Haychecker: a data quality measurement tool

Jacopo Gobbi
University of Trento
194438
jacopo.gobbi@studenti.unitn.it

Kateryna Konotopska
University of Trento
198234
kateryna.konotopska@studenti.unitn.it

ABSTRACT

Nowadays data is playing a key role not only in leading IT companies, its importance has in fact also spread to virtually all sectors, being a determinant factor in strategic decisions and in the construction of machine learning systems, data is defined by some as "the new oil".

Business processes are becoming more and more bound to data, and thus to its quality, moreover, data is being sold, bought and sought at large volumes, this calls for general purpose quality metrics and for scalable tools to compute those.

We present haychecker, a library and a tool that can be run either in a scalable and distributed way or in a quick and local fashion.

Keywords

data quality, distributed, spark, pandas

1. INTRODUCTION

We see data quality as a multi-dimensional view, considering different metrics that can be tailored to the user needs, by computing them only on specific columns or on a subset of the data.

Our aim is to offer a tool that can be easily and transparently run in both a distributed or centralized way, and a library that has the same API independently of what kind of resources we are going to use, be it a cluster or a laptop.

1.1 Metrics

Data quality *metrics* or data quality *dimensions* are useful tools to have some numbers by quickly and objectively analyzing any given dataset. The data quality metrics can be task-dependent or task-independent, as pointed out in [Pipino et al., 2002]. Task-independent assessments describe properties of the dataset without any knowledge of the domain and the applications of the data, while task-dependent metrics require some more tailoring or input knowledge to be run.

The metrics implemented in the project permit to give an insight in both aspects. Dimensions like *completeness*, *deduplication*, *timeliness* and *freshness* can be useful in analyzing any dataset. For analyzing more particular properties, relationships and column dependencies we added column specific metrics as *entropy*, *mutual information*, *constraint* check and *rule/grouprule* checks.

1.1.1 Completeness

Measures how complete is the dataset by counting which entities are not missing (NaN or Null/None) in a column or in the whole table.

1. column

$$(1 - \frac{|nulls\ in\ the\ column|}{|rows|}) \cdot 100 \tag{1}$$

2. table

$$(1 - \frac{|nulls\ in\ the\ dataset|}{|rows| \cdot |columns|}) \cdot 100 \tag{2}$$

1.1.2 Deduplication

Measures how many values are duplicated within a column/dataset, NaN and Null values are considered duplicated.

1. column

$$(1 - \frac{|duplicated\ values|}{|values|}) \cdot 100 \tag{3}$$

2. table

$$(1 - \frac{|duplicated\ rows|}{|rows|}) \cdot 100 \tag{4}$$

1.1.3 Timeliness

Reflects how up-to-date the dataset is with respect to the given date/time.

1. column

$$(\frac{|more\ recent\ values|}{|rows|}) \cdot 100 \tag{5}$$

1.1.4 Freshness

Measures how fresh is the dataset by taking the average distance of the values from the current date/time in days or seconds for date and time distances, respectively.

1. column

$$current(date/time) - date/time \tag{6}$$

1.1.5 Entropy

Shannon entropy of a column [Shannon, 1948].

1. column X

$$H(X) = -\sum_{x \in X} p(x) log_2 p(x) \tag{7}$$

1.1.6 Mutual information

Mutual information [Cover and Thomas, 2006], usually labeled as MI, is a measure of how dependent two variables are; and indicates to what degree we can obtain information about one variable through the other.

1. column X and Y

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) log \frac{p(x,y)}{p(x)p(y)} \tag{8}$$

1.1.7 Constraint check

After filtering data on conditions, measures the percentage of the remaining tuples that satisfy a given functional dependency, which is a dependency where values of one set of columns imply values of another set of columns.

1. table

$$(1 - \frac{|tuples\ that\ break\ the\ constraint|}{|rows|}) \cdot 100 \tag{9}$$

1.1.8 Rule check

Measures the percentage of tuples satisfying a set of conditions.

1. table

$$\frac{|tuples\ that\ satisfy\ the\ conditions|}{|rows|} \cdot 100 \tag{10}$$

1.1.9 Grouprule check

Measures the percentage of groups passing certain requirements (on aggregations, similarly to HAVING in sql) after filtering the dataset using a set of conditions. It is basically a rule metric, but performed on groups instead of single tuples.

