DECEMBER 2019

THESSALONIKI – GREECE

INTERNATIONAL HELLENIC UNIVERSITY

Implementation of a web-based platform for Data Analysis, Visualization and Machine Learning

Konstantinos Mouratidis

SID: 8180012

Supervisor: Prof. Ioannis Magnisalis

Supervising Committee Members: Prof. Berberidis Christos
Prof. Stavrinides Stavros

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Data Science

DECEMBER 2019

THESSALONIKI – GREECE

Abstract

This dissertation attempt to explain the process of building an open-source platform for data science that allows users to easily upload their data or connect to data sources, visualize said data, and create machine learning models to solve common problems in the field. Everything is implemented with a simple-to-use web-based graphical interface, and is designed for usage/deployment in a cloud environment.

This project came about with the help of my supervisor and mentor, Prof. Ioannis Magnisalis, and fellow students – the team that helped in the implementation of the first version - Vaso Tsichli, George Katrilakas, and Chris Timamopoulos. And a shout-out to the thousands of contributors of the open-source tools that this project used, mainly of plotly/dash and the SciPy stack.

Konstantinos Mouratidis

December 1st, 2019

Contents

1. Introduction

1.1 Background

With the recent hype on Data Science, Machine Learning, and Artificial Intelligence a lot of new tools have been created to aid the practitioners of these three domains with their daily work, and a large array of older tools is still relevant and have resurfaced. The popular debates (e.g. “R vs Python”, “PyTorch vs Tensorflow”, and the list goes on) have become commonplace across social media platforms and due to self-posting on platforms like Medium[1] you can find literally thousands of articles on these topics.

Furthermore, each year seems to be having its own “hype word(s)”, usually spanning multiple subdomains of computer science and statistics. Some of them are “Deep Neural Networks” (and their derivatives), “DevOps” and “Micro-Services”, “Big Data”. These guide investments from various funds to startups, as well as hordes of young talent to corporate positions with cool titles. Usually, the hype is an overstatement and when the balloon pops, so do expectations.

Taking a broader perspective, most of what is done today is not much different from what used to be done in earlier “eras”. What has really changed is how almost all of these practices are now freely and openly available to the public, wrapped in nice programmatic interfaces (APIs) that significantly aid the development process. However, even that has its caveats.

Despite our infinite wisdom, we still fall victim to the most recent trend. Big Data however is not a trend, it is simply a niche. This is not to say that big data technologies are useless, just not necessary to a large part of those dealing with data. In fact, as lots of large scale companies have discovered, they need to collect their data, in so-called data lakes and warehouses, and then they find out that they have to mine them for insights,

[1] https://medium.com

and their computers are simply not strong enough. They need to resort to other measures, using clusters and distributed computing.

But not every company can support, or even needs, that sort of infrastructure scale. In my experience, so far, most datasets can easily be handled with an average gaming computer: the GPU would have to work for ~1 week to train on OpenPose or similar academic datasets, recommender systems algorithms trained on 80M ratings would fit nicely into a machine with 8 GB (perhaps using out-of-core algorithms) to 32 GB of RAM, and most datasets from companies that I worked with as a freelancer could be very easily handled by a moderately fast quad-core CPU (heck, I could even handle some of it with 2010's Intel Core 2 Duo E5500). Of course, even such a computer is not always accessible, and other problems exist too. In fact, data are never enough (Wu, 2019), and good and helpful datasets are always hard to find. Considering thus that there is still a market for small- and medium-sized datasets, it is no wonder that a lot of tools have spawned in order to aid those companies who have data but no analysts.

The employees tasked with the analysis face the same problem new practitioners in the field face: a lifted knowledge bar in the sense that they have to span multiple domains, but the various tools aim at easing the learning curve. However, if you are not experienced with analysis techniques, the tools don't really help you. The learning curve is not the only obstacle. But even analysts may lack expertise in specific data domains, and equipment. Even more often, the super-abundance of available tools has a negative effect: when shown a new tool a commonly occurring first though is along the lines of "why do we need another tool?", or when first starting "what tool should I choose?".

1.2 This project

This project is indeed yet another tool. Even worse, EDA Miner started as a joke, something evident from the name: after the course on Exploratory Data Analysis (EDA), and the popular tool RapidMiner. A particular pain point that got us into this was the mentality that certain closed-source, expensive, and over-hyped tools "are better". In this regard, a few stand out in particular; ordered by our increasing annoyance: SPSS, SAS, Matlab, and finally SAP. RapidMiner's core is open-source so it barely didn't make the list.

In all arrogance, the design philosophy was "to make a better version of the combined RapidMiner, SAP Hana, Weka, PowerBI, and Orange3" stack. We didn't, obviously, and we mention this again in the conclusions.

What EDA Miner aspires to be is a free and open-source tool that can not only handle basic tasks encountered in all of the aforementioned tools but will also *learn* from its users and actually incorporate machine learning in its design, so as to completely level the learning curve of data science tools. Also, the target audience is not only data experts but also (and mainly) anyone with at least a dataset and a question. In this era of buzzwords, what is probably truly unique is the "Cloud" and its benefits, so EDA Miner is a web-based application for anyone with internet access but without access to a strong enough computer (Park, 2019).

The development of EDA Miner, at start, followed a "monkey see, monkey do" mentality, simply copying functionality from other tools and tasks, and just doing what "felt right". Midway, it turned into a serious project that is influenced by previous work on "yet another tool", MIT's Data Integration and Visualization Engine (DIVE)[2], among others. Proper scheduling of features and a timetable have been introduced. The project adopted a Code of Conduct[3], created Contributing guidelines[4], and hosts documentation[5], everything online.

1.3 Licensing & private/public version

As mentioned previously, EDA Miner is a free and open-source tools, and the code is publicly available on GitHub. As for the license of the project, the initial team decision was to go with GNU General Public License v3.0[6], mainly for the condition of

---

[2] http://dive.media.mit.edu/
[3] https://github.com/KMouratidis/EDA_miner/blob/master/CODE_OF_CONDUCT.md
[4] See appendix 6, or https://github.com/KMouratidis/EDA_miner/blob/master/CONTRIBUTING.md
[5] http://edaminer.com/docs
[6] https://github.com/KMouratidis/EDA_miner_public/blob/master/LICENSE

“source disclosure”. This decision is not set in stone and, should the need or demand arise, it might be revised or changed but only in favor of a more permissive license.

The private version was dropped from v0.3 onwards, but why did we need a private version in the first place? The public version was mostly a mirror of the private one, the main differences being a PDF-printing functionality and a login menu. There were various reasons why we decided on this, and (nearly) doubling the commit count (contributions) on our GitHub profiles was definitely not one of them. First, at that point in time, these functionalities were neither well-developed nor well-tested and thus security was a serious consideration. Secondly, some of these, like the login, were originally meant to be used by the university (if and) when it is deployed on its servers, but we decided that other users might like this functionality too. Finally, it served as a great discussion and experimentation ground for the core team to work on still unsupported features, however since I’m the only contributor now this isn’t an issue either.

1.4 Structure

This report on the project is divided in five sections. This introductory chapter simply served as a place for me to complain and troll, but also explain some of the motivations behind the whole project.

The second chapter focuses on going over similar tools that are currently in the market and their features. We will review visualization and machine learning tools, both commercial and academic.

The third chapter goes over the tools this project uses. It discusses shortly what each library/framework/tool does, the community behind it, and more.

The fourth chapter is the essence of this report. We will go over all the major design decisions and considerations, how we used each tool, the overall architecture including discussions about its future, the project & directory structure, and more.

The fifth and final chapter contains the closing remarks: missed opportunities, future directions, suggestions for future contributors, and a few final words and comments.

1.5 Absence of definitions and terminology, and final remarks

I expect that people reading this dissertation are at least somewhat familiar with the field, probably due to a degree, online courses, or work experience, and thus I won't get into details about those concepts that I consider basic, e.g.: "regression", "classification", "visualizations", "databases", "object-relational mappers". It is a common adage that you "google it" when working on/with computers all day, so take responsibility for your knowledge gaps. Furthermore, academics and marketing may actually give a [thought] to distinctions between concepts like "data mining vs statistical/machine learning", but I will probably be using them interchangeably -often subconsciously- a lot.

This essay is not meant as a literary discourse, but rather a report / overview / manual / documentation / whatever. The literature review won't be the most comprehensive research (time and money limitations) on the matter, and due to the nature of the task there isn't much literature to begin with. A lot of the material referenced will be non-academic (websites, blog posts and articles, dreaded print-screens), and I loathe that just as much as the next person but it can't be helped.

As a final reminder, the main deliverable is the **software**, which I largely developed alone, often using new technologies I had no idea about, creating my own tools for a lot of tasks since available ones didn't work (e.g. python doc tools either fail or don't work well enough), doing all that within less than 6 months (including the 2 months spent on it before taking it as a dissertation project), and having to handle some important limitations of the framework used. To put things into perspective, this kind of software takes whole teams (of often extremely experienced people, as will become evident later) **years** to develop. I believe that most, if not all, of the code follows high coding standards and best practices, and the project as a whole has been structured equally well.

2. Literature (i.e. competitors') Review

As mentioned in the introduction, this is not the first software of its kind. A lot of others have made similar efforts at assisting the various data workflows, and this is not substantially innovative in terms of provided functionality. What EDA Miner offers is a simple (and yet unpolished) web-based interface for getting your data in, doing the standard visualizations, and offers some machine learning capabilities. One thing that is different, or so we would assume since we cannot possibly have knowledge of closed-source solutions and their code, is that our system is almost entirely able to learn from usage (we discuss this in the final chapter as it is yet unimplemented). Before going into what's unique and why, we need to go over existing solutions.

2.1 Overview of papers on various topics relating to implementations

2.1.1 Data type inference

Before any sort of action on data, any tool that aims at doing any sort of recommendation to the user for visualizations must first be made aware of the data types it has to handle. Different data types mean that different transformations (e.g. standardization for numeric data) and different visualizations (e.g. a pie graph on a float column becomes illegible) can be applied.

The DIVE paper correctly points to this, in the first part of their literature review (Hu, Orghian, & Hidalgo, 2018, p. 2). They mention that popular tools like Power BI and Google Data Studio implement this feature (and so does BigQuery[7]). An implementation they don't mention but we do use as inspiration is the one from Amazon[8]. In DIVE they define their model as an aggregation of these systems: they take them as features and try to detect both semantic and scale types, as well as relationships. This process is aided by

[7] https://cloud.google.com/bigquery/docs/loading-data#loading_json_files
[8] https://docs.aws.amazon.com/kinesisanalytics/latest/dev/sch-dis-ref.html, https://docs.aws.amazon.com/kinesisanalytics/latest/dev/sch-dis.html, https://docs.aws.amazon.com/machine-learning/latest/dg/creating-a-data-schema-for-amazon-ml.html

computing statistics. Finally, they mention a few programming libraries they use. One stands out in particular: **Messytables**, by Open Knowledge Labs[9].

Frictionless data, a listed project by Open Knowledge Labs[10] provides their own library for data type inference, conveniently named tableschema-py[11], which was initially used by us as well. However, this library fails to detect types when they appear in slightly non-ordinary formats, which do however appear in the specification (i.e. "1985-03" is parsed as string instead of *yearmonth*). This is probably because, just as the other libraries mentioned in DIVE, it uses a heuristic-based approach. We will discuss more on this in the third chapter about design decisions and how we deal with this particular issue, but here we would like to go over the underlying specification on Table Schemas (Walsh & Pollock).

We adopt a lot of the definitions about concepts present in the Table Schema specification, such as those about tabular data, physical and logical representation of data. For brevity, and to avoid confusion, we take *data type* to mean the logical representation, which we choose to categorize as one of: integer, float (*number*, in the specification), category, date (covering *date*, *time*, and *datetime*), text (*string*). The *descriptor* is something you might see us referring to as *(data) schema*, which we take to contain all the information about data types, sub-types, meta-information – optionally more – as per the specification.

2.1.2 Other guiding sources

Due to working in Python adherence to other, external, specifications may be loosened in favor to adhering to language-specific best practices and norms, such as:

- *PEP 8 – Style Guide for Python* (van Rossum, Warsaw, & Coghlan, 2001)
- *PEP 20 – The Zen of Python* (Peters, 2004)
- *PEP 257 – Docstring Conventions* (Goodger & van Rossum, 2001)

[9] https://github.com/okfn/messytables
[10] https://okfn.org/projects/
[11] https://github.com/frictionlessdata/tableschema-py

2.2 Overviews of competitors and other products

2.2.1 Visualization Software

There are probably a hundred different platforms and products for data visualization and business intelligence (the buzzword for data analyst of the previous decade) but three stand out in particular. These have been selected based on Gartner's continued faith in their companies' potential. For the past three years (see the diagrams from (Microsoft, 2017), (Microsoft, 2018), (Qlik, 2019) or the appendices) these are the only companies to have made it consistently in the "Leaders" quadrant, which is defined as a function of "completeness of vision" and "ability to execute". We will quickly go over them here, and also review visualization capabilities and libraries of the two most dominant programming languages, Python and R. You will also find a more complete comparison of these tools in Appendix 2.

Microsoft's Power BI[12]

One of the most famous software in the field, with 7.93% according to Datanyze's report (Datanyze), and certainly a great solution. It is an application running both locally and on the cloud with a very simple and polished graphical user interface (GUI). Since it is developed by Microsoft it provides seamless integration with their Office stack. Additionally, they allow imports[13] from various formats (see Figure 1) including but not limited to files (e.g. CSV, XML, JSON), databases (from various vendors and types, e.g. SQL Server, Oracle, IBM, postgreSQL, SAP HANA, Amazon RedShift, Google BigQuery, MySQL, and of course Azure), and about 40 services (e.g. Facebook, GitHub, MailChimp, Google Analytics, Zendesk). It should be noted that their team has done an excellent job in integrating R and Python as well as big data tools such as Spark and the Hadoop Distributed File System (HDFS), and getting data from a URL. After connecting to a data source (or multiple), optionally transforming and cleaning them, you can create stunning visualizations with minimal efforts, often not more than a couple dozen clicks

[12] https://powerbi.microsoft.com/en-us/
[13] https://docs.microsoft.com/en-us/power-bi/service-get-data

(see Figure 2). What makes Power BI differ, from say MS Excel, in terms of visualizations is the ability to create dashboards that you can also publish.

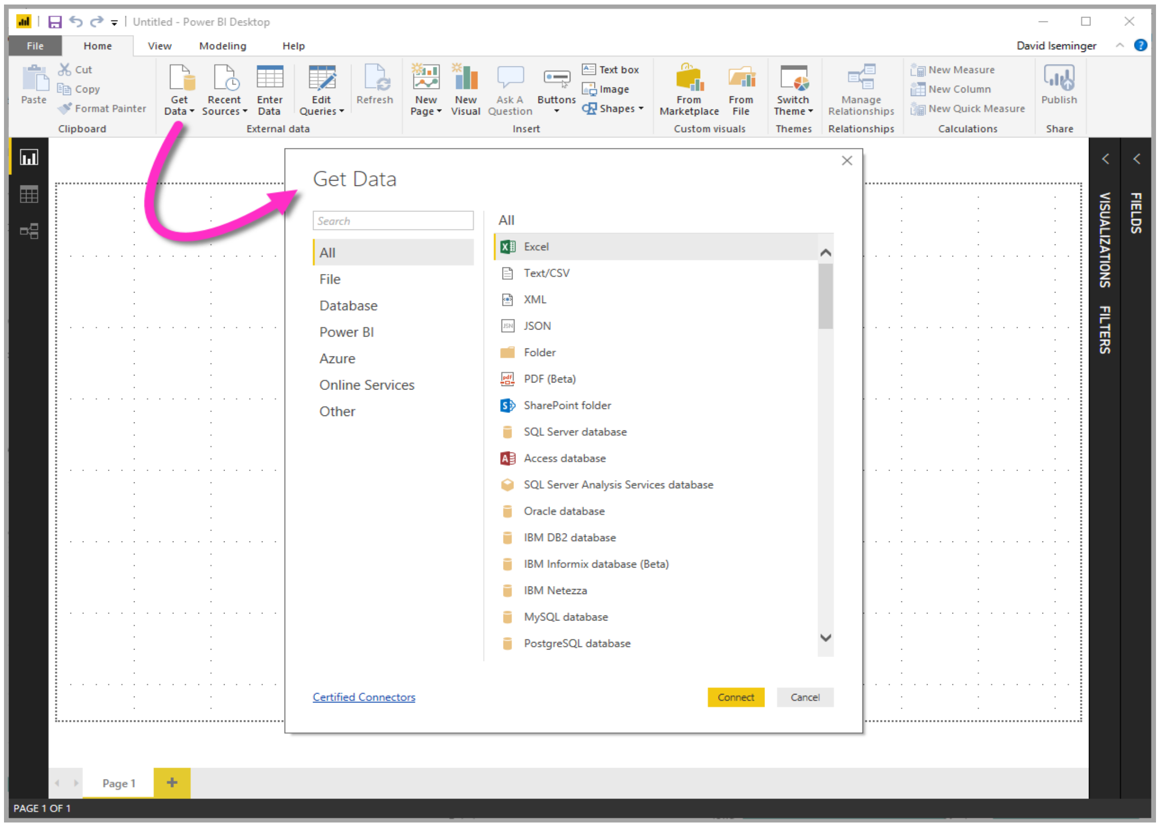


Figure 1 https://docs.microsoft.com/en-us/power-bi/desktop-data-sources

Power BI is also able to detect / infer your table schema (Hu, Orghian, & Hidalgo, 2018, p. 2), and offers a Mixed-Initiative Visualization System which is called Power BI Q&A (see Figure 3).

In all its greatness, it does not offer any Machine Learning, nor is it built for use with other systems such as Linux. This has another serious drawback, that it (seemingly) cannot be integrated with Docker (nish2288, 2018). Furthermore, while the *Power BI Desktop* version is free, the *Power BI Pro* ($9.99 per month per user) and *Power BI Premium* ($4,995 per "dedicated cloud compute and storage resource") are not[14]. A comparison between all three does not exist on their website, but some limitations of the free version include: 1) storage limit of 10 GB per user (which is pretty reasonable), 2) no API embedding (e.g. embedding visuals to PowerApps, SharePoint, etc), 3) no Peer-to-

[14] https://powerbi.microsoft.com/en-us/pricing/

Peer (P2P) sharing. You *can* however use Python, export reports to PDF, infer data schemata, and save/upload/publish your reports online (xello).

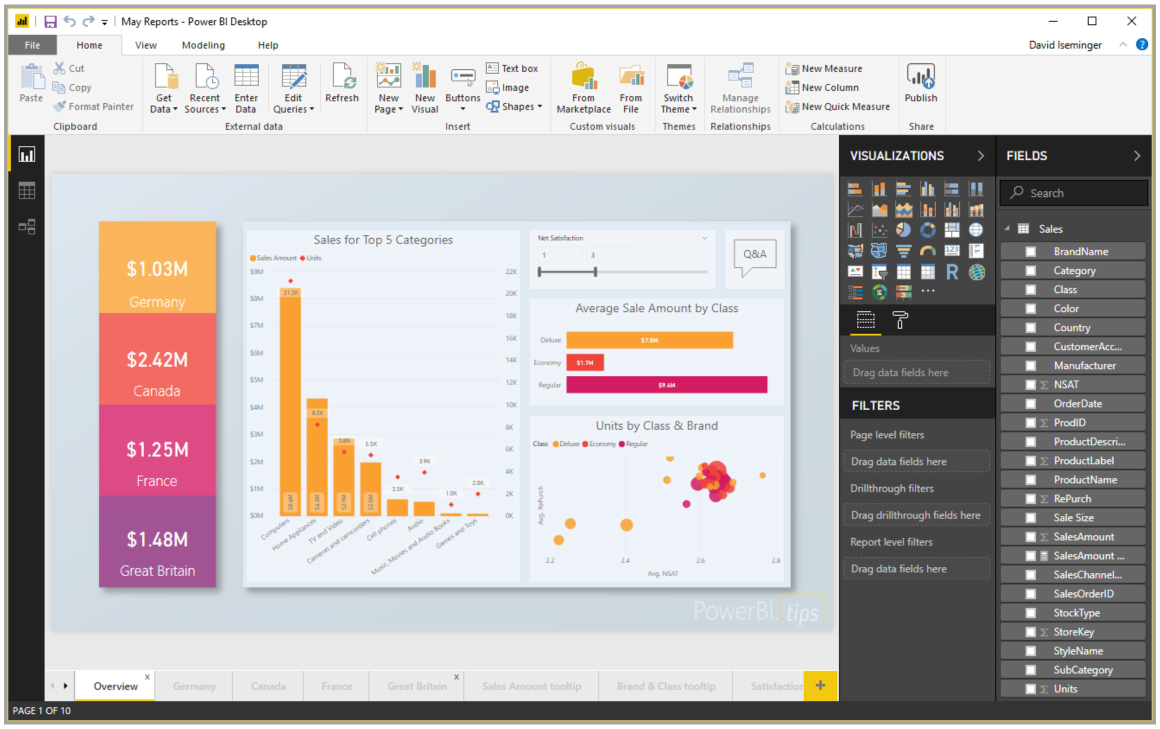
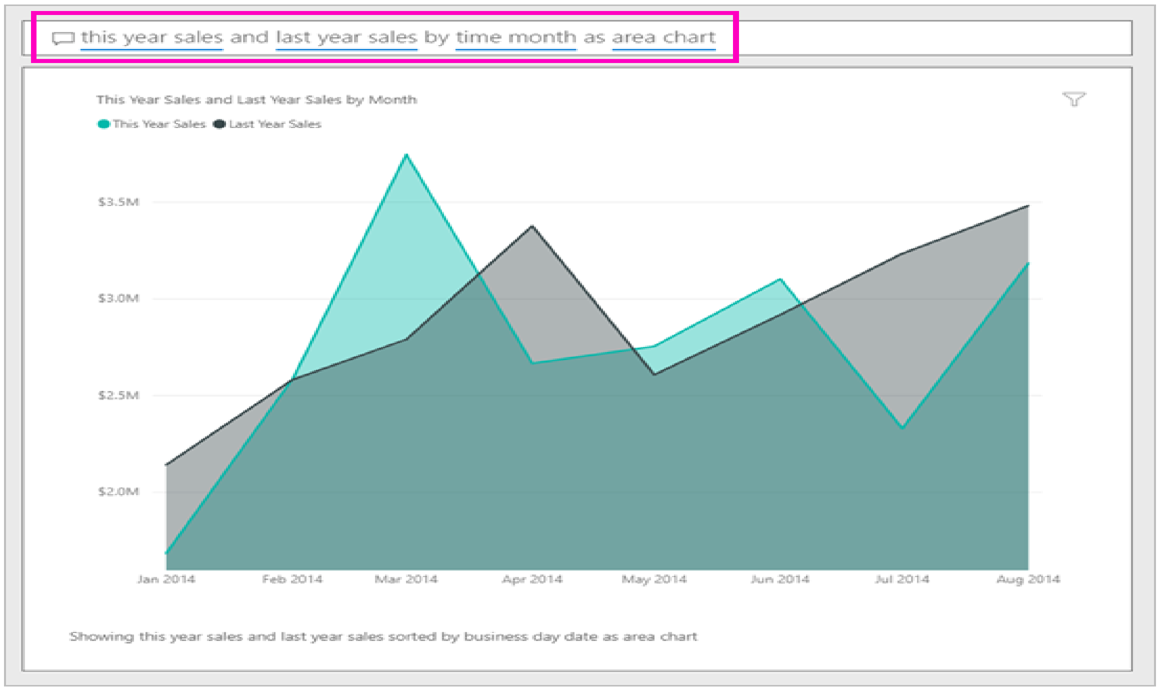


Figure 2 https://docs.microsoft.com/en-us/power-bi/desktop-what-is-desktop


this year sales and last year sales by time month as area chart

This Year Sales and Last Year Sales by Month

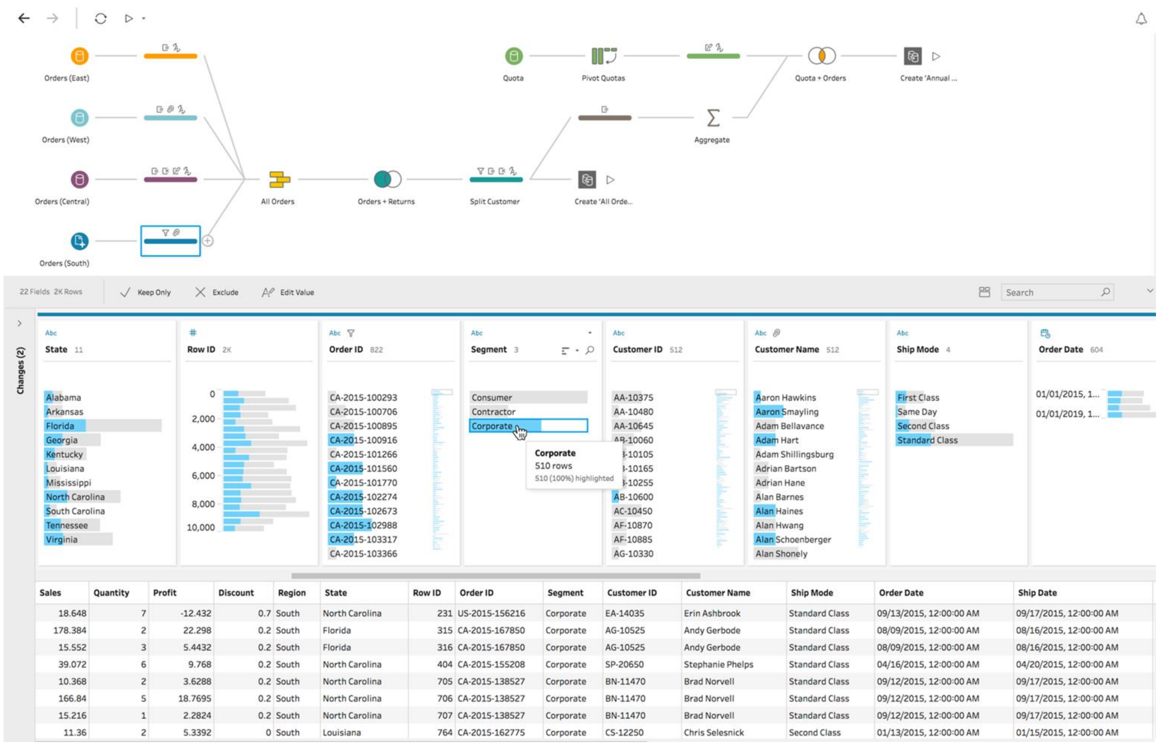
Showing this year sales and last year sales sorted by business day date as area chart

Figure 3 https://docs.microsoft.com/en-us/power-bi/power-bi-visualization-introduction-to-q-and-a

Tableau[15]

With a market share of 14.51% (Datanyze), the self-proclaimed "analytics platform that disrupted the world of business intelligence", released in 2003 (Levy, 2013), or about 8 years before Power BI, Tableau is a highly interactive data visualization software.

It features preprocessing functionalities for combining data sources, reshaping, cleaning, and inspecting the data. Similarly to Power BI it can connect to various data sources and providers including Amazon, Azure, IBM, Google, Cloudera, SAP, and various SQL and NoSQL servers[16].



State: Alabama, Arkansas, Florida, Georgia, Kentucky, Louisiana, Mississippi, North Carolina, South Carolina, Tennessee, Virginia

Row ID: 0, 2,000, 4,000, 6,000, 8,000, 10,000

Order ID: CA-2015-100293, CA-2015-100706, CA-2015-100895, CA-2015-100916, CA-2015-101266, CA-2015-101560, CA-2015-101770, CA-2015-102274, CA-2015-102673, CA-2015-102988, CA-2015-103317, CA-2015-103366

Segment: Consumer, Contractor, Corporate

Customer ID: AA-10375, AA-10480, AA-10645, AB-10060, -10105, -10165, -10255, AB-10600, AC-10450, AF-10870, AF-10885, AG-10330

Customer Name: Aaron Hawkins, Aaron Smayling, Adam Bellavance, Adam Hart, Adam Shillingsburg, Adrian Bartson, Adrian Hane, Alan Barnes, Alan Haines, Alan Hwang, Alan Schoenberger, Alan Shonely

Ship Mode: First Class, Same Day, Second Class, Standard Class

Order Date: 01/01/2015, 1..., 01/01/2019, 1...

Sales	Quantity	Profit	Discount	Region	State	Row ID	Order ID	Segment	Customer ID	Customer Name	Ship Mode	Order Date	Ship Date
18.648	7	-12.432	0.7	South	North Carolina	231	US-2015-156216	Corporate	EA-14035	Erin Ashbrook	Standard Class	09/13/2015, 12:00:00 AM	09/17/2015, 12:00:00 AM
178.384	2	22.298	0.2	South	Florida	315	CA-2015-167850	Corporate	AG-10525	Andy Gerbode	Standard Class	08/09/2015, 12:00:00 AM	08/16/2015, 12:00:00 AM
15.552	3	5.4432	0.2	South	Florida	316	CA-2015-167850	Corporate	AG-10525	Andy Gerbode	Standard Class	08/09/2015, 12:00:00 AM	08/16/2015, 12:00:00 AM
39.072	6	9.768	0.2	South	North Carolina	404	CA-2015-155208	Corporate	SP-20650	Stephanie Phelps	Standard Class	04/16/2015, 12:00:00 AM	04/20/2015, 12:00:00 AM
10.368	2	3.6288	0.2	South	North Carolina	705	CA-2015-138527	Corporate	BN-11470	Brad Norvell	Standard Class	09/12/2015, 12:00:00 AM	09/17/2015, 12:00:00 AM
166.84	5	18.7695	0.2	South	North Carolina	706	CA-2015-138527	Corporate	BN-11470	Brad Norvell	Standard Class	09/12/2015, 12:00:00 AM	09/17/2015, 12:00:00 AM
15.216	1	2.2824	0.2	South	North Carolina	707	CA-2015-138527	Corporate	BN-11470	Brad Norvell	Standard Class	09/12/2015, 12:00:00 AM	09/17/2015, 12:00:00 AM
11.36	2	5.3392	0	South	Louisiana	764	CA-2015-162775	Corporate	CS-12250	Chris Selesnick	Second Class	01/13/2015, 12:00:00 AM	01/15/2015, 12:00:00 AM

Figure 4 Tableau Prep Builder https://www.tableau.com/products/prep

It features a free trial, but after that the price is high with plans starting from $70 per user per month for the Tableau Creator. There are various other offerings as well for teams and organizations or embedded analytics[17].

It also offers all the basic visualization types in an easy-to-use menu (see Figure 5), and has Desktop, Online, and Server versions. It is also able to support more complex

[15] https://www.tableau.com/
[16] https://www.tableau.com/products/prep#data-sources
[17] https://www.tableau.com/pricing/

visualizations and a gallery (see Figure 6) that been building up for more than a decade showcases Tableau’s power. They offer training, consulting, and support as tiered services[18].

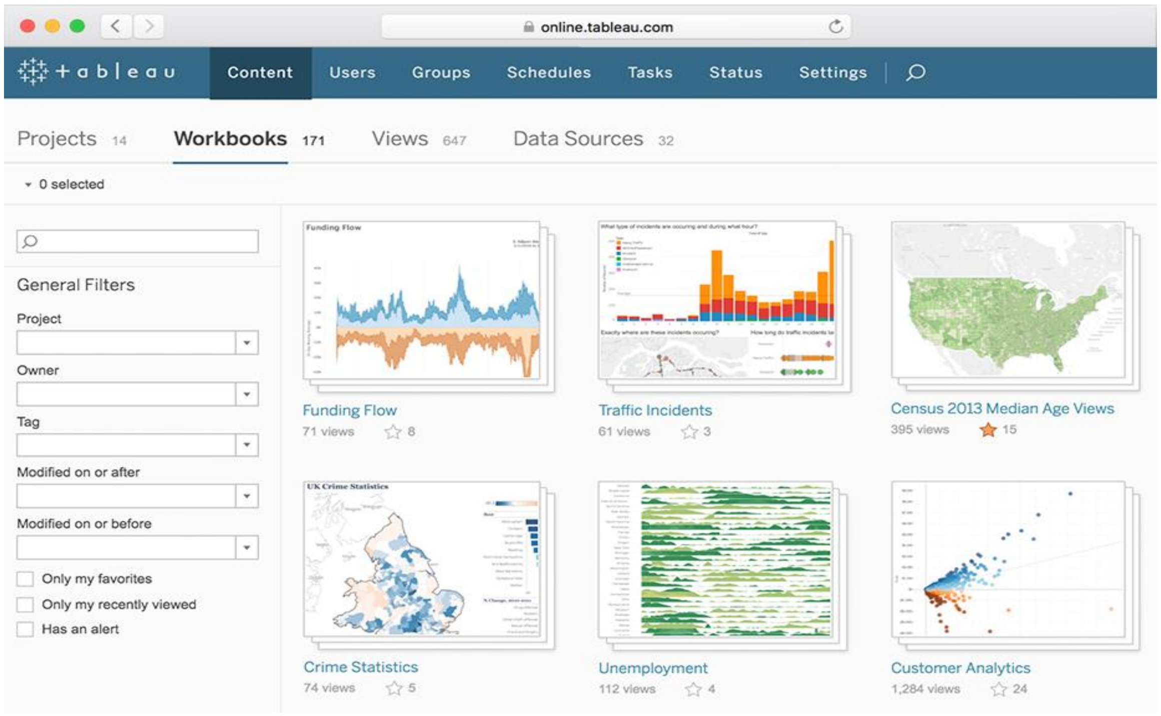


Figure 5 Visualizations https://www.tableau.com/products/what-is-tableau

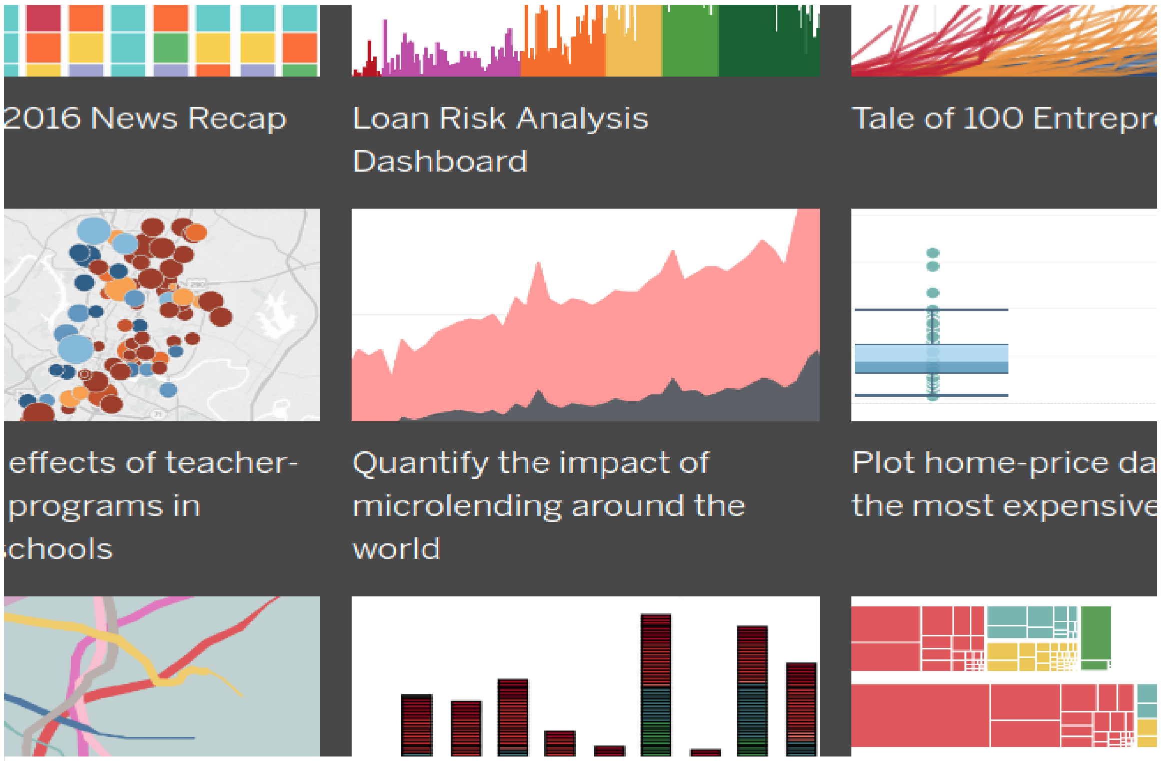


Figure 6 Tableau Viz Gallery https://www.tableau.com/solutions/gallery

[18] https://www.tableau.com/services

Finally, Tableau gives developers tools to create their own extensions and make truly interactive dashboards with custom user interfaces. Not only that, but the extensions are pretty flexible allowing for connecting to non-yet-supported data sources, or to even run their own data science models using languages like R, Python, or Matlab. Embedded analytics enable integrations with Salesforce and Microsoft Sharepoint[19].

Qlik[20]

Qlik is, in their words, a “data analytics for modern business intelligence” tool, with a reported market share of 2.22% (Datanyze). QlikTech is probably the older of the bunch having been founded in 1993 with their first product being focused on data analysis (Industrifonden).

