

Everyone can define their own schemas (like schema.org) by defining their own namespace (URL) and providing entities with their custom namespace. The data store will simply save and return this information - without doing any checking.

5.4 Content negotiation

The only bit where the data store does something clever is that it will be able to return data in a format that the client asked for (a bit like HTTP negotiation). The client can say, "I only understand semantic information defined by these schema namespaces" and the data store will do its best to return data in the required format.

For simple types (e.g. dates) that the data store understands, it will convert data accordingly. For other semantic information that the data store does not understand (e.g. information that a certain cell is missing with a probability p), the data store will filter out the extra information and just return the raw number.

References

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment* 9, 12 (2016), 993–1004.

[2] Finale Doshi-Velez and Been Kim. 2017. *Towards A Rigorous Science of Interpretable Machine Learning*. Technical Report arXiv:1702.08608.

[3] Isabelle Guyon, Imad Chaabane, Hugo Jair Escalante, Sergio Escalera, Damir Jajetic, James Robert Lloyd, Núria Macià, Bisakha Ray, Lukasz Romaszko, Michèle Sebag, Alexander Statnikov, Sébastien Treguer, and Evelyne Viegas. 2016. A brief Review of the ChaLearn AutoML Challenge: Any-time Any-dataset Learning without Human Intervention. In *Proceedings of the Workshop on Automatic Machine Learning (Proceedings of Machine Learning Research)*, Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.), Vol. 64. PMLR, New York, New York, USA, 21–30.

[4] Ihab F. Ilyas and Xu Chu. 2015. Trends in Cleaning Relational Data: Consistency and Deduplication. *Foundations and Trends® in Databases* 5, 4 (2015), 281–393.

[5] Nicholas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. 1997. Wrapper Induction for Information Extraction. In *IJCAI*.

[6] Zachary C. Lipton. 2016. The Mythos of Model Interpretability. In *ICML Workshop on Human Interpretability in Machine Learning (WHI)*.

[7] David Pinto, Andrew McCallum, Xing Wei, and W. Bruce Croft. 2003. Table Extraction Using Conditional Random Fields. In *ACM SIGIR Conference on Research and Development in Information Retrieval*.

[8] Charles Sutton, Tim Hobson, James Geddes, and Rich Caruana. 2018. Data Diff: Interpretable, Executable Summaries of Changes in Distributions for Data Wrangling. (2018).

[9] Kai Xu, Dae Hoon Park, Yi Chang, and Charles Sutton. 2018. Interpreting Deep Classifiers by Visual Distillation of Dark Knowledge. (2018).