DATA SCIENCE THROUGH THE LOOKING GLASS AND WHAT WE FOUND THERE

Fotis Psallidas [1] **Yiwen Zhu** [1] **Bojan Karlas** [1 2] **Matteo Interlandi** [1] **Avrilia Floratou** [1] **Konstantinos Karanasos** [1]
Wentao Wu [1] **Ce Zhang** [2] **Subru Krishnan** [1] **Carlo Curino** [1] **Markus Weimer** [1]

ABSTRACT

The recent success of machine learning (ML) has led to an explosive growth both in terms of new systems and algorithms built in industry and academia, and new applications built by an ever-growing community of data science (DS) practitioners. This quickly shifting panorama of technologies and applications is challenging for builders and practitioners alike to follow. In this paper, we set out to capture this panorama through a wide-angle lens, by performing the largest analysis of DS projects to date, focusing on questions that can help determine investments on either side. Specifically, we download and analyze: (a) over 6M Python notebooks publicly available on GITHUB, (b) over 2M enterprise DS pipelines developed within COMPANYX, and (c) the source code and metadata of over 900 releases from 12 important DS libraries. The analysis we perform ranges from coarse-grained statistical characterizations to analysis of library imports, pipelines, and comparative studies across datasets and time. We report a large number of measurements for our readers to interpret, and dare to draw a few (actionable, yet subjective) conclusions on (a) what systems builders should focus on to better serve practitioners, and (b) what technologies should practitioners bet on given current trends. We plan to automate this analysis and release associated tools and results periodically.

1 INTRODUCTION

The ascent of machine learning (ML) to mainstream technology is in full swing: from academic curiosity in the 80s and 90s to core technology enabling large-scale Web applications in the 90s and 2000s to ubiquitous technology today. Given many conversations with enterprises, we expect that in the next decade most applications will be *"ML-infused"*, as also discussed in (Agrawal et al., 2019). This massive commercial success and academic interest are powering an unprecedented amount of engineering and research efforts—in the last two years alone we have seen over 23K papers in a leading public archive,[1] and millions of publicly shared data science (DS) notebooks corresponding to tens of billions of dollars of development time.[2]

As our team began investing heavily both in building systems to support DS and leveraging DS to build applications, we started to realize that the speed of evolution of this field left our system builders uncertain on what DS practitioners needed (e.g., *Are practitioners shifting to using only DNNs?*). On the other end, as DS practitioners we were equally puzzled on which technologies to learn and build upon (e.g., *Shall we use* TENSORFLOW *or* PYTORCH*?*). As we interviewed experts in the field, we got rather inconsistent opinions.

We thus embarked in a (costly) fact finding mission, consisting of large data collection and analysis, to construct a better vantage point on this shifting panorama. As more and more of the results rolled in, we realized that this knowledge could serve the community at large, and compiled a short summary of the key results in this paper.

The goal of this paper is to create a data analysis-driven bridge between system builders and the vast audience of data scientists and analysts that will be using these systems to build the next one million *ML-infused* applications.[3] To this end, we present the largest analysis of DS projects to date: (a) 6M Python notebooks, publicly shared on GITHUB—representative of OSS, educational, and self-learning activities; (b) 2M DS pipelines professionally authored in ANONSYS within COMPANYX, a planet-scale Web company—representative of the state of DS in a very mature AI/ML ecosystem; and (c) an analysis of over 900 releases of 12 important DS libraries—this captures growth, popularity, and maturity of commonly used tools among practitioners.

The diversity and sheer size of these datasets enable mul-

[1]Microsoft, Redmond, Washington, USA [2]ETH, Zurich, Zurich, Switzerland.

[1]Per arXiV search: https://bit.ly/2lJBoQu.

[2]Per COCOMO software costing model (Boehm et al., 2009) applied to the GITHUB datasets we discuss in this paper.

[3]Per estimates reported in (Agrawal et al., 2019).

tiple dimensions of analysis. In this paper, we focus on extracting insights from dimensions that are most urgent for the development of systems for ML, and for practitioners to interpret adoption and support trends:

- *Landscape* (§3) provides a bird's-eye view on the volume, shape, and authors of DS code—hence, provides the aggregate landscape of DS that systems aim to support.
- *Import analysis* (§4) provides a finer-grained view of this landscape, by analyzing the usage of DS libraries both in isolation and in correlation. As such, it (a) sheds light on the functionality that data scientists rely on and systems for ML need to focus on, and (b) provides a way to prioritize efforts based on the relative usage of libraries.
- *Pipeline analysis* (§5) provides an even finer-grained view by analyzing operators (e.g., learners, transformers) and the shape (e.g., #operators) of well-structured and, often, manageable and optimizable DS code (Schelter et al., 2017; Agrawal et al., 2019).
- *Comparative analysis* (§6 and throughout the paper) is performed, when possible, across time and datasets. In particular, we compare: (a) the evolution of 12 widely used Python libraries for DS across all their releases, investigating their code and API stability; (b) statistics for Python notebooks from GITHUB between 2017 and 2019; and (c) SCIKIT-LEARN pipelines from GITHUB with ANONSYS DS pipelines, studying similarities/differences of public notebooks and mature enterprise DS pipelines.

In addition to reporting objective measures, we feel compelled to provide more subjective and speculative interpretations of the results—we call these Wild Actionable Guesses (WAGs). In order to clearly distinguish between these two classes, we will systematically mark our speculations with the following typography: **WAG:** ***A speculative example***.

As an example, comparing GITHUB in 2017 and in 2019 we observe a 3× increase in the number of DS libraries used. Interestingly, however, the most popular libraries gained even more prominence (+3% for the top-5). Moreover, the top-7 operators in ANONSYS cover 75% of the 2M pipelines we collected from COMPANYX. The wild actionable guess we derive from this is: **WAG:** ***System builders can focus their optimization efforts on a small number of core operations, but need mechanisms to support an increasingly diverse tail of libraries/functionalities.***

Collectively these datasets provide us with a unique vantage point on this important field. We share the key findings in this paper in the form of aggregated statistics and insights we derived. However, to fully empower practitioners along systems and ML audiences at large, we are in the process of releasing the public datasets (raw and processed versions) and our analysis tools. This will enable validation of our process, level the playing field, and enable many more interesting questions to be asked and answered. We plan to maintain a continuous data collection, so that this analysis could become a "living" one, with continuously updated statistics.