1. table

$$\frac{|groups\ that\ satisfy\ the\ conditions|}{|groups|} \cdot 100 \tag{11}$$

2. RELATED WORK

To implement **haychecker** we used **Spark** (pyspark) for the distributed side, and **pandas** for the centralized side, these are industry proven technologies that have stood the test of time, here we briefly present and describe them, along with their strengths and weaknesses.

2.1 Distributed

Spark is an open source framework made to scale-up general-purpose computation tasks through the distribution of the work-load, while this is usually done on clusters, it can be installed and run on single computers, which allows developing and testing before deploying costly tasks to possibly thousands of computers.

Spark has high level APIs for different languages (python, scala, java, R), and its comprised of different modules, such as Spark SQL, Spark MLlib, Spark GraphX and Spark Streaming, in this project we only made use of Spark SQL.

Spark can run different and complex analytics while accessing many different data sources, such as the **HDFS**, **Apache Cassandra**, tables accessed through **jdbc**, and many others [Databricks, 2018]. For this reason we use Spark to import data even when running in centralized mode, converting it to a pandas dataframe before usage.

While **Hadoop** is more two-stage and disk based, **Spark** is multi-stage and tries to run in-memory as much as possible, this allows it to generally perform better and be a resilient and fault-tolerant solution that is not only able to scale, but also to set records [Xin, 2014].

The base building block of Spark are **resilient distributed datasets**, which are a “distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.” [Zaharia et al., 2012] The logical evolution of RDDs are dataframes: as RDDs, they are distributed collections of data, but have a stronger structure (columns, types) which allows for more optimization and planning when executing queries [Michael Armbrust and Zaharia, 2015].

Spark is lazy, which means computations will actually be executed only when results from the (whole) operation are being collected, which allows for better optimization and resources management.

In haychecker, we avoid RDDs and make exclusively use of dataframes, given **1)** their generally better performance **2)** how they conceptually map to tables in relational databases.

While Spark is an incredibly powerful instrument, it has its drawbacks; given its planning engine, distribution of work-load, and overall complexity, it will naturally incur in an overhead which could be avoided by “blindly” running simpler, eager and more naive frameworks on a single machine, or even single thread; for this reason **haychecker** has also an API which makes use of **pandas** to provide results.

2.2 Centralized

“**Pandas** is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language” [Pandas, 2018], its basic data structure is the dataframe, like Spark dataframes, they conceptually resemble tables of relational databases. Pandas is a single thread, eager on computations framework used for general-purpose data manipulation and analysis on -not so big- data.

As Spark and Spark SQL, it provides an API that allows operations that we would usually see in relational databases, like queries, groupby statements or different aggregations, but it also permits more straightforward computations on columns as if they were arrays, i.e. applying scikit-learn [scikit learn, 2018] functions to specific columns.

While there is not much more to say about it, pandas is a good solution for when the dataset is not too large and tasks are not too complex; it has very serious drawbacks when operations become more resource intensive, it cannot spill to disk if needed, lacks a planner, and its memory consumption is not transparent and quite inflated, as the author explains in [McKinney, 2017], its rule of thumb is “you should have 5 to 10 times as much RAM as the size of your dataset”.

3. HAYCHECKER

Building on pandas and spark, we implemented a tool that can be used both as a library and as a script. The script can either be passed to spark-submit to be run in a distributed way, or provided to the python interpreter to be run using pandas. It accepts (and requires) only one parameter, which is the configuration file (json) specifying the data source, if the schema has to be inferred, which delimiter is used and if

