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.

A data science environment of the future needs to support an interaction model that keeps the human in the loop, allows an efficient human-computer interaction and can be extended with new AI asistants that provide advice about data. Such advice needs to be explainable to the human user, as well as machine-readable by other components of the system. In this report, we present Wrattler, a system that is:

Interactive. 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. 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. 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. 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.

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).

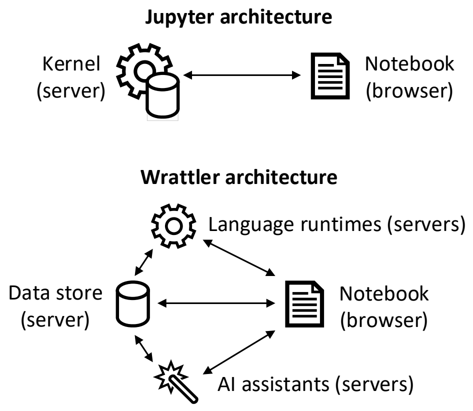

Figure 1. In notebook systems such as Jupyter, state and execution is managed by a kernel. In Wrattler, those functions are separated and enriched with AI assistants.

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 data analyst. They help to write data cleaning scripts or annotate data in the data store 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.

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.

The structure is ilustrated in Figure 2. Top-level nodes (squares) represent individual notebook cells. Code (circles) is represented either as a single node (R, Python) or as a sub-graph with node for each operation (DSLs 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 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 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.

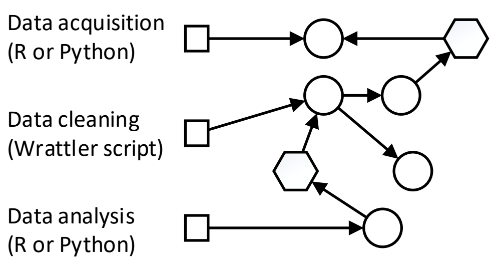

Figure 2. Sample dependency graph. Data acquisiton and analysis are represented as opaque R/Python code cells. Data cleaning is written in a domain-specific langauge understood by Wrattler and represented by multiple cells. Latter two cells depend on data frames exported by earlier cells.

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.

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.)

3.4 AI assistants

The Wrattler architecture enables integration of AI assistants in a number of ways. First, AI assistants have access to the data store (and the dependency graph) and can use those 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, transofrmation or visualization. An AI assistant can work in two ways:

- When invoked, the assistant creates a new frame in the data store with additional information, e.g. probabilistic type inference (ptype) will annotate columns (and possibly also cells) with their probabilistic types.
- The assistant defines a DSL for the task at hand. When invoked from a notebook, it guides the user in creating

a script in the DSL that performs the required operation (such as extracting data or reformatting a data frame).