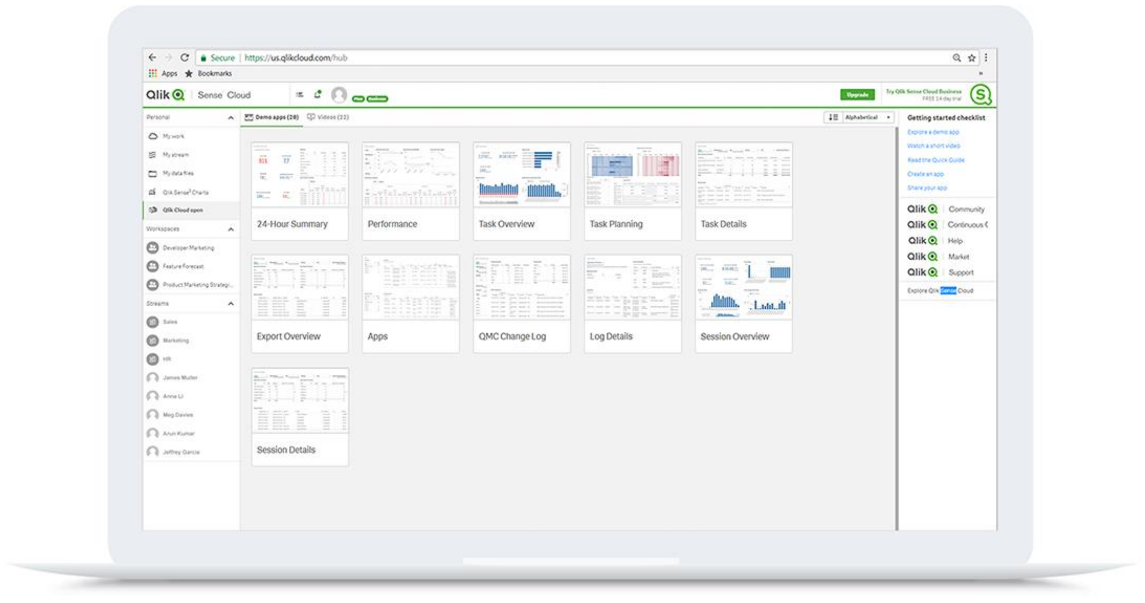


Figure 7 Qlik Analytics Platform, https://www.qlik.com/us/products/qlik-sense?ga-link=HP_Mid1_US

They have a free tier, *Qlik Sense Cloud Basic*, a *Business* tier for $15 per user per month (including a free trial), and an *Enterprise* tier with flexible pricing options. They offer similar services to Tableau, like embedded analytics[21], with their main product being their analytics platform. Furthermore, they offer other products like data indexing[22], reporting[23], conversational analytics[24], and consulting among others. Using

[19] https://www.tableau.com/developer/tools
[20] https://www.tableau.com/
[21] https://www.qlik.com/us/bi/embedded-analytics?ga-link=HP_Mid2_US
[22] https://www.qlik.com/us/products/associative-big-data-index

Qlik Connectors[25], you can integrate your data from various sources such as files, databases, SAP, salesforce, Azure, Amazon, Google, and more, for a total of over 100 sources.

They provide the basic visualizations[26] but you can opt in extra tools such as *Qlik GeoAnalytics*[27] with advanced functionality. In fact, this is a common pattern all around Qlik (buying "products" with extra functionality).

Open-source programming languages, libraries, and frameworks

R project[28] and **Python**[29]

R is primarily a statistical programming language with more than 15,000 currently (R-Project, 2019). Most of these are probably relevant to data analytics tools and trying to map all of them would a nearly impossible feat for a time-limited work such as this one. *Python*, on the other hand, is more of a general-purpose programming language which has been used in lots of fields and for lots of purposes (scientific computing & data science, web development, game development, gluing other languages). Python boast nearly 190K projects and over 350K users[30], being the second most popular (general-purpose) programming language (after JavaScript), second most loved (after Rust), the most wanted (Stack Overflow, 2019, p. Technology), and the one with highest rating change according to TIOBE Index (TIOBE, 2019), with an overall growth shadowing many others (Robinson, 2017).

First on the list is *ggplot2*[31], a project with more than 200 contributors on GitHub, nearly 1.5K forks and 4K stars, with a GPL-2.0 open-source license, and is part of the tidyverse collection of packages[32]. The original publication (book) by Springer (Wickham, 2016) has received 3.6K citations and over 200K downloads. It uses a

[23] https://www.qlik.com/us/products/nprinting
[24] https://www.qlik.com/us/products/qlik-insight-bot
[25] https://www.qlik.com/us/products/qlik-connectors
[26] https://help.qlik.com/en-US/sense/February2019/Subsystems/Hub/Content/Sense_Hub/Visualizations/visualizations.htm
[27] https://www.qlik.com/us/products/qlik-geoanalytics
[28] https://www.r-project.org/
[29] https://www.python.org/
[30] https://pypi.org/
[31] https://github.com/tidyverse/ggplot2
[32] https://www.tidyverse.org/packages/

declarative style of programming, which is often cited as a “mini-language” (Wickham, 2016, p. About). *ggplot2* also has a few python clones, the two most popular being *ggpy*[33] (not maintained) and *plotnine*[34] which is the suggested alternative (ChKwK, 2018).

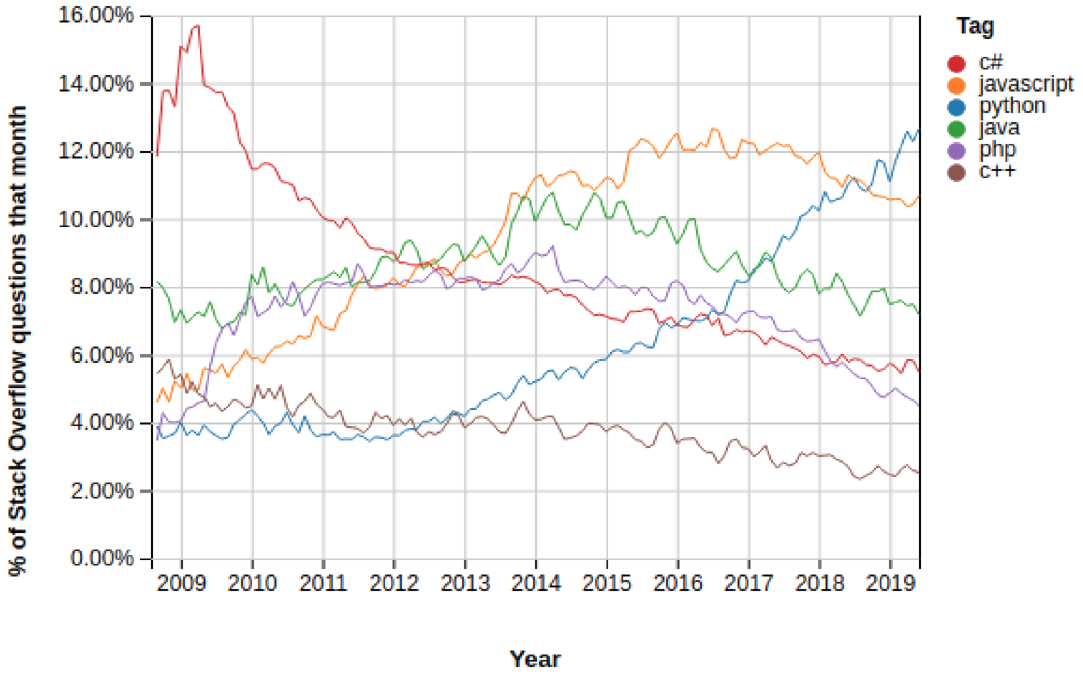


Figure 8 Programming Language Growth,
https://insights.stackoverflow.com/trends?tags=python%2Cjavascript%2Cjava%2Cc%23%2Cphp%2Cc%2B%2B

However, the Python equivalent to ggplot is *matplotlib*[35], by John Hunter (Hunter, 2007), with 165 paper citations[36]. This open-source project uses a *Python Software Foundation*-based license[37], is used by nearly 120K projects, with more than 800 contributors 9.7K stars and 4.3K forks on GitHub. *matplotlib* supports a wide range of highly customizable visualizations, which aided by an array of third-party libraries (including the ones we mentioned previously[38]), covers most visualization needs.

[33] https://github.com/yhat/ggpy
[34] https://github.com/has2k1/plotnine
[35] https://github.com/matplotlib/matplotlib
[36] https://ieeexplore.ieee.org/document/4160265
[37] https://matplotlib.org/users/license.html
[38] https://matplotlib.org/thirdpartypackages/index.html

Leaflet[39] is a free and open-source JavaScript mapping library that has binding for both R and Python. It has a very short and permissive license[40], over 600 contributors, 4K forks, and 25K stars. Furthermore, it is immensely popular and was even used by popular sites such as The New York Times[41], GitHub[42] and The Washington Post[43], among the –at least– 24K other projects.

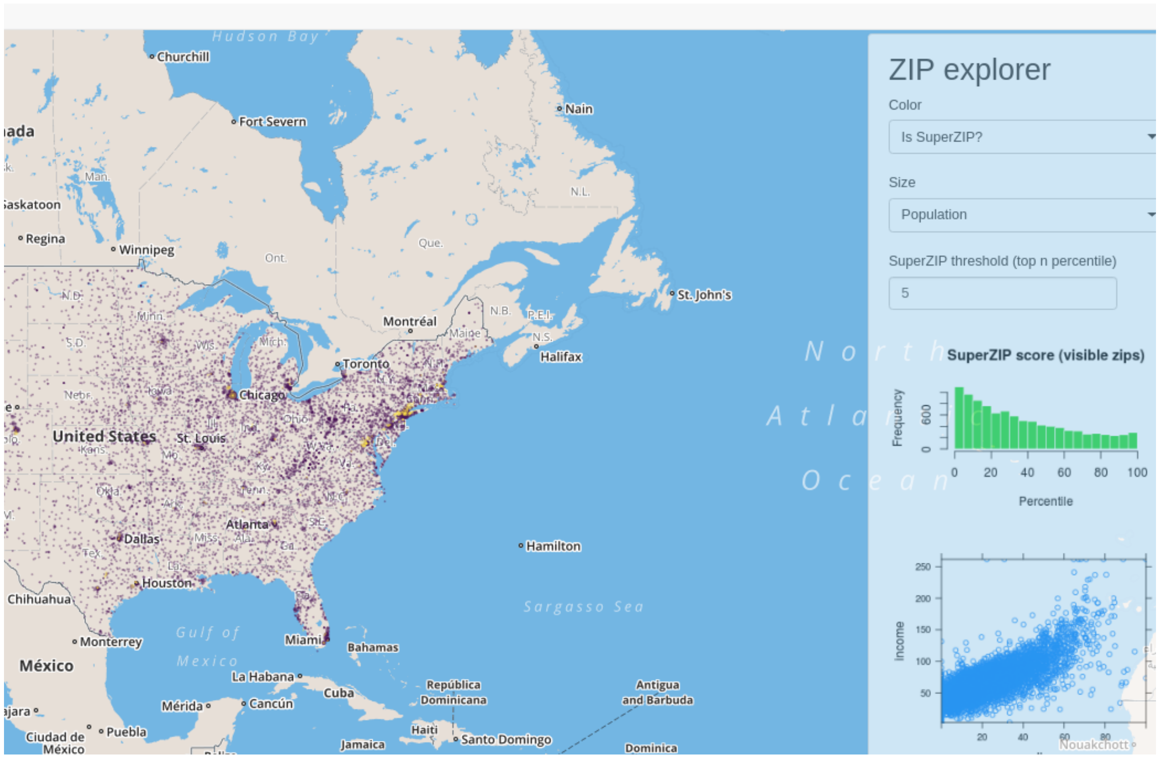
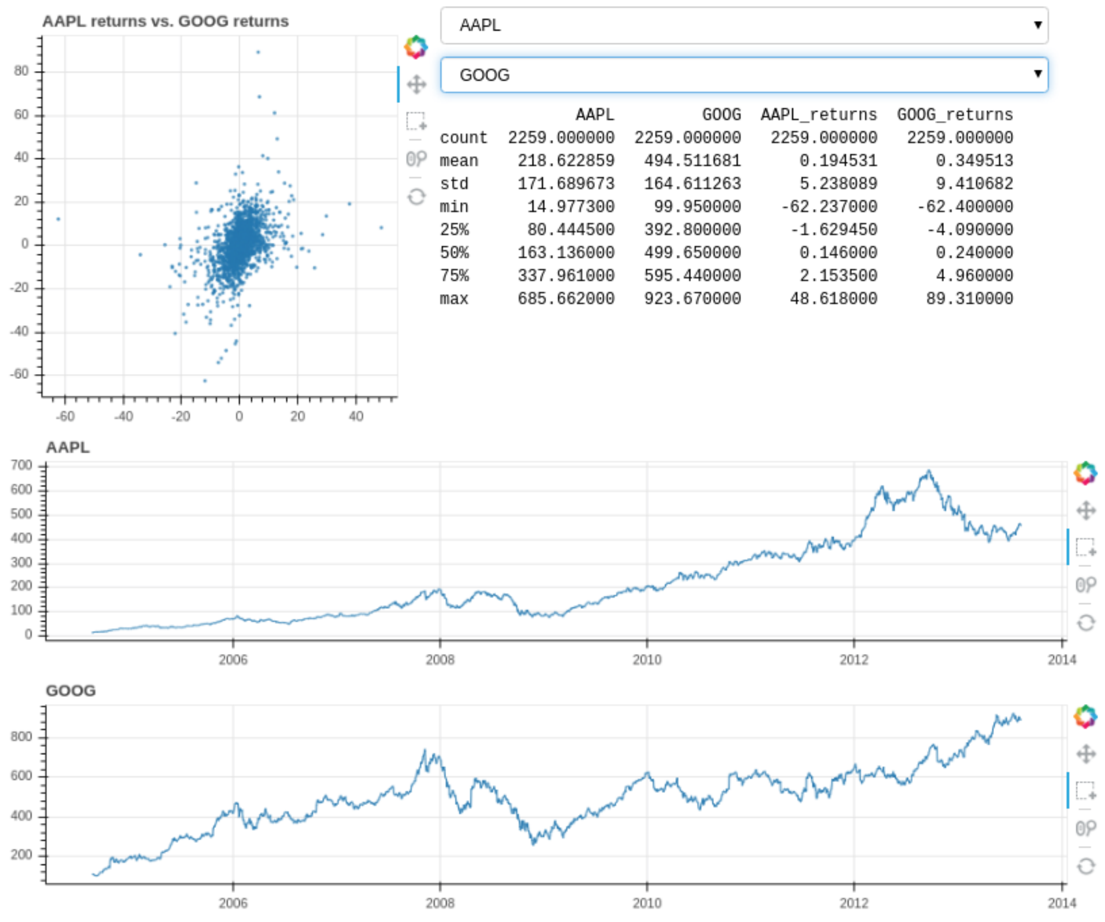


Figure 9 SuperZip Example - Shiny Gallery, https://shiny.rstudio.com/gallery/superzip-example.html

Finally, two more projects stand out, both focused on creating dashboards: *Shiny*[44] for R, and *Bokeh*[45] for Python. *Shiny* has 44 contributors, about 3.5K stars and 1.5K forks and a mixed license mainly distributed under GPL-3[46]. *Bokeh* is more popular, used by 12.8K people, with 11K starts and about 2.8K forks, with 371 contributors and a BSD-3-

[39] https://leafletjs.com/, https://github.com/Leaflet/Leaflet
[40] https://github.com/Leaflet/Leaflet/blob/master/LICENSE
[41] http://www.nytimes.com/projects/elections/2013/nyc-primary/mayor/map.html
[42] https://github.blog/2013-06-13-there-s-a-map-for-that/
[43] http://www.washingtonpost.com/sf/local/2013/11/09/washington-a-world-apart/
[44] https://shiny.rstudio.com/
[45] https://bokeh.pydata.org/en/latest/
[46] https://github.com/rstudio/shiny/blob/master/LICENSE

Clause license[47]. They both build interactive applications focused on web browser, but *Bokeh* has one additional merit: it integrates very well with *Jupyter Notebooks*, one of the most favorite development environments for Python (Stack Overflow, 2019, p. Most Popular Development Environments).



```
              AAPL         GOOG  AAPL_returns  GOOG_returns
count  2259.000000  2259.000000   2259.000000   2259.000000
mean    218.622859   494.511681      0.194531      0.349513
std     171.689673   164.611263      5.238089      9.410682
min      14.977300    99.950000    -62.237000    -62.400000
25%      80.444500   392.800000     -1.629450     -4.090000
50%     163.136000   499.650000      0.146000      0.240000
75%     337.961000   595.440000      2.153500      4.960000
max     685.662000   923.670000     48.618000     89.310000
```

Figure 10: Stocks Example, Bokeh Gallery, https://demo.bokeh.org/stocks

We conveniently left out *Plotly* and its dashboard tool, *Dash*, because these are the ones we use for most of our software[48]. We will go into them in a bit more detail in the next chapters where we will see a high-level overview of the tools used and discuss the design decisions. Before that, let's review some similar tools for building Machine Learning models and pipelines.

[47] https://github.com/bokeh/bokeh/blob/master/LICENSE.txt
[48] You can view the docs on the R version here: https://plot.ly/r/

Kibana[49]

Kibana is part of the Elastic stack[50] (with *Elasticsearch*, *Logstash,* and now *Beats*), and it helps you visualize your data. It handles the basics, and provides additional features like time series analysis, machine learning, graphs /networks and maps visualization and analysis, and PDF exports. It also offers integration with Vega for creating custom visualizations[51].

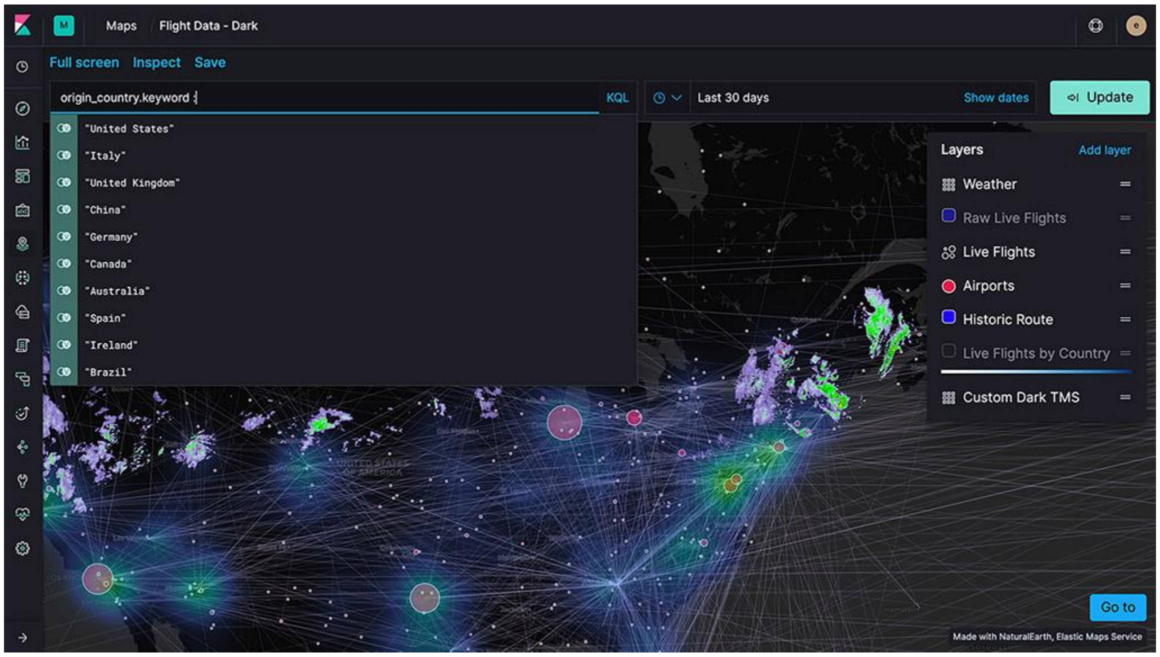
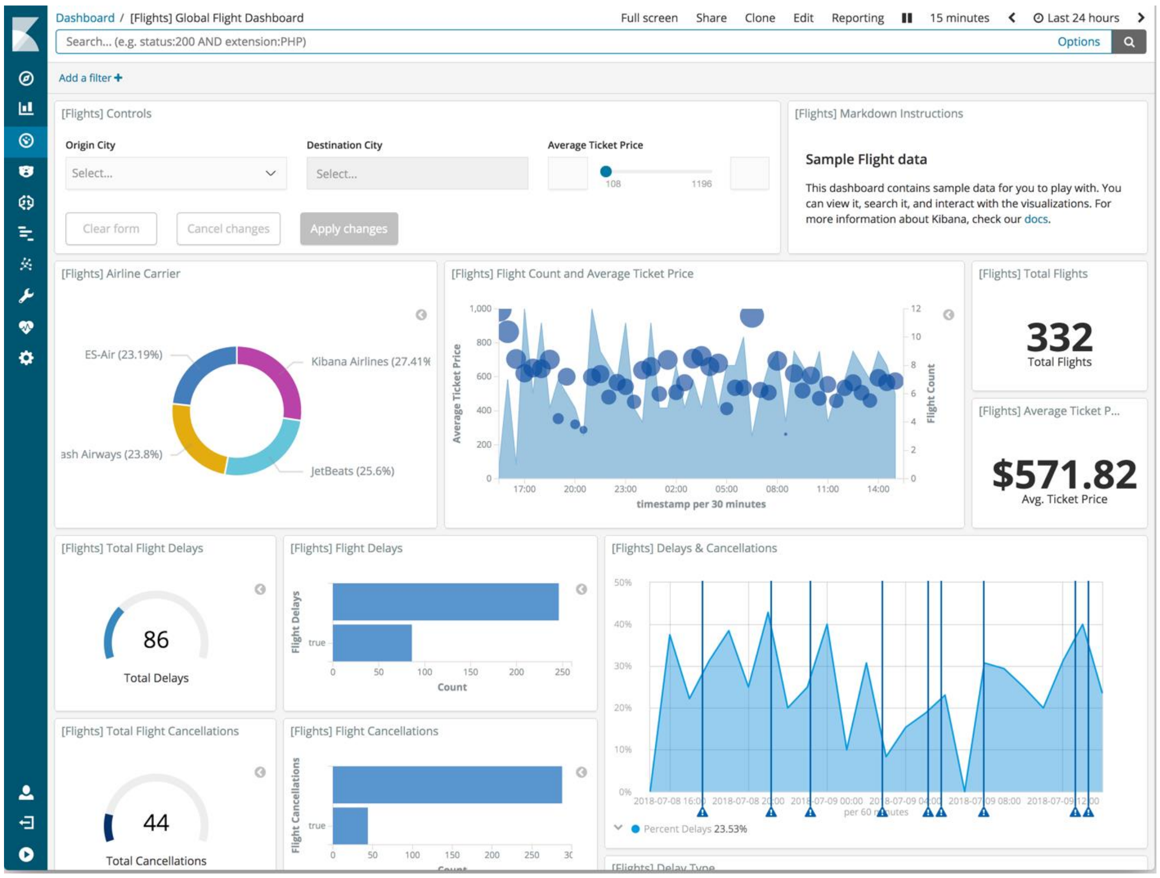


Figure 11 Geospatial Analysis, https://www.elastic.co/products/kibana/features

The complete list of features[52] (many with their own list of modules) gives much more details, but we will mention a few here. Connections to databases include MySQL, MongoDB, PostgreSQL, Microsoft SQL, connections to external tools include Slack and Jira. Perhaps one of the most interesting is the ability to create dashboards, which is advertised as its main feature, with a nicely polished UI.

[49] https://www.elastic.co/products/kibana
[50] https://www.elastic.co/products/
[51] https://www.elastic.co/products/kibana/features#vega--custom-
[52] https://www.elastic.co/products/kibana/features



Sample Flight data

This dashboard contains sample data for you to play with. You can view it, search it, and interact with the visualizations. For more information about Kibana, check our docs.

Figure 12 Kibana Dashboard, https://www.elastic.co/guide/en/kibana/current/dashboard.html

Kibana has an open-source version with about 4.9K forks and 12.5K stars, and 400 contributors[53]. The main license follows Apache License Version 2.0[54]. The open source project has received a lot of traffic (despite the “used by | 5” section, it has about 12.5K closed and 4.6K open issues). It also has a paid (cloud) version starting from $17 per month[55]. A more detailed comparison, with a huge table of features can be found in *Elastic*’s page[56], according to which data management, graph exploration and analytics, all of the machine learning features, and the PDF exports are all in the various premium versions.

[53] https://github.com/elastic/kibana
[54] https://github.com/elastic/kibana/blob/master/LICENSE.txt
[55] https://www.elastic.co/products/elasticsearch/service/pricing
[56] https://www.elastic.co/subscriptions

2.2.2 Machine Learning / Statistical Modeling Software

There are probably as many tools in this category as there are in the previous, and most cloud providers (see Azure, Google, Amazon) also have their own offerings of machine learning and analysis tools. We will only go over three tools here: *KNIME*, *Weka*, and *RapidMiner*. The selection is based on the relevant paper by Al-Khoder and Harmouch (Al-Khoder & Harmouch, 2015), who base their decision on which tools are better fitted to be complete platforms (Al-Khoder & Harmouch, 2015, p. 3). We will add to the list Orange3, because it another completely open-source and popular tool, and Kibana, because even though it is partly open-source (with ML being the non-open-source part), it is already incredibly popular due to being part of the Elastic ("ELK") stack. Additionally, they are all (completely or mostly) open-source. R is being excluded primarily because the whole language is designed for this specific purpose and going over it is redundant (and we briefly went over it in the previous section). Furthermore, this section will focus mainly on presenting the tools, not on any detailed comparisons; a discussion on this can be found in the literature review later in this chapter.

KNIME[57]

KNIME is the recommended tool for novices (Al-Khoder & Harmouch, 2015, p. 2), reported as a "data analytics, reporting, and integration platform" under a GPLv3[58] (Al-Khoder & Harmouch, 2015, pp. 3-4). It has been placed as a "Leader" in Gartner's *Magic Quadrant for Data Science and Machine Learning Platforms*[59]. The latest publication is from the announcement of version 2 in 2009 (Berthold, et al., 2009).

Just like a lot of the tools already mentioned, it supports integrations with various SQL servers and file formats, with *HIVE* connectors also available[60], as well as connections to APIs[61].

[57] https://www.knime.com/
[58] https://www.knime.com/blog/new-version-210-has-been-released
[59] see Appendix 1, illustration 15, also: https://www.knime.com/about/news/knime-recognized-by-gartner-as-a-leader-in-data-science-and-machine-learning-platforms-2019
[60] https://www.knime.com/blog/new-version-210-has-been-released
[61] https://www.knime.com/whats-new-in-knime-210

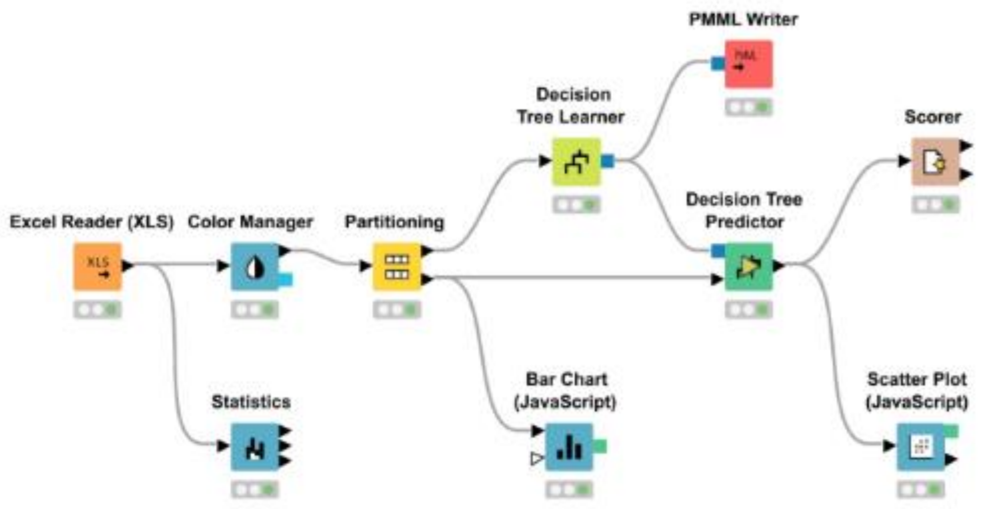


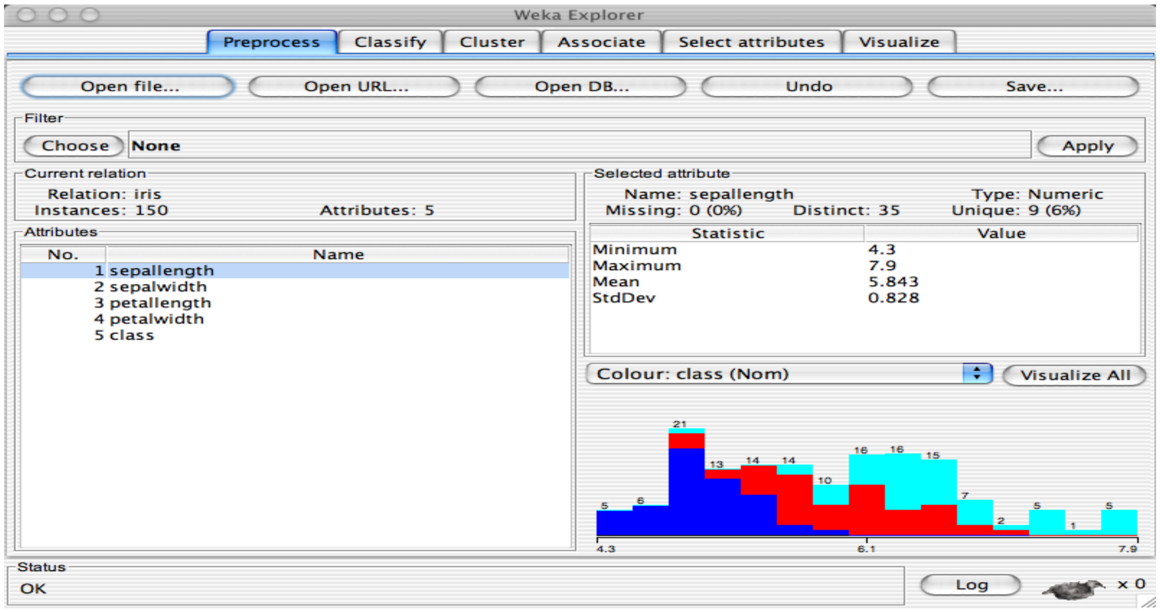
Figure 13 Machine Learning pipelines in KNIME, https://www.knime.com/analytics-expert

One feature that particularly stands out for *KNIME* is the ability to deploy your models as REST APIs and integrate it easily with R and/or Python, neural networks included[62]. Apparently *KNIME* can use *Plotly* (in essence, via JavaScript or Python[63]) or *RapidMiner*[64]. It also showcases some solutions to common problems like building recommendation systems, inventory optimization, sentiment analysis, churn, and anomaly detection[65]. While it is a local application it can also deploy interactive browser applications with *KNIME WebPortal*[66] and they also offer a cloud-based version for more than 2 years now[67]. As per Wikipedia (but couldn't find a reference in *KNIME*'s website), *KNIME* works with out-of-core algorithms[68].

62 https://www.knime.com/analytics-expert
63 E.g. https://gist.github.com/webbres/3c052788ac55df90ea5b22fe65a68b4e
64 E.g. https://hub.knime.com/aborg/extensions/com.mind_era.knime_rapidminer.knime.feature/latest
65 https://www.knime.com/solutions
66 https://www.knime.com/knime-software/knime-webportal
67 https://www.knime.com/knime-software-in-the-cloud-old
68 https://en.wikipedia.org/wiki/KNIME#Internals

WEKA[69]

WEKA is definitely one of the most seasoned software in this list, with its first release dating back to 1994 (Holmes, Donkin, & Witten, 1994) by the university of Waikato, New Zealand, with 33 contributors from the university[70]. It was originally written in C, C++ and LISP (Holmes, Donkin, & Witten, 1994, p. 2) but was later re-written in Java (Witten, et al., 1999, p. 1). *WEKA*'s website and source control[71] are the least user-friendly of what we viewed here, which mirrors the user interface of the actual application, while also lacking own forums like the other software. Similar comments can be made about their documentation[72].



Filter: None

Relation: iris
Instances: 150 Attributes: 5

No.	Name
1	sepallength
2	sepalwidth
3	petallength
4	petalwidth
5	class

Name: sepallength Type: Numeric
Missing: 0 (0%) Distinct: 35 Unique: 9 (6%)

Statistic	Value
Minimum	4.3
Maximum	7.9
Mean	5.843
StdDev	0.828

Colour: class (Nom)

Status: OK

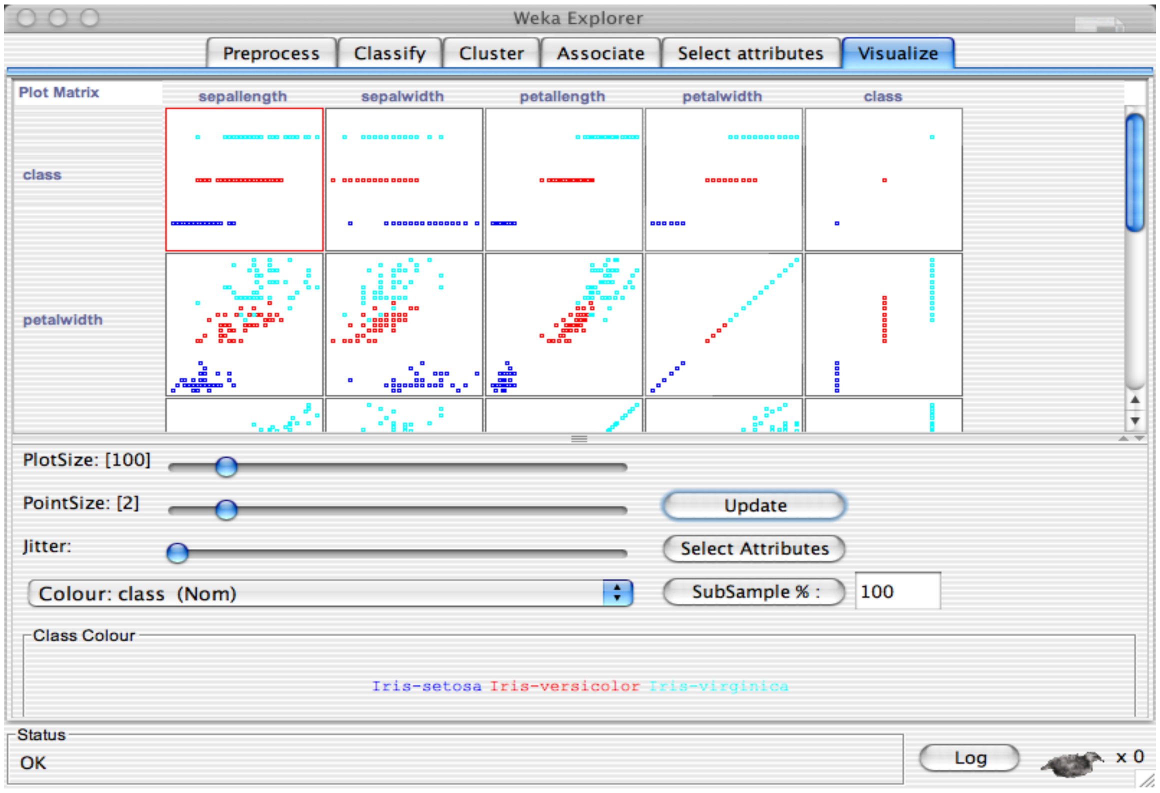
Figure 14 Weka Explorer, https://www.cs.waikato.ac.nz/~ml/weka/gui_explorer.html

That said, *WEKA* does not lack much with respect to its competitors in terms of available machine learning models. It can easily handle time series, decision trees, rule-based models, neural networks, bagging, boosting, Bayesian, and more (Frank, Hall, & Witten, 2016). It also includes preprocessing utilities (Holmes, Donkin, & Witten, 1994, p. 4). It has the least visualizations of every software examined so far, with only 5 out of 27 listed in (Al-Khoder & Harmouch, 2015, pp. 5-6), as well as data sources support.