Our goal with this work is to make a small contribution to help our community to converge to a common understanding of the "shape" of this field, so that *we can all descend a shared gradient towards a global optimum.*

2 CORPORA, ETL, AND LIMITATIONS

For our analysis we leverage all publicly available notebooks from GITHUB, a large dataset of data science pipelines from within COMPANYX, and OSS packages available from PYPI (PyPI). In this section, we describe each data source, and the ETL processes used to prepare data for analysis. Here, we report on overall data volumes for context. Detailed analysis is deferred to later sections.

GITHUB. In our analysis we use two corpora of publicly available notebooks on GITHUB, referred to as GH17 and GH19. Both consist of notebooks available at the `HEAD` of the `master` branch of all public repositories at the time of download: 1.2M notebooks (270G compressed) in July 2017 for GH17 (Rule et al., 2018; Rule et al.) and 5.1M notebooks (1.3T compressed) in July 2019 for GH19.

ANONSYS Telemetry. Our second dataset is representative of the output of a mature data science community within COMPANYX—a large scale web company. The underlying system, ANONSYS, has been developed and used in production for over a decade. We obtained access to a telemetry database from 2015 to 2019, containing over 88M reporting events. While many users opted-out of reporting telemetry, this large scale sample is representative of DS activities within COMPANYX, and provides an alternative (to GITHUB) vantage point.

PyPI. Our analysis in §3 will reveal that both GH19 and GH17 are dominated by notebooks authored in Python. We thus reuse our parsing/analysis infrastructure to inspect not just DS notebooks but also core libraries they use. We select the most popular libraries in GH19 and few emerging ones to download all metadata and source code for all their releases from PYPI. This corresponds to 12 libraries, totalling 936 releases, 490K files, 2.3M functions, and 266K classes.

ETL. We built an in-house system to support code analytics at the scale described above. While the presentation of this system is beyond the scope of this paper, here we briefly describe the extract-transform-load (ETL) process that transforms the raw data provided from each source to a form that we can base our analysis. We employee a parallel crawler, based on (Rule et al., 2018; Rule et al.). Upon downloading, we parse the nbformat format of each notebook, discard malformed notebooks, and upload the valid ones to a PostgreSQL database. We include all metadata (kernel, language, and nbformat versions), and cell-level

Dataset		GH17	GH19	Change
Notebooks	Total	1.2M	5.1M	4.1×
	Empty	21K (1.6%)	89K (1.7%)	4.2×
	Deduped	816K (66%)	3M (59%)	3.7×
Cells	Total	34.6M	148M	4.3×
	Empty	1.5M (4.3%)	6.4M (4.32%)	4.3×
	Code	22M (64%)	98.4M (67%)	4.4×
	Deduped	9M (26%)	36M (24.3%)	4×
Language	Python	1M (84%)	4.7M (92%)	4.7×
	Other	0.2M (16%)	0.4M (8%)	2×
User	Unique	100K	400K	4×

Table 1. Overall statistics for GITHUB.

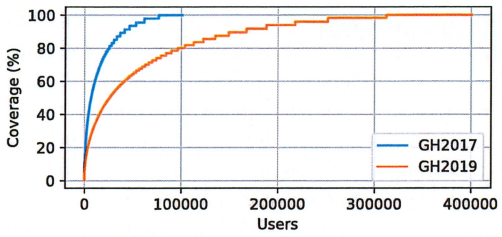


Figure 1. Coverage on #notebooks with varying number of users. Users (x-axis) are ordered in descending order of #authored notebooks. The absence of spikes on the left hand side indicates that no individual user dominates the collection of notebooks.

information (type, order, source code). ANONSYS pipelines are similarly processed, and PYPI libraries are also added to the DB including all metadata and source code. We then perform several extraction passes, where the Python code is parsed and analyzed extracting different features (e.g., which libraries are imported in each code cell)—the features are also stored in the DB. As a result, each statistic in this paper can be achieved by a (more or less complex) SQL query. We are evaluating the feasibility to release the raw DB, the ETL code, and those queries.

Limitations. A comprehensive analysis of all DS activities in the industry is an Herculean task beyond the ambitions of this paper. Here, we take a first stab to construct a representative dataset and some initial measurements. Data availability and parsing/analysis facilities at our disposal pushed us to focus on Python as a programming language, interactive notebooks (instead of all Python programs), and specifically publicly released notebooks. This certainly introduced several biases (e.g., we are more likely to include "learning code" than commercial DS code). To compensate we included the ANONSYS dataset. However here as well we carry deep biases: our ANONSYS dataset is limited by users opt-out choices, and this only captures the DS patterns typical for the use of ANONSYS at COMPANYX, which might differ from other larger companies, and certainly not representative of less mature/smaller players. Finally the sheer volume of data forces us to limit the manual inspections for the interpretation of our results.

3 LANDSCAPE

We start our analysis by providing a birds' eye view of the landscape of data science through coarse-grained statistics from GITHUB notebooks. Tab. 1 presents an analysis of GITHUB notebooks to reveal the volume of (a) notebooks, cells, and code cells; (b) languages associated with them; and (c) their users. We present similar analysis for ANONSYS pipelines in §5 and for PYPI in §6.

We clean the raw data in several steps. First, we remove malformed notebooks. This leaves 1.24M notebooks for GH17 (5.1M for GH19) in our database. We then remove 4.3% empty code cells (those containing only tabs, whitespace, and newlines) and 1.6% empty notebooks (i.e., those made up of only empty cells). Next, we eliminate duplicate notebooks introduced by git forking, and notebook checkpointing[4]. We retain 9M unique code cells for GH17 (36M for GH19). In subsequent sections, we will base our analysis on the full set of notebooks and code cells, and we will report when empty or duplicate code cells and notebooks have an effect on the results.

Overall, we see a roughly 4× growth in most metrics between GH17 and GH19. Code cells growth outpaced notebooks growth (more cells per notebooks in GH19), while duplication was also increased in GH19.

