Jupyter on Steroids

ABSTRACT

This abstract is way too long. Most of it should be in the introduction. We should try to keep it under half column with a flow like: issue -> "This paper proposes..." -> Results if any.

A plethora of tools and frameworks has been proposed, by the database and systems community, for streamlining the admittedly arduous tasks of data retrieval, data curation and data visualization. Since these tasks, typically, have to be carried out in most data science projects, a significant portion of the afore-mentioned tools, focuses on providing a simple-to-use way for performing them. Despite the very promising results, however, these tools often seem to limit flexibility as they only focus either on a predetermined set of use-cases or on only a small fraction of tasks of a typically much bigger pipeline. This lack of flexibility often pushes code-literate data scientists to the tested and proven path of interactive notebooks such as Jupyter.

Interactive notebooks allow the use of popular, high-level and highly expressive imperative languages, such as python, for analyzing data and composing the results into an easily readable notebook-like interface. Due to the wide popularity of such languages, there is also a huge collection of third-party libraries that can be used by data scientists as building blocks of a much bigger analytical process. Furthermore, the web environment of notebooks enables collaboration between data scientists, since it allows them to directly interact with the user interface in order to develop and run code, process data, generate visualizations, and lastly, compose their findings into an interactive (and re-runnable) report-like page, that contains code, visualizations and textual description of the analysis.

However as we show in this work, interactive notebooks are still suboptimal with regard to ease of use and interactivity. Setting up notebook environments and dependencies, obtaining and combining data and generating the respective visualizations, requires technical knowledge that often exceeds the skill-set of a typical data scientist. Lastly, while such notebooks support the generation of interactive visualizations, this interactivity is not an integral part of the data analysis process.

To address these issues, we propose extending interactive notebooks with the ViDeTTe framework. Our proposed extension adds a template language that operates on JSON structures and facilitates – among others – communication with databases and the creation of interactive plots. In this paper we present a sample workflow that highlights the potential of such an extension for use in data analysis tasks.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. INTRODUCTION

User-friendly data analysis tools and frameworks often provide limited flexibility, as they usually focus on a predetermined set of use/analysis cases or a small fraction of the typically large data analysis pipeline. This lack of flexibility often pushes code-literate analysts towards the use of interactive notebooks such as Jupyter.

Interactive notebooks allow the use of popular, high-level and highly expressive imperative languages, such as Python, for analyzing data and composing the results into an easily readable notebook-like interface. Due to the wide popularity of such languages, there is also a huge collection of third-party libraries that can be used by data scientists as building blocks of a much bigger analytical process. Furthermore, the web environment of notebooks enables collaboration between data scientists, since it allows them to directly interact with the user interface in order to develop and run code, process data, generate visualizations, and lastly, compose their findings into an interactive (and re-runnable) report-like page, that contains code, visualizations and textual description of the analysis.

However as we show in this work, interactive notebooks are still suboptimal with regard to ease of use and interactivity. Setting up notebook environments and dependencies, obtaining and combining data and generating the respective visualizations, requires technical knowledge that often exceeds the skill-set of a typical data scientist. Lastly, while such notebooks support the generation of interactive visualizations, this interactivity is not an integral part of the data analysis process.

We address these issues, by extending interactive notebooks with the ViDeTTe framework. ViDeTTe notebooks support a new template language capable of facilitating common data analysis tasks. The main contributions of this extension are:

- *Expressive template language:* Prior work, treats a

page as a database view. Building on that, our template language goes beyond SQL query and view definition in both style and fundamental expressiveness. It is a mixture of query as well as web templating language that works on ordered (arrays) and semi-ordered (JSON) data.

- *Easy data retrieval:* Our framework supports communication with all major database types, such as Postgress, MongoDB, SQL etc, eliminating the need for individual DB drivers. Furthermore, using ViDeTTe, user access credentials for the database server(s) are stored in a configuration file, eliminating the embarassingly insecure practice of typing usernames and passwords in notebook cells vidible to everyone.

- *Inline JSON operations:* The primary data structure used in ViDeTTe is JSON arrays. ViDeTTe combines the intuitive nature of JSON with the ability to write inline JSON operations, resulting in a clean, structured and readable code.

- *Variable binding:* Analysts can easily "bind" variables using our template language. "Binding", results in automatic re-execution of notebook cells that contain those variables upon a change. ViDeTTe will trigger execution of the appropriate cells without any extra coding effort. As we show later, combined with inline JSON operations, binding becomes an incredibly versatile tool.