[69] https://www.cs.waikato.ac.nz/ml/index.html
[70] https://www.cs.waikato.ac.nz/ml/people.html
[71] https://svn.cms.waikato.ac.nz/svn/weka/
[72] http://weka.sourceforge.net/doc.dev/

Even though it lacks more advanced models and does not have equally powerful pipelines (in fact, they probably cannot be called pipelines at all (ben26941, 2017)), it allows integration with languages that data scientists are familiar with (Python[73], and R[74]), and of course Java. Third-party applications have worked to extend *WEKA*, with some of the first GitHub results being:

- *autoweka*[75] (autoML in weka)
- *wekaDeeplearning4j*[76]
- *weka* python wrapper[77]
- *tmweka*[78] (text mining in weka)



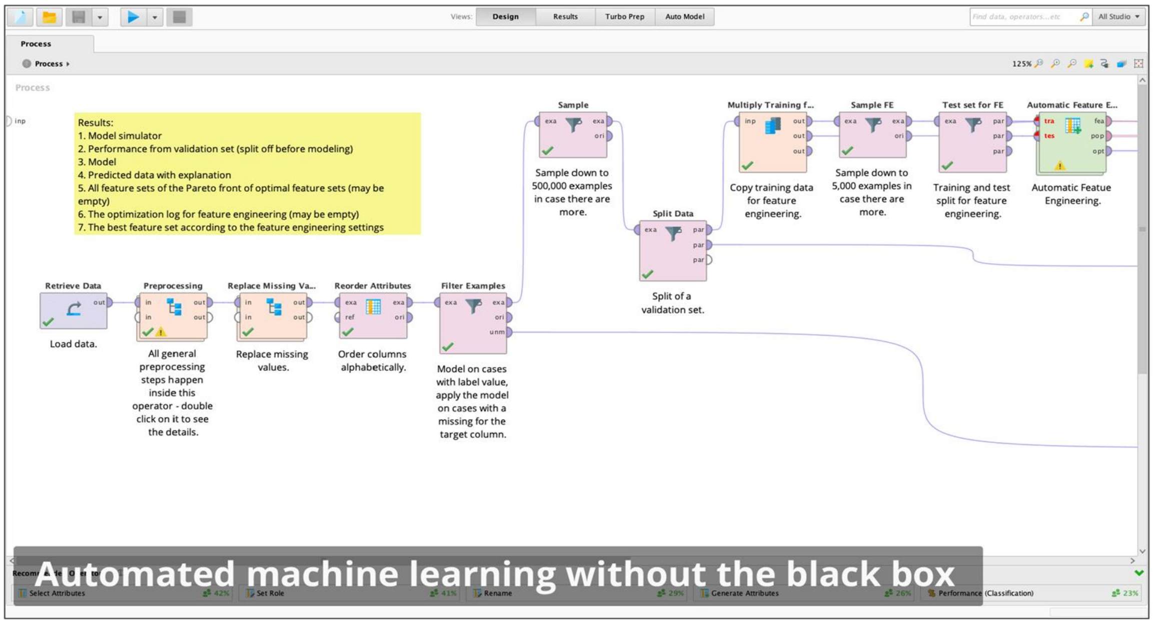
PlotSize: [100]
PointSize: [2]
Jitter:
Colour: class (Nom)
SubSample % : 100
Status
OK

Figure 15 Weka Explorer (2), https://www.cs.waikato.ac.nz/~ml/weka/gui_explorer.html

RapidMiner[79]

[73] E.g. https://pypi.org/project/python-weka-wrapper/, https://www.youtube.com/watch?v=cwbPzumwgNo
[74] E.g. https://forums.pentaho.com/threads/154305-Integrating-R-with-Weka/, http://weka.sourceforge.net/packageMetaData/Rplugin/index.html
[75] https://github.com/automl/autoweka
[76] https://github.com/Waikato/wekaDeeplearning4j
[77] https://github.com/chrisspen/weka
[78] https://github.com/jmgomezh/tmweka

Despite the jokes about *RapidMiner* in the introduction, it is probably the most complete, easy to use, and intuitive tool in this list. As said previously, the core of *RapidMiner Studio*[80] which provides data visualization and visual creation of pipelines, is free and open-source[81], written in Java, and the company's business model is based on offering additional services and extensions on top of it. According to their website, more than half a million of users spread across over 30K organizations and 1K universities use their products[82].


Results:
1. Model simulator
2. Performance from validation set (split off before modeling)
3. Model
4. Predicted data with explanation
5. All feature sets of the Pareto front of optimal feature sets (may be empty)
6. The optimization log for feature engineering (may be empty)
7. The best feature set according to the feature engineering settings

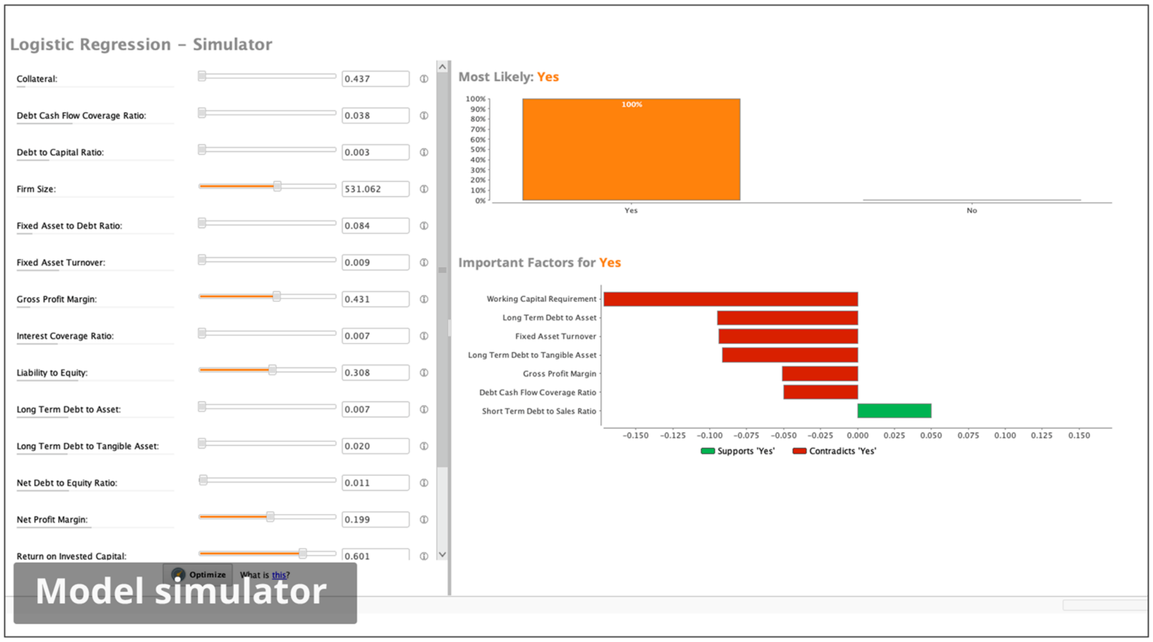
Views: Design Results Turbo Prep Auto Model
Process
Retrieve Data
Load data.
Preprocessing
All general preprocessing steps happen inside this operator - double click on it to see the details.
Replace Missing Va...
Replace missing values.
Reorder Attributes
Order columns alphabetically.
Filter Examples
Model on cases with label value, apply the model on cases with a missing for the target column.
Sample
Sample down to 500,000 examples in case there are more.
Split Data
Split of a validation set.
Multiply Training f...
Copy training data for feature engineering.
Sample FE
Sample down to 5,000 examples in case there are more.
Test set for FE
Training and test split for feature engineering.
Automatic Feature E...
Automatic Featue Engineering.
Automated machine learning without the black box
Select Attributes
Set Role
Rename
Generate Attributes
Performance (Classification)

Figure 16 RapidMiner Pipelines, https://rapidminer.com/products/auto-model/

One such extension, handling data management is *Turbo Prep* (https://rapidminer.com/products/turbo-prep/) which also allows for exploratory data analysis, data transformations and cleaning, merging datasets. Another one is *Auto Model* (https://rapidminer.com/products/auto-model/) which handles even more of the modeling process like calculating statistics and judging the quality of datasets, perform automated model selection & tuning as well as automatic feature engineering, and provides dashboards and integrations (e.g. with MS Excel).

[79] https://rapidminer.com
[80] https://rapidminer.com/products/studio/
[81] https://github.com/rapidminer/rapidminer-studio
[82] https://rapidminer.com/products/

Furthermore, nearly 4K extensions are available in the *RapidMiner Marketplace*[83]. Since RapidMiner allows for Python and R extensions, you can find a lot of those in the *Marketplace*. The other three extensions are the *RapidMiner Server*[84] which handles deployment of models and collaboration, *RapidMiner Real-Time Scoring*[85] which has also targets deployment but focuses on fast inference times with a REST API wrapper, and *RapidMiner Radoop*[86] which is the *RapidMiner Studio* big-data version for *Hadoop* and *Spark*, providing cluster computing and integration with the *Hadoop* ecosystem: *SparkR*, *PySpark*, *Pig*, *HiveQL*, and more.


Logistic Regression – Simulator

Collateral:	0.437
Debt Cash Flow Coverage Ratio:	0.038
Debt to Capital Ratio:	0.003
Firm Size:	531.062
Fixed Asset to Debt Ratio:	0.084
Fixed Asset Turnover:	0.009
Gross Profit Margin:	0.431
Interest Coverage Ratio:	0.007
Liability to Equity:	0.308
Long Term Debt to Asset:	0.007
Long Term Debt to Tangible Asset:	0.020
Net Debt to Equity Ratio:	0.011
Net Profit Margin:	0.199
Return on Invested Capital:	0.601

Figure 17 RapidMiner Auto Model, Simulator, https://rapidminer.com/products/auto-model

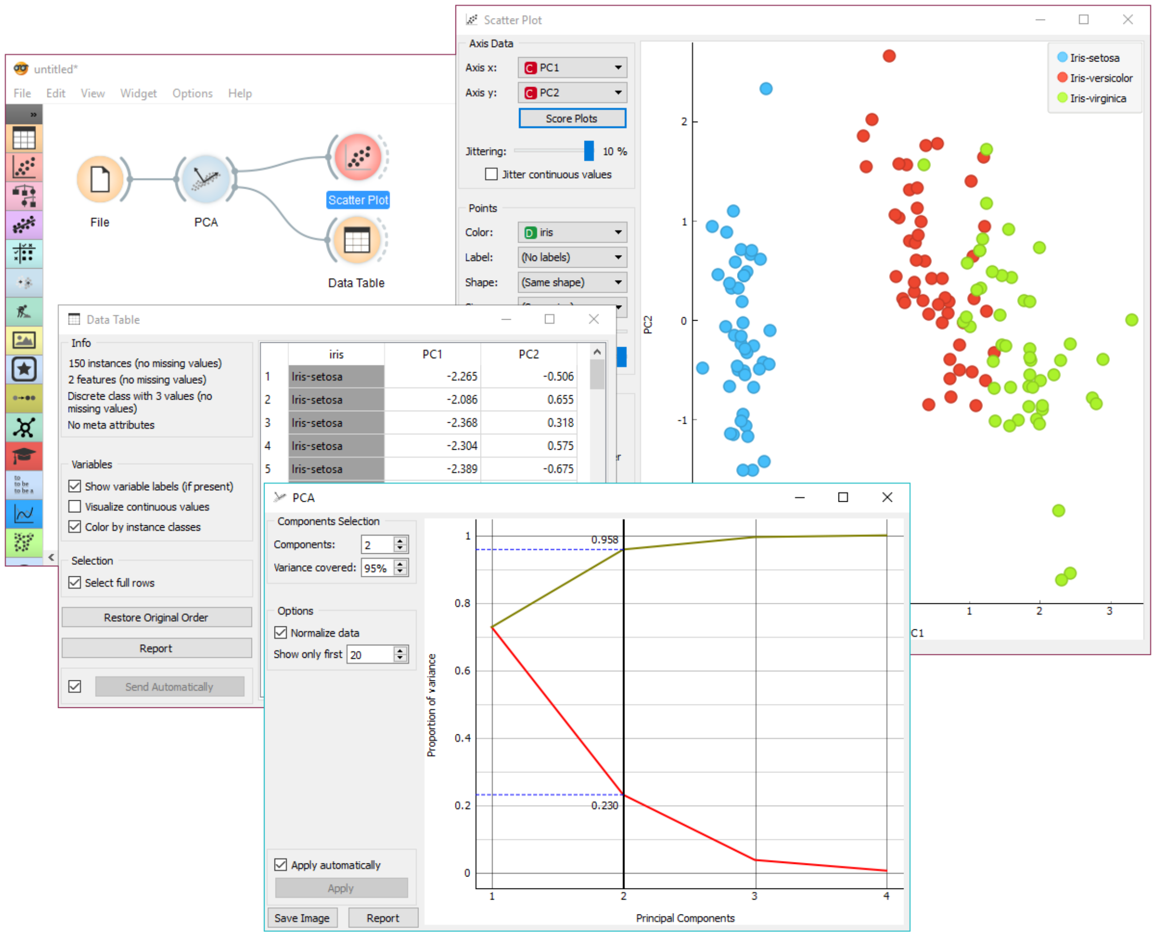
Orange3[87]

Orange is an incredibly modular data mining software which steps largely on scikit-learn and the *scipy* stack[88], with matplotlib and Qt handling most of the graphical interface[89]. It has 1.8K stars, 542 forks, is used by 163 others, and is developed by 71

[83] https://marketplace.rapidminer.com/UpdateServer/faces/index.xhtml
[84] https://rapidminer.com/products/server/
[85] https://rapidminer.com/products/real-time-scoring/
[86] https://rapidminer.com/products/radoop/
[87] https://orange.biolab.si/
[88] https://github.com/biolab/orange3/blob/master/requirements-core.txt
[89] https://github.com/biolab/orange3/blob/master/requirements-gui.txt

contributors[90], with 16 people being mentioned in the original publication (Demsar, et al., 2013) and having contributed the majority of the codebase. The license is based on GPLv3 with the University of Ljubljana being the designated copyright holder[91].

A very simple interface with every sub-menu being a popup comprises the whole GUI. It looks a lot like the other tools we discussed so far (see https://orange.biolab.si/screenshots/ for more), and Qt works seamlessly on most operating systems.



Scatter Plot
Axis Data
Axis x: PC1
Axis y: PC2
Score Plots
Jittering: 10 %
☐ Jitter continuous values
Points
Color: iris
Label: (No labels)
Shape: (Same shape)

Data Table
Info
150 instances (no missing values)
2 features (no missing values)
Discrete class with 3 values (no missing values)
No meta attributes
Variables
☑ Show variable labels (if present)
☐ Visualize continuous values
☑ Color by instance classes
Selection
☑ Select full rows
☑ Send Automatically

	iris	PC1	PC2
1	Iris-setosa	-2.265	-0.506
2	Iris-setosa	-2.086	0.655
3	Iris-setosa	-2.368	0.318
4	Iris-setosa	-2.304	0.575
5	Iris-setosa	-2.389	-0.675

PCA
Components Selection
Components: 2
Variance covered: 95%
Options
☑ Normalize data
Show only first 20
☑ Apply automatically

Figure 18 Orange3 screenshot, PCA, https://orange.biolab.si/screenshots/

The *Workflows* tool also allows the user to define complex pipelines including data ingestation, processing, visualization, and machine learning. As *Orange* uses *sklearn* behind the scenes for most of its machine learning, everything from there is available, or

[90] https://github.com/biolab/orange3/
[91] https://github.com/biolab/orange3/blob/master/LICENSE

can be ported. It even includes many well-defined tasks such as correspondence analysis, manifold learning, regression. Preprocessing tasks like merging data, finding / removing outliers, imputation, normalization are there too. In terms of visualizations, *Orange* offers the basics (trees, boxplots, histograms, scatterplots) and a few less commonplace ones (pythagorean tree, radviz). Everything is a *widget*, and a catalog can be found on their website[92]. SQL connections are limited to MSSQL and PostegreSQL, but files can be acquired from the web, from the typical formats (Excel, txt, CSV), or from other platforms (e.g. Google Sheets). Some basic data type inference is done, which the user can, of course, modify[93].

Although this tool is rather limited when compared to the previous (which is to be expected as this is used mainly as an educational platform), developing new add-ons is easy since *Orange* is directly written in Python and they provide a reference template[94], and thus you can leverage the whole python community and its packages.

2.2.3 Cloud providers and services

In this final part of the competitors' review we will quickly go over the cloud offerings of Amazon (AWS), Microsoft (Azure) and Google. There many more, some of which we already visited (e.g. Kibana is mainly a cloud platform, and Tableau Server can also be deployed in cloud providers like AWS) but due to size constraints we cannot go over them here.

Amazon QuickSight and other ML services[95]

Amazon's main offering is an array of specialized services revolving around *Infrastructure as a Service*, but over the years they have developed tools that can be thought of as *Service as a Service*, or similar. One such offering is Amazon's QuickSight which is a Business Intelligence / Machine Learning service.

[92] https://orange.biolab.si/widget-catalog/
[93] https://orange.biolab.si/widget-catalog/data/file/, https://orange-visual-programming.readthedocs.io/loading-your-data/index.html
[94] https://github.com/biolab/orange3-example-addon
[95] https://aws.amazon.com/quicksight/, https://aws.amazon.com/machine-learning/

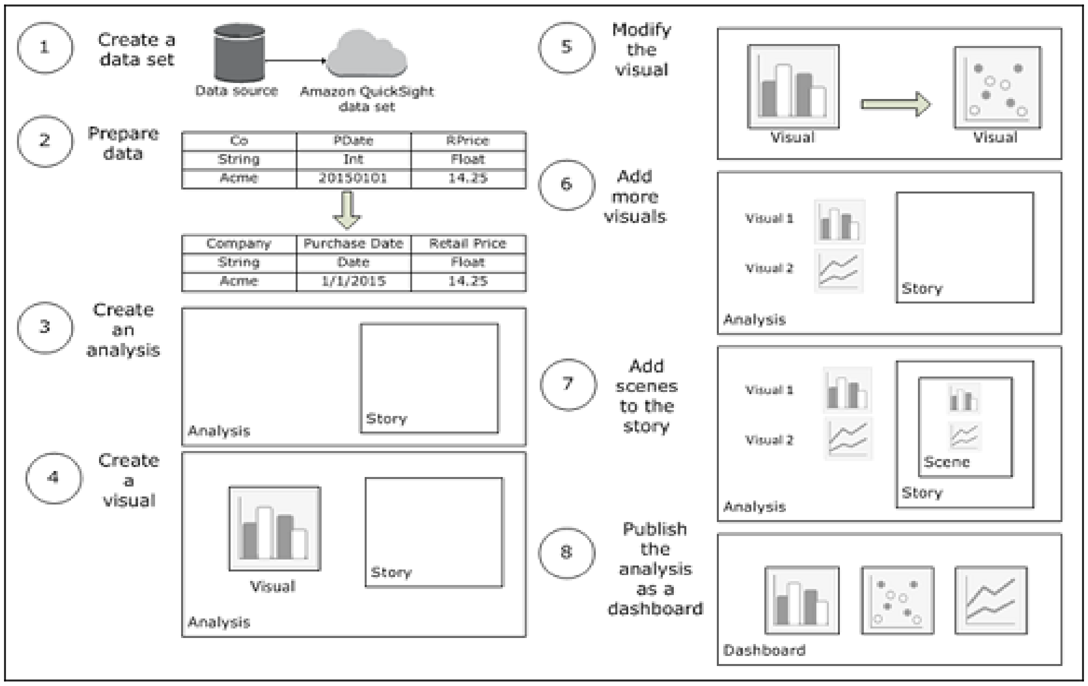


Figure 19 Amazon QuickSight workflow, https://aws.amazon.com/blogs/big-data/10-visualizations-to-try-in-amazon-quicksight-with-sample-data/

As one would expect, this works with a usage-based pricing model, as most AWS offerings. When it comes to integrations with data sources, it obviously supports other AWS service like RedShift, S3, Athena, GLUE, and even third-party applications like MySQL, PostgreSQL, Teradata, Salesforce, or common file formats for user uploads[96]. Is supports dashboard creation and allows for embedding them in your applications. With *AutoGraph* the system chooses the most appropriate visualization type according to the dataset schema. It supports most of the basic visualization/graph types.

When it comes to ML services, there are perhaps more offerings with Amazon *SageMaker*, *Comprehend*, *Personalize*, *Translate*, *Transcribe*, *Ground Truth*, being a few of them providing solutions to tasks like computer vision, natural language processing, recommendation systems, forecasting, and more. It also supports TensorFlow, PyTorch, Apache MXNet, Gluon, and more out of the box, and often with performance improvements[97]. Finally, it comes with various helpers for hyperparameter tuning and model selection.

[96] https://aws.amazon.com/blogs/big-data/10-visualizations-to-try-in-amazon-quicksight-with-sample-data/
[97] https://aws.amazon.com/sagemaker/

Google Data Studio and ML Engine[98]

Similarly to Amazon, Google has its own array of offerings. Diving straight into the available connections that *Data Studio* can handle, it also integrates with other Google services[99] (*BigQuery*, *Campain Manager*, *Google Ads*, *Google Analytics*, *Cloud SQL*, *Google Cloud Storage*, *Google Sheets*), popular databases (MySQL, PostegreSQL), and allows file uploads. Through the "partner connectors" you can also integrate with Amazon, Salesforce, Facebook, Instagram, Mailchimp, LinkedIn, Quora, Reddit, Pinterest, Twitter, and over 100 more. It also allows you to create your own dashboards and an extensive gallery[100] showcases some amazing works. With the exception of 3D scatterplots (for which I have found no reference), *Data Studio* provides all of the basic visualizations.

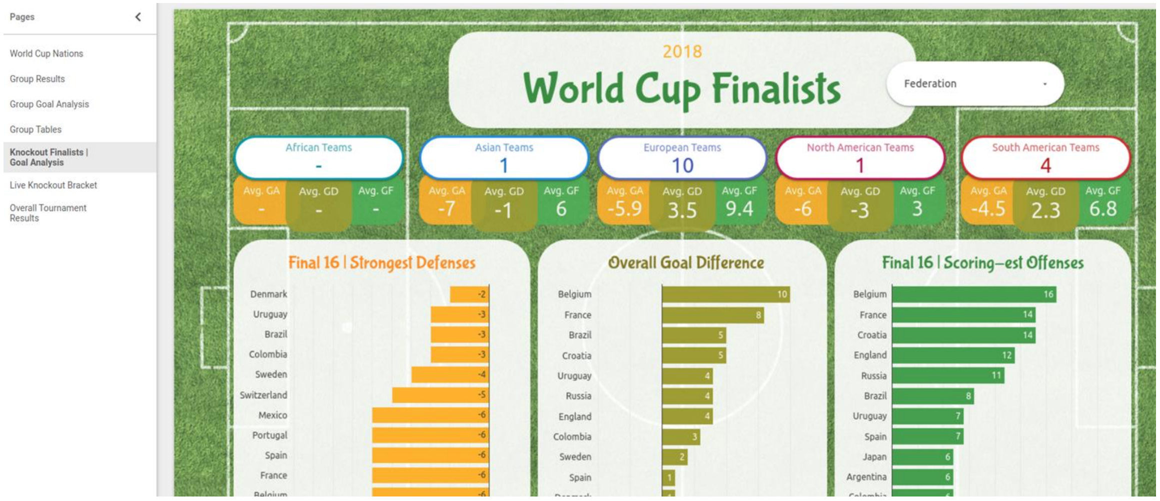


Figure 20 World Cup Finalists, https://datastudio.google.com/reporting/1Rg5y6r0640X8uo2xo2XY48sG9IyMiYEN/page/wcCU

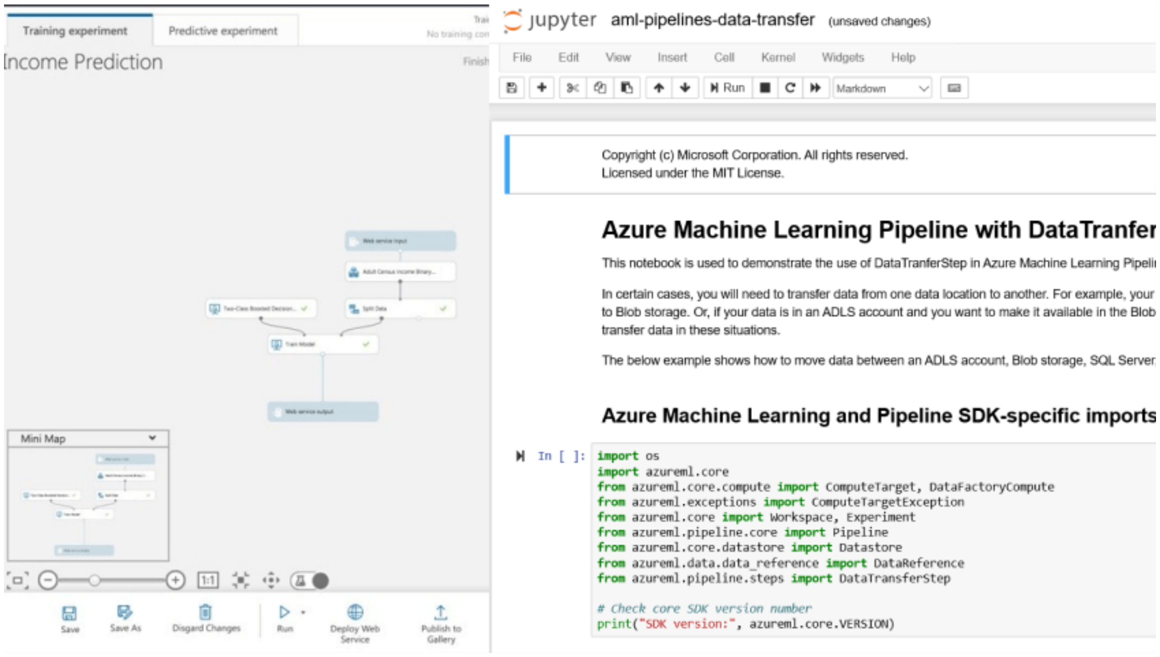
Again similarly to Amazon, Google provides its own Machine Learning services like the *Cloud Machine Learning Engine* which also allows for scalable training and comes preconfigured with scikit-learn, XGBoost, Tensorflow and Keras. If you want to add another there is the option of uploading your own Docker containers. It also provides access to CPU, GPU and TPU hardware. Furthermore, using *HyperTune* can help with

[98] https://datastudio.google.com/overview, https://cloud.google.com/ml-engine/
[99] https://datastudio.google.com/data
[100] https://datastudio.google.com/gallery

hyperparameter tuning. Finally, it too handles common usecases with more specific offerings in computer vision and natural language processing[101].

Azure[102]

The last on the list is Azure, which also provides a dashboard functionality, which is somewhat limited and we won't go into detail about that (after all, Microsoft's main visualization product is Power BI, and we already covered it).



Copyright (c) Microsoft Corporation. All rights reserved.
Licensed under the MIT License.

Azure Machine Learning Pipeline with DataTranfer

This notebook is used to demonstrate the use of DataTranferStep in Azure Machine Learning Pipeli

In certain cases, you will need to transfer data from one data location to another. For example, your
to Blob storage. Or, if your data is in an ADLS account and you want to make it available in the Blob
transfer data in these situations.

The below example shows how to move data between an ADLS account, Blob storage, SQL Server

Azure Machine Learning and Pipeline SDK-specific imports

```
In [ ]: import os
        import azureml.core
        from azureml.core.compute import ComputeTarget, DataFactoryCompute
        from azureml.exceptions import ComputeTargetException
        from azureml.core import Workspace, Experiment
        from azureml.pipeline.core import Pipeline
        from azureml.core.datastore import Datastore
        from azureml.data.data_reference import DataReference
        from azureml.pipeline.steps import DataTransferStep

        # Check core SDK version number
        print("SDK version:", azureml.core.VERSION)
```

Figure 21 Azure Machine Learning Service, https://azure.microsoft.com/en-us/services/machine-learning-service/

Azure's AI/ML offering comprises of services targeting the same fields as its competitors, with integrations with Databricks, ONNX, PyTorch, Tensorflow, scikit-learn. *Azure Machine Learning Service* promises to handle all stages from training up to deployment with "automated machine learning capabilities and open-source support". With *Jupyter Notebook* integration and the ability to run Python code, it is certainly very extensible and can handle custom data input. Furthermore, it provides both a flowchart for helping you choose a ML model[103], and automated ML[104].

[101] https://cloud.google.com/vision/, https://cloud.google.com/translate/
[102] https://azure.microsoft.com/en-us/overview/ai-platform/, https://docs.microsoft.com/en-us/azure/azure-portal/azure-portal-dashboards
[103] https://azuremlsimpleds.azurewebsites.net/simpleds/,

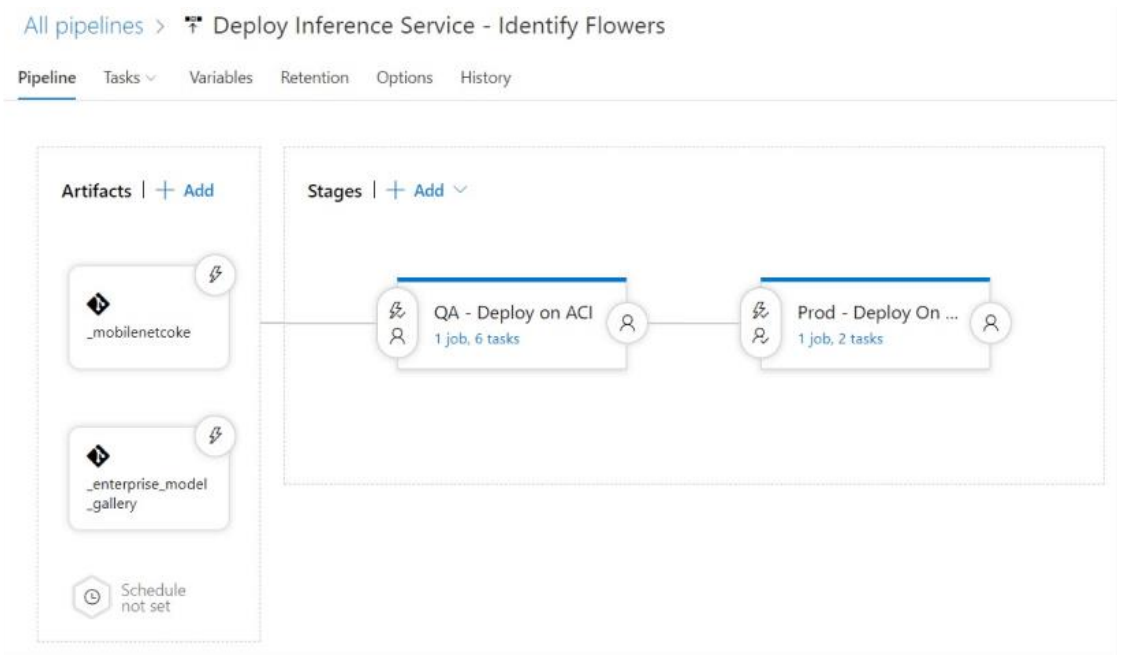


Figure 22 Azure Machine Learning Service, https://azure.microsoft.com/en-us/services/machine-learning-service/

[104] https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-automated-ml

3. Designing an open-source solution for data visualization and analysis

This projects stands on the shoulders of (mostly open-source) giants, without the work of whom this endeavor would not have been possible given the time window. These giants roughly sum up to around 40-50 independent projects. Although we will explain the whys and hows for each tool that was used in the next chapter, this small chapter will focus on very briefly introducing them and their capabilities. Seeing that Python was introduced in the previous chapter we will skip over it (same for JavaScript, it being *the* most popular language).

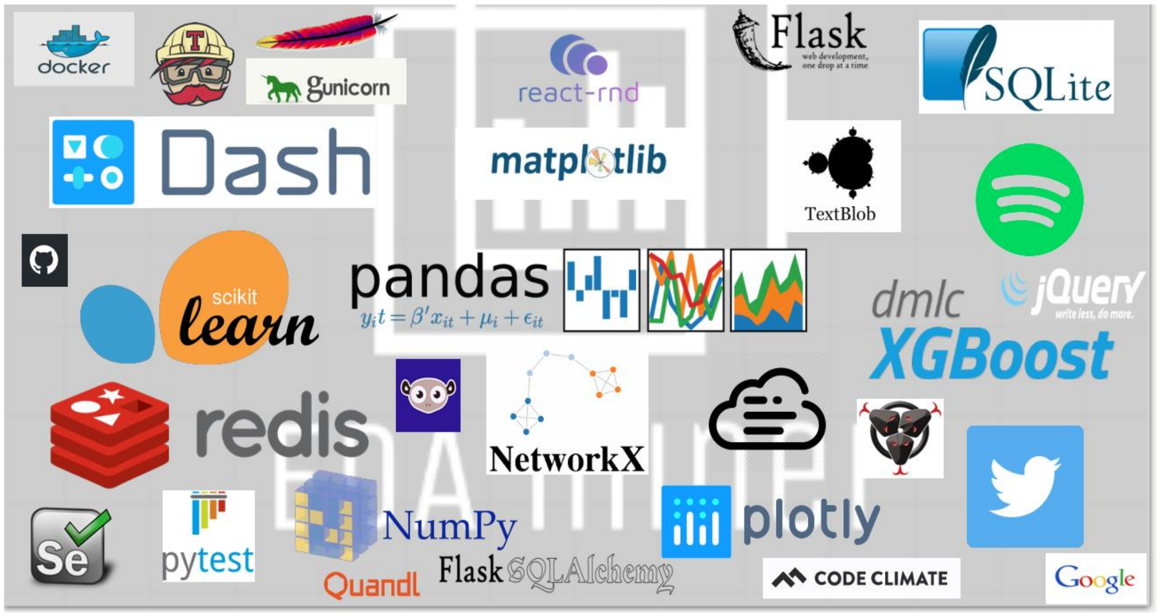


Figure 23 EDA Miner's main tech stack, v0.2+

3.1 Python libraries

3.1.1 Tech and computing stack

scikit-learn[105]

We mentioned *scikit-learn* (or *sklearn*) before multiple times, but briefly. It is a 12-year-old project that started as part of Google Summer of Code project by David Cournapeau[106] with the first publication dating back to 2011 (Pedregosa, et al., 2011) which has received a tremendous community following donations from individuals, organizations, universities. The GitHub repository numbers over 17.9K forks, 36K stars and 64K projects in which it is used, with a total of 1380 contributors and a *New BSD* license[107].

In its core, sklearn is a machine learning library with a very concise and well-designed "bare-bones" programmatic API (Pedregosa, et al., 2011, p. 3), that is build using *NumPy*, *SciPy*, and *matplotlib*, and boasts great performance with highly optimized and parallelizable models. Since the underlying libraries use *C* or *C++* code (which is precompiled) they can be made extremely efficient (Pedregosa, et al., 2011, p. 2), and sklearn added to that by adding multi-threading to the mix, a lot of it written in *Cython* (Pedregosa, et al., 2011, p. 3). It also uses *Fortran* libraries such as *LAPACK* (Pedregosa, et al., 2011, p. 3). A comparison with other libraries can be found in Appendix 3, as in their original paper (Pedregosa, et al., 2011, p. 4), which attributes *sklearn's* performance improvements to the reduced overhead due to less copying, among others (Pedregosa, et al., 2011, pp. 4-5).

The project provides a lot of documentation, which paired with all previous points makes evident how it came to dominate the ML world in Python. The sklearn docs provide a very detailed User Guide which goes into great depths explaining the models, their theory, and the typical use cases[108]. The complete user guide at the initial publication was estimated at over 300 pages (Pedregosa, et al., 2011, p. 3), and it probably has grown much more since then. A very popular aid is the *Flow Chart* which

[105] https://scikit-learn.org/stable/index.html
[106] https://scikit-learn.org/stable/about.html
[107] https://github.com/scikit-learn/scikit-learn
[108] https://scikit-learn.org/stable/user_guide.html

really helps beginners select the appropriate models for their tasks quickly. Not only that, but sklearn provides tutorials and examples (along with the hundreds other external sources like YouTube videos, Medium articles, etc).

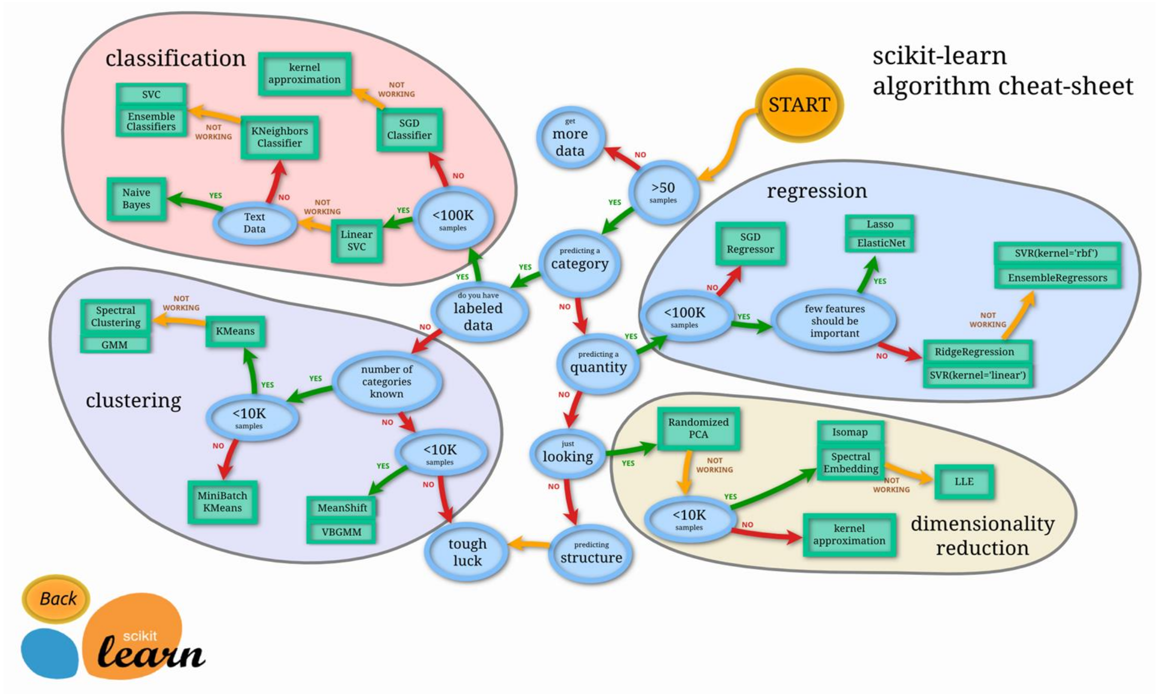


Figure 24 : SciKit-learn Flow Chart, https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

Probably the best feature of *sklearn* is the ability to choose to use the models in standalone mode or stack them in *Pipelines*[109] which are a great way to define more complex models ready for production environments. With simple inheritance from base classes and mix-ins, creating your own estimator and adding it to a pipeline is very easy[110]: just subclass the base and mix-ins you want, define the appropriate methods (e.g. fit, predict, transform) and you're good to go. It is this ease of use and clean API that made us choose it as a base for our project, and the same probably applies to *Orange3* and others.

SciPy[111]

[109] https://scikit-learn.org/stable/modules/compose.html
[110] https://scikit-learn.org/stable/developers/contributing.html#rolling-your-own-estimator
[111] https://www.scipy.org/

SciPy, or Scientific Python, is not only a Python library originally published in 2001 (Jones, Oliphant, Peterson, & et al, 2001), but a whole stack of libraries ("ecosystem") geared towards scientific computing sprung forth later. Both *sklearn* and the next few packages are also part of this ecosystem. It has its own conferences, a very large community, and a lot of other libraries that greatly enhance Python's performance (e.g. *Cython*, *Dask*, *PyTables*). It is a project used by nearly 120K others, with over 6.1K stars, 2.9K forks, and 750 contributors, with a BSD-3-Clause license[112]. It has lots of applications, from signal processing to image processing, sparse data, Fourier transforms, clustering, and more (e.g. peak finding, convolutions, decompositions, equation optimizers/solvers).