The second case is illustrated in Figure 1. An AI assistant reads data from the data store and returns suggestions in a small DSL to the notebook, which the user can review (using a live preview mechanism) and accept. For example, an AI assistant based on datadiff can suggest the following script (written using the Wrattler DSL framework):

```
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 above script is constructed interactively. When the user types the first line and types "." the datadiff assistant offers the most likely patches and the user can choose. While choosing, a preview of the result is computed on-the-fly, giving the user an immediate feedback. Finally, the script is represented in a fine-grained way (second cell in Figure 2). Once it is interactively constructed, it can be translated to multiple supported languages such as R, Python or JavaScript.

4 AI assistants

An AI assistant guides data analysts through a single data collection, integration, preparation, analytics, or reporting task. 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

AI assistants have access to information in the Wrattler data store and can use two separate aspects for their guidance:

- 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 idea is that assistants may need to know the context of transformations that have already been performed in order to decide what to suggest next.

The assistants should be self-contained in the sense that they interact only through the data store. This restriction is designed to encourage the Unix "do one thing well" design for assistants.

There are two important design aspects of AI assistants. First, they always produce code, rather than being a black-box that transforms data. Second, they are designed to be used interactively.

Domain specific languages. The output of an assistant is code in a small domain-specific language designed for the task that the AI assistant is solving. The code can be used in two ways – during the interaction with the AI assistant, the code is exposed through a Wrattler DSL framework as discussed in Section 3.4. For production use, the code is translated to a language like R or Python. A typical code produced by an AI assistant will implement a data transformation or produce a visualization.

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.

However, at the same time, our data preparation tools should be usable in "batch mode" by analysts who already know what they want to do and don't want to mess with a wizard. We should aim to produce the equivalent of scikit-learn for data preparation.

The output snippet should be readable, and something that a data analyst can look at to verify that the transformation accomplishes what they had wanted. As far as possible, the snippet should be "reusable" in a sense that it could be copy-pasted into another script that was processing a similar dataset and would still work. You can think of a snippet as something that a person might write in a single cell of a Jupyter notebook.

4.2 Types of AI assistants

AI assistants can be imagined for every step of the data analytics process. We attempt a fairly c omprehensive list of the data analytics tasks for which can imagine targetting assistants. Then, for each task, we list a few examples of specific agents. Even though the list of agents is incomplete, it is still too long to implement fully in the current project.

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, more information about countries can be offered.

Data integration. Assistants can identify pairs of variables across multiple data sources that are likely to be coding the same variable.

Schema inference. Inferring types of variables and doing this probabilistically.

Data parsing. Quick tool for specifying a scraper for web pages (related to wrapper induction, Sumit Gulwani work, possibly this paper Robust and Noise Resistant Wrapper Induction Assistant to automatically infer format of csv files Assistants to extract tables from pdf's, html Identify which values are likely to be "special", like missing data markers

Data cleaning Identifying values that seem to be out of range, e.g. missing value indicators Identifying possible inconsistent coding of strings Identifying potential functional dependencies (1:1 correspondences) and returning violations Manual cleaning of subset of data (a la Sample Clean) Identifying rows that seem "jointly" unusual, i.e., that seem not to conform to statistical dependencies among the columns Identifying distinct records that may refer to the same entity (record linkage)

Data monitoring. Data diff: Returning a report of the differences between two data sets Monitoring classifier to detect shifts in its input distribution Detecting change points and covariate shift Montioring classifier to suggest when/how often new data points shoud be labelled for monitoring

Exploratory visualization. Summarizing most representative rows of a data table Automatically styling scatterplots and bar charts

Performing analytics. Which classification methods should I use? e.g., by search through similar problems on Kaggle and Open ML (a la AutoML) Simiarly for feature selection methods and so on Hyperparameter exploration

Debugging and interpreting predictive analytics. Debugging predictive analytics, e.g., deciding whether and how to add more features Error analysis assistant: Summarize the output on validation data Error analysis based on visualization of classifier score (see work of Saleema Amershi) Tools for explaining the predictions of a model on validation data (a la LIME) * Research directions: Improved model families for the interpretable explanation. Decision lists on marginal distribution rather than joint "Training set blame": Given an error in the validation data, what examples from the training set caused me to make that error? Explaining differences between predictions: why was instance 24601 treated differently than 24675?

4.3 Possible AI assistants

Data Parsing. Simple scraper assistant: Simple information extraction wizard. Create a data table based on HTML pages that have been automatically generated, like product review pages, for which a simple pattern on the HTML tree will be sufficient to extract the desired information. This is called wrapper induction in the information extraction literature.

* *Input*: Web page that contains information in unstructured form, such as a list of search resuts or an HTML list.

* *Output*: A data frame (i.e. a relational table) that contains the information that has been extracted from the web page.

* *User interaction*: User needs to initially highlight and specify which items to be scraped. Need to summarize the results of the scraping and highlight cases which might require correction from the user. Update scraper accordingly.

* *Potential method*: A wrapper is a path through the HTML DOM. Use a language (like XPath) for navigating through the DOM. Search for the shortest path that retrieves the labels provided by the user.

* *Research issues*: Not clear, given the large amount of related work in this area. We would need to read around carefully before we can identify opportunities. Possible opportunities include: (a) Using a language like [Scrapy](https://scrapy.org/) to induce the wrappers; (b) Using a language model over the scraped values to fine-tune the wrapper (e.g. if a wrapper returns both dates and cities, it's probably wrong), or (c) using a probabilistic model over the program representation of the wrapper to focus attention in the search for good wrappers. Perhaps there are enough scrapers on line (e.g. Github projects that use scrapy) that you can get a training set for this. * *Related work*: Gulwani's work on data wrangling, lots of work on wrapper induction. There is a lot of work on this. Scrapy is a popular open source framework that several of my BSc/MSc students have found useful.

Data cleaning. *Inconsistent string assistant*: Identify potential typos in string data. The idea is to look for strings that do not occur commonly in the data set, but are very close in edit distance to strings that do occur commonly.

* *Input*: A column of a data table whose values are strings, like addresses, cities, states/provinces, occupations, and so on.

* *Output*: A list of suggestions for strings that should be renamed: e.g. '[("Stocland", "Scotland"), ("Untied Kringdom", "United Kingdom")]'

* *Potential method*: Simplest cut: Define a threshold δ on Levenshtein distance, and $0 < \alpha < 1$ on the number of occurrences. Return the set of all strings y that occur less than αN times in the data set but where there is a corrected string x which is less than δ in string edit distance from y. More ambitious: Noisy channel model for correction. Learn a probability distribution $p(x)$ over the strings in the column, and define an error model $p(x'|x)$. Then search for ways to transform the dirty column x' to a clean version x in such a way to maximize $p(x|x')$. This formulation deserves some more thought, though, before we try to implement it.

* *Research questions*: Need to connect this to existing concepts in the databases and data mining literature, such as U-repair.

* *User Interaction*: The user should be able to reject suggestions in the output list, and this should rapidly update the models used in such a way that similar suggestions to

the bad ones are also removed from the list. A noisy channel model would be nice for this, not sure about the edit distance one.

Exploratory visualization. *Data table summarization assistant*: Displays a short subtable of the most "representative" columns of a given data frame. The number of rows in the summary should be small enough to fit on a given screen. Essentially an alternative to the ‘head()‘ function in pandas/R.

Input: A data frame, along with type information (continuous/categorical) for each of the columns. Desired number k of rows in output.

Output: A list of k indices of rows in the dataframe

Potential method: Simplest possible implementation: Run k-medoids and return the medoid values.

Research questions: First, are there research questions here, or is this so simple that it is part of the infrastructure? If k-medoids is so simple for this, why doesn't everyone also do it? How to evaluate whether the summary is good? Maybe there are examples of data sets where medoids is not good. For example, in the Karpathy ICML example, medoids would not make sense, listing the institutions that have the most papers is most informative, because it's a "long tail" type of column. How do you tell "long tail" data from "just show the clusters" type data? Perhaps a model-based framework could distinguish? Maybe you want to summarize the ways in which two data sets differ? Is the best summary the cluster centroids or the "top k" along some value? * *User interaction:* I'm not sure what interactions are enecessary. Users could ask for more rows, mark two items as similar, or mark an item as uninteresting. Or perhaps better would be to allow drill down, i.e., to make it easy to explore the clusters represented by each example.

5 Data store

TP: My rough notes from discussion with JG

Data store stores data frames with metadata. This is a good start for data science work, because most languages can provide mappings to/from data frames.

5.1 Semantics annotations

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.