Throughout this paper, we present ViDeTTe via an example data analysis: We assume a scenario where a data analyst must (a) retrieve (and curate) website access information from a database, (b) plot a histogram of access count Vs timestamp, as well as a histogram of access count Vs age groups, and (c) be able to select a time range from the first histogram using mouse input and automatically filter and display data in the second plot to present age-group-based activity within that time frame.

The remainder of this paper is organized as follows: Section ?? discusses the architecture of the ViDeTTe framework. Due to space limitation, we focus on those aspects that can be useful as a notebook extension. Section ?? presents our example data analysis. Finally, Section ?? concludes the paper.

> *Ok I thought about it and I propose the following modifications in the paper structure (Nothing major - just moving text around to make it more readable): We begin with a good discussion about the vidette features that can help notebooks in section 2. We then show the entire walkthrough in one section (Section 3). By now, the reader knows what vidette can do so it will be easier to follow and understand the code we show.*

2. ViDeTTe ARCHITECTURE

SOME INTRO TEXT (1-2 paragraphs)

2.1 Source Wrappers

2.2 Visual Units

2.3 Template(?)

2.4 Interactive Interface

> *Costa, the algorithms you have for this section are completely unreadable. I don't think it helps anyone who reads it and it will take forever if we actually want to explain it in text. I assume (hope) these are algorithms that run in the background (under the hood vidette functions) and not something the analyst needs to write. If that's the case, let's describe the operation of those functions logically in text. If the analyst actually needs to write that code, I don't think we have a submit-able paper here.*

3. WALKTHROUGH EXAMPLE

DESCRIBE THE PROBLEM AGAIN WITH MORE DETAIL

3.1 Data Retrieval

Just move what we already have in the Data retrieval section here.

3.2 Simple Visualization

Here we show simple non-interactive visualization using vidette. We get a chance to say how intuitive everything is and how awesome it is that we can inline functions in these units.

3.3 Interactive Visualization

Here we show the modifications needed to go from simple to interactive visualizations. This way we point out that going from naive to advanced visualizations take minor modifications in the units and everything is taken care of under the hood.

4. DATA RETRIEVAL

> *It might be a good idea to merge these 3 sections into 1 called "Walkthrough example" or something like that. Depends on how large these sections become.*

Data retrieval is one aspect where ViDeTTe shines. It enables the analyst to write inlined, source specific database queries in order to retrieve data from the respective database servers. In a Python implementation, analysts need to perform a series of tasks before being able to retrieve data. Some of these tasks potentially fall way beyond the skill set of the average data analyst.

Costas: We should also say that we simplify file imports. For cases, when the data analysts wants to use csv or json files from their local computer, they can use an import button, which is part of our UI, to upload them to the notebook server. After that step is completed we trigger a function that automatically converts this file to JSON and we assign it to a ViDeTTe variable, so that it can be used in the notebook. Currently, this task requires ssh-ing or scp-ing the files to the notebook server, and potentially moving files to the directive/file-system that is seen by the notebook, which also requires system administrator-linux knowledge

```
sources : [ {
  driver   : "postgres",
  host     : "edu.db.domain",
  expose   : [ {
   schema : 'website_info',
   tables: [visits, page_views] } ]
  port     : 5432,
  username : "dbadmin"
  password : 'myP@ss'
}]
```

(a) DB Access Configuration file

```
<% let readings =
   SELECT count(time) as visits, time
   FROM (SELECT * FROM page_views pv
               join visitors v
         on pv.v_id = v.vid) AS joined_table
   GROUP BY time
   ORDER BY time ASC %>
```

(b) Data retrieval

```
readings = [
   {visits: 15, time: '08:00'},
   {visits: 10, time: '09:00'},
   {visits: 25, time: '10:00'},  ...]
```

(c) Query Result

```
<% unit highcharts %> {
  title: 'Visitor information' ,
  xAxis : {
    labels : ['08:00','09:00'...],
    min : '08:00'
    max : '22:00'
  }
  series: [{ data: [ {y:15}, {y:10}...] }]
} <% end unit %>
```

(d) Unit with evaluated unit state

```
<% unit highcharts %>
{
  title: 'Visitor information',
  type: 'line'
  xAxis : {
    labels : [
      <% for reading in readings %>
        <% print reading.time %>
      <% end for %>],
    min : <% bind min_time %>,
    max : <% bind max_time %>
  }
  series: [{
    data: [ <% for reading in readings %>
        {
          y  : <% print reading.count %>
        }
      <% end for %> ]
  }]
}
<% end unit %>
```

(e) Template `temp_view`

Figure 1: Template, template instance, and UAS configuration file for the running example

First, our analyst needs to install and configure database drivers. This step will either introduce a dependency between the analyst and some system administrator, or the analyst will need to have the required access rights in order to perform the installation. The later can be a security concern or can lead to corrupted systems if not performed correctly. The complexity of this step increases as the number of different database systems, our analyst wants to access, increases. More specifically, for cases when an analyst needs to access data stored in a MongoDB and a MySQL database, two drivers will need to be installed.