Languages. Looking at languages, we confirm one of our main hypothesis: Python grew its lead from 84% in GH17 to 92% in GH19. All other 100 languages combined grew by roughly 2×, well below Python's 4.7×. **WAG: *Python is emerging as a de-facto standard for DS; systems builders and practitioners alike should focus on Python primarily.***

Users. User growth is similarly paced at 4×, reaching an impressive 400K unique users in GH19. Note that on GITHUB, large organizations may appear as a single "user". Hence, this is a lower bound to the number of contributing individuals. The top 100 most prolific users, in this definition, have over 432 notebooks in GH17 (over 812 in GH19) each, yet no individual user provides more than 0.5% of the notebooks in the corpus. This is best seen in Fig. 1. It shows the coverage of users on notebooks. Note the absence of spikes on the left hand side of the figure. Interestingly, the average number of notebooks per user has remained roughly the same between GH17 and GH19 (i.e., 12.3 and 12.8 notebooks per user, respectively), with the standard deviation slightly increasing (i.e., 54.8 and 59.4, respectively).

Code shape. To get a better understanding of the shape of code in Python notebooks, we parsed (using `parso` (David Halter et al.)) and analyzed every code cell to extract salient structural features. The 3M unique notebooks in GH19 contain a combined 7.68B AST nodes, grouped into 13M functions, 1M classes, 10.7M `if` code blocks, 15.4M `for`

[4]Users often uploads both a base notebooks, and the intermediate checkpoint files generated during editing.

loops, 0.78M `while` loops. Nesting is also present but reasonably rare with 3.3M functions in classes, 0.26M functions in functions, 3.1K classes in classes, and 3.8K classes in functions. Beside overall statistics we are curious to see how many notebooks or cells are *linear*, i.e., contain no conditional statements, or *completely linear*, i.e., contains neither conditionals statements nor classes or functions. At the notebook level for GH19, 28.53% are completely linear while 35.11% are linear. At the cell level, however, 83.04% are completely linear while 87.52% are linear. GH17 has similar characteristics for all metrics reported here.

Overall this analysis validates an interesting line of thinking: **WAG:** ***DS code is a mostly linear orchestration of libraries for data manipulation (+ some UDFs). It is thus amenable to be statically analyzed and transformed into declarative dataflows and optimized as such.***

4 Import Analysis

Recall that our goal with this analysis is twofold: inform practitioners of emerging common practices (e.g., pick a commonly used data preprocessing library) and assist system builders to focus their investments to better support users (e.g., optimize systems for more prominent use cases). To this purpose, after the broad statistical characterization of §3 we turn our attention to which libraries are used more prominently and in which combination in Python notebooks. This implicitly helps us characterize what people do in these notebooks (e.g., visualization, statistical analysis, classical ML, DNNs). We look at this through the lens of import statements (i.e., import... or from...import...).

We begin by discussing the volume of imports and libraries across notebooks (§4.1). Then, we present an analysis of frequencies of imports per library (§4.2), followed by an analysis of statistical correlations (positive and negative) between library imports (§4.3). We conclude the section with a coverage analysis of libraries on notebooks (§4.4).

4.1 Landscape

Tab. 2 reports the total number of imports and the fraction based on from...import... code structures for the GitHub datasets. The growth of imports at 4.85× is outpacing the growth of notebooks and code cells we observed in Tab. 1. Also note that after deduplication, per our discussion in §3, there is a reduction of 27.3% and 31.5% in the number of imports in GH17 and GH19, respectively. Interestingly we observe a large number of unique libraries, over 116K in GH19 (up 2.8× from GH17). *This indicates the field is still substantially expanding.*

4.2 Important Libraries

Having presented the landscape in aggregate terms, we now focus our analysis on identifying what we informally refer to "important" libraries (i.e., libraries that have high import or usage frequencies).

Dataset		GH17	GH19	Change
Imports	total	7M	34M	4.85×
	(from)	2.7M (38.6%)	13M (37.8%)	4.2×
Libraries	unique	41.3K	116.4K	2.8×

Table 2. Import statistics for GitHub.

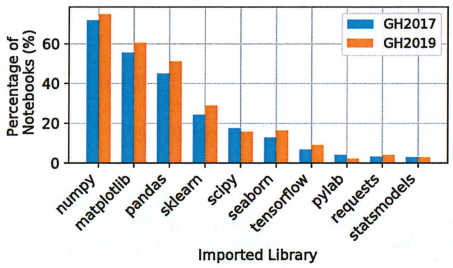


Figure 2. Top-10 used libraries across notebooks.

Most used libraries. As our first metric, we consider the percentage of notebooks that import them at least once. Fig. 2, shows the top-10 libraries in that metric. We make few observations. First we confirm a key assumption of our study: *notebooks are used primarily for DS activities*—the top-10 libraries imported by frequency focus on common data science tasks (i.e., downloading and communicating with external sources, processing data, machine learning modeling, exploring datasets and explaining results through visualizations, and performing scientific computations). Second, we verify our intuition that NumPy, Matplotlib, Pandas, and Scikit-Learn are quite popular (each being imported directly[5]). However, these frequencies exceed our expectations. Third, by comparing the usage between GH17 and GH19, it is really interesting to see that "big" (i.e., most used) libraries are becoming "bigger", while several libraries have lost in popularity by means of relative usage (e.g., Scipy and Pylab, as shown in Fig. 2). This indicate a consolidation around a core of well maintained libraries.

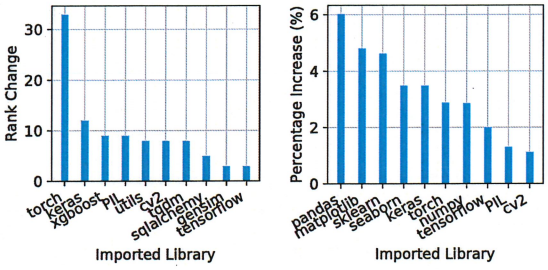
WAG: ***These results suggest that systems builders can focus optimization efforts on few core libraries (such as NumPy), but must also provide pluggable mechanisms to support a growing tail of less frequently used libraries.***

Highest ranking differentials in usage. Comparing GH17 to GH19, the ranking in terms of usage changed dramatically for a few libraries. Fig. 3a shows the top-10 libraries that increased their usage ranking[6] the most over the last two years[7]. Our first observation is that PyTorch increased

[5] More imports could happen indirectly, e.g., Pandas internally uses NumPy, so what we provide is a lower bound of usage.