Numpy[113]

NumPy, or Numerical Python, is primarily used for operations with and on arrays. *NumPy* allows vectorization of operations, which essentially takes optimized and parallelized code written in *C* / *C++* / *Fortran* and provides a simple API. The *NumPy* project is used by nearly 234K others, with 11.4K stars, 3.8K forks and 800 contributors, with the same license as SciPy. Just like *SciPy*, *NumPy* is pretty old, with the initial publication dating back to 2006 (Oliphant, 2006).

pandas[114]

pandas, or Python Data Analysis Library, is the part of the SciPy ecosystem that focuses on data analysis tasks like data cleaning, missing values management, computing statistics, providing functionalities for time series, aggregations, I/O, merging / joining datasets, basic visualization, most of what a Data Analyst / Scientist needs for data handling (McKinney, 2010). It's main feature is the *DataFrame* which should be familiar to *R* programmers. This project is also immensely popular, being used by over 125.8K others, with over 20.7K stars, 8.2K forks, and 1550 contributors, with the same license as the previous[115].

[112] https://github.com/scipy/scipy
[113] https://www.numpy.org/
[114] https://pandas.pydata.org/
[115] https://github.com/pandas-dev/pandas

SymPy[116]

SymPy, or Symbolic Python, is the last library of the SciPy stack that we will go over. It is a library for symbolic mathematics but is evolving towards becoming a comprehensive "computer algebra system" (Meurer, et al., 2017). It handles differentiation and integrals, logic, concrete and discrete mathematics, geometry, working with polynomials (e.g. expansion / factoring) and many more (see example below). Perhaps the most interesting feature is the ability to convert symbolic expressions to Python / NumPy functions. The *SymPy* project is used by more than 11.3K projects, and has about 6.1K stars, 2.7K forks, 770 contributors and, again, a BSD-type license[117].

```
>>> from sympy import Symbol, cos
>>> x = Symbol('x')
>>> e = 1/cos(x)
>>> print e.series(x, 0, 10)
1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + 277*x**8/8064 + O(x**10)
```

Other libraries

We cannot introduce every library used (and their dependencies which usually are in the tens) so here is a list of other notable scientific libraries we used in our code:

- *networkx*[118] (Hagberg, Schult, & Swart, 2008)
- *peakutils*[119] (Hermann Negri & Vestri, 2017)
- *pygraphviz*[120]
- *python-Levenshtein*[121]
- *textblob*[122]
- *xgboost*[123]

[116] https://www.sympy.org/en/index.html
[117] https://github.com/sympy/sympy
[118] https://github.com/networkx/networkx
[119] https://bitbucket.org/lucashnegri/peakutils/src/master/
[120] https://github.com/pygraphviz/pygraphviz
[121] https://github.com/ztane/python-Levenshtein
[122] https://github.com/sloria/TextBlob
[123] https://github.com/dmlc/xgboost

- *fuzzywuzzy*[124]

3.1.2 Interface / web stack

Dash[125]

Dash is a Python library, developed by Plotly Technologies Inc. that, behind the scenes, uses Flask as the back-end server and React as the front-end JavaScript framework. It is also a rather popular project, used by over 4K other projects, with about 9.6K stars, 1K forks, and released under an MIT license[126]. The Dash App Gallery is another great place to take a first look at the project[127].

With Dash you can create web applications with HTML and CSS all within Python, and with the use of callback functions (a Python function decorated by *dash.Dash.callback*) that specify *Output* (what Dash/HTML element to change, e.g. a graph or a div), *Input* (what Dash/HTML element triggers -on change- the functions, e.g. "button X was clicked"), and optionally *State* (something that is read when a function is triggered by the Inputs, e.g. "what is the value of dropdown X when button Y is clicked?") you can manage the whole interactivity without writing a line of JavaScript. This comes at a small price, and we will discuss that in the next chapter, as well as other limitations and work-arounds.

Dash is extendable by creating your own (or from the community) components in *React.js*, and using a project boilerplate[128] you can easily make them available in Dash. You can also integrate *D3.js* with *Dash*[129]. The documentation is pretty extensive and covers various usage patterns in depth, allowing you to create complex applications, while there is also a repository with examples of advanced functionalities available on GitHub[130].

Dash is broken down in several libraries according to their domains. Some were incorporated from community contributions, others were paid by *Plotly* clients but were

[124] https://github.com/seatgeek/fuzzywuzzy
[125] https://dash.plot.ly
[126] https://github.com/plotly/dash
[127] https://dash-gallery.plotly.host/Portal/
[128] https://github.com/plotly/dash-component-boilerplate
[129] https://dash.plot.ly/d3-react-components
[130] https://github.com/plotly/dash-recipes

open-sourced. Here we will view a few of them, and some that are community-maintained (i.e. outside of *Plotly*), all of which we used in EDA Miner.

- *Dash Core Components*[131] is responsible for high-level components like graphs, dropdowns, sliders, check-boxes, and more.
- *Dash HTML Components*[132] contains (almost) every HTML tag like *div*, *button*, *p*, *span*, *h1-h6*, and even *script*.
- *Dash DAQ*[133], which was recently open-sourced, contains high-level components that are more geared towards controls like switches, gauges, joysticks, knobs, thermometers, and more.
- *Dash DataTable*[134] is a library for creating tables that are highly interactive, and much like spreadsheets you can edit values, add or remove lines or columns, sort, filter, have paging, and more.
- *Dash Cytoscape*[135] was also recently released (merely 1.5 months prior to this project's start, or 6 month at the time of writing), and is an extension of the popular *Cytoscape.js* library for network / graph visualizations.
- Dash Bootstrap Components[136] is an independent community project that ports the Bootstrap project into Dash, giving access to components like modals, navbars, tabs, jumbotrons, cards, and more.
- *Dash Core Components for Visualization*[137] is another external project that provides a few extra components: a component to run JavaScript, a network component, and a data-table.

Flask[138]

Flask is a Python web micro-framework, which means it doesn't make too many assumptions for you, nor does it wrap everything around a rigid and opinionated API[139].

[131] https://dash.plot.ly/dash-core-components
[132] https://dash.plot.ly/dash-html-components
[133] https://dash.plot.ly/dash-daq
[134] https://dash.plot.ly/datatable
[135] https://dash.plot.ly/cytoscape
[136] https://dash-bootstrap-components.opensource.faculty.ai/
[137] https://github.com/jimmybow/visdcc
[138] https://flask.palletsprojects.com/en/1.1.x/
[139] https://flask.palletsprojects.com/en/1.1.x/foreword/

That said, due to their design decisions their API is very simple and polished[140], and the scaling of *Flask* applications is not an issue[141]. It works by using *Werkzeug* and *Jinja* as the underlying mechanisms for a web server gateway interface (WSGI) and template engine respectively. Since sub-classing the *Flask* object is supported, a lot of applications are based off that, and most notably, the one we used: *Dash* (to access the *Flask* server object from the *Dash* object you call *Dash.server*).

Flask is, of course, open source as well. The GitHub page shows that is is an immensely popular project with more than 45.8K stars, 12.8K forks, and 550 contributors with a BSD-3-Clause license. It is used by more than 303.7K projects[142], with large companies like Pinterest (Steven Cohen cites that it handles "over 12 billion requests" daily (Cohen, 2015)) and LinkedIn (0:45 – 1:15 LinkedIn talk (Sanders, 2014)) being some of them.

As we said, Flask can be sub-classed to create wrappers, but the primary focus of the development community is designing extensions (Sanders, 2014) and we will see a few of them here:

- *Flask-Login*[143] is an extension that helps with user authentication, session management, "remember me", and anonymous users, among others. What is lacks is handling of permissions (an interesting Flask extension is *Flask-Security*), registration, and account recovery. As a GitHub project, it has nearly 2.2K stars and 500 forks, with 78 contributors and an MIT license[144].
- *Flask-WTF*[145] is an extension that helps with Forms, the validation of input, CSRF protection, reCAPTCHA and more. The GitHub project uses a BSD License, is used by about 45.3K other projects with nearly 1K stars and 240 forks, with 69 contributors[146].

[140] https://flask.palletsprojects.com/en/1.1.x/design/
[141] https://flask.palletsprojects.com/en/1.1.x/becomingbig/
[142] https://github.com/pallets/flask
[143] https://flask-login.readthedocs.io/en/latest/
[144] https://github.com/maxcountryman/flask-login
[145] https://flask-wtf.readthedocs.io/en/stable/
[146] https://github.com/lepture/flask-wtf

- *Flask-Mail*[147] is an extension for that provides a simple API to handle mails via SMTP, using a simple dictionary for passing configurations. It is also popular, with 22 contributors, more than 100 forks, 400 stars, and 12.8K projects using it, with a BSD license[148].
- *Flask-SQLAlchemy*[149] is the Flask extension developed by the core Flask team that supports the Python library SQLAlchemy. It is, essentially, an Object-Relational Mapping (ORM). Used by more than 79.6K projects, with about 2.7K stars, 700 forks, and 82 contributors, it is also shared on GitHub under a BSD-3-Clause license[150].
- *Flask-Caching*[151] is the last Flask extension on our list, which provides caching support using any of Werkzeug's caching back-ends (we use it with the preconfigured Redis) plus custom ones via sub-classing. The GitHub projects sits on the less used side, with about 2.2K projects using it, 400 stars, 70 forks, 60 contributors, and a mixed BSD license[152].

Others

We used other frameworks and libraries too, but the limited usage (and the limited space here) won't allow us to go into too much detail, especially since the Dash code-base is bigger by far. We will only focus on JavaScript (JS) which, in itself, needs no introduction. We used a bit of "vanilla" JavaScript but the two frameworks we mostly used were React and jQuery:

- *React*[153] is an open-source library / framework for creating component-based interactive elements. Writing JSX (essentially JS with a few extras) to extend the React base Component class, compiling with Babel, and optionally using any library (e.g. installed via the Node Package Manager), you can easily create complex components and single-page apps. React is developed by Facebook,

[147] https://pythonhosted.org/Flask-Mail/
[148] https://github.com/mattupstate/flask-mail
[149] https://flask-sqlalchemy.palletsprojects.com/en/2.x/
[150] https://github.com/pallets/flask-sqlalchemy
[151] https://flask-caching.readthedocs.io/en/latest/
[152] https://github.com/sh4nks/flask-caching
[153] https://reactjs.org/

initially released in 2013 (Occhino & Walke, 2013), and the GitHub code is distributed with an MIT license. It is the most popular library of what we saw with 1.3K contributors, 134K stars, 25K forks, and a whooping 2.33M projects using it[154].

- jQuery[155] also a JS library but also comes packages with User Interface elements, and even though it is a bit older, being first published in 2006 (York, 2009), and despite its GitHub page being much less popular (52K stars, 18.4K forks, 347K usages, 275 contributors, MIT license[156], it has been steadily increasing in usage across the most popular websites: among the top million websites it was reportedly used in 63% of them in 2015 (https://www.maxcdn.com/blog/maxscale-jquery/), 69% in 2017 (http://web.archive.org/web/20170219042532/https://libscore.com/) and 73% in 2018 (https://trends.builtwith.com/javascript/jQuery).

3.2 Database technologies

The use of databases throughout the application serves multiple purposes and we will go over them in the next chapter. Before that, we will briefly take a look at the ones we currently use and a few details about them. For SQL databases we are using SQLA*lchemy* as an intermediary (see previous section on Flask-SQLalchemy).

3.2.1 Redis

Redis[157] is an open-source (with a BSD-3-Clause license) NoSQL database, whose purpose is to be an "in-memory data structure store". The most popular use case is a key-value store, where according to G2[158] and DB-engines[159] it is considered the top contender. The GitHub repository has over 38.1K stars, 14.7K forks, and 300

[154] https://github.com/facebook/react/
[155] https://jquery.com/
[156] https://github.com/jquery/jquery
[157] https://redis.io/
[158] https://www.g2.com/categories/key-value-stores
[159] https://db-engines.com/en/ranking/key-value+store

contributors, with very active development having more than 700 open pull requests[160]. It has front-ends to around 50 programming languages[161] and a very simple API with clear documentation[162]. Related to key-value storage, use cases include caching, and shared-memory-type access of resources. It is also capable of working as a message broker, handle streams and PUB/SUB, and even on-disk persistence. Finally, it is able to scale really well with extensions like Redis Sentinel and Redis Cluster. And it is fast; you will find some benchmarks in the next chapter.

3.2.2 SQLite

SQLite[163] is probably *the* most popular database. If you used Chrome or Mozilla to navigate the internet, or have an Android phone or an iPhone, or have a Windows 10 or Mac OS-X computer, then chances are it is backed by an SQLite database, or a few. Of course, measuring popularity can be done differently, and according to DB-engines it ranks 11th [164]. According to the SQLite website, there are more than a trillion databases in use[165].

As the name implies it is a "lite" version, which means it doesn't support all the commands you would typically expect of the larger SQL databases. It, too, is an open-source database which is in the Public Domain and boasts to be "uncontaminated", which means that they don't take lightly to new contributors[166]. Furthermore, SQLite is by default supported in Python, and is ported as part of the standard library[167]. According to the same docs, a common usage pattern is to use it for prototyping / development, and then move on to another provider like PostgreSQL or Oracle. That said, SQLite is not a "dummy database", it is in fact used in production by some of the biggest companies like Facebook, Dropbox, Apple, Google (Chrome Web Browser, Android) for various of their products[168].

[160] https://github.com/antirez/redis
[161] https://redis.io/clients
[162] https://redis.io/documentation and https://redis.io/commands
[163] https://www.sqlite.org/index.html
[164] https://db-engines.com/en/ranking/
[165] https://www.sqlite.org/mostdeployed.html
[166] https://sqlite.org/copyright.html
[167] https://docs.python.org/3/library/sqlite3.html
[168] https://www.sqlite.org/famous.html

3.3 Other libraries and tools

This project also uses a variety of other tools, methods, and techniques. We will see them in the next chapter in more detail so here we will only list them here for completeness' sake.

- REpresentational State Transfer (REST) APIs are software architecture techniques and self-imposed rules on the behavior of applications to make interfacing multiple of them easier (W3C Working Group, et al., 2004). These rules, or *constraints*, instruct that the REST service follows a client-server architecture, it does not track state, it is cacheable, and has a uniform interface (usually using URIs, HTTP, and JSON) (Erl, Carlyle, Pautasso, & Balasubramanian, 2012). The "Micro-Services" architecture is very often using REST-type APIs for communication among services. It is also a great way to lessen the load of your servers (e.g. both Wikipedia and Twitter share their data via APIs). REST APIs are expected to also return appropriate status codes, some examples can be found in restfulapi.net[169].
- *Oauth2* is used to handle authorization, mainly for accessing Google and other web API resources (e.g. Google Drive). It is a widely accepted protocol that allows authorization of only specific resources (Barbettini, 2018). We are using the *google-auth*[170], *google-auth-oauthlib*[171], and *google-api-python-client*[172] Python packages.
- Other APIs and the Python libraries used for interfacing with them include Quandl / *quandl-python*[173], Reddit / *praw*[174], and Spotify / *spotipy*[175].
- *redis-py*[176] is used for interfacing with Redis.

[169] https://restfulapi.net/http-status-codes/
[170] https://github.com/googleapis/google-auth-library-python
[171] https://github.com/googleapis/google-auth-library-python-oauthlib
[172] https://github.com/googleapis/google-api-python-client
[173] https://github.com/quandl/quandl-python
[174] https://github.com/praw-dev/praw
[175] https://github.com/plamere/spotipy

- *matplotlib* is also used for creating pairplots, and it is converted into plotly's format by the latter's tools.
- *word_cloud*[177] to generate word-cloud images.
- *Selenium* (via Python) for controlling the browser and handling testing.
- *py.test*[178] for testing Python applications, together with *pytest-cov*[179] for parsing coverage reports.
- *dill*[180] as a more capable alternative of Python's *pickle* module.
- *requests*[181] is a library that simplifies HTTP requests in Python.
- *Docker*[182] is a layer between your application and the host operating system allowing for virtualization and delivery of software in small packages called *containers*. It is also used often in Micro-Services architectures. It started in 2013[183] and its widespread adoption along with tools that interface with it (cloud providers' specialized services, Kubernetes, Docker-Swarm) have brought a small revolution in the "DevOps" world.

[176] https://github.com/andymccurdy/redis-py
[177] https://github.com/amueller/word_cloud
[178] https://github.com/pytest-dev/pytest
[179] https://github.com/pytest-dev/pytest-cov
[180] https://github.com/uqfoundation/dill
[181] https://github.com/psf/requests
[182] https://www.docker.com/get-started
[183] See "Milestones" section at https://www.docker.com/company

4. Implementing the EDA Miner tool

In this chapter we will go over the considerations and design decisions, the implementation, and everything else related to the codebase. We will explain our motivations and thinking process, explore benchmarks, and speak of concrete future plans that don't fit in the "future work" chapter. We will also go over the specific issues we identified (e.g. project code organization, inter-app authentication) that iteratively changed the agile design of our project. It should also be noted that, unless otherwise specified, all benchmarks in this chapter are not rigorous[184], that is they are might vary across computers or other supporting software versions, and are run under no external load, only on a local computer with the following specs:

- CPU: Intel i7-8700K
- RAM: 32GB RAM
- GPU: GTX 1080, driver 418.56, CUDA 10.1
- OS: Ubuntu 18.04.3 LTS 64-bit, Linux: 4.15.0-58-generic

It should be made clear upfront that not much optimization has been made during development. We consider the application to still be in (pre-) alpha stage. Furthermore, although we did try to improve styling, it has not been our main focus. The main focus of the development so far was to roll out as many features as fast as possible (up to July), clean the code and use object-oriented design patterns to improve code quality even further including documentation writing and unit testing (it is a continuous process but August was "refactoring month"), and then once again add new features primarily geared towards actual use-cases (e.g. recommender systems, SQL integration, PDF / report generation) and more UI/UX.

Furthermore, the main target group of the application has been data scientists and non-technical people that are working with datasets smaller than 25 GB, that is datasets that could potentially fit the RAM of a (strong yet) local machine or a contained cloud

[184] No effort was expended in isolating benchmark processes from the OS and/or other processes, or trying to negate the effects of (unplanned) hardware/OS caching.

instance. This means that no Big Data tools were used, and although scaling of each individual component *was and is* a major consideration, discussions about scaling beyond one instance of the main server have mostly been theoretical. This decision was made after conducting a draft poll, on July 23, in a popular Facebook group (due to budget and time constraints) called "Artificial Intelligence & Deep Learning" with about 280K members. The results of the poll (multiple answers can be selected, and the denominations were arbitrarily decided) can be viewed in the histogram below. Although the results of this poll should not be taken too seriously as being representative, about 45.1% of the votes use datasets less than 2 GB, and 56.7% less than 25 GB. Additionally, we should consider that of those working on the higher end of the spectrum, a lot are working with images, videos, and other multi-media content, applications for which this

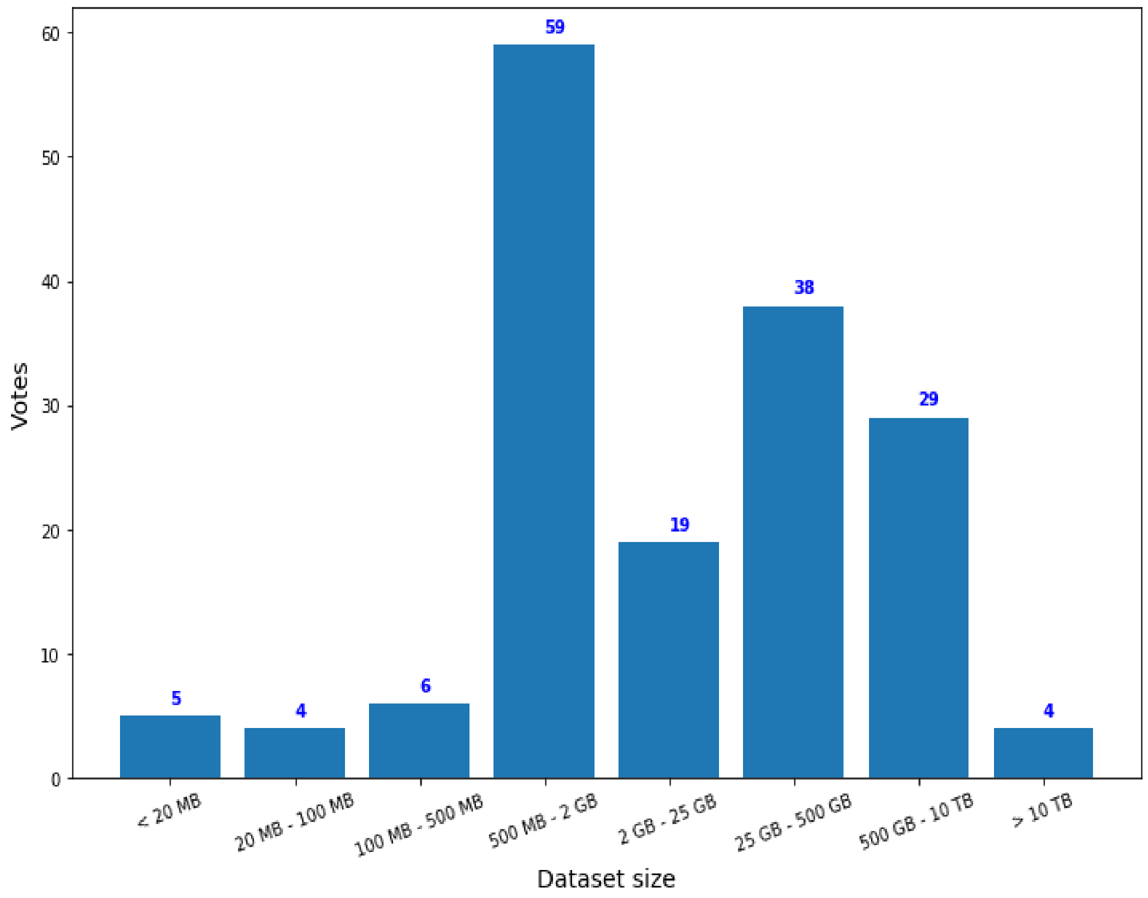


Figure 25 Dataset size poll in Facebook group Artificial Intelligence & Deep Learning (https://www.facebook.com/groups/DeepNetGroup/permalink/911122442613972/)

project does not account for.

4.1 Project Structure: Dash & Flask

Dash is the backbone of the project, and the early versions (up to v0.3) were Dash-only. Applications with Dash work by defining a layout, which is a hierarchically structured container of components that map to HTML or React elements, and then defining callbacks, which are decorated Python functions that take their arguments in the form of Input (Dash element that triggers the callback) and State (Dash element that is read when the callback is triggered), and output a JSON-serializable Python object (e.g. dict, Dash elements, str, list). These callbacks are registered in the main *dash.Dash* instance.

The first direct consequence of the above is that only one callback function can mutate a certain Dash element. For example, two different buttons cannot be tied to two different functions to mutate a certain *div*. There are only two known workarounds for this. The official way is to combine the callbacks in one function, which very often leads to increased complexity, function size, and lots of if/else branching within functions; increasing more than linearly in complexity as the number of Input/State elements increases. A second way, which is mostly a hack, is to use custom JavaScript, from Python (e.g. using extra package *visdcc.Run_js*) or otherwise. We use both methods (e.g. see *utils.py* > *interactive_menu,* and in v0.2 see *apps* > *analyze* > *Model_Builder.py* > *modify_graph*). Although not exactly different ways, one could break individual operations into separate functions that return data to separate hidden divs and combine those, and/or offload the logic completely to Python (both of which usually decrease complexity, e.g. in v0.4 see *modeling* > *model_builder.py* > *update_nodes*).

The second consequence is that callbacks that are triggered by some input will throw errors if not all of the Input/Output/State elements are present. This can mostly be fixed either by tweaking Dash parameters (swapping Input/State elements), or, in more advanced cases, adding the elements in the layout beforehand but keeping them hidden.

The third, and worst of all, is that "all callbacks must be defined before the server starts". This has a lot of practical implications and directly bars some very common uses such as the "add another item" pattern. The suggested workaround from the documentation and forums is to define all possible elements beforehand and keep them hidden (see this extreme example: callback for dynamically created graph[185]).

If you noticed that the previous two paragraphs ended with "define and keep hidden", then you are in a good spot. This is probably the most common pattern of Dash (hereafter referred as "hidden div" pattern/solution) and is the suggested way for moving data across callbacks, having callbacks share the same data, caching, and reducing network traffic.

Which brings us to our next topic. Novice Dash developers that want to pass data across functions will be tempted to use global variables, which usually works fine if only one user uses the app, but that is often not the case. Variables in the global scope are not scoped on a per-session basis, which will lead to one user overwriting another's data[186]. We dealt with this both with the hidden div solution (e.g. reporting metrics for trained models) and with Redis.

As was mentioned briefly before, using custom JavaScript is not well-integrated into Dash, but is possible. However, the best way to extend Dash is to use React to develop custom components. This tends to increase complexity quite a lot, and usually this results to a lot of bloat (node packages), but great resources exist to help with both of these issues. Adding scripts is similar to adding CSS[187], but even then there are limitations (e.g. due to the way Dash elements are rendered, scripts might be unable to find some of them, thus causing errors). Technically, using other JS frameworks is not possible[188] but it can be made to work (both in Dash, and within React when designing custom components).

Furthermore, it needs to be expressed again that the price paid for dealing with the issues mentioned above is more than compensated by how Dash rewards you with interactivity. Despite all the problems that cause us to find workarounds and hacks, Dash is still incredibly expressive, and this is what makes this whole project interesting. In terms of judging the size of the codebase in source lines of code, although it is still a

[185] https://community.plot.ly/t/callback-for-dynamically-created-graph/5511/4
[186] https://dash.plot.ly/sharing-data-between-callbacks
[187] https://dash.plot.ly/external-resources
[188] https://dash.plot.ly/faqs

work in progress and lacks in various aspects, this project ranks pretty well when compared to peers: it consists of only ~6.2**K** lines of Python code. Here is a small list of other similar software:

- DIVE: 32**K** (back-end: 9.2**K**, front-end: 22.8**K**)
- Orange3: 74.7**K**
- RapidMiner Studio: 383.2**K**
- Kibana: 704.4**K**

Before moving further we would like to stress the last point a bit more. One side-benefit of the brevity of the code is that the latter is easy to go through for newcomers, and debug for maintainers and contributors. We are actively using the *Looks Good To Me* (LGTM[189]) tool (which also provided the line count above) to monitor code quality and fix spurious parts in our code (see illustrations below).

[189] https://lgtm.com/

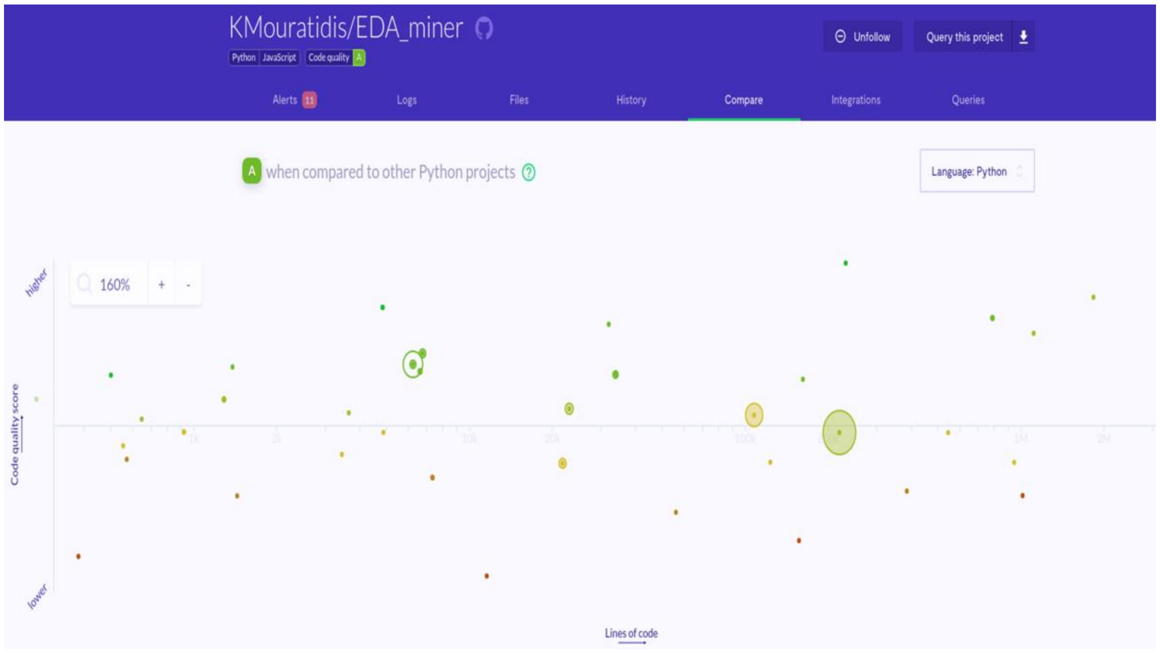
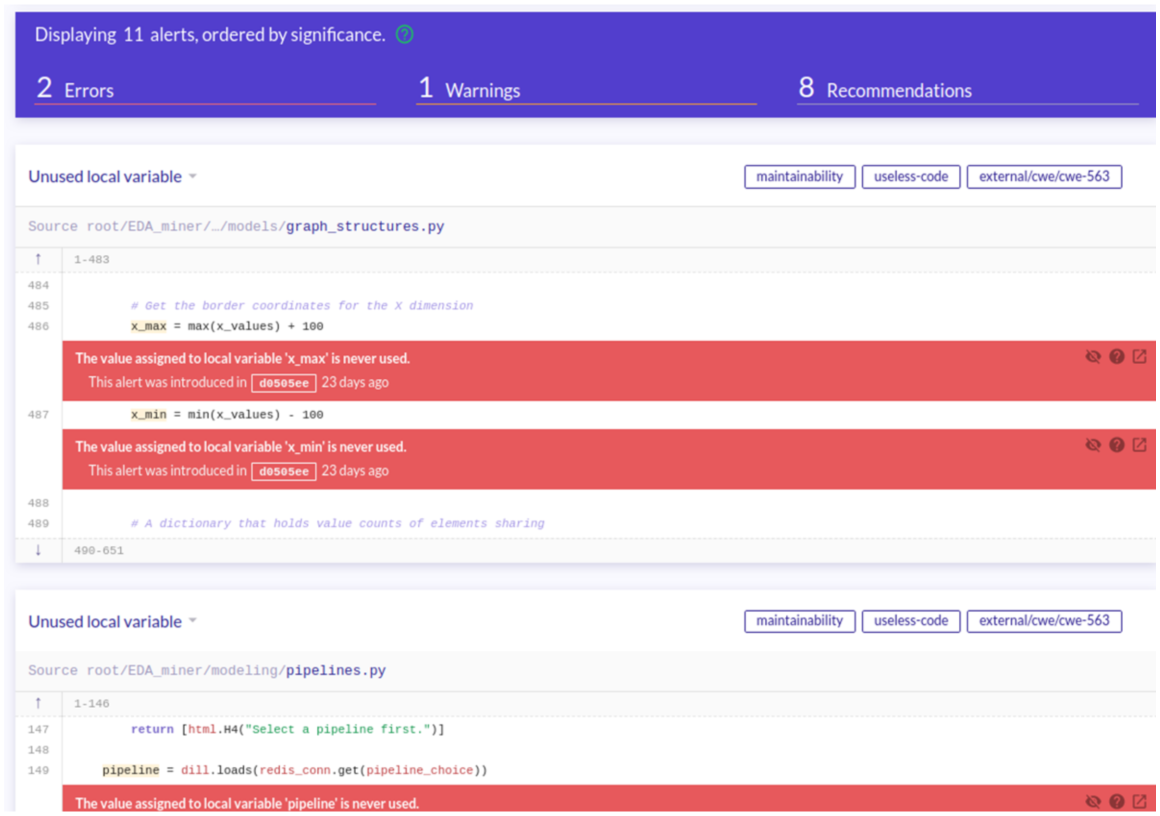


Figure 26: LGTM: EDA Miner comparison to other Python projects, https://lgtm.com/projects/g/KMouratidis/EDA_miner/context:python


Displaying 11 alerts, ordered by significance.

2 Errors 1 Warnings 8 Recommendations

Unused local variable

maintainability useless-code external/cwe/cwe-563

Source root/EDA_miner/…/models/graph_structures.py

```
        # Get the border coordinates for the X dimension
        x_max = max(x_values) + 100
```

The value assigned to local variable 'x_max' is never used.
This alert was introduced in d0505ee 23 days ago

```
        x_min = min(x_values) - 100
```

The value assigned to local variable 'x_min' is never used.
This alert was introduced in d0505ee 23 days ago

```
        # A dictionary that holds value counts of elements sharing
```

Unused local variable

maintainability useless-code external/cwe/cwe-563

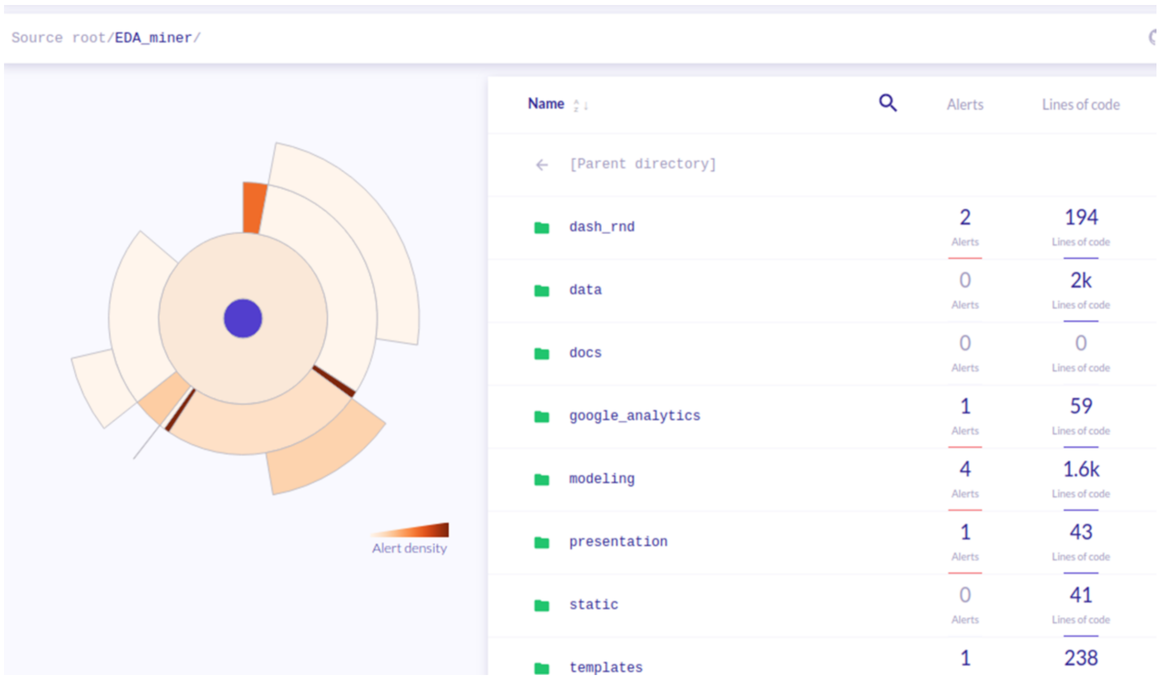
Source root/EDA_miner/modeling/pipelines.py

```
        return [html.H4("Select a pipeline first.")]

    pipeline = dill.loads(redis_conn.get(pipeline_choice))
```

The value assigned to local variable 'pipeline' is never used.

Figure 27: LGTM: EDA Miner alerts & code quality, p.1



Source root/EDA_miner/

Name	Alerts	Lines of code
[Parent directory]		
dash_rnd	2	194
data	0	2k
docs	0	0
google_analytics	1	59
modeling	4	1.6k
presentation	1	43
static	0	41
templates	1	238

Figure 28: LGTM: EDA Miner alerts & code quality, p.2

One good example of the code's expressiveness is how we have implemented navigation throughout the app. Especially in earlier versions (prior to v0.2), using three HTML/Dash tabs (the choice doesn't matter, buttons, URL, or anything else would work too) the user navigated through sub-apps, each of which was a layout (list of Python & Dash components) returned by a central callback. Each of the apps had an identical callback managing its own sub-pages. As a result, the whole project was a big one-page app that was quite fast (see illustrations below for before/after).

However, since v0.3 the need for compartmentalization and scaling led to breaking up each app into its own Dash app (we did keep the navigation inside apps the same), and all of them are connected as the WSGI level with the main Flask app. It is very easy now for completely independent Dash or Flask (and potentially Django) apps to be integrated. It is also quite easy to share Flask extensions across apps (e.g. the login, the main database). The performance was hit considerably though, and ironic as it may seem that performance took a hit when one of our goals was scaling it might not be entirely true. One of the main drivers of the performance hit is mostly due not having optimized distribution of static files (which are now smaller on a per-app basis), and, secondly, because each app is essentially its own server. This leads to a need for more restructuring.