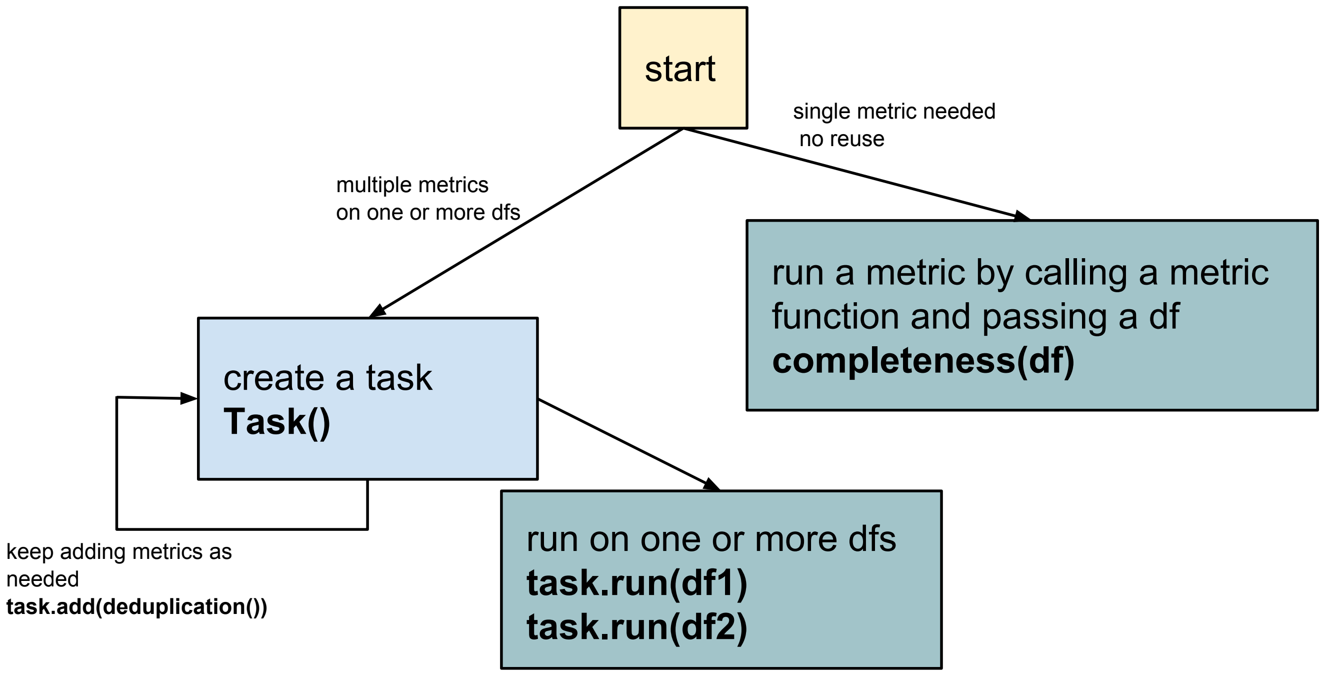

Figure 1: haychecker library workflow

it has an header (for csv files), where to write results (json), and a list of metrics; an example config file is provided in the repository [Gobbi and Konotopska, 2018].

The script imports data from a various number of resources, local files, tables and files in a distributed file system, csv and json files from the web, and tables accessed through jdbc.

The library is based on 2 sub-packages, **dhc** (distributed hay checker) and **chc** (centralized hay checker), those two packages expose the exact same API, allowing for easily switching from centralized to distributed mode during development (i.e. by simply changing the import from haychecker.**dhc**.metrics to haychecker.**chc**.metrics).

Each package is based on a single class, **Task**, which is a container for **metrics** that will/can be computed later or on different dataframes, and a number of functions (metrics) which will either compute and return results when called or return a Task instance, to be later run or expanded by adding more metrics or Tasks to it.

The point of the Task class is to **1)** allow defining things once, but to use those defined metrics multiple times, **2)** trying to optimize/combine metrics that can be run together in a single call, instead of calling them one at a time, when possible.

Haychecker can be found on GitHub at the following link: https://github.com/fruttasecca/hay_checker, can be installed via pip3, and its documentation can be found at https://fruttasecca.github.io/hay_checker/html/index.html (the link to the documentation website is visible from the GitHub repository).

4. EXPERIMENTS

Here we have **1)** benchmark running times, **2)** check on what size of data pandas can be pushed to run to, and **3)** runs of haychecker on two popular open-source datasets. The datasets are: the **Global Terrorism Database** [Consortium, 2017], a worldwide collection of terrorist attacks and their characteristics, ranging from 1970 to 2016, and the **Mushroom Classification** dataset [Learning, 1987], where each tuple describes some properties like odour, shape, color, etc. and if it is poisonous or not.