Once the system is configured, the analyst needs to read lengthy documentation pages and stackoverflow posts in order to properly issue queries to the databases via imperative code by using the library API, then consume the results by using internal constructs of the API, and potentially convert them into a format that assists in data processing. Lastly, the analyst will have to convert the format of the data again in order to feed them into the respective visualization library.

Finally, when implemented in a Jupyter notebook, the analyst's credentials for the database server might lie on plain sight to anyone who has access to the notebook, disrupting the valuable ease of results communication through notebooks.

Costas: Some of these steps, mentioned up to here are common for the data retrieval and visualization stages (e.g installing database drivers and visualization libraries, reading documentation pages to figure out how to use the and so on. Can we group them into a single section for ipython and then explain how things differ in ViDeTTe?

ViDeTTe addresses each one of the aforementioned imperative code issues efficiently: The analyst generates a configuration file, containing information required for establishing a successful connection with the respective database systems. Figure 1a shows an example of the configuration file that is used for connecting to a postgres database that contains the data that will by used in our analysis. The configuration file must contain the type of the database system, the host name, port and the credentials that will be used for obtaining a connection. Additionally, it contains the database tables that will be accessible by queries, only the tables that are explicitly defined in the configuration file will be accessible in the notebook. After this configuration file is imported, (which is done directly from the notebook's UI), it will be hidden from the UI and encrypted so that this information is no longer visible to the notebook reader. Furthermore, as an added benefit, the schema of the accessible tables will be displayed on the UI, this allows the notebook user to get a glimpse of that information when generating the queries.

Once this step is completed, the analyst is free to issue queries in order to access database systems. Figure **??** contains a sample query that is used in our analysis. The query joins the two tables: *visitors* and *page_views* on the id of the visitor, then groups the result on the *time* attribute and runs a *count* aggregate to count the visitors per unit of time. Lastly, it sorts the resulting dataset by time in ascending order. Figure 1c shows the result of the query. Notice that the result has been converted into a JSON array by ViDeTTe. In the next section we will show how to use this dataset to generate our first visualization, without the use of any imperative logic.

4.1 Source Wrappers

ViDeTTe contains a set of source wrappers that enables data retrieval from various different sources both relational and non-relational. These source wrappers come pre-installed with ViDeTTe so that the notebook user will not have to take on any system administrator duties. The user simply provides the configuration file and writes the query in the language supported by each database system. The source wrapper uses that information to connect to the respective database system, retrieve the requested data, convert it into JSON format and make it available to the user. For database systems that do not contain tables with schema, the notebook user has to provide the respective record container under the "expose" attribute of the configuration file (i.e if the user accesses a MongoDB database, she will have to declare a collections of documents instead of database tables). Additionally, in cases when the accessed database system does not have a schema the notebook will simply display a small portion of the dataset, thus assisting the notebook user to compose queries.

Costas: Maybe put the last two sentences in a side-note?

5. DATA CURATION

DATA CURATION

6. VISUALIZATION

Costas: Mention that the problems are: 1) Initial installation 2) Data conversions, 3)Use of imperative code, 4) Most of these libraries do not generate interactive visualizations

The main challenge when generating visualizations in interactive notebooks is the variance between the APIs of the visualization libraries. For each visualization library the analyst decides to use, she has to perform an installation of the respective packages (which requires advanced technical knowledge on its own), read lengthy documentation pages that dictate how to use functions provided by each library and then engage in tedious imperative programming in order to "massage" the existing datasets into a set of formats accepted by each employed API function.

After the data analyst has completed this tasks, she ends up with visual components that, while they can be informative, they cannot be used for further data exploration, without the need of additional imperative code. Specifically, depending on the employed visualization libraries, the data analyst either ends up with non-interactive visualizations, that are essentially static images, or with visualizations, that while they are interactive, this interactivity cannot be used as a part of the analysis, as it will not trigger any changes to other parts of the notebook. In either case, the analysis does not gain much value from such visual components.

6.1 ViDeTTe Visual Units

To shield analysts from the laborious task of constructing visualizations, ViDeTTe abstracts out each visual component as a ViDeTTe *visual unit* (or simply *unit*). In the eyes of the analyst a visual unit is simply a black box that takes as input a JSON value that describes the visualization, namely, unit instance. The visual unit internally uses the unit instance to invoke the appropriate renderer calls that will generate the expected visualization. As such a particular instantiation of the unit can be described as **<% unit** U **%>** v **<% end unit %>**, where U the type of the visual unit and v the JSON value corresponding to the input of the unit.