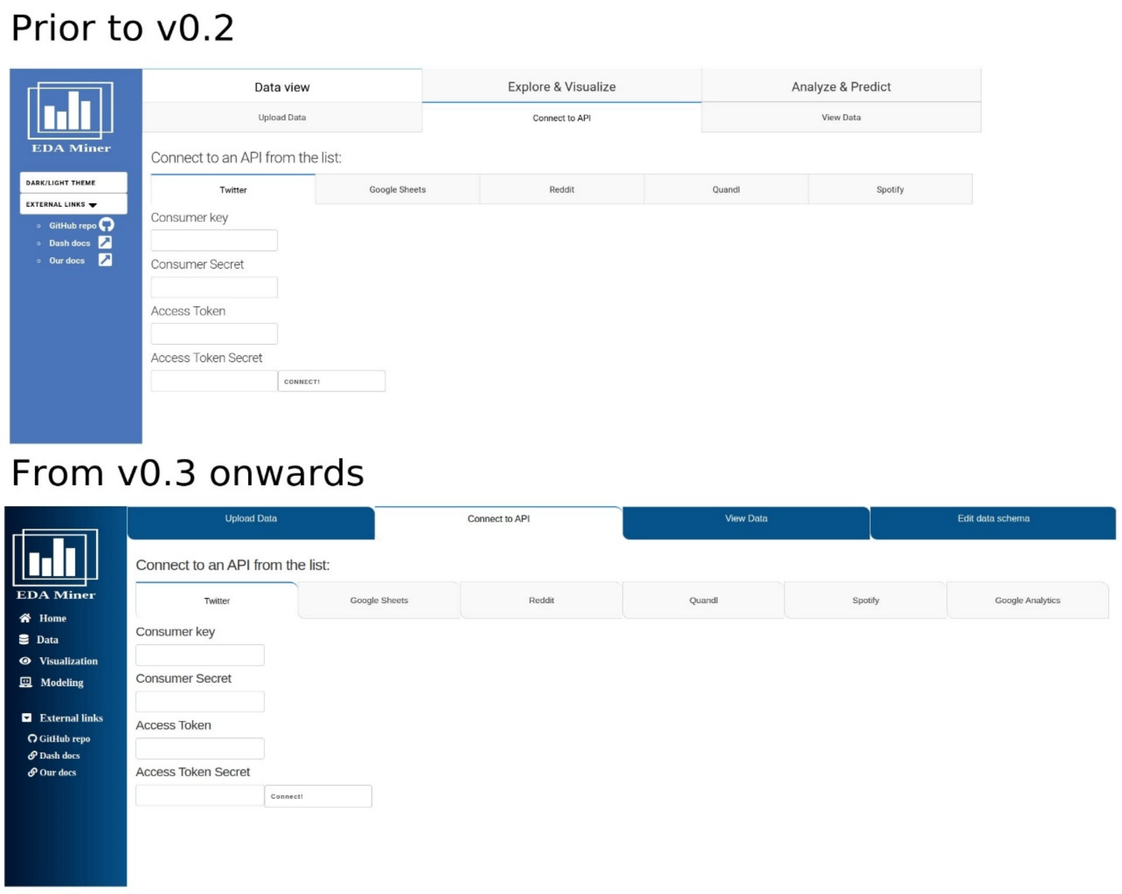
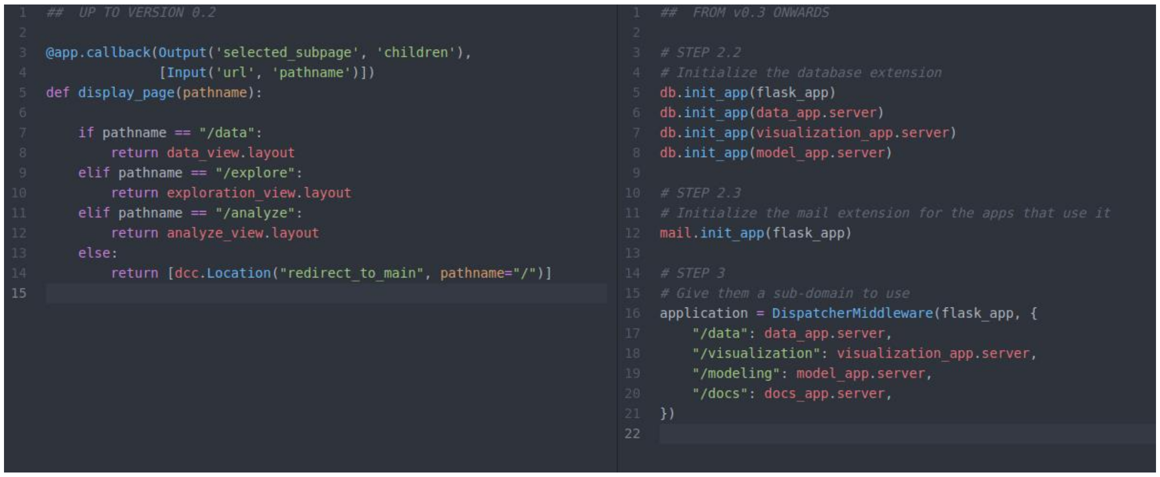


Figure 29: EDA Miner, app navigation, pre-v0.2 vs. v0.3+



```python
##  UP TO VERSION 0.2

@app.callback(Output('selected_subpage', 'children'),
              [Input('url', 'pathname')])
def display_page(pathname):

    if pathname == "/data":
        return data_view.layout
    elif pathname == "/explore":
        return exploration_view.layout
    elif pathname == "/analyze":
        return analyze_view.layout
    else:
        return [dcc.Location("redirect_to_main", pathname="/")]
```

```python
##  FROM v0.3 ONWARDS

# STEP 2.2
# Initialize the database extension
db.init_app(flask_app)
db.init_app(data_app.server)
db.init_app(visualization_app.server)
db.init_app(model_app.server)

# STEP 2.3
# Initialize the mail extension for the apps that use it
mail.init_app(flask_app)

# STEP 3
# Give them a sub-domain to use
application = DispatcherMiddleware(flask_app, {
    "/data": data_app.server,
    "/visualization": visualization_app.server,
    "/modeling": model_app.server,
    "/docs": docs_app.server,
})
```

Figure 30: EDA Miner, app navigation code, pre-v0.2 vs. v0.3+

At start, we did not intend for the apps to be connected as they are today at the WSGI level, but rather for each to be a completely independent entity connected via an HTTP Proxy server. This would allow for much easier load balancing across the whole app, but also within apps, especially as a containerized application with Docker-Swarm (or similar) integration. The problem with this approach is that we would need to implement an inter-app authentication and authorization system (which could of course be implemented in a less-correct but much simpler way), as well as the actual HTTP proxy configuration. As a result this was left for later than v0.4.

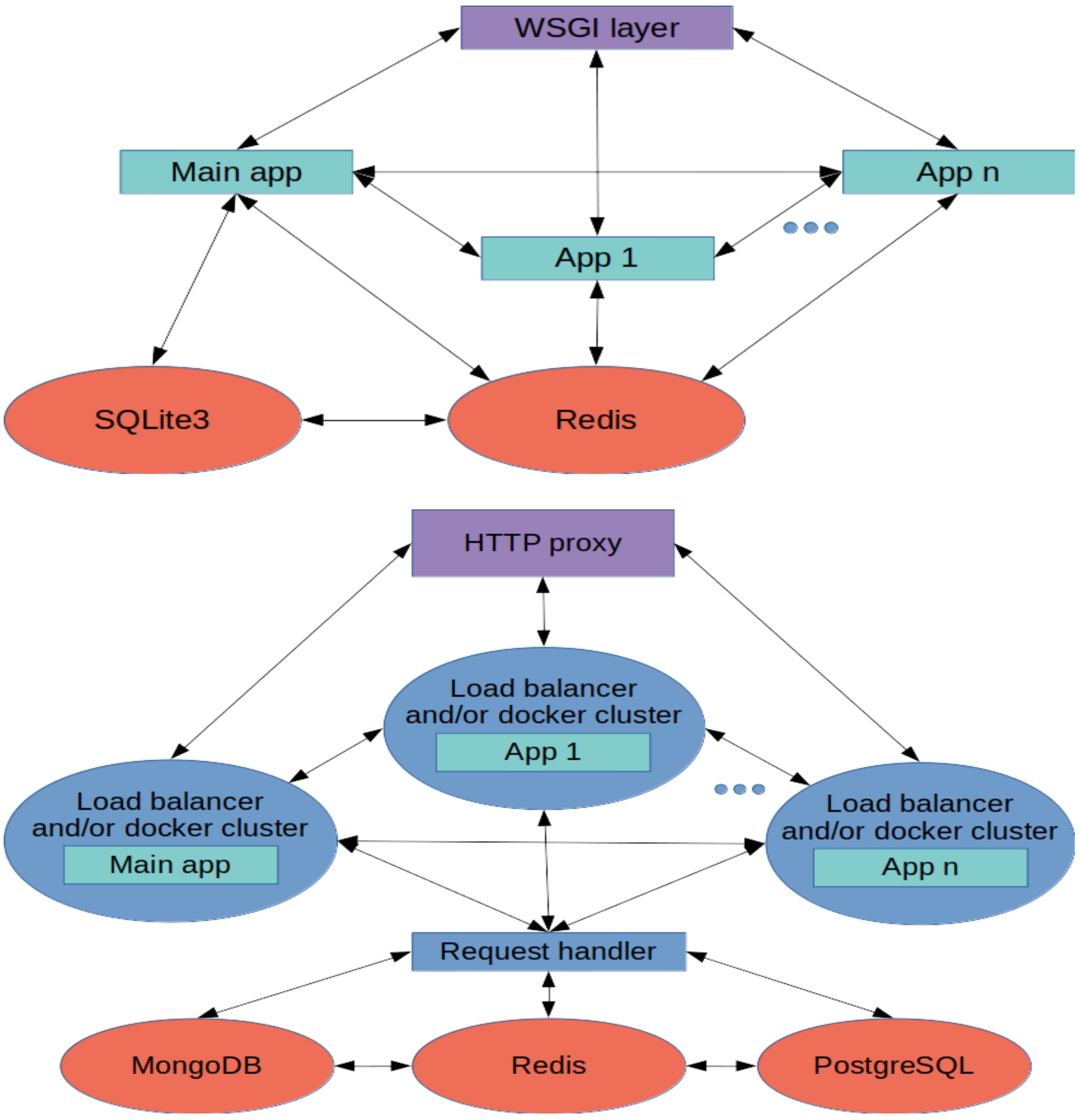

WSGI layer
Main app
App n
App 1
SQLite3
Redis
HTTP proxy
Load balancer and/or docker cluster
App 1
Load balancer and/or docker cluster
Main app
Load balancer and/or docker cluster
App n
Request handler
MongoDB
Redis
PostgreSQL

Figure 31: EDA Miner architecture, high-level overview of v0.3 - v0.4, and future plans.

4.2 Per-app implementation notes

4.2.1 Data app

The Data app is responsible for handling all ingestation and modifications operations of data. Currently there are four processes that are supported, each under its own tab. We will quickly review each one of them, going into detail only when necessary. We will also discuss future plans and extensions.

First is the "Upload Data" tab where the users can upload data with a simple button or drag-n-drop process. We used *dcc.Upload*[190] with the parser that is given in the docs. We also added support for a few more file types using Python's and pandas' functionalities. Additionally, we used Weston John's fork[191] of *dash_resumable_upload* to handle larger files (we tried with file-sizes up to 1GB).

Second is the "Connect to API", which takes user credentials from a form, or gets authorization with some other method (e.g. uploading credentials, or via oauth2), and presents a user interface so that the user can select the data they want. For example, for Twitter it presents an *input* field where the user can input an account name and the app will fetch their tweets. Initially, when developing this you had to write code in a few different places, and include Dash elements and callbacks to the layout for every new API connection. Instead of that, everything is now abstracted away as follows: programmers are now expected to implement an abstract base class called APIConnection, and then the rest of the code will use its data and functions to construct the UI. This is easily achieved in Python by using the *inpect* module which has methods that provide access function signatures. It provides the following functions: *save_data_and_schema* (given a dataframe and a name, it infers the schema and saves the data) and *render_layout* (depending on whether the user has previously connected to this API or not, return a form for getting authorization or a UI for fetching data). The user is expected to implement 2 methods, 1 static method, and 2 properties. The two properties are simply Dash layouts (the authorization form & the UI). The two methods, which map to the two aforementioned properties, respectively: *connect* which handles all the authorization

190 https://dash.plot.ly/dash-core-components/upload
191 https://github.com/westonkjones

login including changing the state of the object, and *fetch_data* which handles the requests for data from the connected API. Finally, there is a static method called *pretty_print* which given a dataframe (after *fetch_data* success) it displays the first few results in a user friendly manner. Each of the implementation classes is used to create objects for each user attempting a connection, and hold all of the relevant data for that connection. The existing implementations use memoization to lessen the load on the server, and future contributors are advised to follow this example but adjust the cache expiration timer according to each API's needs (e.g. Quandl's data don't change as often

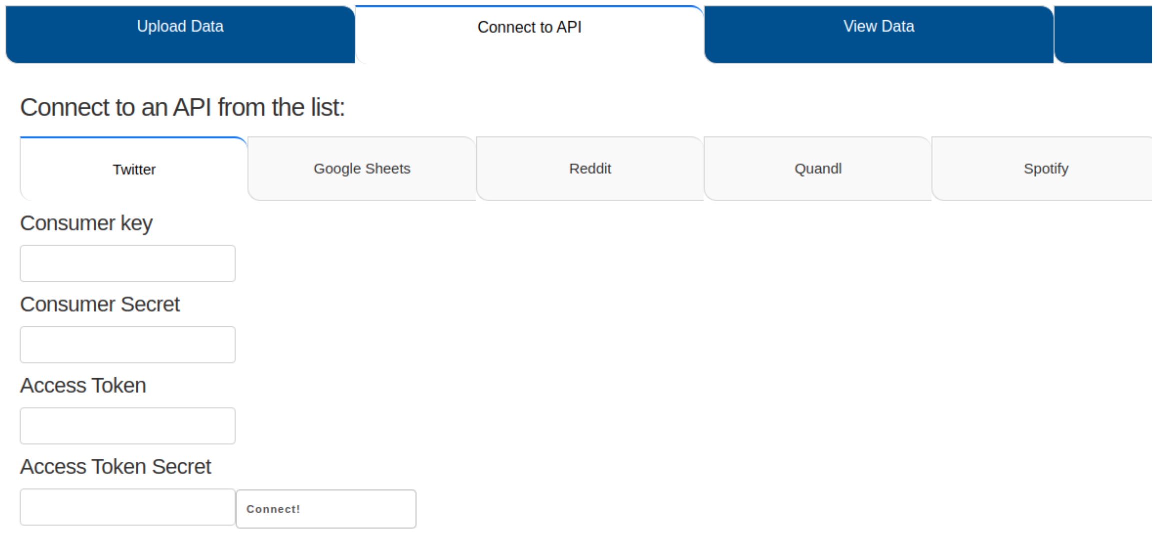
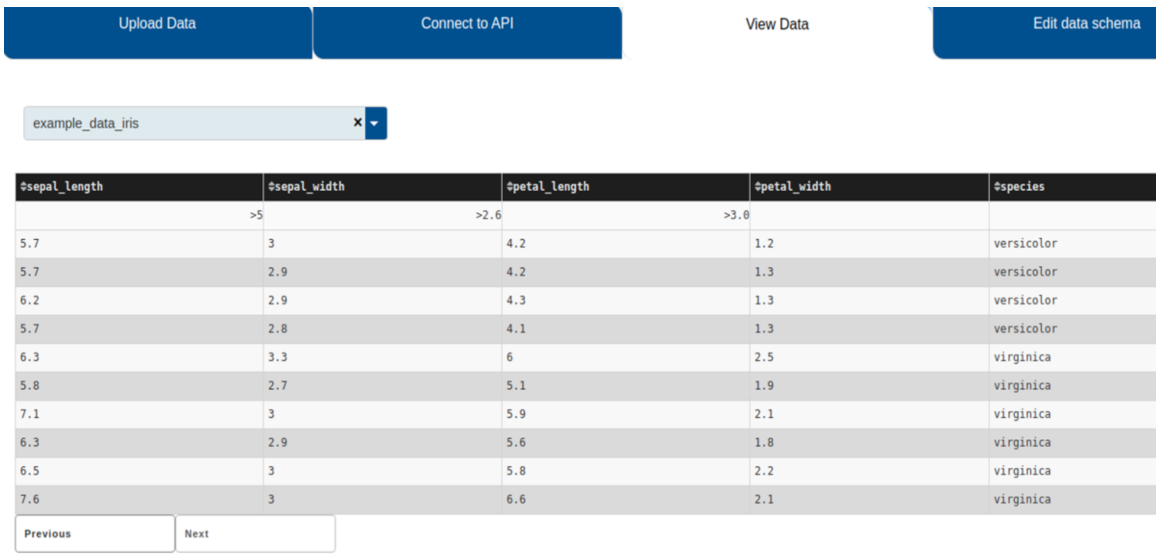


Figure 32: EDA Miner / Data / Connect to API

as Twitter's).

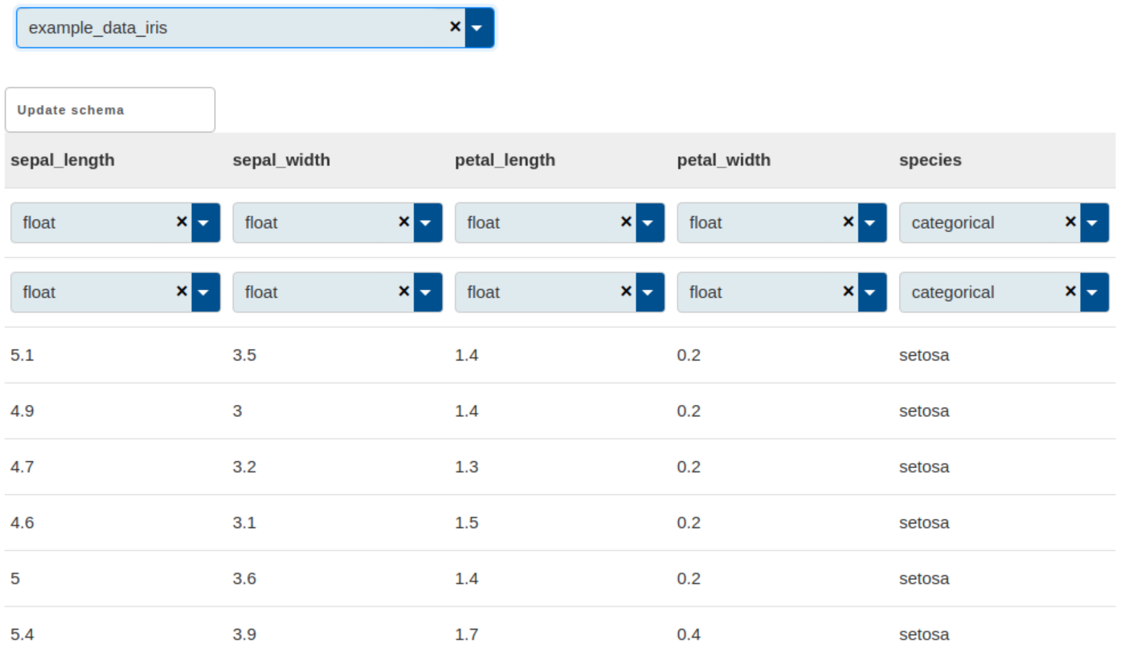
Next on the list is the "View Data" tab, which aspires to be a spreadsheet. Currently, you can only inspect, use filters, and sort your data. Behind the scenes it uses Dash's DataTable library. Immediate plans include allowing for editing cells and adding rows and columns. In the long-term, we want to implement creation of new columns using custom formulas.



⇕sepal_length	⇕sepal_width	⇕petal_length	⇕petal_width	⇕species
>5	>2.6	>3.0		
5.7	3	4.2	1.2	versicolor
5.7	2.9	4.2	1.3	versicolor
6.2	2.9	4.3	1.3	versicolor
5.7	2.8	4.1	1.3	versicolor
6.3	3.3	6	2.5	virginica
5.8	2.7	5.1	1.9	virginica
7.1	3	5.9	2.1	virginica
6.3	2.9	5.6	1.8	virginica
6.5	3	5.8	2.2	virginica
7.6	3	6.6	2.1	virginica

Figure 33: EDA Miner / Data / View Data

Finally, there is the "Edit data schema" tab, where you can inspect the inferred schema for upload datasets and correct any mistakes. It is simply an HTML table with a few dropdowns. Let's quickly go over how this was implemented.



sepal_length	sepal_width	petal_length	petal_width	species
float	float	float	float	categorical
float	float	float	float	categorical
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Figure 34: EDA Miner / Data / Edit schema

Having knowledge of the data types can help guide visualization and modeling recommendation, as well as preemptively inform the user of potential actions for data preparation. The DIVE paper mentions this very important aspect and how all the major tools do it (take a peek at how DIVE does it here[192]).

We do so too but we take a slightly different approach. We chose not to focus too much in getting the data schema as accurately as possible, we simply defined a few simple rules to handle basic types and some "subtypes" (e.g. binary, latitude, email), and leave it to the user to correct any mistakes. We decided on the types based on types of possible visualizations (for example a latitude column can be drawn in a scatterplot but wouldn't make sense in a pie chart). In code, it mainly works by trying to explicitly convert to certain data type and match against a list of predefined keywords (e.g. "year", "age", "gender"). The general idea is that user corrections will be used as explicit feedback to train a machine learning model that, given the first few rows and the column names, can make accurate predictions. Here's an overview of the types and subtypes that are currently handled:

- Types
    - interger
    - float
    - date
    - categorical
    - string
- Subtypes
    - binary (categorical)
    - longitude (float)
    - latitude (float)
    - email (string)
    - ipv4 (string)
    - ipv6 (string)

[192] https://github.com/MacroConnections/DIVE-backend/blob/29bc55ff82718238164425232bdcc24ce6e3e113/dive/worker/ingestion/type_detection.py

- mac address (string)

Part of this is achieved by keeping track these information in the data schema. Our data schema is a dictionary comprising of three elements: a dictionary mapping each column name to a type, another mapping to subtype, and a pandas.DataFrame with the first five rows. We decided on this format because it is lightweight and covers what we need these data for. That said, we didn't come up with this; instead, it is a mutation of format proposed by Frictionless Data and we even defined a quick helper to convert to a format compatible to the their Table Schema specification[193].

4.2.2 Visualization app

The visualization app is implemented in a similar way in terms of layout and design patterns. It currently contains six tabs, each responsible for a different type of visualization.

First on the list is the "ChartMaker". Inspired by Plotly's *Online Graph Maker*[194], this tab allows the users to create a custom graph from their datasets by adding new elements, or *traces*, that use the selected dataset's variables. The user can then export the create figures and they are accessible elsewhere within the app, e.g. for presentation purposes.

Second, the "Key performance indicators" is a sub-page designed to help users create and track metrics they themselves create, or use prebuilt ones. Currently the only KPI implemented is a baseline (see illustration below). Our implementation of the baseline uses a moving average, an exponential moving average, an alternating least squares baseline[195], and the baseline calculated with the python package *peakutils*. We are using this formula to combine them, which we found through trial and error, and didn't test thoroughly.

$$2/3 * (2/3 * peakutilsBaseline + 1/3 * expMovAvg) + 1/3 * ALSBaseline$$

Figure 35: Baseline formula

[193] https://frictionlessdata.io/specs/table-schema/
[194] https://plot.ly/create/#/
[195] https://stackoverflow.com/a/50160920/

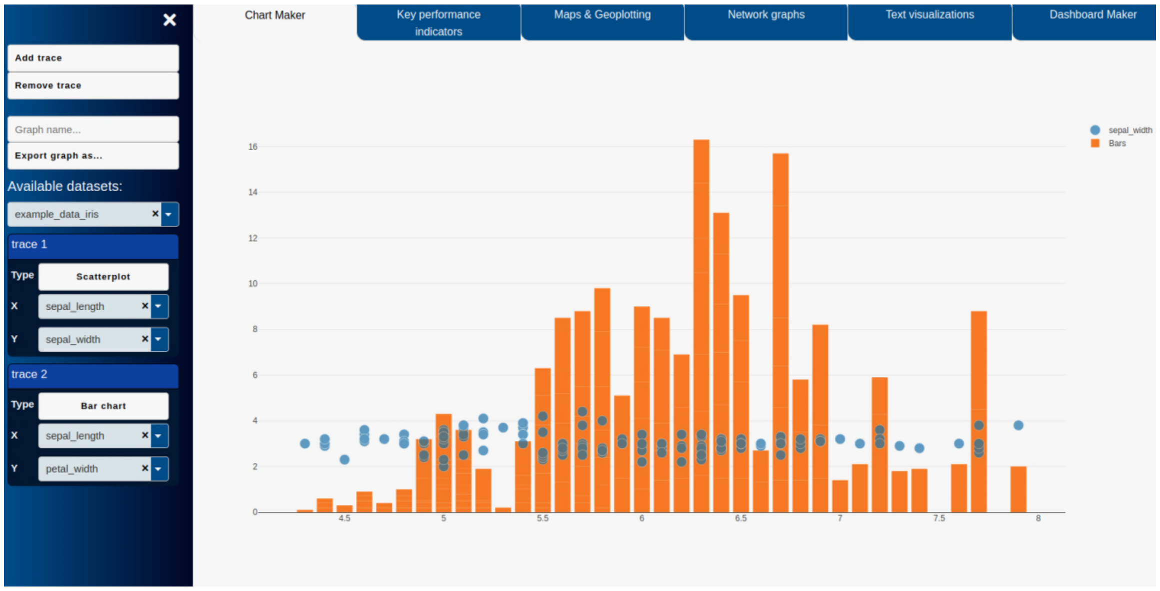


Figure 36: EDA Miner / Visualization / ChartMaker

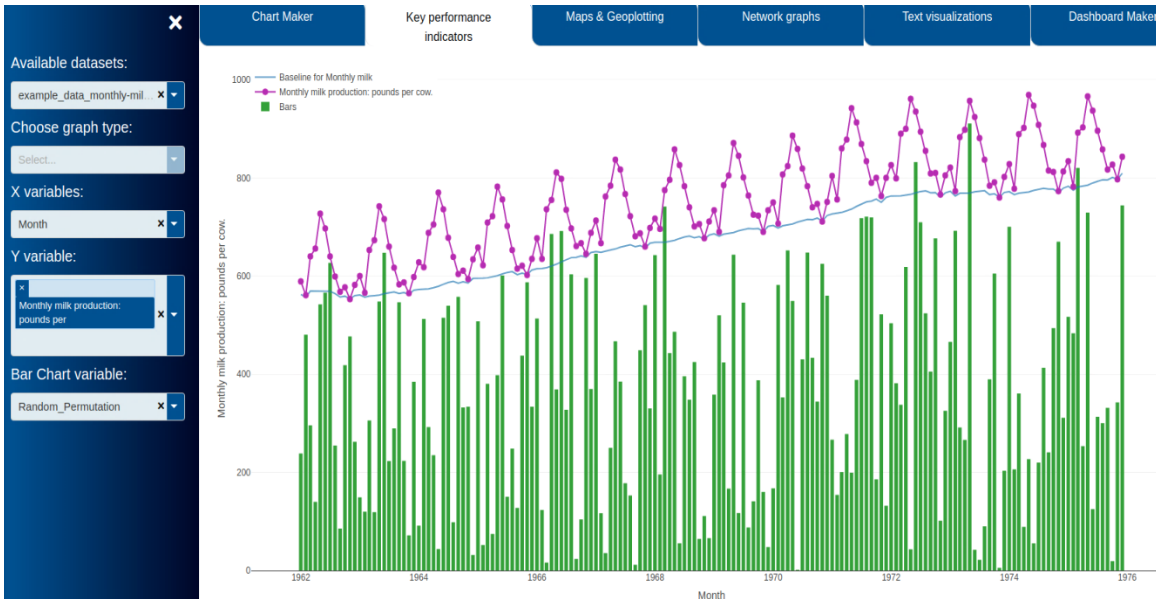
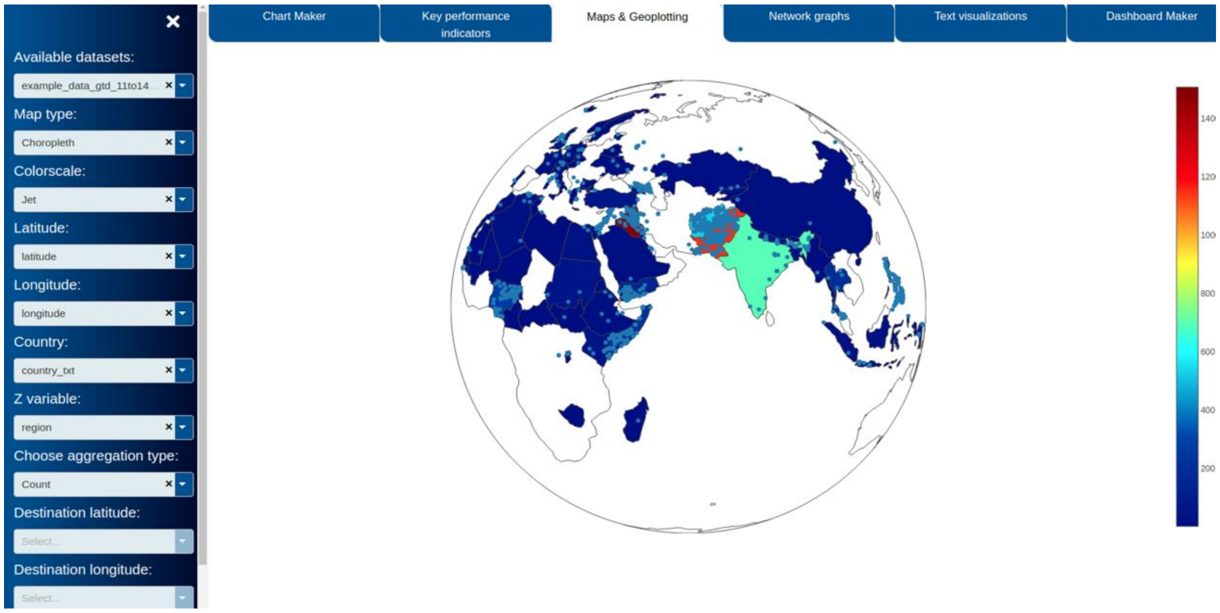


Figure 37: EDA Miner / Visualization / Key performance indicators

The third sub-page is "Maps & Geoplotting". Currently it supports three map types: simple geoscatter (points on the map), choropleth (points, and aggregation with coloring per country), and lines on map (draws lines given from-to points). Also, three map projection styles are supported: equirectangular, azimuthal equal area, and orthographic. This, just like most other visualizations, use Plotly's library, but future plans include integration with other mapping libraries and providers (e.g. mapbox, leaflet, folium).



Available datasets: example_data_gtd_11to14

Map type: Choropleth

Colorscale: Jet

Latitude: latitude

Longitude: longitude

Country: country_txt

Z variable: region

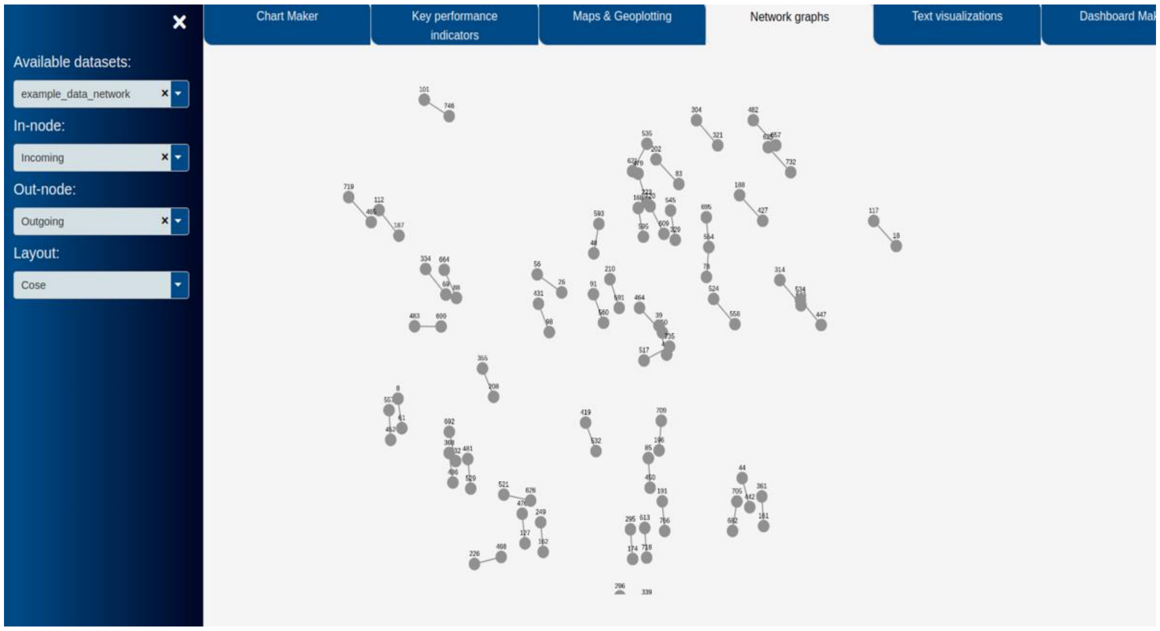
Choose aggregation type: Count

Destination latitude: Select...

Destination longitude: Select...

Figure 38: EDA Miner / Visualization / Maps & Geoplotting

The next two tabs are there providing only basic functionality, and improvements on them are not on the intermediate development plans. They are "Network graphs" (visualizing graph/network data) and "Text Visualizations" (currently only creates word-clouds but will include word vector and other visualizations).



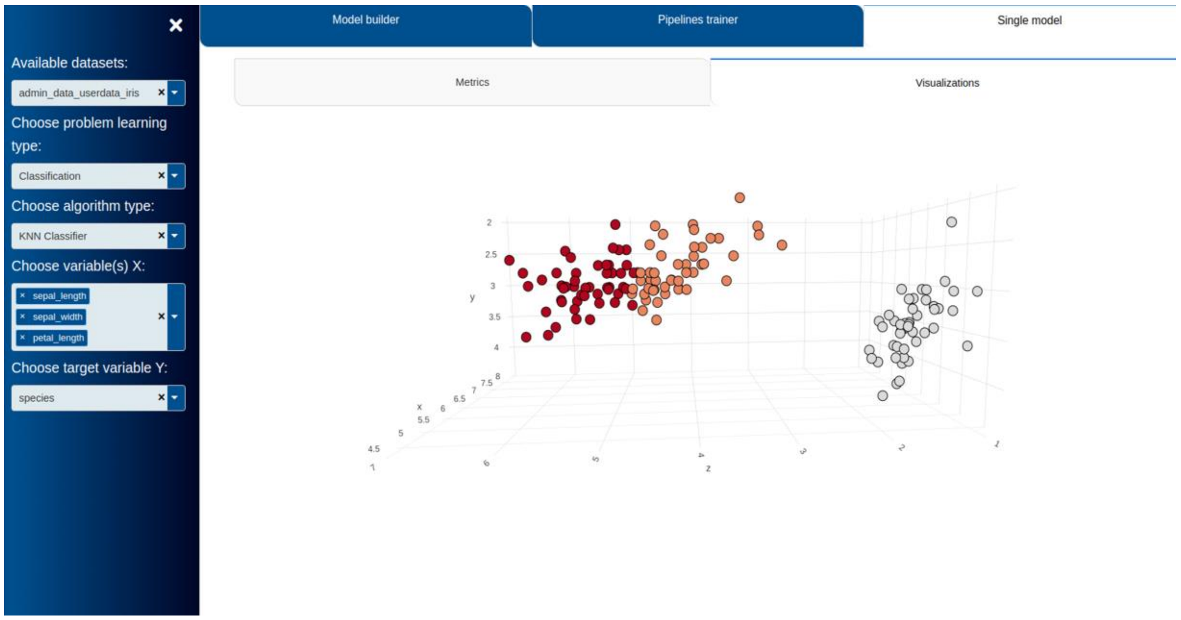
The final tab, "DashboardMaker", is still incomplete. Its purpose is to serve as a drawing board where the users can place their graphs, add text, and maybe some controls so as to create interactive interfaces for presentations and stories. This is mostly inspired by Microsoft's Power BI as well as DIVE's visual narratives. Currently it uses react-rnd[196] to create a Dash-compatible component that we use to wrap elements that are supposed to be resizable.

4.2.3 Modeling app

Finally we come to the last of the three main apps, and the one responsible for all the machine learning and statistical modeling. Similarly to the previous two, it contains tabs to navigate across the sub-pages. We will review them in reverse order since the first two are more complex.