4.1 Benchmarks

As a modus operandi, we tried to push for as many rows as possible while benchmarking metrics that can compute on single columns, like completeness or deduplication, while using less rows when multiple columns were required; benchmarks have been carried out on a laptop running Ubuntu 16.04, with 8 GB of RAM and a i7-7700HQ CPU, and each metric has been run ten times for each number of rows.

Timeliness and freshness have been run with a date column of type string, so a conversion was required at run time by both modules implementing the metrics. Rule has been run while checking for three conditions (one for each column), while grouprule has been run with three "having" conditions (again, one for each column). Constraint has been run on two "when" columns and two "then" columns. Approximately, columns had 50 unique values.

For simple tasks, and when the data size is limited, the centralized version wins over the distributed, this happens in example for the completeness, deduplication, timeliness and freshness metrics. Once the task becomes more complex or the data grows larger the distributed module performs better, sometimes this is thanks to the optimizer planning a better course of action, sometimes it's because of pandas RAM usage and inability to spill to disk, causing it to fail.

metric	| columns |	| rows |	exec. time (s) distributed	exec. time (s) centralized
completeness	1	30,000	0.10	0.00
	1	300,000	0.10	0.00
	1	3,000,000	0.20	0.01
	1	30,000,000	1.27	0.06
	1	300,000,000	9.10	0.42
deduplication	1	20,000	0.27	0.00
	1	200,000	0.15	0.00
	1	2,000,000	0.15	0.01
	1	20,000,000	0.51	0.08
	1	200,000,000	4.19	0.74
timeliness	1	10,000	0.26	0.00
	1	100,000	0.28	0.02
	1	1,000,000	1.34	0.21
	1	10,000,000	12.02	1.96
	1	100,000,000	120.90	19.40
freshness	1	10,000	0.19	0.02
	1	100,000	0.20	0.08
	1	1,000,000	1.29	0.81
	1	10,000,000	11.92	7.75
	1	100,000,000	121.85	fail
rule	3	10,000	0.26	0.00
	3	100,000	0.19	0.00
	3	1,000,000	0.16	0.04
	3	10,000,000	0.35	0.53
	3	100,000,000	2.64	6.21
grouprule	3	10,000	0.39	0.06
	3	100,000	0.29	0.19
	3	1,000,000	0.47	1.45
	3	10,000,000	1.85	17.69
	3	100,000,000	21.84	fail
constraint	4	10,000	1.60	0.06
	4	100,000	1.75	0.13
	4	1,000,000	2.22	0.54
	4	10,000,000	8.79	5.07
	4	100,000,000	91.98	fail
entropy	1	20,000	0.30	0.00
	1	200,000	0.25	0.01
	1	2,000,000	0.41	0.16
	1	20,000,000	1.78	2.47
	1	200,000,000	18.55	33.13
mutual information	2	10,000	0.70	0.00
	2	100,000	0.59	0.02
	2	1,000,000	0.74	0.34
	2	10,000,000	0.89	5.12
	2	100,000,000	1.75	fail

Table 1: Runtimes of the distributed and centralized modules. Fail indicates the inability to terminate due to excessive RAM usage and consequent swapping, which led to pandas taking an extremely long time to terminate.

4.2 Global Terrorism Database

The Global Terrorism Database (GDT) contains data about terrorist attacks worldwide during the last few decades. It consists of information about date, time, location, specificity, target type, success, attack type and other details.

Table 2 contains some resulting metrics obtained by running haychecker on the dataset. We can observe that the dataset completeness is only **43.6%** , this partiality can be explained by the fact that local governments may or may not divulge such information in a transparent way, especially regimes. As an interesting side note, there seems to be a relation between the type of attack and the target, which is quite intuitive.