For instance, Figure 1d shows a unit instance of type `highcharts`. The unit instance describes all the information that will be displayed in the visualization (such as the title of the chart, the labels on the x axis and so on). Each visual unit, comes with a unit instance schema that describes the format of the unit instance.

6.2 Template

Costas: We should make it clear in the intro that ViDeTTe operates on a JSON datamodel. This means that source wrappers, visual units and templates all operate on the same model, no conversions are required from one to another. Additionally. since our datamodel is plain-old javascript objects, the analyst does not need to learn a new API in order to interact with our values.

Templates are declarative specifications that can be used to generate ViDeTTe variables or unit instances. The template language supports a set of *template directives*, all of which operate on the JSON datamodel. These directives are used to describe computation, define variables and set up data collection. Due to lack of space we do not include a figure containing a formal BNF grammar for the template language, instead we simply describe each of them in detail and provide a concrete example that illustrates their use.

Defining variables. A template may define variables that are added to the notebook's environment so that they can be used in subsequent computation. The **<% let** x **=** E **%>** directive defines variable x and assigns to it the result of the expression E. E can denote three types of expressions: path navigation on JSON data, invocation of a python function that performs computation by using as input and output JSON values or a source-specific language (such as a SQL query). For instance, the template shown in Figure 1b employs a **let** directive to create a variable `readings` containing the visitor information that will be displayed in the chart (retrieved from a relational DBMS through an SQL query).

Reporting syntax. Value assignments and iterations over collections are specified by using the **print** and **for** directives. The **<% print** E **%>** directive evaluates the expression E and returns the result, while the **<% for** x **in** E **%>** B **<% end for %>** directive specifies that variable x iterates over the result of E and, in each iteration, it instanciates the body B of the **for** loop. In Figure 1e, in lines 7-9 and 14-18 a **for** directive is used to iterate over the readings retrieved from the database and for each reading, it generates a new JSON value, with the use of the **print** directive. Particularly, in line 8 the template generates a string (the time label), which is added to the labels array (which contains the labels that will appear on the x axis of the chart). In lines 15-17 it generates a JSON object of the form {y: ...} and adds it to the data array. The data array in highcharts contains the points that will appear in the chart.

Collecting data. In addition to specifying how to compute a template instance, the template's **bind** directive allows the analyst to specify user input collection. Specifically, the **<% bind x %>** directive describes a two-way bind that will be created between the part of the unit instance appearing on its left side and to variable x. For instance, in Figure 1e in lines 10-11 we create a two-way binding between the min and max boundaries of the chart and the min and max values of

```
1  <% let age_groups =
2     SELECT agegroup, count(*) AS total
3     FROM (SELECT CASE
4      WHEN age BETWEEN 0 AND 9 THEN '0 to 9'
5      WHEN age BETWEEN 10 and 19 THEN '10 to 19'
6       ...
7      FROM (SELECT * FROM page_views pv join visitors v
8            on pv.v_id = v.vid where time BETWEEN
9            <%=min_time and <%=max_time) joined_data) jd
10    GROUP BY agegroup
11    ORDER BY agegroup ASC %>
```

(a) Data retrieval

```
1  age_groups = [
2     {age_group: '0 to 9', total: 12},
3     {age_group: '10 to 19', total: 67},
4     {age_group: '20 to 29', total: 84},  ...]
```

(b) Query Result

```
1   <% unit highcharts %>
2   {
3     title: 'Visitor information',
4     xAxis : {
5       labels : [
6         <% for v in age_groups %>
7           <% print v.age_group %>
8         <% end for %>]
9     }
10    series: [{
11      data: [ <% for v in age_groups %>
12          {
13            y  : <% print v.total %>
14          }
15        <% end for %> ]
16    }]
17  }
18  <% end unit %>
```

(c) Template `temp_view`

Figure 2: Template, template instance, and UAS configuration file for the running example

the time labels that have been retrieved (namely min_time and max_time). While the user interacts with chart he can select a particular region in the chart which in turn updates the values *min_time* and *max_time*. This event invokes an internal propagation algorithm (which will be described in the next section) which triggers the revaluation of the ViDeTTe statements that use the values *min_time* and *max_time* in other parts of the notebook.