196 https://github.com/bokuweb/react-rnd

The last tab, “Single model”, gives a simple menu consisting of dropdowns for the user to pick a dataset, a problem type (e.g. regression, classification), a model (e.g. “KNN classifier”), and then two more for picking the regressor (X, multiple allowed) and target (Y) variables. It then fits the model and returns metrics according to the problem type (e.g. for classification it returns a classification report and accuracy, for regression



Available datasets: admin_data_userdata_iris

Choose problem learning type: Classification

Choose algorithm type: KNN Classifier

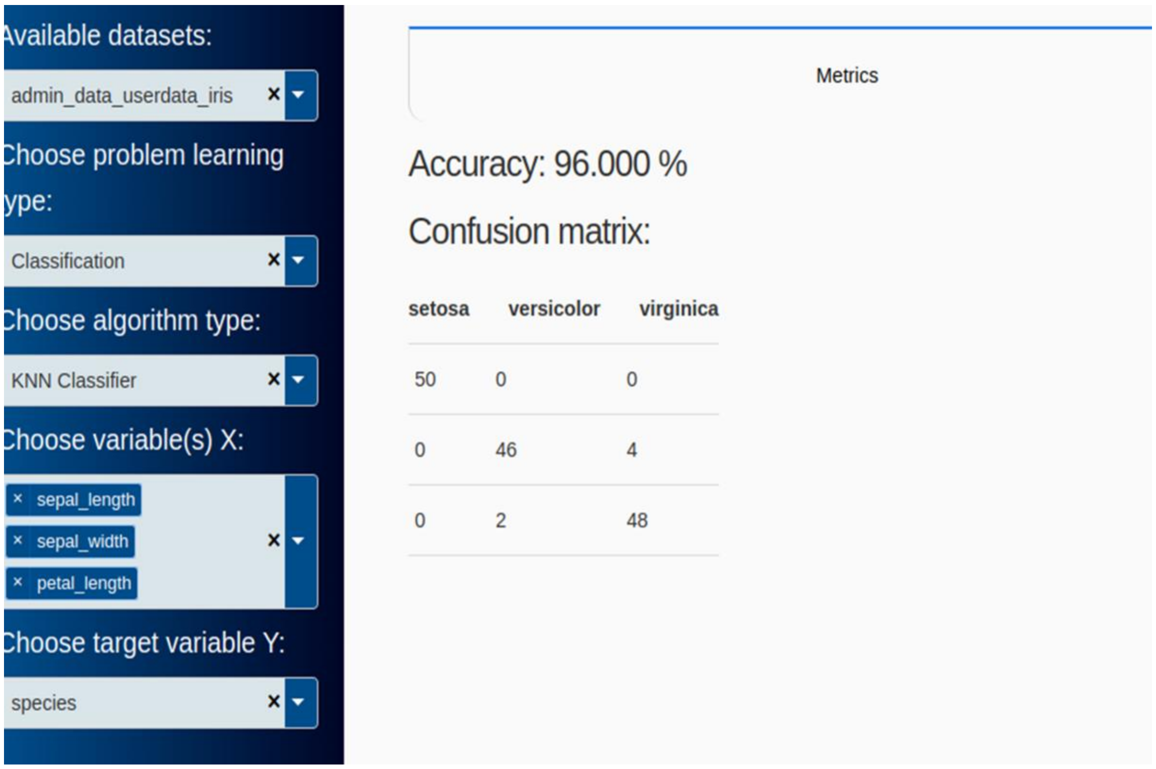
Choose variable(s) X: sepal_length, sepal_width, petal_length

Choose target variable Y: species

Figure 40: EDA Miner / Modeling / Single Model, Visualization

the MSE) and provides an interactive visualization.

The second tab, “Pipelines trainer” has the exact same interface for the results of model fitting but differs in how selecting models works. The models in this case are pipelines that are exported from the ModelBuilder (see below), which also means that the datasets need to be defined there (although this might change later). As soon as you select a pipeline and X/Y variables, the model will be trained and the results displayed. This model will then be saved on Redis as a pickled object for one hour. If the user chooses to “export” the model, then it saved (using *PERSIST* from Redis). Exported models become available in the Model Devops app (not analyzed here due to being in the early stages of development).



Available datasets:

admin_data_userdata_iris

Choose problem learning type:

Classification

Choose algorithm type:

KNN Classifier

Choose variable(s) X:

sepal_length, sepal_width, petal_length

Choose target variable Y:

species

Metrics

Accuracy: 96.000 %

Confusion matrix:

setosa	versicolor	virginica
50	0	0
0	46	4
0	2	48

Figure 41: EDA Miner / Modeling / Single Model, Text metrics

In the first tab is the "ModelBuilder". Let's first inspect the front-end implementation. It provides a small menu that lets you define a complex model / pipeline. The pipeline is implemented as a directed graph (for now it is expected to be acyclic) using dash-cytoscape. The nodes are the various model classes (*estimators* in sklearn terms) that map 1:1 to sklearn-like model classes, while the edges are simply defining links among them. The nodes belong in exactly one of the following categories, or "parents": Inputs (choosing an input source), Cleaning (not implemented), Preprocessing (e.g. *StandardScaler*, *MinMaxScaler*, *PolynomialFeatures*), Dimensionality Reduction (*PCA*, *NMF*, *TruncatedSVD*), and Estimators (e.g. regressors, classifiers). Each of these categories has an order parameter symbolizing its place in the pipeline (higher order means closer to the output). Adding and removing nodes are handled with dropdowns, connecting two nodes needs to click two or more of them (both click order and the order parameter matter) and then click the "Connect selected nodes" button, and there is also another dropdown where the user can select a prebuilt model tailored for a specific task. When the user finishes with the model definition, there is a "Save graph and export pipelines” button that both saves the current graph and creates one separate pipeline for every output (Estimators) node.

Let’s peek into the implementation with more detail, and the class hierarchy. We have a total of 6 classes: Node, NodeCollection, Edge, EdgeCollection, _Graph, and Graph. The nodes are implemented as data classes with Python’s __slots__ for faster attribute access and less memory consumption, and node-only attributes. Nodes’ model classes are expected to provide a typical sklearn interface (that is to implement *train*, *fit*, and *predict* methods). Each node stores its position (*xpos*, *ypos*), its *parent* and the parent’s *order*, a *label*, a *node_id* (e.g. “linr_001”), and a *node_type* (a string that maps to a sklearn-like class, e.g. “linr”). A NodeCollection contains Node objects, points to the Graph it is part of, holds information about how many nodes of each *node_type* it

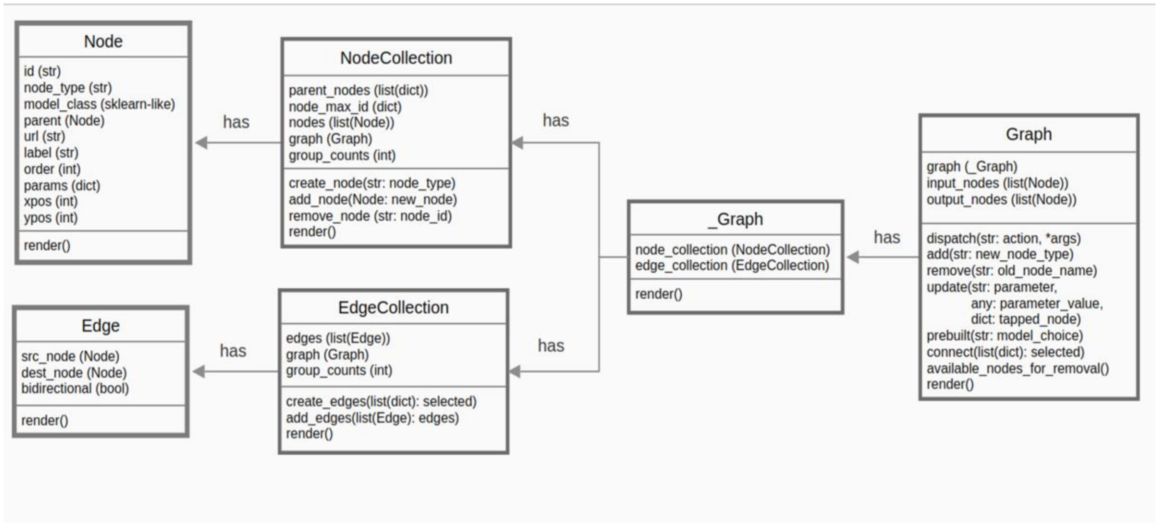


Figure 42: EDA Miner / Modeling / ModelBuilder / Class hierarchy

contains, and defines a few helper methods for creating, removing, and adding nodes. In a similar fashion, the Edge class holds a source and destination nodes, and whether the edge is bidirectional or not (all edges are directed), and the EdgeCollection holds Edge objects, a pointer to the parent Graph, and utility methods to create and add new edges. The _Graph class is the old implementation of the Graph class which will probably be removed in the future when the current code refactoring is finished; it contains a NodeCollection and an EdgeCollection. Finally, the Graph class contains the _Graph object, and keeps track of the input and output nodes. It defines a *dispatch* method that uniformly handles all the other methods and allows us to create Dash callbacks and UI with ease like in the Data app for APIs (see above). Every object contains a *render* method which returns the representation of the object for plotting in dash-cytoscape.

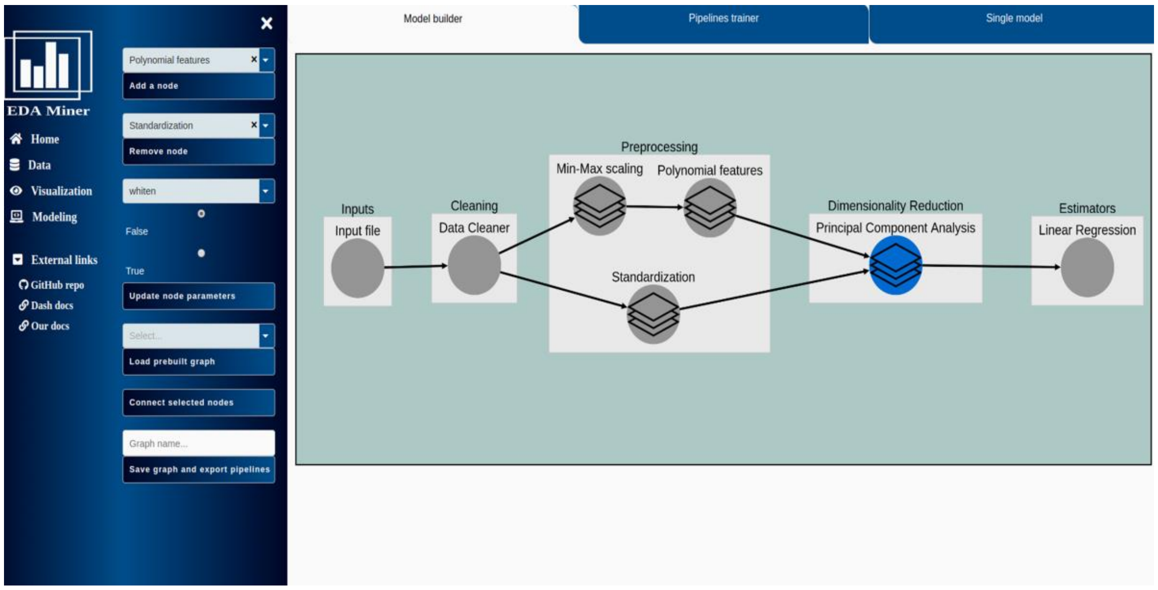


Figure 43: EDA Miner / Modeling / ModelBuilder

A few important points must be stressed out. When it comes to connecting nodes, for now, we allow connections only among nodes belonging to the same category, or among nodes where the source node's order is smaller than the target node's (e.g. starting from a *Preprocessing* node only to *Preprocessing*, *Dimensionality Reduction*, and *Estimators* nodes). Also, graphs must be Directed and Acyclic (or "DAG"), so support for Markovian models is not there yet. Also, the way these Graphs are turned into actual pipelines is not fully implemented in the sense that some core features are still missing: multiple input nodes are not handled well (e.g. no table union nodes, just dataframe concatenation across one axis), and both multiple input and multiple output nodes are still not well-tested. The reason for this is that we have not yet implemented a correct parser; instead we rely on a recursive *_traverse_graph* function that uses sklearn's *FeatureUnion* (when one or more nodes are the input of another) and *Pipeline* (to actually connect the output of *FeatureUnion* with the next node) to create a model. While this approach has simplified coding, it is not without error (at least until a more careful implementation of the Graph class is done). We also didn't implement graph traversal (and won't until it is absolutely necessary); for that we convert the Graph into a *networkx.DiGraph* object and use its methods. Finally, all model classes, aside from the interface requirements mentioned above, also need to define a *modifiable_params* dictionary where the keys are the model parameters and the value is a list of available choices (the first is the default). This was done so as to avoid future circumstances where users would input incorrect values, or not desirable (e.g. an *n_jobs=1000* that can crash the server).

4.3 Database Technologies & Models

In our app we use both SQL and NoSQL. Detailed usage of each is described below, but first let's explore an overview of how they fit into the overall architecture. SQL databases interact with the authentication system and the individual Dash apps, providing both long-term storage and data for the main functionalities. NoSQL, and Redis in particular, serves as temporary storage, handles data shared or moved across apps, and also handles temporary links/tokens. Here's an overview:

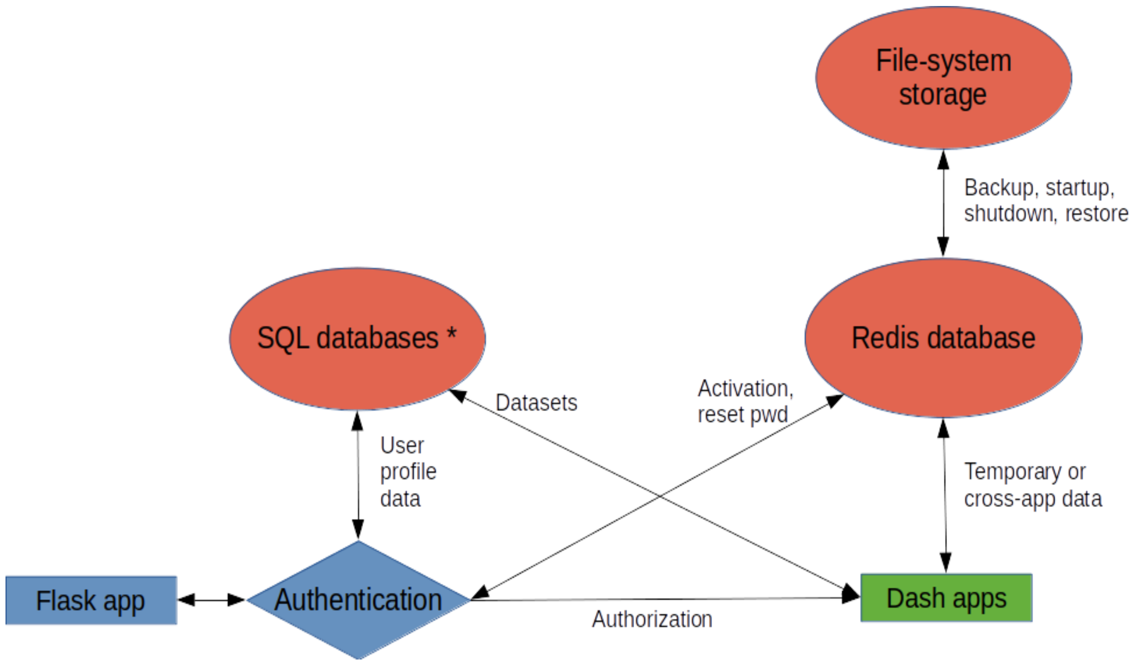


Figure 44: Databases in the overall architecture

The choice for SQLite was made for two reasons: 1) because of its easy integration with Python and being a “lite” database, and 2) since we use SQLAlchemy as an Object-Relational Mapping engine the choice of database doesn’t matter a lot since we can easily change it without introducing the need for code rewriting across the applications. In fact, for a production environment, and when more advanced uses are needed, we do recommend a more robust SQL server; in particular, we are looking into PostgreSQL.

So how are SQL databases used in our application? First, it is important to note that there will be multiple databases: a main database, and one database per user (it is not implemented yet, but is a top priority). The main database schema consists of four tables, all of which can be seen in more detail in the illustration below. The first table, *User*, holds data for user management, and along the columns listed below, it inherits everything from *SQLAlchemy.Model* and *flask_login.UserMixin* base classes. The second table, *DataSchemas*, contains a pickle with the schema (2 dictionaries and a *pandas.DataFrame*) for each user dataset and an indicator (whether the schema was auto-inferred or user-provided). The third table, *UserApps*, contains for every user a row for every app they have access to. Finally, the fourth table, *UserDatabases*, keeps track of where the database of each user is stored.

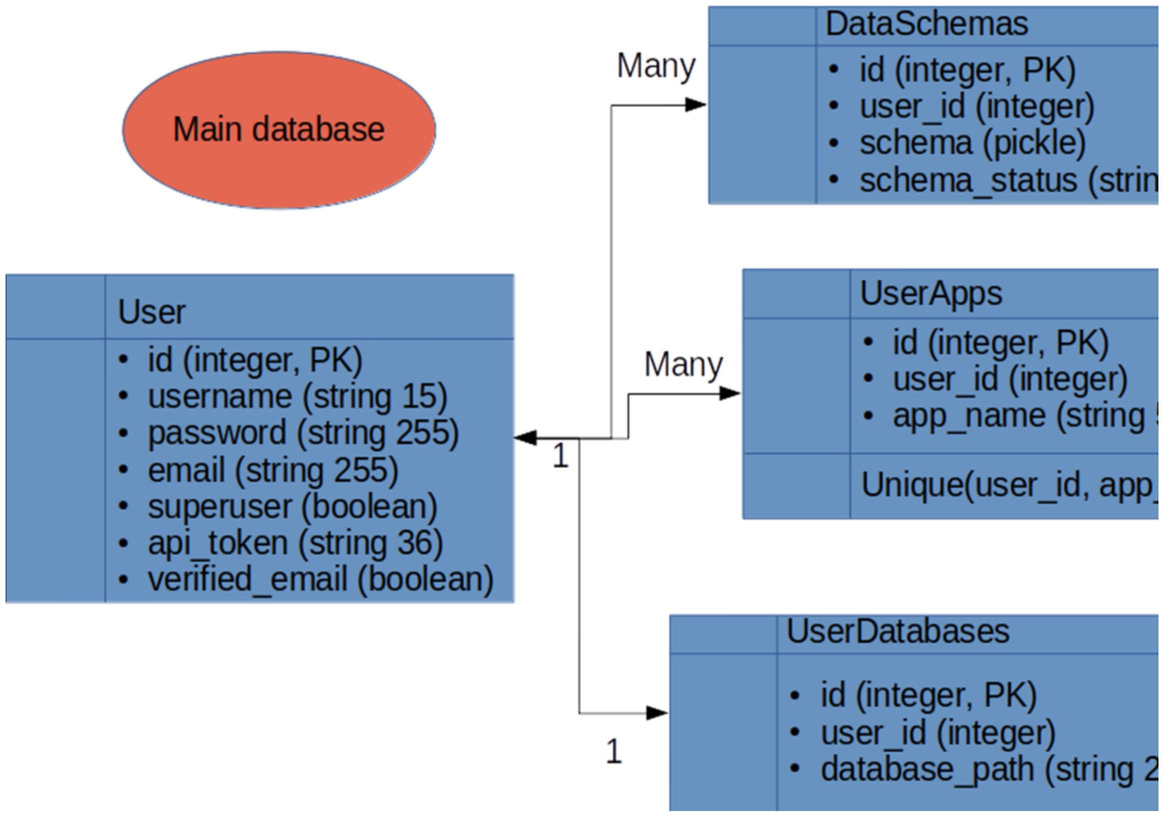


Figure 45: Main database schema

For each of the users, we are also creating a database to store their uploaded data. Since we are treating everything as a *pandas.DataFrame* it is easy to translate everything into SQL, even more so since we are already inferring data types. Furthermore, it is possible to get a schema from the user that can connect two or more tables, e.g. a table containing company's list of clients and user data, with a table containing tweets or emails from the accounts of the former (see example illustration below, and a zoomed overview).

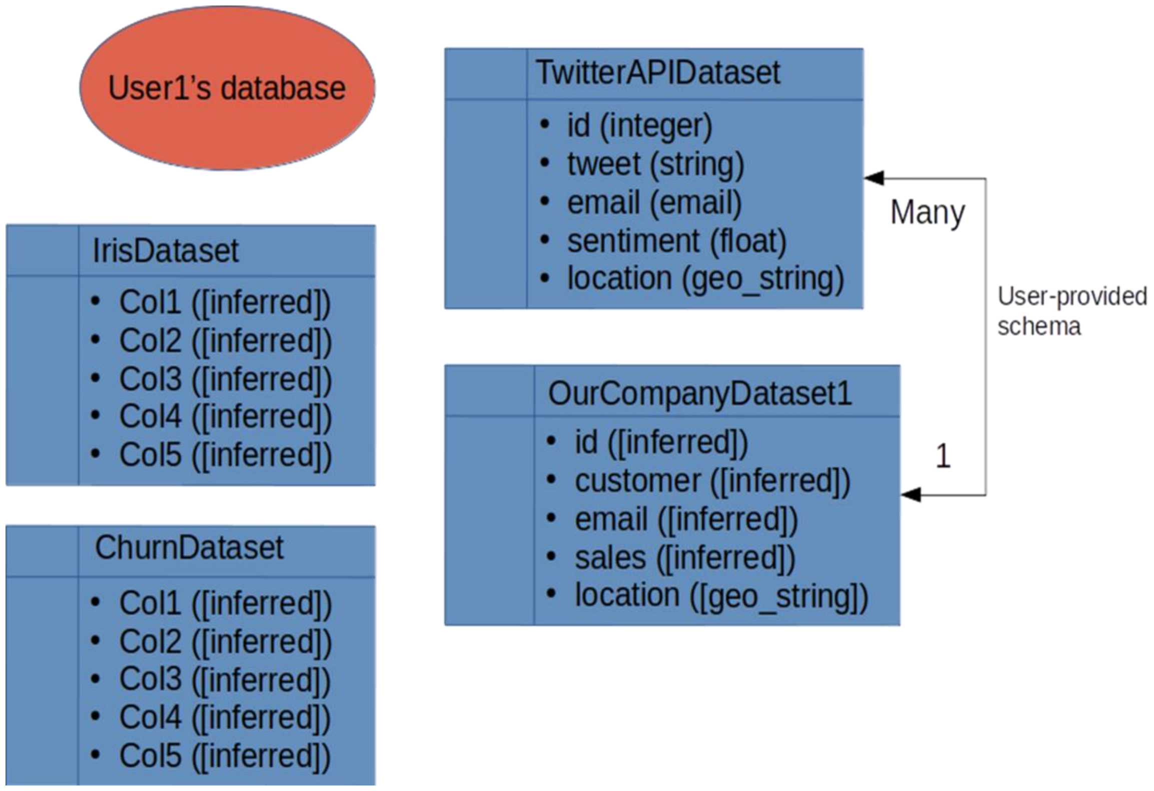


Figure 46: Database for user 1, with a custom schema connecting two tables

SQLite3 was an easy choice to make. Redis on the other hand was a more difficult choice and one we need to elaborate on. During early development, Redis was chosen primarily because in the Dash docs it is the go-to example of a "shared memory space" needed to pass data across threads and processes in a safe manner[197]. Their examples only use Redis as a back-end for Flask-Caching. They also use a hidden div that holds a random string (a session id) for making sure each user only accesses their own data. We used the same technique in early development but for reasons including authorization, permanent storage, and security (a hidden div can easily be modified), we moved to a complete authentication system with the typical email/username plus password login. This is primarily performed with SQL, and Redis is used only for temporarily storing (using the expiration feature) activation links or similar. Furthermore, our main use of

[197] https://dash.plot.ly/sharing-data-between-callbacks

Redis (and its main use) is as a “global dict”, that is, a key-value storage that can be accessed by multiple threads/processes. At the point we started considering other choices it didn’t make sense to transition to a similar database like *Memcached*[198] because even if it added any performance gains it wouldn’t add any extra features (in fact, it provides less[199]), and we definitely didn’t need them at the time. What was worth trying out was database management systems like *Kyoto Cabinet*[200] or the Python built-in DBM module[201], and on-disk key-value storage systems like Python’s *shelve*[202] whose main difference with a DBM is its ability to store Python objects that can be pickled, and finally DiskCache which a pure-Python alternative to shelve/dbm and advertises itself as an “on-disk cache”[203]. As a result, we only chose to benchmark *shelve* and *DiskCache*, but first let’s first see how Redis fares on its own.

Setting aside its popularity, especially among Python programmers, it is an incredibly performant database server and here are two lists of benchmarks of 7 Redis

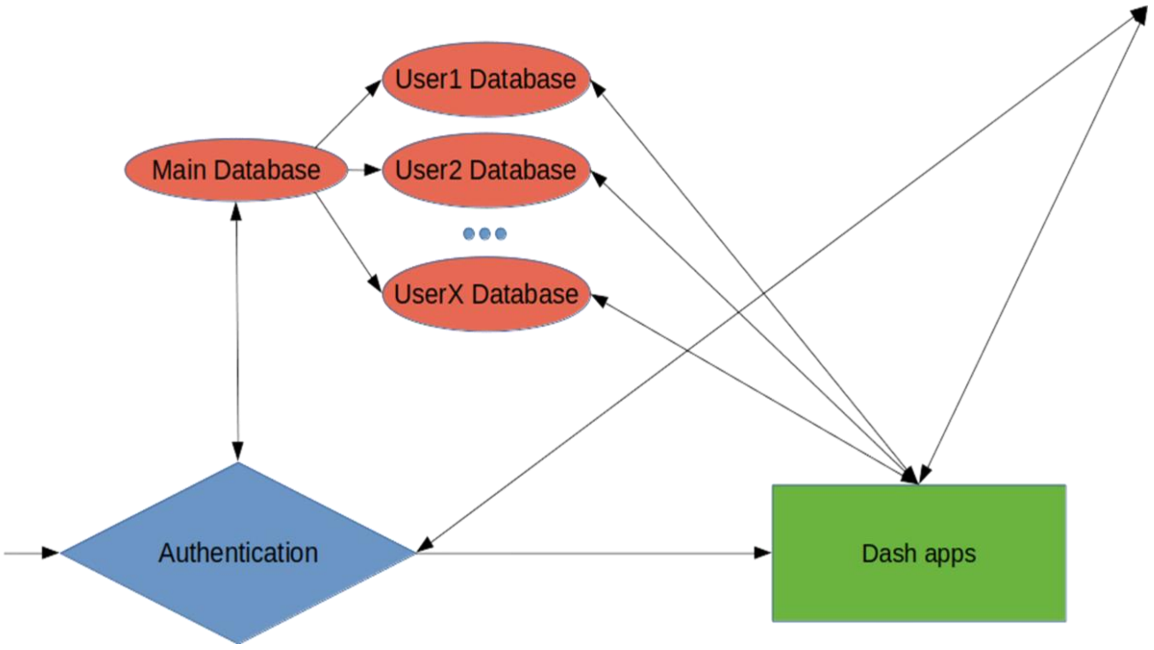


Figure 47: Architecture, zoom into databases

[198] https://memcached.org/
[199] https://stackoverflow.com/a/11257333
[200] https://fallabs.com/kyotocabinet/
[201] https://docs.python.org/3/library/dbm.html
[202] https://docs.python.org/3/library/shelve.html
[203] https://github.com/grantjenks/python-diskcache

commands measured on localhost, using 1 million requests, with random keys spanning a range of 1 million (so to stress cache misses). The first sends and receives requests one at a time in a blocking manner, while the other shows how Redis' pipelines can speed up response speeds (by not waiting for individual replies and reading them all at once[204]). Currently we are not using the pipeline feature in our implementation so raw requests per second more closely matches our case.

Raw requests per second (all requests completed within 1 ms):

- **GET**: 152,718.39
- **SET**: 156,201.19
- **LPUSH**: 157,282.17
- **LPOP**: 156,494.53

Pipelined requests per second (-P 100, see appendix for cumulative response graphs):

- **GET**: 1,240,694.75
- **SET**: 914,076.81
- **LPUSH**: 1,216,545.00
- **LPOP**: 1,225,490.25

Seeing only raw GET and SET requests, the ~152-156K requests per second translates to one every ~6.5 μs, however timing the requests for a single key-value pair from within Python[205] seems to introduce additional delay making them as slow as ~37-38 and 40-41 μs for SET and GET respectively (or 24-27K requests per second). In comparison, shelve seems to take about 1.6 - 9μs for GET and about 1.3 - 5 μs for SET (we provide ranges due to high fluctuation), while DiskCache is at about 7-8 μs and 120-130 μs respectively. So why not use shelve?

When comparing Redis to shelve five major considerations stirred the decision: 1) shelve does not support concurrent read or write access, which can be done by extending it, but would introduce the need for manual process and thread synchronization and working with low-level locks and/or queues, 2) shelve suffers the same limitations of

[204] https://redis.io/topics/pipelining

[205]For details see the attached Jupyter Notebook with the performance tests. These were done with the
%%timeit magic command for 50 runs of 1000 loops.

DBMs[206], 3) As we are working with larger objects the gap between Redis and shelve closes, e.g. for a 1000x1000 pandas DataFrame of random numbers shelve takes about 2.4 ms (GET) and 2.8 ms (SET) while Redis takes about 5.3 ms and 4.8 ms respectively[207], 4) Redis is already working as a server which means that it can be accesses from many clients in a network without additional effort, and 5) Redis already has an image and a lot of online support for Docker.

One could argue that we could have used MongoDB or other solution that is friendlier to JSON data. While it is true that most of the data we store are Python dictionaries or can be converted to JSON (e.g. pandas DataFrames), we also want to store complex data types (i.e. Python objects). We could do that, too, by extending the objects with constructors and serializers but that would introduce unnecessary complexity. Also, even for the case where JSON supposedly would be better, our benchmarks show that serializing datasets to JSON is definitely not faster than pickle (we use dill, an extension to pickle that can handle more objects). The 1000x1000 dataset takes 28-29 ms to save with pickle (~38 ms for dill) and ~85 ms - 1s for JSON. Changing to MongoDB and still use BLOBs for pickles kind of defeats the purpose. These differences are mainly because the data need to change format first, but that is a significant overweight; and we didn't even account for reading times and extra data copying[208].

[206] https://docs.python.org/3/library/shelve.html#restrictions
[207] Interestingly enough, DiskCache becomes faster than both with 738 μs for GET and 3.65 ms for SET.
[208] pandas do tend up to perform a lot of copying when initialized, something that doesn't apply for when data are loaded via pickle, which skews benchmarks but doesn't affect the result.

4.4 Directory structure for project & apps, and extensions

We mentioned how v0.3 changed the overall structure of the project and how it is now comprised on individual “mini-apps”. We presented how we use Dash and the various designs we employed at the top level. We also went over the major apps in chapter 4.2, and we also reviewed, in chapter 4.3, how these apps use and interact with the various database technologies that we use. What we did not view, however, was how individual applications are structured. In this chapter, we will view the “demo” template app, the directory structure, and the various forms that extensions can take as of v0.4 (if a later version implements the HTTP proxy, then extensions can become arbitrary servers in arbitrary programming languages).

4.4.1 Directory structure: project

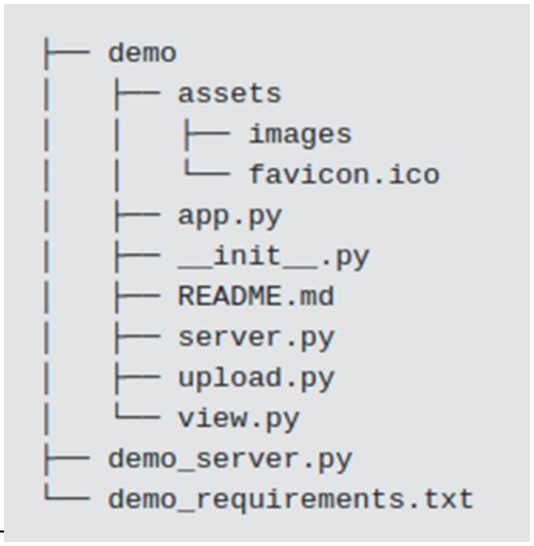
The project’s top-level directory is structured as a typical Python project. It contains a *requirements.txt* file with all the package dependencies, a *project_info.txt* file with some automatically calculated statistics about the project (e.g. test and documentation coverage), YAML configuration files for CI/CD tools (*.codeclimate.yml* and *.travis.yml),* git and GitHub files (*.gitignore*, *.github* folder) and some markdown/text files like the license, the code of conduct, a readme, and contribution guidelines (which provides about 9 pages of important additional information and is given as appendix 6). The top-level directory also contains a few more directories: *example_data* (contains a few datasets in various file formats), *images* (screenshots, an older version of the dash callback chain, and a screenshot of the directory structure), *tests* (a folder containing scripts for unit-testing and testing docker builds), and the main folder containing all of the source code, *EDA_miner*.

The source folder contains a folder for *static* files and another for *templates* (HTML files rendered with Flask), one folder for each “mini-app”, one folder for the *docs* (which are an app of their own found at /docs), and a folder with a custom Dash component, *dash_rnd*. It also contains the main database (*users.db*), a file with database management utilities (*users_mgt.py*), two files with utilities for all the apps (*utils.py* and *exceptions.py*), two for defining forms (*forms.py*) and models (*models.py*), one for Flask extensions (*app_extensions.py*), various configuration files (*env.py*, *config.py*, *layouts.py*), and the WSGI file to pull together all apps and initialize for each of them the main database, the user login, and more (*wsgi.py*). The output of the *tree* command in Linux is given in Appendix 7.

4.4.2 Directory structure: Demo app

Small Dash applications and dashboards are usually in a file (by convention *app.py*), but Dash provides examples on how to structure larger applications, including how it supports URLs[209]. We also provide demo app to server as a template for adding new functionality[210]. As said previously, each new app should have its own folder in */EDA_miner/*, preferably with a short lowercase name and optionally underscores. New apps can of course be their own files, but the folder is preferable even for smaller apps.

When it comes to structuring these extension apps, we would expect that there are two files in the top-level directory and a folder containing the app. The files are: an *x_requirements.txt* file (*x* stands for the app name) and an *x_server.py* which is there mostly as dummy for the case an app would run in standalone mode (also, since apps use relative imports, Python needs the executed script to be in a parent folder). Regarding the app folder, this is largely left to the developer. You could either have one *x/app.py* file (most Dash apps), or a more refined structure. In the demo app we have the following: a *demo/server.py* file for handling the app-only configurations (e.g. its own database, if it



```
├── demo
│   ├── assets
│   │   ├── images
│   │   └── favicon.ico
│   ├── app.py
│   ├── __init__.py
│   ├── README.md
│   ├── server.py
│   ├── upload.py
│   └── view.py
├── demo_server.py
└── demo_requirements.txt
```

Figure 47: Demo app directory structure, tree

[209] https://dash.plot.ly/urls
[210] https://github.com/Kmouratidis/EDA_miner-template-app

has one) and initializing the Dash app, a *demo/app.py* file for handling the main page (the *dash.Dash* instance will be imported from here), a *demo/__init__.py* (so the folder can be interpreted as a Python package), a *demo/README.md* with information about the app, an *demo/assets* folder containing any app-only CSS and JavaScript (the equivalent to a *static* folder), and two files for this particular app, each representing one sub-page or tab, *demo/upload.py* and *demo/view.py* that provide a simple interface to drag-n-drop a file and view it in a table. You can see it graphically using the *tree* command in Linux (as seen above).

4.4.3 A demonstration: Google Analytics REST API

Extensions to the project can have their back-end written with any technology or versions the developer wants. This will become even better when we transition to an HTTP Proxy solution. For now, all you have to do is write your server logic in whatever framework, and write a mini-application that integrates with Flask. Essentially, apply the Adapter software design pattern. Here will view one such example, and we will also review how Redis can be used with it.

In our initial implementation we were connecting to Google Sheets via a library called gspread. This library's requirements were compatible with the recent versions of Google's libraries for oauth2 (specifically oauth2client v4.1.3), but the one we wanted to use for Google Analytics wasn't (it needed oauth2client v1.5.2). Needless to say that these two versions are completely incompatible. It seems that the primary way of dealing with this is messing with paths and/or package naming[211], both of which seemed too tedious and would probably make deployment harder. Instead, we came up with the idea of packaging the connection to Google Analytics as a REST API deployed in its own docker container. When we transitioned to a custom implementation of connecting to Google Sheets this micro-service wasn't any longer needed (no package dependency conflicts) but we kept it as an example of yet-another-way to integrate services and APIs, potentially with back-end languages other than Python. REST principles were mentioned in the previous chapter so we won't go over them here. Instead, we will go over a few benchmarks and intended usage.

[211] https://stackoverflow.com/a/6572017/

We tested our Google Analytics REST API using an HTTP load generator (*hey*[212]). It is built with Flask, and uses memoization as a caching policy (per user and metric) with Flask-Cache and Redis. We tested it with 10,000 requests, with 50 concurrent clients, requesting the same (one) metric from the same (one) user, each of which transferred a total of 67 bytes. The results show that the slowest at 3.2022**s** (base value) and the fastest request was at 0.0331**s** (~96x difference), with the average being at 0.0522 (61x), equating ~955 requests per second. In essence, only the first batch took significant time to fetch, and the rest were retrieved from the cache. Running the same query again (which is now cached) returns a slowest of 0.0425s (75x), a fastest of 0.0041s (781x), and an average time of 0.0361s (88x), equating ~1,381 requests per second. You can find both reports in Appendix 5.

We mentioned previously that we kept this as an example, so let's delve into this a bit more. Since this micro-service is receiving and dispatching data via HTTP methods it can be implemented in any programming language with whatever framework (which is one of the benefits of micro-services), and using universal formats like JSON is can language agnostic. Although currently it connects to the central Redis server (this is considered an anti-pattern, (Richardson, 2019, p. 40) and (Fowler, 2014) and (Schmitz, 2017)), it can easily be modified to have its own connection.

The usage splits in two parts: the server and the client. Server-side, the app accepts only two routes, with one method each. First, you are expected to send a POST request to the main page ("/") with the user_id (username), and the private_key and client_email (credentials by Google), then the server saves that info and returns a 201 status code to indicate success. Then you can send GET requests to a URL defined by the comma-separated list of metrics you want, and the user_id ("/my_user_id/pageviews,sessions"). The server returns 401 ("Unauthorized") if the user is not authenticated, 400 ("Bad Request") if no metrics were given, and 200 ("OK") if the request was successful. Client-side, the programmer is expected to implement an interface (an abstract base class, *APIConnection*) and handle the sending of requests within that. Our implementation is just a wrapper around *requests.post* and *requests.get* with some custom logic for displaying data, and a menu for requesting data.