4.3 Mushroom Classification

The Mushroom Classification dataset incorporates information about random samples of different species of gilled mushrooms. Each entry is composed by attributes corresponding to mushroom characteristics, such as color, shape,

metric	arguments	score
completeness		43.61
mutual_info	"when": "country", "then": "success"	0.03
	"when": "attacktype1", "then": "success"	0.02
	"when": "attacktype1", "then": "targtype1"	0.63
	"when": "country", "then": "targtype1"	0.02
	"when": "suicide", "then": "success"	0.002
	"when": "gname", "then": "success"	0.022
deduplication (max)	"columns" : ["gname"]	2.01
deduplication (min)	"columns" : ["attacktype1"]	0.01

Table 2: GTD dataset most relevant metric scores.

size, odor, class (edible or poisonous) and others; the dataset was created with the intent to use these simple attributes in order to learn how to distinguish poisonous, edible and not recommended mushrooms. **Table 3** presents some results provided by haychecker, aside from appreciating the completeness of the dataset, we can observe a correlation between cap color and odor, and the existence of a functional constraint between odor (when) and poisonous (then), at least for a subset of the data.

metric	arguments	score
completeness		99.99
mutual_info	"when": "cap-shape", "then": "class"	0.03
	"when": "cap-color", "then": "class"	0.02
	"when": "cap-color", "then": "odor"	0.63
	"when": "cap-color", "then": "veil-color"	0.02
constraint check	"when": ["odor"], "then": ["class"]	56.57
	"when": ["cap-shape"], "then": ["class"]	0.44
deduplication (max)	"columns" : ["gill-color"]	0.15
deduplication (min)	"columns" : ["veil-type"]	0.01

Table 3: Mushroom dataset most relevant metric scores.

5. CONCLUSION

Distribution, planning, and resilience come at price, but allow us to operate on data which centralized solutions will not.

Given that centralized solutions will usually limit themselves to more humble data sizes, a bench-marking contest between distribution and centralization is surely interesting, but not that useful, in our opinion. This is because distributed frameworks can provide the same API for low and big scale operations and are thus more general purpose, while by using centralized ones we will eventually be forced to pick up and learn a new (distributed) framework.

Considering the importance of data quality metrics, we think it is reasonable to expect already established frameworks to implement them, eventually.

As a future work, aside from implementing more metrics, like some from [Pipino et al., 2002], it would be interesting to implement a third sub package based on **Dask** [core developers, 2018], which uses pandas dataframes as a building block, while providing distribution a la Spark.

You can find haychecker at https://github.com/fruttasecca/hay_checker, together with its documentation and a readme which should allow you to have a general idea on how to use the library. You can install it either with pip3 or by using the package contained in the "dist" directory.

6. REFERENCES

[Consortium, 2017] Consortium, S. (2017). Global terrorism database. https://www.kaggle.com/START-UMD/gtd. Last accessed 2018-09-02.

[core developers, 2018] core developers, D. (2018). Dask parallel computing library. http://dask.pydata.org/en/latest/. Last accessed 2018-09-02.

[Cover and Thomas, 2006] Cover, T. M. and Thomas, J. A. (2006). *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, New York, NY, USA.

[Databricks, 2018] Databricks (2018). Databricks data sources. https://docs.databricks.com/spark/latest/data-sources/index.html. Last accessed 2018-09-02.

[Gobbi and Konotopska, 2018] Gobbi, J. and Konotopska, K. (2018). hay checker, a data quality tool. https://github.com/fruttasecca/hay_checker.

[Learning, 1987] Learning, U. M. (1987). Mushroom classification. https://www.kaggle.com/uciml/mushroom-classification/home. Last accessed 2018-09-02.

[McKinney, 2017] McKinney, W. (2017). Apache arrow and the "10 things i hate about pandas". http://wesmckinney.com/blog/apache-arrow-pandas-internals/. Last accessed 2018-09-02.

[Michael Armbrust and Zaharia, 2015] Michael Armbrust, Yin Huai, C. L. R. X. and Zaharia, M. (2015). Apache spark officially sets a new record in large-scale sorting. https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html. Last accessed 2018-09-02.

[Pandas, 2018] Pandas (2018). Python data analysis library. https://pandas.pydata.org/. Last accessed 2018-09-02.

[Pipino et al., 2002] Pipino, L. L., Lee, Y. W., and Wang, R. Y. (2002). Data quality assessment. *Commun. ACM*, 45(4):211–218.

[scikit learn, 2018] scikit learn (2018). scikit-learn machine learning library. http://scikit-learn.org/stable/. Last accessed 2018-09-02.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.

[Xin, 2014] Xin, R. (2014). Apache spark officially sets a new record in large-scale sorting. https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html. Last accessed 2018-09-02.

[Zaharia et al., 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA. USENIX Association.