[6] All ranking differentials are statistically significant per Student's t-test (Student, 1908).

[7] Ranking differentials are computed among the top-100 used



(a) Rank change (b) Percentage change

Figure 3. Top-10 libraries with most increased usage

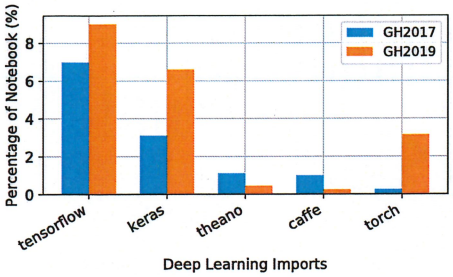


Figure 4. Deep learning libraries

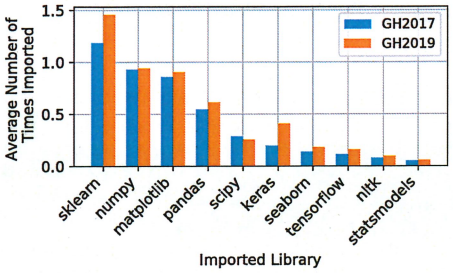


Figure 5. Average number of times to be imported

the most (33 positions in our ranking) followed by KERAS and XGBOOST. These results confirm the overall increased interest in deep learning and gradient boosting. Furthermore, the ranking differentials for PILLOW and OPENCV suggest an increased interest in image processing, while the increased rank of GENSIM indicates an interest for text processing. Finally, the increase of TQDM indicates a growing interest for showing progress bars (which, in turn, indicates long-running computations or file downloading), while the increase for SQLALCHEMY suggests a need to efficiently access data stored in databases.

Most increased in usage. We complement the ranking differential analysis with a percentage increase in absolute terms. The top-10 libraries by highest percentage increase[8] are shown in Fig. 3b. This shows that "big" libraries are getting "bigger" at a faster rate than average. We observe a similar pattern for libraries related to deep learning (e.g., KERAS, PYTORCH, and TENSORFLOW). For this reason, we next dive deeper into a comparison of usage among deep learning libraries.

Comparison of usage among deep learning libraries. Fig. 4 shows the percentage of notebooks that use TENSORFLOW, KERAS, THEANO, CAFFE, and PYTORCH—which were the top-5 used libraries with focus on deep learning in both GH17 and GH19. We make two observations. First, TENSORFLOW, KERAS, and PYTORCH have drastically increased their usage rate (with PYTORCH having the highest increase). Second, for both THEANO and CAFFE the usage rates have dropped considerably. Note that for both observations, the changes in percentages are of statistical significance based on Student's t-test.

WAG: ***Deep Learning is becoming more popular, yet accounts for less than 20% of DS today.***

Most imported libraries and coding patterns. We conclude our analysis on important libraries by ranking libraries based on their import rate and the coding patterns associated with imports. More specifically, the metric we used so far considers usage of a library in a notebook if the library is used in at least one import of the notebook. Another metric, that also helps us reveal coding patterns, is the total number of times a library is involved in imports of a notebook. In this direction, Fig. 5 shows the average number of times a library is imported per notebook. We make two observations by comparing Fig. 5 and Fig. 2. First, there is a change in the overall ranking of libraries: the top-4 libraries (i.e., SCIKIT-LEARN, NUMPY, MATPLOTLIB and PANDAS) getting rearranged, new libraries appearing in the top-10 of Fig. 5 (i.e., KERAS and NLTK), some other libraries changing position (e.g., TENSORFLOW and SEABORN), and others getting out of the initial top-10 (i.e., PYLAB, REQUESTS, and STATSMODELS). The main reason for these results are due to coding patterns of data scientists in importing and using libraries. For instance, SCIKIT-LEARN users may prefer to import its submodules and operators explicitly, using multiple import statements. In contrast, NUMPY users may use its submodules and operators directly in the code (e.g., numpy.*), and import NUMPY just once.

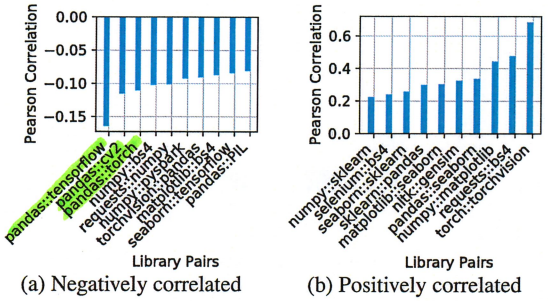
4.3 Correlation

An interesting statistic is the co-occurrence (or correlation) of libraries in practice. This could indicate practitioners the need for expertise in certain combinations of libraries, and for a system builder which libraries to co-optimize. We present positive and negative Pearson correlations among libraries (Pearson, 1896)—we focus on the GH19 dataset for this analysis.

Negative Correlations. Regarding negatively correlated

libraries, as this is more meaningful given the larger baseline.

[8]The increases are statistical significant based on Student's t-test (Student, 1908).



(a) Negatively correlated (b) Positively correlated

Figure 6. Top-10 correlated library pairs

libraries, Fig. 6a projects the top-10 negatively correlated library pairs. We make four main observations. First, PANDAS, a commonly used library for processing of tabular data, is anti-correlated with neural network frameworks (i.e., TENSORFLOW and PYTORCH)—this is due to typically different data types (images vs. tabular) and support for pandas-like functionalities within neural network frameworks. This aligns with our second observation: PANDAS is anti-correlated with image processing frameworks (i.e., OPENCV and PILLOW). Our third observation is similar in nature, as we see TENSORFLOW and SEABORN being anti-correlated, likely because tensorflow carries its own visualization facilities. Finally, BS4, a web-page information extraction library, is anti-corraleated with NUMPY. This hints at a negative correlation between array manipulation and processing of web pages, this is further confirmed by the negative correlation between NUMPY and REQUESTS.