[212] https://github.com/rakyll/hey

5. Conclusions & future work

It has been difficult getting this project so far. I had to overcome architecture and design issues and think ahead about issues like scaling, security, and even possible business models for anyone wishing to continue development efforts. The following paragraphs will comment on issues faced, as well as suggestions for future work.

The biggest hurdle in this project has been my lack of experience, which made everything more time-consuming. There are a lot of processes which could’ve been improved but weren’t, lots of code that wasn’t well-written at first, limited use of design patterns and a clean project architecture. A related issue has been writing clean code. Using Dash’s model for all the interactivity means that the whole codebase needs to conform to a certain pattern. This doesn’t necessarily reflect well on the back-end logic. Another problem regarding code quality has been that, at times, development of components has been hurried, perhaps a bit too much (e.g. trying to complete a new feature before a presentation). Most of these were dealt with since v0.3 came out, but future contributions should take some time to revise existing code and use more rigorous procedures, including cleaning up the code.

Speaking of procedures, almost the whole codebase was contributed by me which in turn meant that use of testing, documentation, version control, and many other good practices may have been too superficial. A team that decides to pick up where this project left off would need to actually implement proper code review with branches and pull requests, more closely monitor code quality, better handle CI/CD, continue the writing docstrings and add more generic documentation, including for the overall usage, manage project requirements, and last -but certainly not least- continue with serious unit-testing and, in general, greater test-writing efforts.

Speaking of incomplete work, a lot of decisions have been deferred implicitly or explicitly (e.g. SQL database choice), while for some others we avoided hard-committing (e.g. the one-app vs multi-app dash architecture is easy to reverse). This has mostly been a conscious stance, following the words of Robert Martin, that “good architecture defers decisions” (Martin, 2015). However, a more serious and calculated approach is needed, as some of these decisions should definitely be made.

Some of the mini-apps that we went over in the previous section are not finished, and some that we didn’t go over but are currently in the live demo and/or GitHub repository may even lack a complete implementation. We will go over these next.

First, the Data mini-app now has an extra functionality to allow uploads of larger files, reportedly in the gigabytes. Although we briefly mentioned this before, these files are not actually handled yet, as simply loading them into Redis would be wasteful, while reading them in memory to perform visualization and modeling would also cause problems. The use of this component has also not been tested a lot. Viewing data is also not implemented fully, as it is currently not handling editing the data. The schema tab doesn’t handle paging either, and it should also have a lot more choices for sub-categories. A very interesting new direction is creating integrations and solutions that involve operations on data, for example using technologies such Elasticsearch (e.g. text search) or Dremio (e.g. data cataloging), and perhaps adding some data management functionality for creating new columns with custom features (e.g. created from user-specified formulas) or connecting specifying more complex schemas connecting datasets in an SQL fashion. The latter could be achieved by using dash-cytoscape (or whatever the new tool for the Model Builder is). This is especially important since the only possibility of connecting datasets right now is to add two or more of them in the Model Builder while taking care that their dimensions are aligned either in rows or columns (and thus are concatenated with Pandas).

Next, the Visualization mini-app needs a lot more improvements. First and foremost, the ChartMaker needs to handle cases like two traces sharing the X-axis but having a secondary Y-axis with different values, or having no common axis at all, or plotting multiple graphs at once in a grid. The “Key performance indicators” tab needs to implement more KPIs (including user-defined). The “Maps & Geoplotting” should be extended to allow connections with other mapping libraries and APIs, and become more optimized. The same apply to the “Network graphs” tab. The “Text visualizations” can only create a word cloud, so that needs additional options as well. Additionally, the “Dashboard Maker” still needs to implemented (and corrected), or perhaps dropped in favor of the Presentation mini-app (see below). All of the above are in serious need of UI/UX improvements. Finally, a very interesting proposition is to allow integrations with other commercial tools, e.g. exporting data to Qlik.

The Modeling app is perhaps the best part of the app. It also needs UI/UX improvements, especially the “Model Builder” which should probably be reworked completely using a better interface (and probably lots of JavaScript) than the current one built with dash-cytoscape. Part of the code cleaning effort should also go here. It is important that future contributions that seek to scale this project should be able to handle partial/incremental training all the way from reading data to potentially augmenting sklearn’s models. Future contributions could also mean adding more models and integrations with other libraries, as well as adding a model builder for deep learning. Also, new model classes could be added that just provide utilities (e.g. connecting datasets, plotting, progress reports).

Two new mini-apps didn’t make it in the previous chapter, since they are still in the proof-of-concept stage and non-functional. The Presentation mini-app is meant to become a different way to create and present exported visualizations, stories, reports, in much the same way as presentation software and competitor tools handle it. This app needs to be implemented either by creating new Dash components using React, or by implementing a custom interface from scratch (which still needs to get graphs from Dash). The other demo mini-app, named “DevOps”, is meant as a demonstration of how the models built and trained within the Modeling app can become useful in other applications. Currently, it only provides the option to download a trained model as a pickle, and showcases (with a graphical interface) of how a REST API exposing the models should work; providing endpoints to predict new data with the selected model, retrain it, or download it. While implementing the rest of this shouldn’t be too hard, a few interesting (or necessary) directions would be allowing users to send their own models to train, and making sure to provide meta-data (e.g. the required schema) to the users of the API, taking a special care for implementing security and authentication correctly, and maybe some functionality for non-technical people like auto-generating embeddable forms for inference (which should help with building innovative apps).

Although we discussed a lot about what was and was not created, there are still a few more, but they need to be mentioned separately since they represent a new category: lost opportunities. The biggest "lost opportunity" was that no 'intelligence' was infused in the software. Contrary to the DIVE paper, we didn't implement a visualization recommendation (let alone mixed-initiative) system. Neither did we implement the ML model recommendation. These were some of the important initial goals of the project that got sidelined by the need to focus on more pressing issues regarding the main functionality of the applications. Working on this, though, is well-defined and could be implemented in different ways for each mini-app. For Visualization, I was considering two ways: 1) a large collection of heuristics can be used to define a subset of all possible visualizations that are potentially interesting, and 2) treat this a machine learning problem where column types/subtypes are mapped to potential visualizations. Both techniques can be used, either by using one to refine the other, and/or asking the user's input. More or less the same applies for the Modeling mini-app, but there is one more way that Modeling can be 'infused with intelligence': collective intelligence. The idea is that rather than train an ML model after collecting user data, you can directly allow the users to interact and assist each other. In either case, implementing these is definitely not impossible, and for the simplest case you don't even need to train a model.

The second miss was my inability to correctly implement an HTTP proxy (neither with werkzeug's ProxyMiddleware nor with Nginx) along with Dash. One of the promises of the v0.3+ architecture was that each mini-app would be able to work and scaled (and load-balanced) independently. Since this failed, the overall performance of the app is worse than when it was a unified Dash app, without so much gained in terms of cleaner structure. However, the ability to easily extend the whole program with easy-to-make mini-apps still compensates overall. Whenever the HTTP proxy solution will be implemented, we expect much bigger performance improvements. That said, an architecture involving an HTTP proxy means that the main Flask app won't be able to share user credentials and app configurations the same way, so more components might need to be added in the architecture to account for that.

Bibliography

Al-Khoder, A., & Harmouch, H. (2015, 3). Evaluating four of the most popular Open Source and Free Data. *International Journal of Academic Scientific Research, 3*(1), 13-23.

Barbettini, N. (2018). OAuth 2.0 and OpenID Connect (in plain English). OktaDev. Retrieved from https://www.youtube.com/watch?v=996OiexHze0

ben26941. (2017, 11 21). *Weka equivalent of sklearn's pipelines and feature-unions.* Retrieved from Stack Overflow: https://stackoverflow.com/a/47416412

Berthold, M., Cebron, N., Dill, F., Gabriel, T., Kotter, T., Meinl, T., . . . Wiswedel, B. (2009, 6). KNIME - the Konstanz information miner: version 2.0 and beyond. *ACM SIGKDD, 11*(1), 26-31. doi:10.1145/1656274.1656280

ChKwK. (2018, 8 22). *What is a good ggplot2 equivalent to Python?* Retrieved from Reddit: https://www.reddit.com/r/datascience/comments/996zkv/what_is_a_good_ggplot2_equivalent_to_python/e4ljloz/

Cohen, S. (2015, 1 8). *What challenges has Pinterest encountered with Flask?* Retrieved from Quora: https://www.quora.com/What-challenges-has-Pinterest-encountered-with-Flask/answer/Steve-Cohen

Datanyze. (n.d.). *Business Intelligence Market Share Report.* Retrieved 8 2019, from Datanyze: https://www.datanyze.com/market-share/business-intelligence

Demsar, J., Curk, T., Erjavec, A., Gorup, C., Hocevar, T., Milutinovic, M., . . . Zupan, B. (2013, 8). Orange: Data Mining Toolbox in Python. *Journal of Machine Learning Research*, 2349-2353.

Erl, T., Carlyle, B., Pautasso, C., & Balasubramanian, R. (2012). *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST.* New Jersey, US: Prentice Hall.

Fowler, M. (2014, 3 25). *Microservices.* Retrieved from Martin Fowler: https://martinfowler.com/articles/microservices.html

Frank, E., Hall, M., & Witten, I. (2016). *The WEKA Workbench, Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann.*

Goodger, D., & van Rossum, G. (2001, 5 29). *Docstring Conventions.* Retrieved from Python.org: https://www.python.org/dev/peps/pep-0257/

Hagberg, A., Schult, D., & Swart, P. (2008). Exploring network structure, dynamics, and function using NetworkX. *7th Python in Science Conference* (pp. 11-15). Pasadena, CA, USA: Gael Varoquax, Travis Vaught and Jarrod Millman (Eds).

Hermann Negri, L., & Vestri, C. (2017, 9 8). peakutils. doi:10.5281/zenodo.887917

Holmes, G., Donkin, A., & Witten, I. (1994). WEKA: a machine learning workbench. *Proceedings of ANZIIS '94 - Australian New Zealnd Intelligent Information Systems Conference.* IEEE. doi:10.1109/ANZIIS.1994.396988

Hu, K., Orghian, D., & Hidalgo, C. (2018). DIVE: A Mixed-Initiative System Supporting Integrated Data Exploration Workflows. *ACM SIGMOD Workshop on Human-In-the-Loop Data Analytics (HILDA* (p. 7). Houston, TX, USA: ACM Digital Library. doi:10.1145/3209900.3209910

Hunter, J. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering, 9*(3), 90-95. Retrieved from https://ieeexplore.ieee.org/document/4160265

Industrifonden. (n.d.). *Qlik.* Retrieved from Industrifonden: https://industrifonden.com/success-stories/qlik/

Jones, E., Oliphant, T., Peterson, P., & et al. (2001). SciPy: Open source scientific tools for Python.

Levy, A. (2013, 5 16). *Seattle's Tableau raises $254M in year's biggest tech IPO.* Retrieved from Seattle Times: http://old.seattletimes.com/html/businesstechnology/2021001333_tableauipoxml.html

Martin, R. (2015, 12 15). *Principles of Clean Architecture.* Retrieved from YouTube: https://www.youtube.com/watch?v=o_TH-Y78tt4

McKinney, W. (2010). Data Structures for Statistical Computing in Python. *9th Python in Science Conference*, (pp. 51-56).

Meurer, A., Smith, C., Paprocki, M., Certik, O., Kirpichev, S., Rocklin, M., . . . Scopatz, A. (2017, 1). SymPy: symbolic computing in Python. *PeerJ Computer Science*. Retrieved from https://doi.org/10.7717/peerj-cs.103

Microsoft. (2017, 2 16). *Microsoft breaks through in the Gartner Magic Quadrant for Business Intelligence and Analytics Platform.* Retrieved from Microsoft blog: https://blogs.microsoft.com/blog/2017/02/16/microsoft-breaks-gartner-magic-quadrant-business-intelligence-analytics-platforms

Microsoft. (2018, 2). *Microsoft recognized as a leader in analytics and BI platforms for 11 years.* Retrieved from Microsoft info: https://info.microsoft.com/ww-landing-gartner-bi-analytics-mq-2018-partner-consent-test.html

nish2288. (2018, 5 27). *docker.* Retrieved from Power BI Community: https://community.powerbi.com/t5/Desktop/docker/td-p/426116

Occhino, T., & Walke, J. (2013). JS Apps at Facebook. USA: JSConfUS 2013. Retrieved from https://www.youtube.com/watch?v=GW0rj4sNH2w

Oliphant, T. (2006). *A guide to NumPy.* Trelgol Publishing.

Park, H. (2019, 7). *The Death of Big Data and the Emergence of the Multi-Cloud Era*. Retrieved from KD Nuggets: https://www.kdnuggets.com/2019/07/death-big-data-multi-cloud-era.html

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., . . . Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2825-2830. Retrieved from http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf

Peters, T. (2004, 8 19). *The Zen of Python.* Retrieved from Python.org: https://www.python.org/dev/peps/pep-0020/

Qlik. (2019, 2). *2019 Magic Quadrant for Analytics and Business Intelligence Platforms.* Retrieved from Qlik: https://www.qlik.com/us/gartner-magic-quadrant-business-intelligence

Richardson, C. (2019, 1). *Microservices adoption anti-patterns: Obstacles to decomposing for testability and deployability .* Retrieved from LinkedIn: https://www.slideshare.net/chris.e.richardson/melbourne-jan-2019-microservices-adoption-antipatterns-obstacles-to-decomposing-for-testability-and-deployability

Robinson, D. (2017, 9 6). *The Incredible Growth of Python.* Retrieved from Stack Overflow Blog: https://stackoverflow.blog/2017/09/06/incredible-growth-python/

R-Project. (2019, 10). *Contributed Packages.* Retrieved from R-Project.org: https://cran.r-project.org/web/packages/

Sanders, R. (2014). Developing Flask Extensions. PyCon 2014. Retrieved from https://www.youtube.com/watch?v=OXN3wuHUBP0

Schmitz, D. (2017, 8 30). *10 Tips for failing badly at Microservices.* Retrieved from YouTube: https://www.youtube.com/watch?v=X0tjziAQfNQ

Stack Overflow. (2019). *Stack Overflow Developer Survey.* Retrieved from Stack Overflow Insights: https://insights.stackoverflow.com/survey/2019#technology

TIOBE. (2019, 10). *TIOBE Index.* Retrieved from TIOBE: https://www.tiobe.com/tiobe-index/

van Rossum, G., Warsaw, B., & Coghlan, N. (2001, 7 5). *Style Guide for Python Code.* Retrieved from Python.org: https://www.python.org/dev/peps/pep-0008/

W3C Working Group, Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., . . . Orchard, D. (2004, 2 11). *Web Services Architecture.* Retrieved from W3.org: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest

Walsh, P., & Pollock, R. (n.d.). *Table Schema.* Retrieved from Frioctionless Data: https://frictionlessdata.io/specs/table-schema/

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis.* New York: Springer-Verlag.

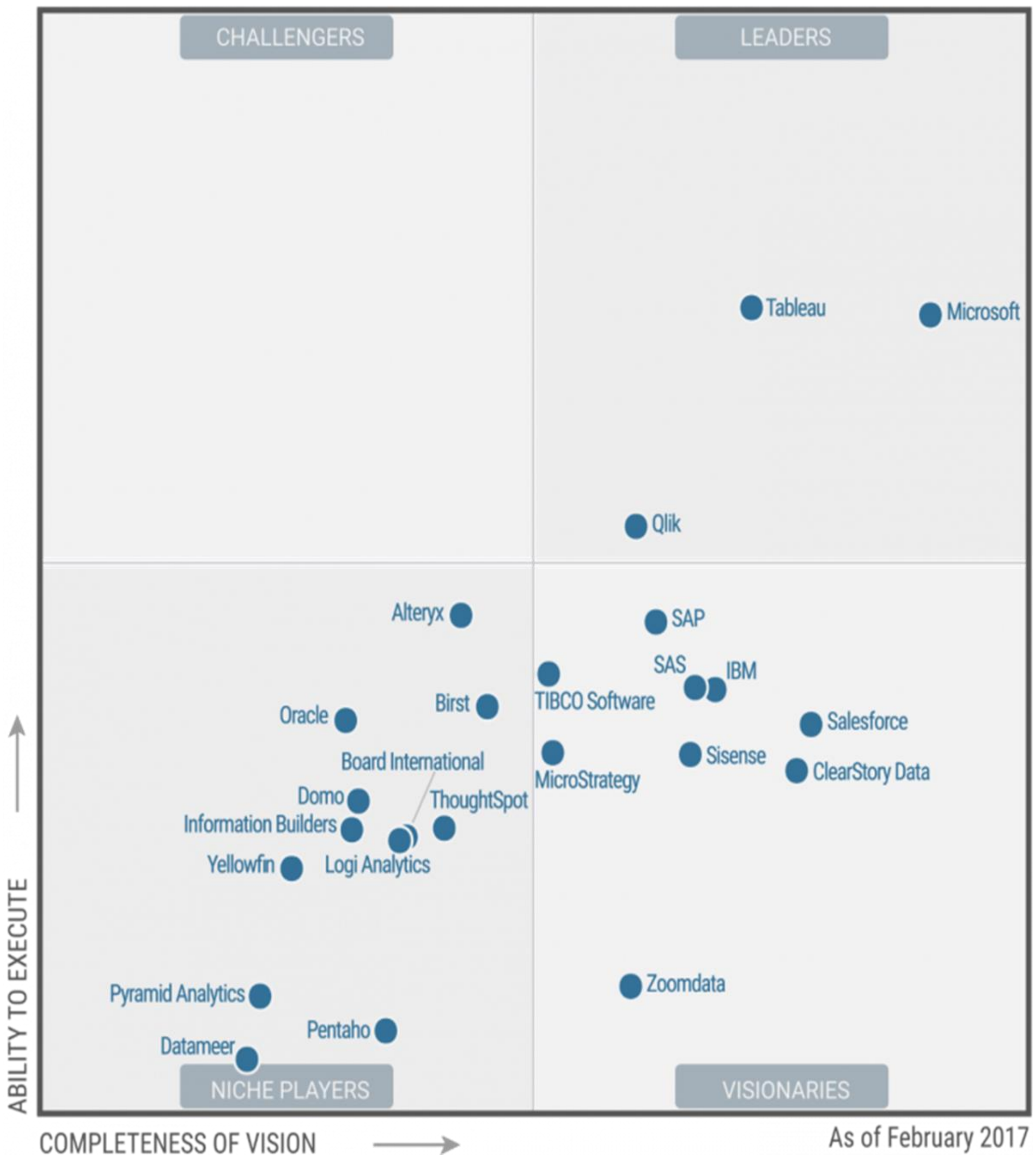
Witten, Frank, Trigg, Hall, Holmes, & Cunningham. (1999). Weka: Practical machine learning tools and techniques with Java implementations. *Computer Science Working Papers,* *University of Waikato.* Hamilton, New Zealand: University of Waikato, Department of Computer Science.

Wu, T. (2019, 6 16). *What are the top ten problems for data scientists?* Retrieved from Quora: https://www.quora.com/What-are-the-top-ten-problems-for-data-scientists

xello. (n.d.). *Power BI Desktop vs Power BI Pro and Premium: What's the difference?* Retrieved from xello: https://xo.xello.com.au/blog/power-bi-desktop-vs-power-bi-pro-and-premium-differences

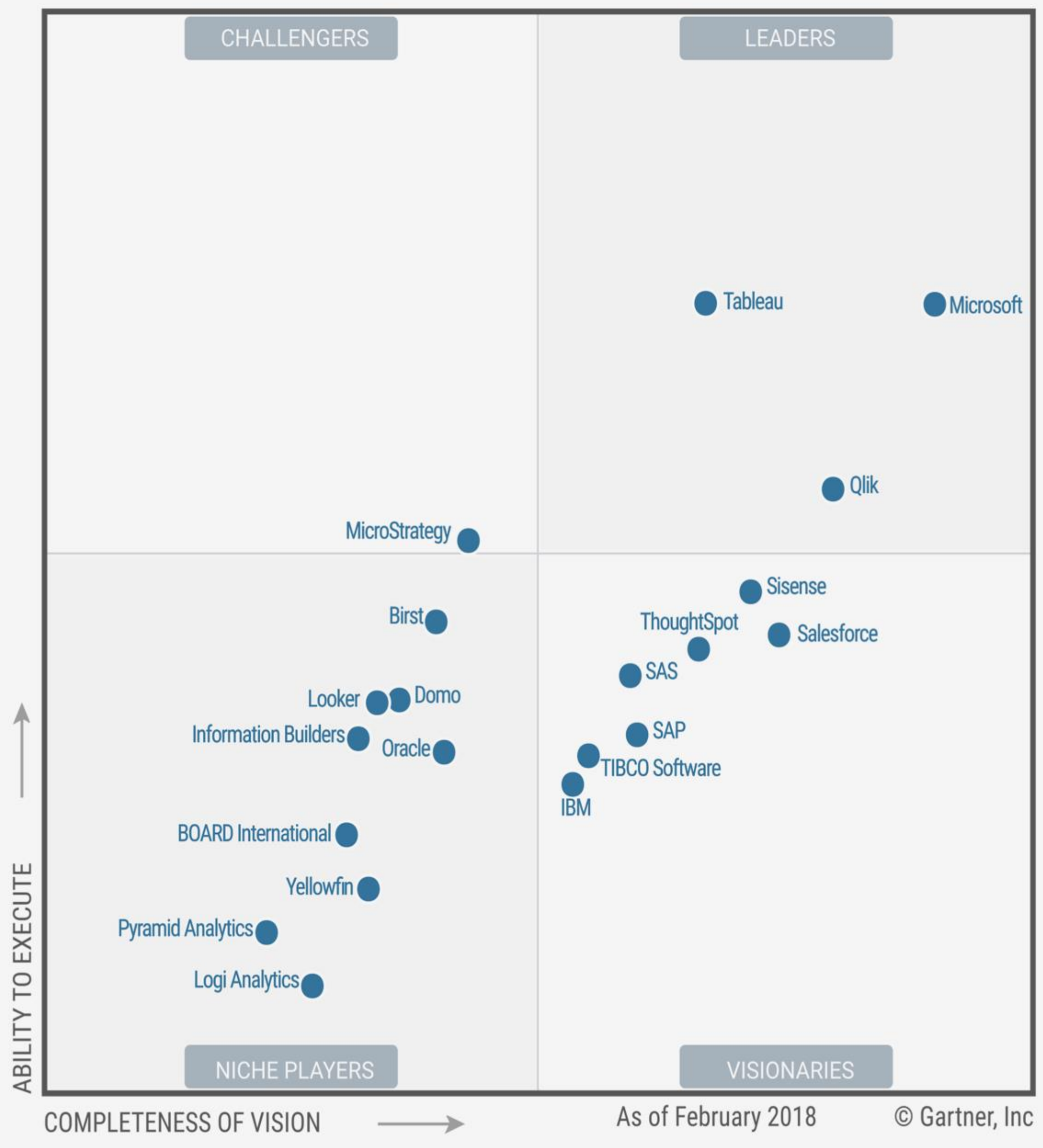
York, R. (2009). *Beginning JavaScript and CSS Development with jQuery.* Indianapolis, IN, USA: Wiley Publishing, Inc.

Appendix 1, Gartner reports



Gartner Magic Quadrant, **2017**, https://blogs.microsoft.com/blog/2017/02/16/microsoft-breaks-gartner-magic-quadrant-business-intelligence-analytics-platforms/

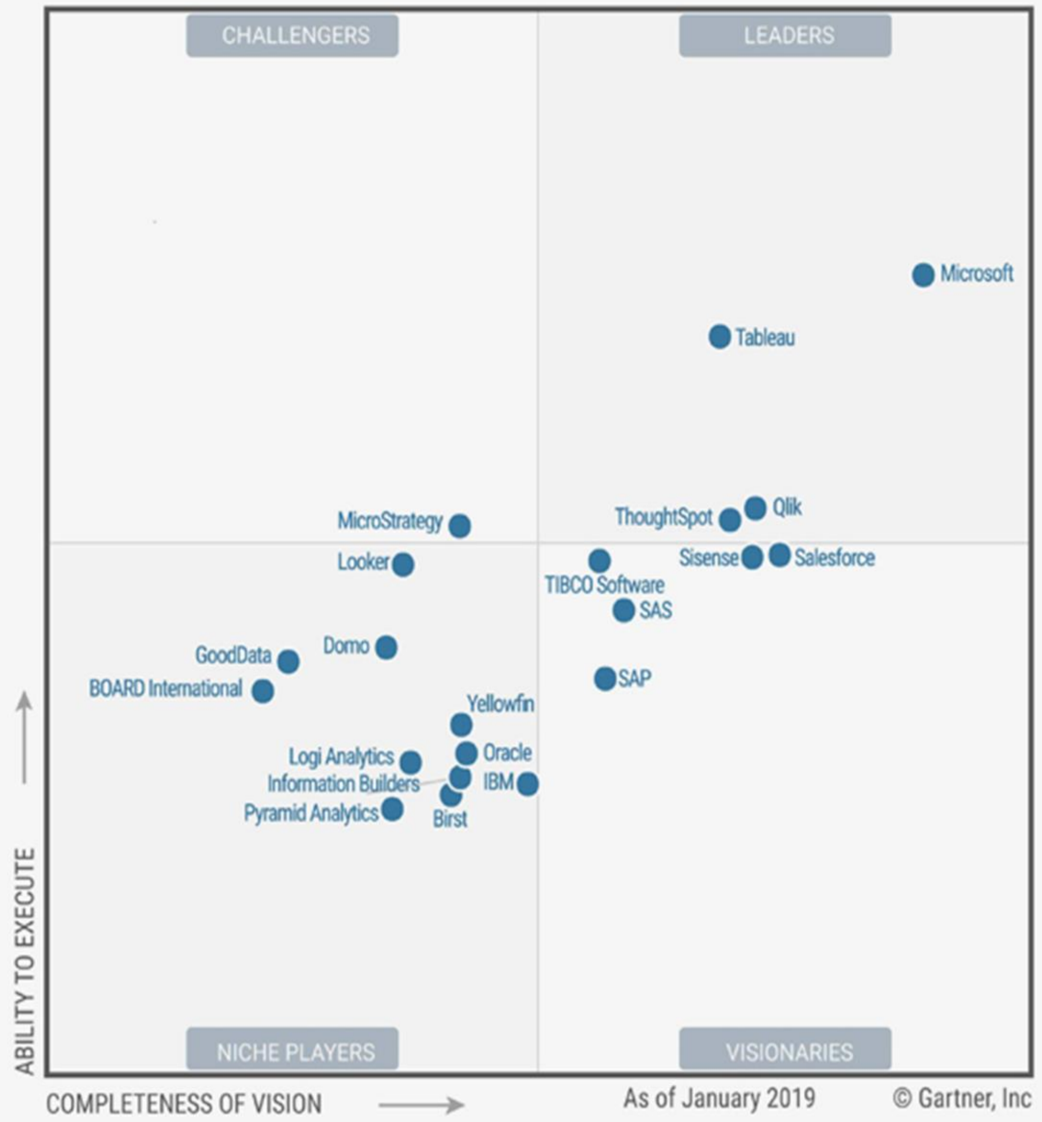
Figure 1. Magic Quadrant for Analytics and Business Intelligence Platforms



Source: Gartner (February 2018)

Gartner Magic Quadrant, **2018**, https://info.microsoft.com/ww-landing-gartner-bi-analytics-mq-2018-partner-consent-test.html

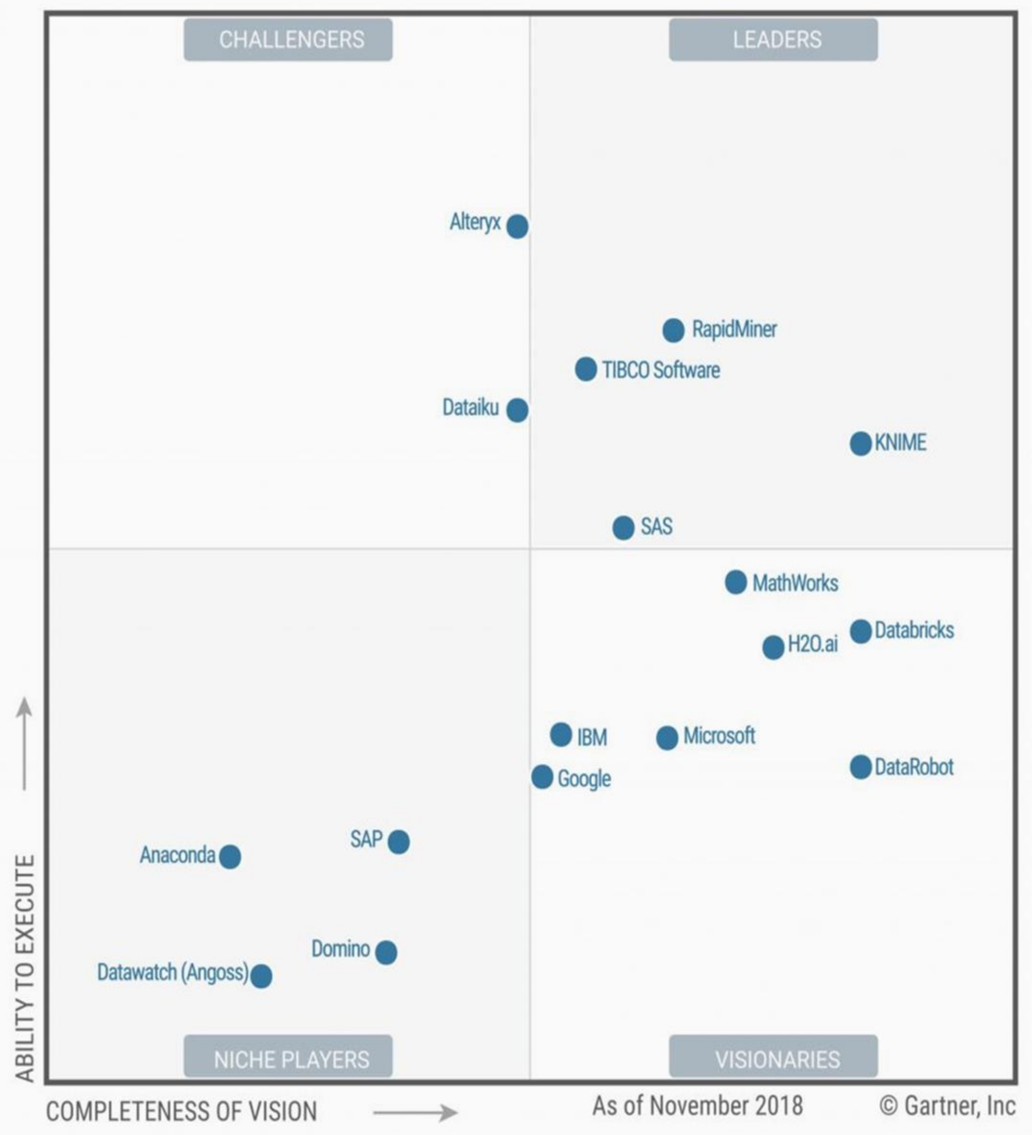
Figure 1. Magic Quadrant for Analytics and Business Intelligence Platforms



Source: Gartner (February 2019)

Gartner Magic Quadrant, **2019**, https://www.qlik.com/us/gartner-magic-quadrant-business-intelligence

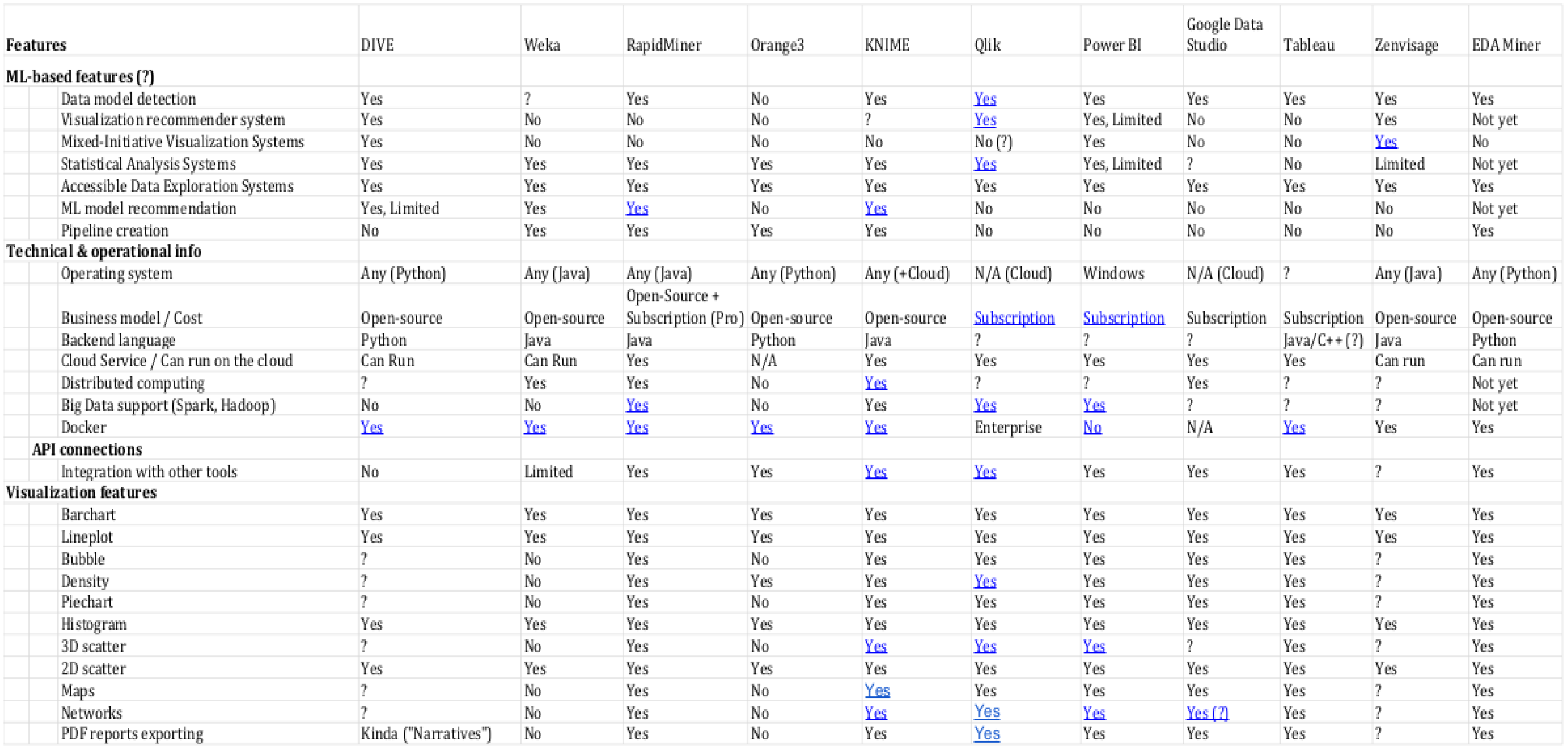
Figure 1. Magic Quadrant for Data Science and Machine Learning Platforms



Source: Gartner (January 2019)

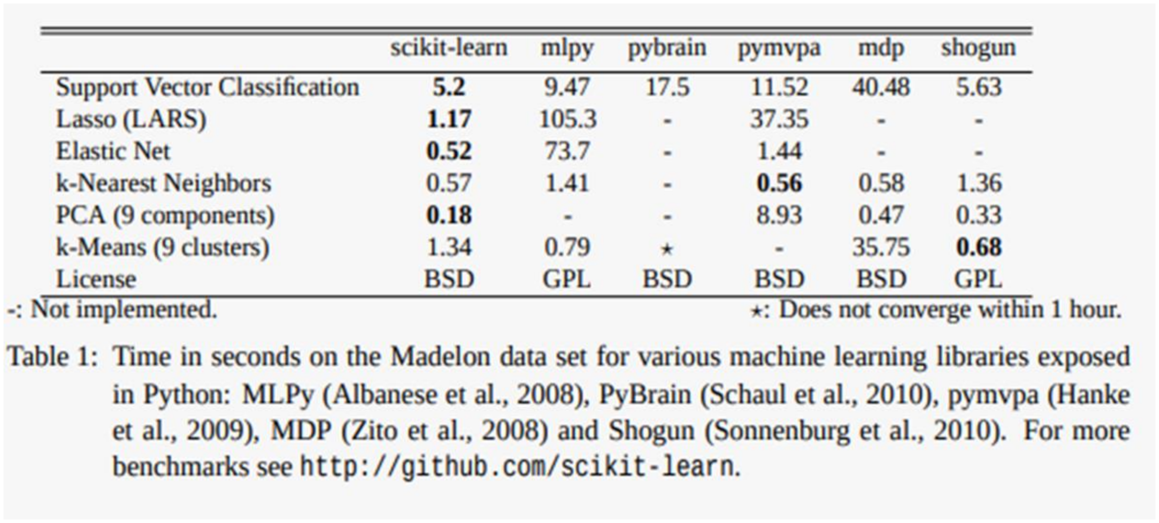
Gartner Magic Quadrant, 2019, https://rapidminer.com/resource/gartner-magic-quadrant-data-science-platforms/