7. INTERACTIVE INTERFACE

As the notebook reader interacts with a visualization the underlying visual unit triggers actions, which make the visualization behave in a certain way. For instance, as the user selects a particular region of the chart (shown in figure *Costas: ADD FIGURE*) by dragging and dropping the mouse over that particular area, she causes the visualization to zoom into the selected area. This behavior is dictated by the underlying visual unit which listens for events (or rather series of events) and actively cause mutations to the visualization. Other than mutating the visual layer, these units also cause mutations on the unit state. In this particular case, zooming into a region, causes the min and max values, shown in Figure 1d, in lines 5-6, to be updated accordingly. If the template language contains variables bound to these parts of the unit instance, ViDeTTe propagates that mutation to the respective variables. Furthermore a change propagation algorithm that operates on the entire notebook identifies all the statements that depend on those variables and triggers their re-evaluation.

For instance, the template shown in Figure 1e contains two variables, namely *min_time* and *max_time*, which are bound to the *min* and *max* unit instance variables respectively. These variables are later used in the parameterized query shown in Figure 2a which retrieves the users that visited the website in the time-frame specified by those two variables, groups the result in age groups and sums up the hits made by each age group. Figure 2b, contains the result of that query. The result of this query is assigned to variable *age_groups* which is then used to produce the bar chart appearing in Figure *Costas: add Figure* (the template shown in Figure 2c shows the template that created that chart)

ALGORITHM 1: MVVM-IVM

```
1  function change-propagation(Notebook N, Environment env,
   Set of Input Diffs D)
2      Set of output diffs D_out ← empty bag;
3      for each statement s in each block of N do
4          if s is <% let x = E %> and E contains directive:
           <%for v in E''%> or directive: <% print E' %> then
5              if There is Δ(t;p) ∈ D with t that matches E' or
               E'' then
6                  Reevaluate E and assign result r to x;
7                  Update env with new value of x;
8                  Construct Δ(x;r) and add it to D;
9          else if s is <%unit U%> B <%end unit%> and B
           contains directives <%for v in E''%> or <%print E'%>
           then
10             if There is Δ(t;p) ∈ D with t that matches E' or
               E'' then
11                 Reevaluate E and invoke unit U with result r ;
12     end
13     return D_out;
```

8. DISCUSSION

DISCUSSION

9. CONCLUSION

10. REFERENCES

ALGORITHM 2: MVVM-IVM

```
1  function MVVMIVM(Template T, Model M, Set of Input Diffs D_in)
2      Set of output diffs D_out ← empty bag;
3      for top-level directive d ∈ T do
4          d.path ← path to d in template T
5          if d is <% print Path ê %> then
6              Δ ← IVMPath(ê, M, D_in);
7              D_out = D_out ∪ {prefixDiffs(d.path, {Δ})}
8          else if d is <% for Var y in Path ê %>B<% end for%> then
9              Δ^type(t̂; p) ← IVMPath(ê, M, D_in);
10             if t̂ = empty then
11                 if type = update then
12                     Let T′ be the template rooted at d, where ê is replaced by p;
13                     p′ ← instantiateTemplate(T′, M);
14                     D_out = D_out ∪ {Δ^update(d.path; p′)};
15             else if t̂ = [k] and type = insert array then
16                 M′ ← M#{y ← p};
17                 p′ ← instantiateTemplate(B, M′);
18                 D_out = D_out ∪ {Δ^insert_array(d.path[k]; p′)};
19             else if t̂ = [k]ŝ then
20                 D′_in ← D_in ∪ {Δ^type(y ŝ; p)};
21                 D_nested = MVVMIVM (B, M, D′_in);
22                 remove first path step of each diff in D_nested;
23                 D_out ← D_out ∪ prefixDiffs(d.path[k], D_nested);
24             D_nested = MVVMIVM (B, M, D_in);
25             if D_nested not an empty bag then
26                 l ← length of array e targeted by Path ê
27                 for i in [0 ... l] do
28                     D_out ←
                       D_out ∪ prefixDiffs(d.path[i], D_nested);
29                 end
30     end
31     return D_out;
```

ALGORITHM 3: IVMPath

```
1  function IVMPath(Path Expression ê, Model M, Set of Input Diffs D_in)
2      for each Δ^type(t̂; p) ∈ D_in do
3          if t̂ is êŝ then
4              return Δ^type(ŝ; p)
5          else if ê is t̂ŝ then
6              if type = update then
7                  p′ ← navigate(p, ŝ) ;
8                  return Δ^update(empty; p′)
9              if type = delete then
10                 return Δ^delete(empty)
11             else
12                 p′ ← navigate(M, ê) ;
13                 return Δ^update(empty; p′)
14     end
15     return null;
```