Positive Correlations. Fig. 6b shows the top-10 positively correlated library pairs. This charts provides evidence backing up common wisdom: for instance, REQUESTS and BS4 are expected to be highly correlated (i.e., REQUESTS allows users to download urls whereas BS4 allows users to extract information from web pages). Through the lenses of large corpora, such as the GITHUB ones we use here, we can test this hypothesis and quantify the confidence for accepting it. In the same direction, NLTK and GENSIM are commonly used together for processing text corpora and building language models over them, and their correlation is reflected in Fig. 6b. Furthermore, SCIKIT-LEARN is commonly used together with NUMPY and PANDAS because the input to transformers and learners of SCIKIT-LEARN are typically PANDAS dataframes and NUMPY arrays, and this hypothesis is accepted per the correlation in Fig. 6b. Similar reasons of correlations exist for the other positive correlations in Fig. 6b and we omit further details here.

4.4 Coverage

We conclude this section with an coverage analysis of libraries on notebooks—i.e., if we only were to support K libraries how many notebooks would be supported. We include all libraries used by GH17 and GH19 notebooks in this analysis.

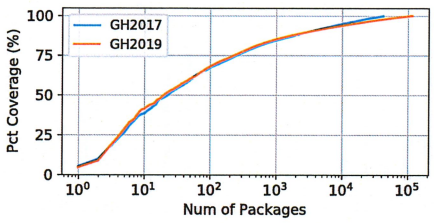


Figure 7. Percentage of notebooks to be covered

Fig. 7 shows the cumulative percentage of notebooks covered (y-axis) while varying the number of libraries (x-axis). As a simple heuristic, we sort libraries from the most to the least used and pick a prefix of size K. Our main observation is that by including just the top-10 most used libraries (i.e., the ones shown in Fig. 2), we can reach a coverage of ~40% in both GH19 and GH17, while a coverage of 75% can be achieved through including the top-100 most used libraries. The increase in coverage, however, is diminishing as less used libraries are added in. More interestingly, a coverage of 100% is much harder to achieve in GH19 than in GH17. This further confirms that the DS field is expanding.

5 PIPELINES

Our analysis so far has focused on understanding data science projects based on the libraries they are using (§4). In this section, we dive deeper into well-structured pipelines (namely, SCIKIT-LEARN and ANONSYS pipelines) to provide an ever finer-grained view of the data science logic, that is also optimizeable and manageable (Agrawal et al., 2019; Schelter et al., 2017). More specifically, we start by providing volume statistics of pipelines in GITHUB and ANONSYS to get a better view of the landscape (§5.1). Then, we focus on the length of pipelines as a measure of their complexity (§5.2). Next, we provide an overview of the number of operators used in pipelines, and we make the case for the need of further functionality (§5.3). Furthermore, we present an analysis of frequencies of learners and transformers in pipelines to point out common practices (§5.4). We conclude this section with a coverage analysis of operator on pipelines to better understand the complexity of supporting pipeline specifications in systems for ML (§5.5).

5.1 Landscape

We start by providing a description of pipelines that we use in the analysis of this section along with their volume.

Description of SCIKIT-LEARN pipelines. To understand the complexity of well-formed pipelines in the GITHUB corpora, we focus on the SCIKIT-LEARN pipelines primarily because (a) of their popularity; (b) multiple systems for ML aim to manage and optimize them (Agrawal et al.,

2019; Schelter et al., 2017); and (c) their specification resembles the one of ANONSYS, enabling us to compare public with enterprise-grade pipelines. Specification-wise, SCIKIT-LEARN provides Pipeline and make_pipeline constructs for generating pipelines. Such pipelines consist of a sequence of transformers (e.g., StandardScaler) and learners (e.g., LogisticRegression). (Intermediate steps of each pipeline are transformers and the last step can be either a learner or a transformer). Hence, their main purpose is to transform input data for either training or inference purposes. Finally, note that pipelines can incorporate sub-pipelines through the use of FeatureUnion and make_union constructs that concatenate the results of multiple pipelines together.

Volume of SCIKIT-LEARN pipelines. We now compare the volume of SCIKIT-LEARN pipelines between GH17 and GH19. From GH17, we managed to extract only 10.5k pipelines. The relatively low frequency (with respect to the number of notebooks using SCIKIT-LEARN discussed in §4) indicates a non-wide adoption of this specification. However, the number of pipelines in the GH19 corpus is 132k pipelines (i.e., an increase of 13× or an average of 181.5 pipelines getting committed daily on GITHUB since 2017). As such, we believe that the "declarative" specification of data science logic, that opens up optimization and management opportunities, is gaining in momentum. We note, however, that an interesting future work in this space is to compare the explicitly specified SCIKIT-LEARN pipelines with the ones specified implicitly (i.e., using SCIKIT-LEARN operations and tightening them together imperatively).

Description of ANONSYS. ANONSYS API is similar to SCIKIT-LEARN: operators are assembled into a data flow graph. Each operator is either a transformer or a learner. ANONSYS is much richer in "data-massaging" operators than SCIKIT-LEARN, as it was developed with the goal of capturing end-to-end pipelines from domain object to domain decision. Users of ANONSYS can author their pipelines in several ways: through a typed imperative language, in Python through bindings, or using ANONSYS's scripting language. For our analysis here we use telemetry data that records the scripting API usage.

Volume of ANONSYS pipelines. Starting from the 88M telemetry events we extracted 29.7M pipelines from 2015 onwards. We found that many pipelines use the exact same set of operators (albeit with different parameters). This is because ANONSYS provides suggested defaults for many DS tasks as well as built-in parameter sweeps. We kept only one copy of these pipelines, which left us with 2M unique pipelines. We analyze those in this section.

5.2 Pipeline Length

As a proxy for pipeline complexity we consider their length. Fig. 8 shows the #pipelines per pipeline length for GH17, GH19, and ANONSYS to drive our discussion.

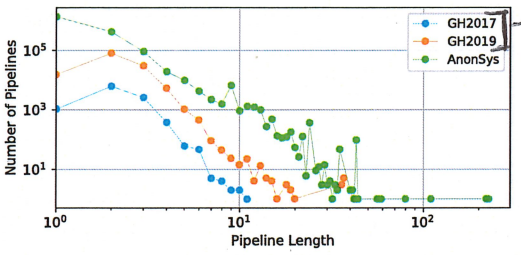


Figure 8. Number of pipelines per pipeline length.