Appendix 2, table comparison of tools



Features	DIVE	Weka	RapidMiner	Orange3	KNIME	Qlik	Power BI	Google Data Studio	Tableau	Zenvisage	EDA Miner
ML-based features (?)											
Data model detection	Yes	?	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Visualization recommender system	Yes	No	No	No	?	Yes	Yes, Limited	No	No	Yes	Not yet
Mixed-Initiative Visualization Systems	Yes	No	No	No	No	No (?)	Yes	No	No	Yes	No
Statistical Analysis Systems	Yes	Yes	Yes	Yes	Yes	Yes	Yes, Limited	?	No	Limited	Not yet
Accessible Data Exploration Systems	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ML model recommendation	Yes, Limited	Yes	Yes	No	Yes	No	No	No	No	No	Not yet
Pipeline creation	No	Yes	Yes	Yes	Yes	No	No	No	No	No	Yes
Technical & operational info											
Operating system	Any (Python)	Any (Java)	Any (Java)	Any (Python)	Any (+Cloud)	N/A (Cloud)	Windows	N/A (Cloud)	?	Any (Java)	Any (Python)
Business model / Cost	Open-source	Open-source	Open-Source + Subscription (Pro)	Open-source	Open-source	Subscription	Subscription	Subscription	Subscription	Open-source	Open-source
Backend language	Python	Java	Java	Python	Java	?	?	?	Java/C++ (?)	Java	Python
Cloud Service / Can run on the cloud	Can Run	Can Run	Yes	N/A	Yes	Yes	Yes	Yes	Yes	Can run	Can run
Distributed computing	?	Yes	Yes	No	Yes	?	?	Yes	?	?	Not yet
Big Data support (Spark, Hadoop)	No	No	Yes	No	Yes	Yes	Yes	?	?	?	Not yet
Docker	Yes	Yes	Yes	Yes	Yes	Enterprise	No	N/A	Yes	Yes	Yes
API connections											
Integration with other tools	No	Limited	Yes	Yes	Yes	Yes	Yes	Yes	Yes	?	Yes
Visualization features											
Barchart	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Lineplot	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Bubble	?	No	Yes	No	Yes	Yes	Yes	Yes	Yes	?	Yes
Density	?	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	?	Yes
Piechart	?	No	Yes	No	Yes	Yes	Yes	Yes	Yes	?	Yes
Histogram	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
3D scatter	?	No	Yes	No	Yes	Yes	Yes	?	Yes	?	Yes
2D scatter	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Maps	?	No	Yes	No	Yes	Yes	Yes	Yes	Yes	?	Yes
Networks	?	No	Yes	No	Yes	Yes	Yes	Yes (?)	Yes	?	Yes
PDF reports exporting	Kinda ("Narratives")	No	Yes	No	Yes	Yes	Yes	Yes	Yes	?	Yes

Appendix 3, sklearn performance



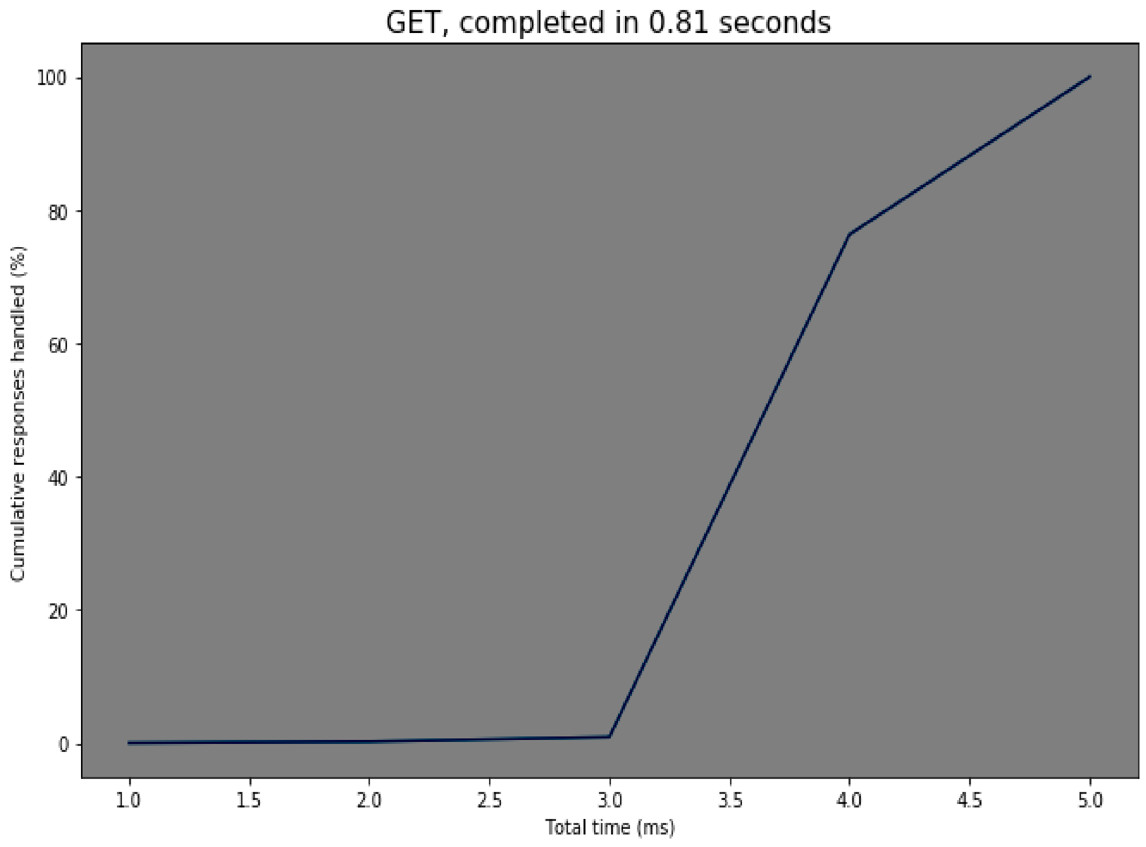
	scikit-learn	mlpy	pybrain	pymvpa	mdp	shogun
Support Vector Classification	**5.2**	9.47	17.5	11.52	40.48	5.63
Lasso (LARS)	**1.17**	105.3	-	37.35	-	-
Elastic Net	**0.52**	73.7	-	1.44	-	-
k-Nearest Neighbors	0.57	1.41	-	**0.56**	0.58	1.36
PCA (9 components)	**0.18**	-	-	8.93	0.47	0.33
k-Means (9 clusters)	1.34	0.79	⋆	-	35.75	**0.68**
License	BSD	GPL	BSD	BSD	BSD	GPL

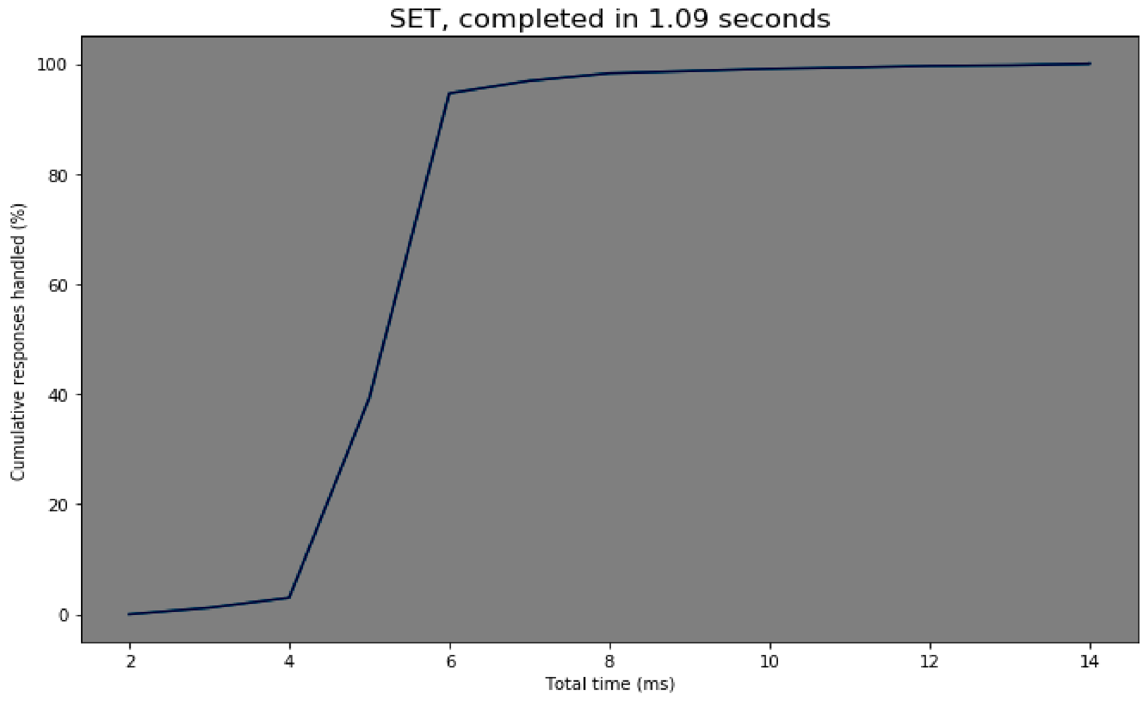
-: Not implemented. ⋆: Does not converge within 1 hour.

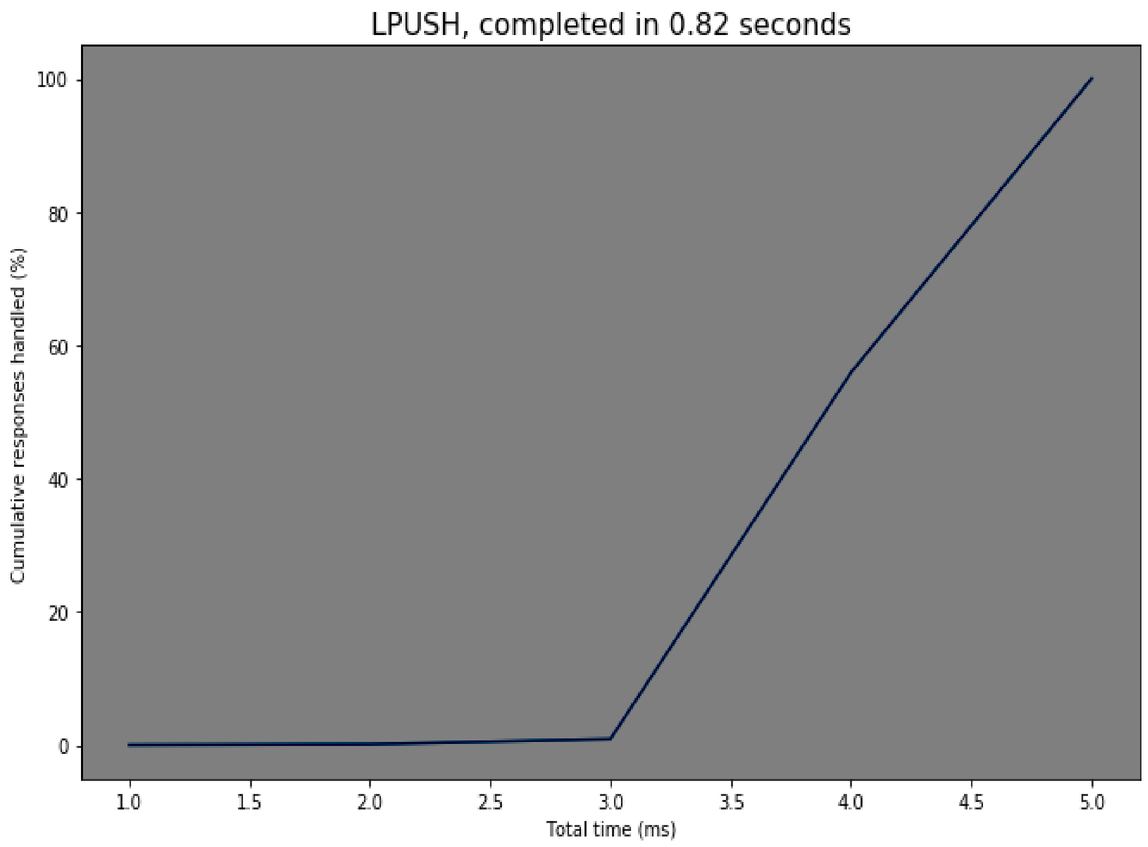
Table 1: Time in seconds on the Madelon data set for various machine learning libraries exposed in Python: MLPy (Albanese et al., 2008), PyBrain (Schaul et al., 2010), pymvpa (Hanke et al., 2009), MDP (Zito et al., 2008) and Shogun (Sonnenburg et al., 2010). For more benchmarks see `http://github.com/scikit-learn`.

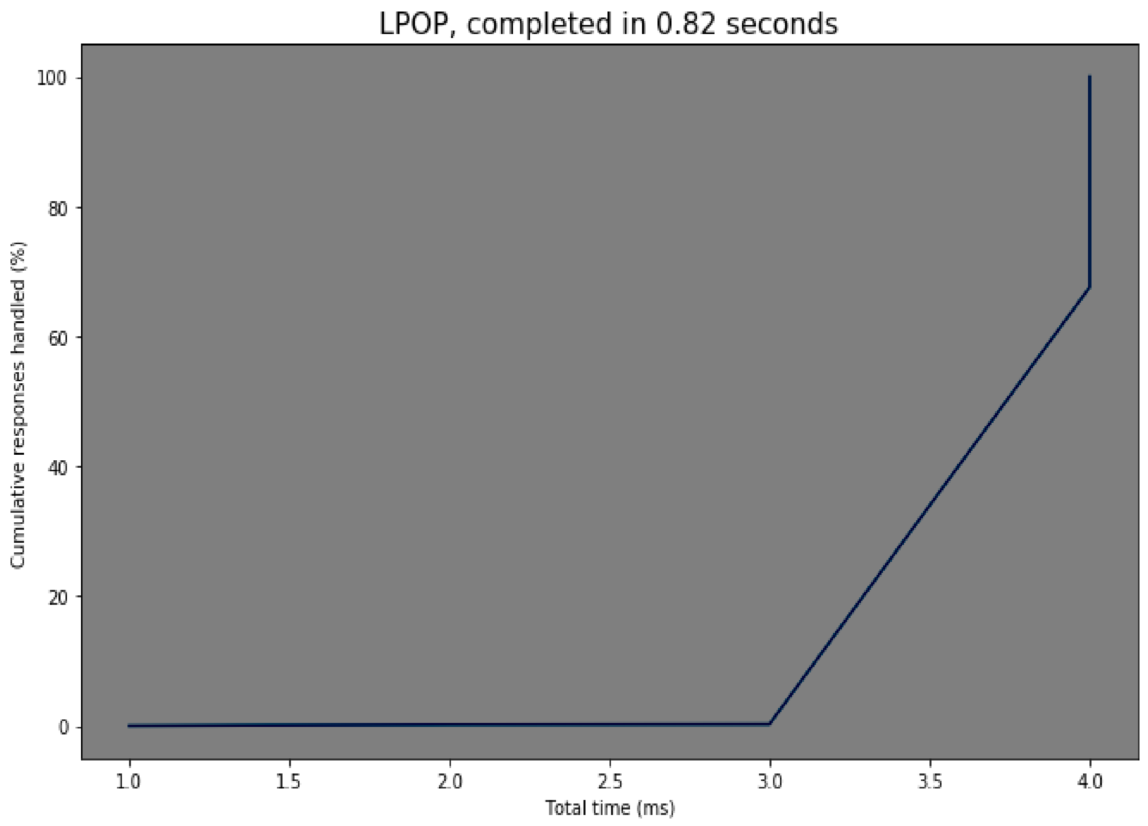
(Pedregosa, et al., 2011, p. 4)

Appendix 4, Redis benchmarks






LPUSH, completed in 0.82 seconds
Cumulative responses handled (%)
Total time (ms)
0
20
40
60
80
100
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


LPOP, completed in 0.82 seconds
Cumulative responses handled (%)
Total time (ms)
0
20
40
60
80
100
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Appendix 5, Flask + hey Benchmarks

Report 1

```
  Total: 10.4634 secs
  Slowest:  3.2022 secs          Fastest:  0.0331 secs          Average:
0.0522 secs
  Requests/sec:  955.7123        Total data:  670000 bytes      Size/request:
67 bytes

Response time histogram:
  0.033 [1]     |
  0.350 [9949]  |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.667 [0]     |
  0.984 [0]     |
  1.301 [0]     |
  1.618 [0]     |
  1.935 [0]     |
  2.251 [0]     |
  2.568 [0]     |
  2.885 [0]     |
  3.202 [50]    |

Latency distribution:
  10% in 0.0342 secs
  25% in 0.0352 secs
  50% in 0.0364 secs
  75% in 0.0374 secs
  90% in 0.0392 secs
  95% in 0.0400 secs
  99% in 0.0424 secs
```

```
Details (average, fastest, slowest):
  DNS+dialup: 0.0001 secs, 0.0331 secs, 3.2022 secs
  DNS-lookup: 0.0000 secs, 0.0000 secs, 0.0000 secs
  req write:  0.0000 secs, 0.0000 secs, 0.0024 secs
  resp wait:  0.0521 secs, 0.0329 secs, 3.2009 secs
  resp read:  0.0000 secs, 0.0000 secs, 0.0003 secs
```

Report 2 (cached)

```
  Total: 7.2396 secs
  Slowest:  0.0425 secs            Fastest:  0.0041 secs            Average:
0.0361 secs
  Requests/sec:  1381.2998         Total data:  670000 bytes        Size/request:
67 bytes

Response time histogram:
  0.004 [1]     |
  0.008 [3]     |
  0.012 [7]     |
  0.016 [4]     |
  0.019 [6]     |
  0.023 [5]     |
  0.027 [5]     |
  0.031 [6]     |
  0.035 [2698] |■■■■■■■■■■■■■■■■■■
  0.039 [6119] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.042 [1146] |■■■■■■■

Latency distribution:
  10% in 0.0338 secs
  25% in 0.0347 secs
  50% in 0.0360 secs
  75% in 0.0373 secs
  90% in 0.0389 secs
  95% in 0.0400 secs
  99% in 0.0411 secs

Details (average, fastest, slowest):
  DNS+dialup: 0.0001 secs, 0.0041 secs, 0.0425 secs
  DNS-lookup: 0.0000 secs, 0.0000 secs, 0.0000 secs
  req write:  0.0000 secs, 0.0000 secs, 0.0008 secs
  resp wait:  0.0360 secs, 0.0020 secs, 0.0424 secs
  resp read:  0.0000 secs, 0.0000 secs, 0.0004 secs
```

Appendix 6, Contributor guidelines

As you are probably aware, this project is almost entirely written in Dash, Python. There are very few Javascript snippets here and there, and there are also some custom Dash components made with React. For CSS this project uses part of Dash's defaults, some Bootstrap, and of course some custom sheets.

Table of contents

Learning resources

- Basics: To write Dash, you just need a basic knowledge of HTML. It is probably impossible to contribute without first going over the The Dash docs, which are also the best place to start your reading. Going through their tutorial, and then taking a quick look on each component library listed should get you up to speed with what you need to know about callbacks, the main feature of Dash, and more.
- Visualizations: Dash steps on the shoulders of plotly for its visualizations, so if you are interested in working on visualizations take a look at the Plotly docs page, and/or at D3.js for Dash from the Dash docs.
- Custom components / React: If you're a JS wiz with a react spellbook, then we love you a bit more (you can design your own components!). You can start working with custom components, and React, Javascript. Take a quick look at what these Dash docs page say: React for Python Developers, and Writing your own components.
- Machine Learning: There are hundreds of material out there and we won't go over them here but familiriaty with sklearn and its programmatic API is strongly desired (see Quickstart, User guide, API reference, Developers guide, especially the Estimators chapter).

Style guide recommendations

You need to write at least somewhat pythonic code. A few good starting points are: PEP-8 and The Hitchhiker's Guide to Python. Also take a quick look at this quick discussion about project structure (mentioned again later).

Regarding imports, it seems to me far easier to find stuff when they are in a particular order, with a line separating each category:

1. The core Dash libraries (e.g. `dash`, `dcc`, `html`, `daq`, `dash_table`) and other Dash components including any custom (e.g. `visdcc`, `dash_rnd`)
2. Import modules from the app: first `server.app`, then other high-level modules, then anything else ordered however you like.
3. Import any other libraries necessary, ordered by fancy points.

Example

```
import dash_core_components as dcc
import dash_html_components as html
import dash_table
import visdcc

from server import app
from utils import cleanup, r
from menus import SideBar, MainMenu, landing_page
from apps import data_view, exploration_view
from apps.exploration_tabs import KPIs

import turtle
from functools import partial
import pandas as pd
```

Naming is extremely important. Use plural for lists (e.g. `traces = [scatter]`) and singular with optional numbering for sole items (e.g. `trace1 = scatter`). Edit away others' work if necessary, and feel free to suggest more rules that will help us keep our sanity. Also, if you find some names weird, flame the author hard and suggest alternatives.

General info on project structure

The older project structure was a mess, but I hope the new one (post v0.3, aka 19th August 2019) is much cleaner. The app is structured as a main Flask app (responsible for user management, mainly) which connects various other Flask and Dash apps at the WSGI level with Werkzeug's DispatcherMiddleware. If you want to add a new app all you need to do is follow the instruction in the `wsgi.py` file (i.e. register the flask extensions and give it a URL path).

You can get acquainted with the project by looking at `/docs` after starting the app, or (equivalently) reading the module and package docstrings. Another way is by visiting the `/images` folder where you will find an old version of `eda_miner_callback_chain.html` which was the Dash-rendered graph showing every function in the project (also found as images, divided over several screenshots) on `v0.2`. Another way is to look directly at the `directory_tree.png`, (made with the linux `tree` command).

It is extremely likely that after these you will still want an explainer; don't worry, anyone working on the project will be more than happy to help you (see Contact section). Anyhow, here is the rough idea:

- Top level modules define the overall app (Flask server, models, forms, flask extensions) including some utils; probably not the place to start playing.
- `/static` and `/templates/` are the place to go if you feel like adding some custom JS/CSS (or HTML for the main Flask app) or storing your images.
- App folders; these are (almost-)standalone applications in either Dash/Flask (maybe Django?). If you want to create an extension that does something more elaborate or separate than existing apps, or just want to add your pre-existing Dash/Flask apps, you create a new directory and do it (remember to visit `wsgi.py`!). Here are a three of them:
    - `data`: This Dash app contains all the logic for the user to handle data, be it uploading data or fetching them from an API. It also detects a data schema following some heuristics, and allows the user to edit it. Found at `/data`, and needs login.
        - Contents of each app are largely up to you. We will soon release a "mini-app" template with more detailed instruction so you can use that as a reference.
        - Current apps usually include a `server.py` (Dash config & definition) and an `app.py` (index page) file as well as different python files for each sub-page (we usually use tabs as a navigation method, feel free to deviate).
    - `docs` is a standalone app that parses the rest of the python codebase as strings, extracts docstrings and automatically creates an API reference page found at `/docs` (no login required).
    - `google_analytics` is a REST API built with Flask and runs completely independently of the main Flask server. It was initially created as a separate "micro-service" because of an incompatibility with the libraries that we used but that is no longer the case. Still, we decided to keep it mainly for demonstration purposes. Go over to `/data/data_utils/api_layouts` to see how `GAnalyticsAPI` interacts with it.

Suggestion on project structure are also welcome.

General info on contributions

Almost everything here is a soft suggestion, not a hard requirement. If you're confident in your coding, don't pay too much attention to these guidelines.

Do the thing
-Varrick, inventor, businessman, rebel.

For now, modules in the top level of the app should probably not be modified, unless you're an experienced dev (optionally with Flask knowledge). For other modules, there are docstrings with a few more words on whether work is encouraged in those modules. Do note: *Any and every contribution is welcome*; those modules are just marked as such to notify beginners to be cautious, or they merely mark areas where we think improvements are needed most - and are easy to implement. Each module as well as each package has its own guidelines (see `Notes to others` section of each), which are there to give you a gereral idea of what you could do. That said, docstrings are somewhat outdated as of this writing (19th August 2019).

Advanced users can safely ignore all that. If you want to contribute but don't know where to start, find issues marked with `good first issue` in GitHub Issues. If you have some standalone code that works but don't know where that fits into the project, open an issue or a pull request. A few examples:

- Stylistic elements (UI/UX, CSS, themes, graph coloring)
- API integrations, data gathering & handling
- Machine Learning Algorithms (classes with a sklearn-like API: `.fit`, `.predict`, `.transform`)
- Custom React components (e.g. resize-n-drag).

Also, if you have suggestions open an issue with a "feature request" tag. A few suggestions on good (and potentially easy) contributions that are really needed (and not mentioned below):

- Writing function/class/module docstrings
- Writing tutorials on the usage of the app
- Writing tutorials for helping other contributors
- Writing (unit-)tests
- Benchmarking (parts or all of the app, libraries, other tools used)

Contributions for code quality

With v0.3 I tried quite a lot to improve code quality; that is to make everything more readable, with more/better comments and docstrings, cleaner and visually appealing, as well as make it simpler for new contributors to do their thing. I believe I did well (not perfect) with `/data/apis` where you have clear instructions on how to make a new API connection, and then the rest is handled for you. I probably didn't help with `/visualization`. I know I failed super hard with `/modeling/model_builder`; a few times it made me want to cry (thank you, *pizza*, for the emotional support!). **Helping here is HUGELY APPRECIATED**. Really. If you are a brave soul and attempt it, do not hesitate to spam and flame me with questions till I cry.

Contributions for visualization

Dash graphs are mainly done in plotly, and we don't promise much for other libraries (but fire away anyway!). We have tried our hands at using matplotlib which somewhat works (plotly has some integration). D3 visualizations should also be possible and are welcome. If you want to wrap some JS library as a collection of React components then that is great as well. You will also probably find discussions about integrating Dash with other visualization tools (such as for folium, e.g. here and here).

Visualization is currently handled by the `/visualization` app. It is split across 6 modules but might be trimmed in the future. These modules are (and some ideas on what can be done):

- ChartMaker: Handling basic 2D and 3D visualizations which are built using individual traces, much like a poor man's ripoff of Plotly's Online Graph Maker. Other modules (e.g. maps, kpis) might be integrated here.
- KPIs: Handling... KPIs! Currently it only calculates a simple Baseline (without any sort of filtering). You could augment the baseline by adding filters (e.g. for promo dates), or new KPIs, or even allow users to create their own.
- Maps: Currently only two types of maps are supported, choropleths, geoscatters (and a combination), and "lines on map", all from Plotly. You could add more, or provide more options, even mapping functionality with/from other libraries and providers.
- Networks: Very basic network/graph visualizations. Currently it has trouble handling more than a couple hundred nodes so that is one possible way to contribute. Also, styles and interactivity for the cytoscape graph. You might also want to try integration with networkx or other libraries.
- Text visualizations: For now it only handles a simple wordcloud. Feel free to add additional visualizations, but do leave word-vector visualizations for later (or use very small models) since increasing server boot time is an issue.
- DashboardMaker: Create your own dashboards, Power BI style. Not quite, because the drag-and-resize is not yet complete, and a lot of stuff still need to be done; so that's your cue, React wizs ;)

Contributions for data

Everyone needs data to work with, and currently there are two ways ways to get data: the user can upload a file, or we can connect to their account somewhere and fetch data via an API. After that, the user should be able to inspect and edit both the data and the inferred schema. Here are a few suggestions on what you could do:

- Connections to APIs: Currently only 6 connections to APIs are supported (Google Analytics, Google Drive / Spreadsheets, Twitter, Quandl, Spotify, and Reddit). For these connections, we only access a very limited subset of what their APIs offer (e.g. only playlists for Spotify), so this is one thing you can work on. If you want to customize the API connections or create a new one, you need to subclass

`APIConnection` from `/data/data_utils/api_layouts.py`. Basically it needs two Dash layouts (giving credentials, and getting data after successful authentication), a (potentially dummy) method to prettily display fetched results, a `connect` method (to connect to the API), and a `fetch_data` method to actually get the data and save it as a `pandas.DataFrame` (use the inherited `save_data_and_schema`).

- Uploading data: A simple box where the user can drag-n-drop files (or click/navigate/open) to upload. A few things can be done here like adding supported for different filetypes (currently only csv, json, xls/x, and feather are supported), or handling large file uploads (see also Dash Resumable Upload).
- Schema inference: Responsible for detecting the data types of the various columns, using (currently) heuristics. You can improve the heuristics, create a better interface for viewing the data (e.g. with pagination), or other (?).
- View data: A Dash DataTable for inspecting the data. Potential improvements here include handling pagination, editting better (e.g. save edits), and schema-relevant operations (e.g. categorical columns svisualizationshould have a dropdown).
- **New features**:
    - Ability to connect datasets with a schema, like in SQL.
    - Concatenate datasets (columns/rows).
    - Permanent storage of data fetched from APIs, e.g. by concatenating previous results. This is important.
    - Query the data with an SQL-like syntax (with or without a front-end GUI) or natural language (text2sql).
    - Create new columns/features using formulas. Spreadsheets can handle it, so why not us too? *This used to be part of the FeatureMaker class in the ModelBuilder but was removed for convenience*.

Contributions for modeling

This app is responsible for training Machine Learning (Data Mining, Business Intelligence, Statistics, whatever) models. If you want to train a simple model, the `single_model.py` module handles that case, but if you want something more complex then you would have to go to the `model_builder.py`, define a custom pipeline, and switch over to `pipelines.py` to train it. Each is accessible by selecting its respective tab.

- Single model: A simple GUI with few dropdowns to select dataset and variables, and a div with tabs for showing the various result types (currently 2: text metrics like accuracy, confusion matrix, MSE, and graphical results).
- Model Builder: Using `dash-cytoscape` we use graph nodes to represent the various "estimators" (using sklearn language). Every model class **MUST** conform to the sklearn API (fit, predict, transform). You can add new models/classes (see `/modeling/models/pipeline_classes.py`). If you are brave enough, you can try converting cytoscape graphs to models (see `/modeling/models/graph_structures.py`) and/or creating pipelines from them (see `/modeling/models/pipeline_creator.py`) as well as training them (see `/modeling/pipelines.py`).

Contributions for deployment and scaling

Up to now, deployment has not been that much of a concern because I was mostly handling it on my own. However, as the app grows and is tied to other services (Redis, google_analytics, some other SQL soon?) we will need a better way to handle this. Currently I'm working a bit on a `docker-compose` script, especially since docker seems the best way to package and set-up the whole project.

The idea behind splitting up the various apps was due to two reasons: allowing for easier addition of new apps (which was achieved), and allowing each app to scale independently (which is currently not done at all). Connecting Flask and Dash apps at the WSGI level is easy and allows for the `login_manager` and other extensions to be connected easily. It is possible that connecting these extensions does not need all apps to be under the same server, but I simply don't know about that stuff and will look into them later on. If not, then each app will have to be separable and get its own url and connected via another dispatcher / proxy (e.g. see Werkzeug's HTTP Proxy). If you have other suggestions, or know a way to pass user session / information in a secure way across Dash/Flask apps, those would be **extremely welcome** as well.

Scaling and performance are issues I didn't concern myself with so far, at least not much. Since v0.3 some early attempts have been made in improving performance:

- Instead of loading datasets from Redis every time we need to create options according to dataset columns, we load the data schema which is a small dict growing only according to number of columns. The same principle applies to other parts where data copying has decreased.
- Caching (memoization) and expiration for Redis data were added to a lot of API calls, and will probably be added to a lot more. That said, a lot of performance upgrades can still be done, including overall scaling. Here are some ideas:
- Some plotly visualizations can benefit from either using OpenGL or performing aggregations in Python before plotting. This lessens the burden both for internet bandwidth and the browser, at the potential cost of graphing precision. For more than a few hundred/thousand data points these are probably worth it.
- The pre-v0.2 application could be scaled by simply creating more containers. This can be done now, too, but it would be a waste (and the user database would need some extra handling). Instead, each app should be scaled on its own (see previous paragraph).
- Currently CI/CD is lacking. I am using a custom Python script to copy files, create docs, run tests, update coverage, and remove non-public code from the private version (as of 19th August 2019 there is no such code: the login has been just integrated, export to PDF has been dropped completely). Integration with TravisCI and other tools exist but are at an early stage. I will be looking into this a bit more over the coming months, as well as tools like Ansible and Jenkins. If you know about these, do lend a hand or tips.
- Deployment: I used to have it deployed on an old computer of mine but now it is deployed on Amazon (pre-v0.2), sponsored by Prof. Ioannis Magnisalis (website). Neither of us look hard enough into this; a different Amazon service may be more fitting, or a combination of them. Reach out to either of us for suggestions or details.

List of contributors

Notice: try to keep them in alphabetical order, edit if you notice inconsistencies! Also, note that most contributions are not visible in the public version of the repository.

- **Active** (as of 19th August 2019):
    - **Gkoustilis George**, promotion, and asking tough questions.
    - **Magnisalis Ioannis**, oversight of the project, including academic and implementation guidance.
    - **Mouratidis Kostas**, everything related to code.
    - **Tentsoglidis Iordanis**, mostly the theoretical part, and university promotion.
- **Past**:
    - **Katrilakas George**, mostly the theoretical part, and a bit on treemaps.
    - **Timamopoulos Chris**, mostly the theoretical part, and a bit on 3D scatteroplot.
    - **Tsichli Vaso**, most of the graphs, a bit on model fitting reporting, LOTS of suggestions for the interface. Patient receiver of my spam.

Appendix 7, Directory Tree

Output of running the at the top-level directory the command: *tree -I "*.png|*.pyc|*pycache*|*.css*|*.jpg*" --dirsfirst* .

```
.
├── EDA_miner
│   ├── dash_rnd
│   │   ├── dash_rnd.dev.js
│   │   ├── dash_rnd.min.js
│   │   ├── _imports_.py
│   │   ├── __init__.py
│   │   ├── metadata.json
│   │   ├── original_source_credits.txt
│   │   ├── package.json
│   │   └── ResizeDraggable.py
│   ├── data
│   │   ├── assets
│   │   ├── data_utils
│   │   │   ├── api_connectors.py
│   │   │   ├── api_layouts.py
│   │   │   ├── ganalytics_metrics.py
│   │   │   ├── __init__.py
│   │   │   └── schema_heuristics.py
│   │   ├── apis.py
│   │   ├── app.py
│   │   ├── __init__.py
│   │   ├── schemata.py
│   │   ├── server.py
│   │   ├── upload.py
│   │   ├── users.db -> ../users.db
│   │   └── view.py
│   ├── devops
│   │   ├── app.py
│   │   ├── __init__.py
│   │   ├── server.py
│   │   ├── upload.py
│   │   └── view.py
│   ├── docs
│   │   ├── all_layouts.py
│   │   ├── app.py
│   │   ├── doc_maker.py
│   │   └── __init__.py
│   ├── google_analytics
│   │   ├── app.py
│   │   ├── Dockerfile
│   │   ├── README.md
│   │   └── requirements.txt
│   ├── modeling
│   │   ├── models
│   │   │   ├── graph_structures.py
│   │   │   ├── __init__.py
│   │   │   ├── pipeline_classes.py
│   │   │   └── pipeline_creator.py
│   │   ├── app.py
│   │   ├── __init__.py
│   │   ├── model_builder.py
│   │   ├── pipelines.py
│   │   ├── server.py
│   │   ├── single_model.py
│   │   ├── styles.py
│   │   └── users.db -> ../users.db
│   ├── presentation
│   │   ├── app.py
│   │   └── __init__.py
│   ├── static
│   │   ├── css
│   │   ├── images
│   │   │   ├── graph_images
│   │   │   └── icons
│   │   ├── favicon.ico
│   │   └── navbar_interactivity.js
│   ├── tasks
│   │   ├── assets
│   │   ├── app.py
│   │   ├── __init__.py
│   │   └── server.py
│   ├── templates
│   │   ├── apps
│   │   │   ├── data.html
│   │   │   ├── modeling.html
│   │   │   └── visualization.html
│   │   ├── base_dash.py
│   │   ├── base.html
│   │   ├── change_password.html
│   │   ├── create_presentation.html
│   │   ├── forgot_password.html
│   │   ├── index.html
│   │   ├── login.html
│   │   ├── profile.html
│   │   ├── register.html
│   │   ├── reset_password.html
│   │   ├── show_presentation.html
│   │   └── user_apps.html
│   ├── visualization
│   │   ├── graphs
│   │   │   ├── graphs2d.py
│   │   │   ├── __init__.py
│   │   │   ├── kpis.py
│   │   │   ├── textviz.py
│   │   │   └── utils.py
│   │   ├── app.py
│   │   ├── chart_maker.py
│   │   ├── dashboard_maker.py
│   │   ├── __init__.py
│   │   ├── kpis.py
│   │   ├── maps.py
│   │   ├── networks.py
│   │   ├── server.py
│   │   ├── text_viz.py
│   │   └── users.db -> ../users.db
│   ├── app_extensions.py
│   ├── config.py
│   ├── data_server.py
│   ├── devops_server.py
│   ├── docs_server.py
│   ├── env.py
│   ├── env_template.py
│   ├── exceptions.py
│   ├── flask_app.py
│   ├── forms.py
│   ├── initialize_project.py
│   ├── __init__.py
│   ├── layouts.py
│   ├── model_server.py
│   ├── models.py
│   ├── presentation_server.py
│   ├── tasks_server.py
│   ├── users.db
│   ├── users_mgt.py
│   ├── utils.py
│   ├── visualization_server.py
│   └── wsgi.py
├── example_data
│   ├── boston.feather
│   ├── churn.csv
│   ├── data.csv
│   ├── gtd_11to14_0615dist.csv
│   ├── gutenberg_sentences.csv
│   ├── iris.csv
│   ├── monthly-milk-production-pounds-p.csv
│   ├── network.csv
│   └── population.csv
├── images
│   ├── screenshots
│   └── directory_tree.txt
├── tests
│   ├── data
│   │   ├── data_utils
│   │   │   ├── __init__.py
│   │   │   ├── test_data_utils.py
│   │   │   └── test_schema_heuristics.py
│   │   └── __init__.py
│   ├── chromedriver
│   ├── coverage_report.txt
│   ├── __init__.py
│   ├── test_app.py
│   ├── test_docker.sh
│   ├── testing_utils.py
│   ├── test_tabs.py
│   ├── test_user_auth.py
│   ├── test_utils.py
│   └── users.db
├── CODE_OF_CONDUCT.md
├── CONTRIBUTING.md
├── Dockerfile
├── LICENSE
├── project_info.txt
├── README.md
└── requirements.txt

28 directories, 136 files
```