SCIKIT-LEARN. We make two observations for SCIKIT-LEARN pipelines. First, both GH19 and GH17 are right-skewed. Most pipelines have length of 1 to 4. Second, both corpora contain long pipelines. However, the length has increased substantially in GH19. Many pipelines have length above 11 (i.e., max length in GH17), with a max length of 43. Finally, note that in some cases (5% in GH17 and 5.6% in GH19) the length reported here is a lower bound. In these cases, the pipeline is assembled out of multiple sub-pipelines, each held in a variable. Our current analysis treats these sub-pipelines as a single operator. We leave a taint analysis to future work to remedy this.

ANONSYS. Fig. 8 also reports the number of pipelines per pipelines length for ANONSYS. Similarly to the SCIKIT-LEARN pipelines, ANONSYS pipelines are right-skewed. Most pipelines have a length of 1 to 4. In contrast to SCIKIT-LEARN, however, ANONSYS pipelines can get lengthier, reaching a max length of 227.

Comparison. ANONSYS pipelines tend to be longer than those in SCIKIT-LEARN. This can be attributed to the end-to-end nature of ANONSYS pipelines: They contain the full data flow from I/O through transformation to model. In SCIKIT-LEARN, much of the early stages of this are handled by Python code or libraries not captured in Fig. 8. **WAG:** ***The design of* ANONSYS *points to a possible future where DS pipelines are fully declarative.***

5.3 Operators and the need for external functionality

To better understand the complexity of Data Science across time and datasets, we consider the number of (unique) operators in this section.

SCIKIT-LEARN. The pipelines in GH17 and GH19 contain 25.4K and 309.2K operators in aggregate. Most operators in both datasets are specified as function calls inlined in pipelines: 72.5% in GH17 and 75.6% in GH19. Of those function calls, 587 and 3,397 are unique in GH17 and GH19, respectively. This indicates that not only do pipelines get longer from GH17 to GH19, they also get more diverse.

ANONSYS. We identified 110.6M total operators. 536 are unique. Focusing on operators that have a equivalents in SCIKIT-LEARN, we identify 463 unique operators.

Comparison. On first glance, SCIKIT-LEARN pipelines have more unique operators than the ANONSYS ones. However, both ANONSYS and SCIKIT-LEARN and allow for user defined operators (UDO) in their pipelines. However, the two systems go about this differently: In ANONSYS, a UDO is wrapped by a single operator and therefore, all 23k UDOs in our dataset show up as a single operator. In SCIKIT-LEARN, UDOs are introduced as separate operators. This explains the large number of unique operators observed in the GITHUB datasets.

The need for more functionality. The pipelines analyzed here require functionality that is not natively supported by ANONSYS or SCIKIT-LEARN, leading users to introduce UDOs. Some of those UDOs are unavoidable due to the rich set of domains DS is applied to. However, we speculate:

WAG: ***Data Science evolves faster than the systems supporting it. And large corpora of data science workloads, such as the ones we focused on here, can help reveal of "what is missing?" from current libraries.***

5.4 Learners and Transformers

In our discussion above, we focused on operators in aggregate terms. To analyze individual operators, we proceed into ranking them by frequency–which is helpful for prioritizing efforts in developing systems for ML, and exposing common practices among practitioners. We do so by first classifying operators natively supported by SCIKIT-LEARN and ANONSYS to learners and transformers:

SCIKIT-LEARN. Top-5 transformers in GH19 are StandardScaler, CountVectorizer, TfidfTransformer, PolynomialFeatures, TfidfVectorizer (in this order). Same are the results for GH17 with the difference that PCA is 5th instead of TfidfVectorizer. Regarding learners, Top-5 in both GH17 and GH19 are LogisticRegression, MultinomialNB, SVC, LinearRegression, and RandomForestClassifier (in this order). Analyzing the frequency of operators per position reveals that StandardScaler dominates the first position.

ANONSYS. Top transformers are OneHotEncoder, TfidfVectorizer, Imputer, Tokenizer, CountVectorizer (in this order), while the top learners include Gradient Boosting, Random Forest, SDCA, PoissoneRegression, and Averaged-Perceptron (in this order).[9]

Comparison. Normalizers are not within the top-10 for ANONSYS, while dimensionality reduction operators (e.g., PCA) are not even in the top-30. Both are very popular in SCIKIT-LEARN pipelines. These differences are easily attributed: ANONSYS adds these automatically based on the needs of downstream operators. The tree learners in ANONSYS are optimized to deal with sparse, high-dimensional data, which lessens the need for dimensionality reduction.

[9]For anonymity purposes, we report the SCIKIT-LEARN operators that are the closest equivalent to the ones found in ANONSYS.

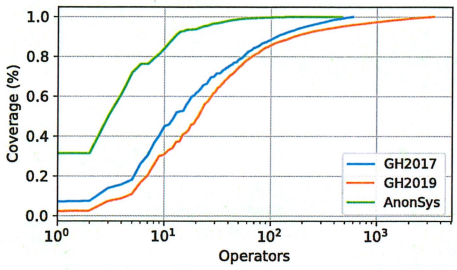


Figure 9. Coverage of operators on pipelines.

Regarding learners, Gradient Boosting and Random Forest are more popular in ANONSYS than SCIKIT-LEARN. We believe this to be due to their relative quality and the tasks observed in COMPANYX.

5.5 Coverage

Here, we focus on the full set of operators of ANONSYS and SCIKIT-LEARN to compute their coverage on the respective set of pipelines. Note that we consider a pipeline covered if all of its operators have been supported. Such analysis is helpful for developers of DS systems to prioritize their efforts.

Fig. 9 shows the coverage for GH17, GH19, and ANONSYS while increasing the number of operators in the descending order of operator frequency. We make three main observations. First, the top-100 operators cover more than 80% of pipelines across all datasets. Second, the same number of operators can cover less of GH19 than GH17. This is indicative of the overall increased complexity of SCIKIT-LEARN pipelines. Finally, the top-10 operators in ANONSYS cover ~80% of all pipelines.

WAG: ***Optimizing only relatively few operators can have a tremendous impact on the overall set of pipelines.***

6 LIBRARIES

We conclude our analysis by taking a closer look at individual Python libraries that data scientists rely on: SCIKIT-LEARN, NUMPY, MATPLOTLIB, PANDAS, SCIPY, SEABORN, THEANO, NOLEARN, KERAS, LASAGNE, NIMBUSML, TENSORFLOW, and MXNET. Our goal with this analysis is to better understand their temporal evolution and whether they have reached a consensus on the functionality they expose (both of which can drive decision making of developers and practitioners). To do so, we first analyze their release frequencies as reflected on PYPI, followed by a more in-depth analysis on their source as provided in PYPI.

6.1 Release Analysis

We start with the analysis of releases by means of release frequencies over time, followed by a break down of releases to release types (i.e., major, minor, and maintenance).

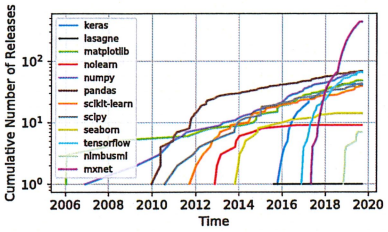


Figure 10. Release frequency of libraries.

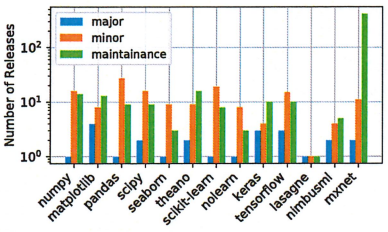


Figure 11. Major, minor, and maintenance #releases per library.

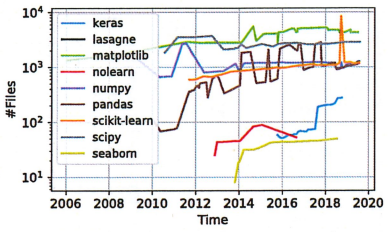


Figure 12. #files per library over time.

Release frequency. Fig. 10 shows the cumulative number of releases per library over time. We make three observations. First, MATPLOTLIB, NUMPY, and PANDAS are constantly being updated. Especially after 2016, releases become more frequent for MATPLOTLIB and NUMPY. Second, we observe a rapid increase in releases of packages such as KERAS, TENSORFLOW, and THEANO. This is likely driven by the current attention to the DNN area. We also observe that both "classical" ML and DNN toolkits see constant releases: SCIKIT-LEARN and TENSORFLOW are good examples of this.

Major, minor, and maintenance releases. The simple number of releases does not reflect the type of changes provided by releases. To do so, we look into the release versions. Following the version scheme defined in (PEP440), for a library release defined as X.Y.Z, we consider major to be X, minor to be X.Y, and maintenance to be X.Y.Z. Fig. 11 shows the number of major, minor, and maintenance releases for each library. We make the following main observations. First, MATPLOTLIB, KERAS, and TENSORFLOW have the largest number of major releases. Second, we observe that many libraries did not have multiple major releases. In particular, NUMPY, PANDAS, SEABORN, SCIKIT-LEARN, and NOLEARN have only one major release. Third, PANDAS, SCIKIT-LEARN, and TENSORFLOW have the largest number of minor releases. Fourth, MXNET has a much larger number of maintenance releases compared to all other libraries.

6.2 Source Analysis

To better understand the code evolution in terms of files, functions, and classes we also explore the source of every release of every library. For this analysis, we do not consider TENSORFLOW, NIMBUSML, and MXNET, as well as a few releases of other packages, because their PYPI package contains native code that we cannot currently analyze. Finally, note that our analysis relies on inspecting the source of releases without installing them.

Files. Fig. 12 shows the number of files per library release over time. We observe that all libraries have gone through one or more phases of rapid increase in number of files. Also, all libraries have gone through one or more phases of source file removals (e.g., due to removal of deprecated functionality and documentation). Lastly, some libraries (e.g., PANDAS) have a big variance in the number of files across releases. This is due to the first two observations, but also because some libraries maintain multiple major/minor releases at the same time. For instance, two minor releases maintained (through multiple and different maintenance releases) in parallel over time may result in high variance in this analysis if the maintenance releases have a big difference in terms of number of files. Finally, although several libraries have started stabilizing around a specific number of files, this should not be interpreted definitively as consensus in exposed functionality. Files in this analysis do not contain only the Python files that exposes functionality that data scientists use in their work, but also documentation and testing files. For this reason, we next dive deeper into the analysis of classes and functions from Python files of each library.

Classes. Fig. 13a shows the distribution of Python classes per library over time. We make three main observations. First, similarly to files, we observe that every library has gone (or is going) through phases of rapid increase, this time in terms of Python classes. Second, we can see that, for most of the libraries, the number of classes is increasing steadily and at significantly different scales. For example, MATPLOTLIB, SCIPY, NUMPY, PANDAS, and SCIKIT-LEARN have a lot more classes than other libraries. Third, in contrast to files, classes tend to experience lower variance in their volume over time. Finally, note that classes in Fig. 13a include both classes used for testing or example purposes, as well as classes that are part of the library APIs.

Classes after filtering. To get a better understanding of classes exposed as part of the API of libraries, we removed classes (a) that appear in files under directories whose name contains one or more of the strings "examples", "test", and "doc"; and (b) whose name either begins with _ (in Python such classes are considered hidden) or contain one or more of the strings "examples", "test", and "doc". Upon removal, the number of classes per library over time is shown in Fig. 13b. Comparing this with the unfiltered counts leads us to two interesting observations. First, the reduction on the number of classes ranges from ~13% and ~31% for KERAS and MATPLOTLIB, respectively, to ~74% and ~79% for

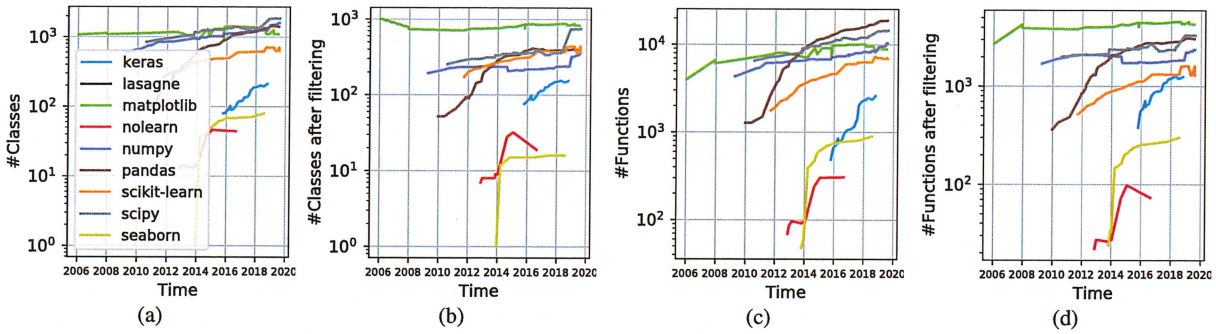


Figure 13. Number of Python classes (a), classes after filtering (b), functions (c), and functions after filtering (d) per library over time. (The legend in (a) serves as the legend for all (a), (b), (c), and (d).)

SEABORN and NUMPY, respectively. These results indicate that these libraries are getting thoroughly tested and provide many examples and documentation of their code. **WAG:** ***We believe that this is a good reason for their high usage in the data science projects***. Second, it becomes more clear that MATPLOTLIB and SCIPY have the most classes; 1,004 and 753 classes, respectively.

Functions. Besides classes, we also performed the same analysis on functions. Fig. 13c shows the distribution of functions for every library over time. We make two main observations. First, the results are similar to the ones on classes, and what we discussed as observations on the growth of classes also apply on functions. Second, however, we observe that the number of functions of every library is larger than the number of classes (typically one order of magnitude more). Finally, it is interesting to see that the relative order of libraries with respect to the number of functions and classes remains similar. Yet, we note that PANDAS and SCIPY have significantly more functions than MATPLOTLIB and NUMPY, which was not the case for classes.

Functions after filtering. Finally, similarly to classes, we performed an analysis by removing hidden functions and functions related to tests, documentation, and examples from the set of functions above. Fig. 13d shows the distribution of Python functions per library over time. After filtering, one of the most important observations is that again there is a big reduction, this time in number of functions. This reduction results in MATPLOTLIB becoming the library with the most functions, while PANDAS and SCIPY now drop to second and third place in this ranking. These results are again indicative of the efforts of making example/test suites and documentation by the respective communities. Furthermore, the (often exponential) increase in the number of functions per library, although it has certainly decreased over time for many libraries, it is by no means indicative of APIs that have reached consensus. **WAG:** ***Hence, we speculate that systems for ML that aim to manage and optimize the functionality of these libraries need to account for potential additions or deprecations over time.***

7 RELATED WORK

Understanding data science processes is crucial as most applications are becoming ML-infused (Agrawal et al., 2019). Our work sheds some light on the topic by performing an analysis of Python notebooks and ANONSYS pipelines. Other approaches (Amershi et al., 2019) include extensive discussions with data scientists about their software engineering practices.

The work in (Bommarito & Bommarito, 2019) presents a coarse-grained analysis on the PyPI repository. Our work targets the Python language as well but with a special focus on data science applications. Thus, we look at a much larger corpus (including PyPI) and provide fine-grained analysis on data science related packages. The work in (Decan et al., 2016) compares the package dependency graphs of CRAN, PyPI and NPM. This work targets various languages and thus does not contain a detailed analysis on Python packages. The study in (Rule et al.) performs an analysis of GitHub notebooks with a focus on the interaction between exploration and explanation of results. Our work incorporates the dataset used for this study (GH17) but focuses more on the structure of data science code rather than the usage of notebooks when explaining the results of the analysis.

8 CONCLUSIONS

Machine Learning is becoming an ubiquitous technology, leading to huge engineering and research investments that are quickly reshaping the field. As builders of ML infrastructure and DS practitioners, we felt the need for a better vantage point on this shifting panorama. We thus amassed a large amount of DS projects (tallest pile to date to the best of our knowledge), and climbed it by means of static analysis, and statistical characterization. From this vantage point, we make several immediate observations and to inform our future work we dare to guess what's happening further in the distance. This analysis, like any other, has several limitations, but we believe is pragmatically very useful. This paper, and future releases of the underlying data and tools,

is an invite to the community to join us in enjoying this view, so that we can have a common understanding on the space we all operate in.

REFERENCES

Agrawal, A., Chatterjee, R., Curino, C., Floratou, A., Gowdal, N., Interlandi, M., Jindal, A., Karanasos, K., Krishnan, S., Kroth, B., Leeka, J., Park, K., Patel, H., Poppe, O., Psallidas, F., Ramakrishnan, R., Roy, A., Saur, K., Sen, R., Weimer, M., Wright, T., and Zhu, Y. Cloudy with high chance of DBMS: A 10-year prediction for Enterprise-Grade ML. 2019. Preprint, https://arxiv.org/abs/1909.00084v1.

Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., and Zimmermann, T. Software engineering for machine learning: A case study. In *ICSE*, 2019.

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. J., and Steece, B. *Software Cost Estimation with COCOMO II*. Prentice Hall Press, 2009.

Bommarito, E. and Bommarito, M. An Empirical Analysis of the Python Package Index (PyPI). *CoRR*, abs/1907.11073, 2019.

David Halter et al. https://parso.readthedocs.io.

Decan, A., Mens, T., and Claes, M. On the Topology of Package Dependency Networks: A Comparison of Three Programming Language Ecosystems. ECSAW '16, pp. 21:1–21:4. ACM, 2016.

Pearson, K. Vii. mathematical contributions to the theory of evolution.iii. regression, heredity, and panmixia. *Philosophical Transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character*, (187):253–318, 1896.

PEP440. Version Identification and Dependency Specification. https://www.python.org/dev/peps/pep-0440/.

PyPI. https://pypi.org/.

Rule, A., Tabard, A., and Hollan, J. D. Data from: Exploration and Explanation in Computational Notebooks, UC San Diego Library Digital Collections. https://doi.org/10.6075/J0JW8C39.

Rule, A., Tabard, A., and Hollan, J. D. Exploration and explanation in computational notebooks. In *CHI*, 2018. ISBN 978-1-4503-5620-6.

Schelter, S., Böse, J.-H., Kirschnick, J., Klein, T., and Seufert, S. Automatically tracking metadata and provenance of machine learning experiments. In *NIPS*, 2017.

Student. The probable error of a mean. *Biometrika*, 1908.