The Generic and Semantic Profiling of Big Datasets

Ankush Jain
NYU Tandon School of Engineering
New York, NY, USA

Theodore Hadges
NYU Tandon School of Engineering
New York, NY, USA

Ruinan Zhang
NYU Tandon School of Engineering
New York, NY, USA

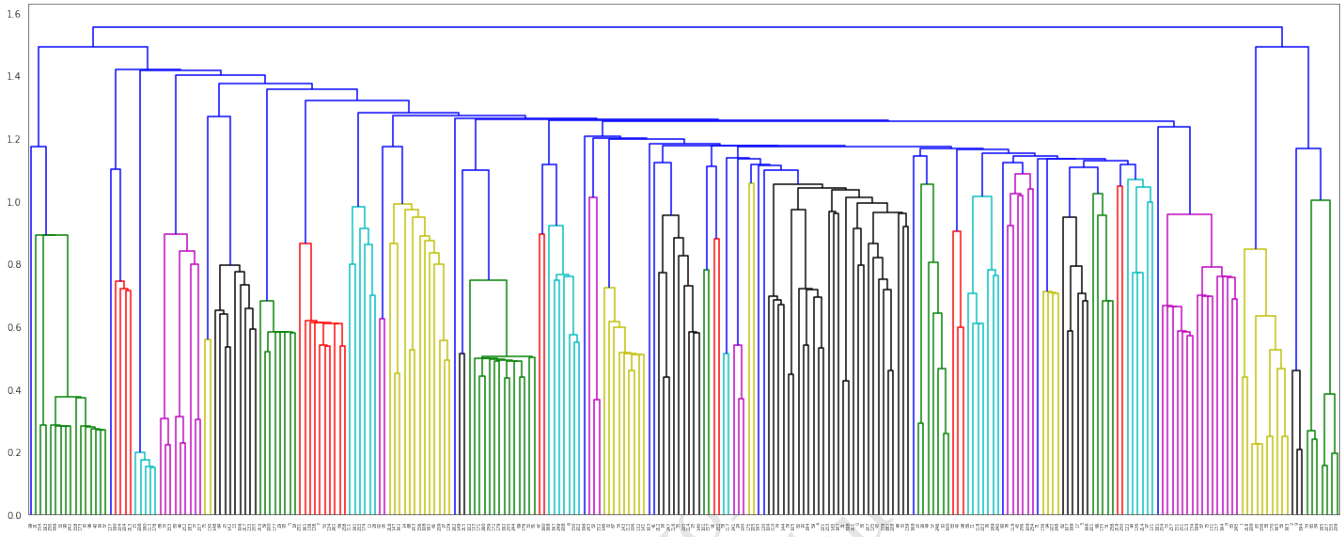

Figure 1: 260 Filenames Clustered Using Cosine Similarity

ABSTRACT

Profiling big data is among the most challenging endeavors a data scientist can take on. Millions of human hours have been spent parsing and cleaning data so that it can fit the user's needs. Recently, designers and researchers have set out to solve (or alleviate) this problem. Notable big data researchers are R. Agrawal and R. Srikant who released their 1994 paper on Frequent Itemsets, a data mining algorithm which changed the landscape of big data. A notable technology is Apache Spark, described by its developers as "an open-source distributed general-purpose cluster-computing framework". In this study, we ran Apache Spark over NYU's 48-node Hadoop cluster, running Cloudera CDH 5.15.0, to generically and semantically profile 1900 datasets from NYC Open Data.

We processed hundreds of datasets for testing and optimization of our code, yet still run into memory issues or errors when we attempt to process all 1900 datasets in one pipeline. Therefore, we present the following quantitative results of a small subset, 150 datasets, with the disclaimer that the full collection consists of 1900 datasets and our results describe only this small sample set. However, the methods we defined are big data methods and can be used for the entire 1900 dataset collection; at the time of writing this, we have processed 1159 of the 1900 sets but present the results of a 150 dataset sample.

The two-step process, generic profling and semantic profiling, can be broken down into two tasks, Task 1 and Task 2, respectively. For Task 1, we sampled 150 of 1900 files. The average number of types for each dataset, across the 150 files is as follows: 10.0 integer types, 11.4 real (double) types, 1.1 data_time types, and 4.7 text types. We found an average of 175460.3 empty values out of all values in the dataset which is approximately 22%. For Task 2, we analyzed 260 columns and we were able to identify the semantic types for 210 columns with a precision of 72.40

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

CCS CONCEPTS

• **Information systems** → **Data cleaning**; *Clustering*; Association rules.

KEYWORDS

data cleaning, datasets, big data, apache spark, profiling, data analytics, parallel programming, hadoop, cloudera, clusters

ACM Reference Format:
Ankush Jain, Theodore Hadges, and Ruinan Zhang. 2018. The Generic and Semantic Profiling of Big Datasets. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The purpose of generic data profiling is to produce metadata about one or more (usually many) datasets to discover general information about those datasets. In our case, for generic profiling, Task 1, we processed 1159 datasets, and present the results of a sample subset of 150 datasets. Our goal was to find the number of non-empty cells, number of empty cells, number of distinct values, the top-5 most frequent value(s), and the different data types (integer, real, string and date-time). For different data types, we performed different statistical operations (min, max, most frequent values and so on). We were also tasked with identifying the columns that are candidates for being keys of that specific table.

The next step in understanding the dataset successfully is to study its underlying semantics. Semantic profiling enables us to gather knowledge about the domain of the data. If we are aware of what the domain of the data source is, we can profile it better and gain useful insights. For semantic profiling, Task 2, we worked with a subset of Task 1's 1900 datasets; this subset consists of 260 columns from NYC Open Data datasets. We and identified and summarized the semantic types of thesecolumns. For different types, we used different methods to identify its types. Regular Expressions and Patterns were used to identify phone number, website, zip code, building codes, locations, addresses, house numbers, school names, parks etc. These types are either very distinct patterns e.g. phone number(ddd-ddd-dddd) or share a lot of similar words e.g. most of the street names have St, Ave and so on. For the types that don't have these features like person names, we took one of two approaches, depending on the type. For categories deemed to contain mostly distinct values but with context, we used Named-Entity Recognition (NER), Soundex and natural language processing to label them. Otherwise, for sets with many repetitive items, we generated Frequent Item Lists against which a given column's row value can be compared. If the item under consideration is in a given set, then the row containing that item is classified as the set's category. All code for this project can be viewed in the 'develop' branch of this repository: https://github.com/theodorehadges/big_data_course_project/tree/develop/.

2 METHODS

2.1 Task 1: Generic Profiling

We imported all data as pyspark dataframes and used inferSchema() to narrow down the possible data types before classifying. If spark classifies a column as int, real, or date_time then every row in that column is of that type. Otherwise, if there is one entry which is of a different type, spark will classify the whole column as a string. Therefore, if it it infers any type other than string, we can use that type as the classification. In most cases, this does not work since many columns are heterogeneous. Our next approach is to iterate through the columns of each file and perform builtin functions or use regex. Our main bottleneck is date_time, since for high accuracy classifications of this type, many regex checks need to be made. We solve this by having three different date_time checks ranging from low accuracy and fast to high accuracy and slow, and apply the most approproate one for a given column given its size.

For task 1, we first iterate over every column for each data set in the NYC open data, and used pyspark.sql.DataFrameReader(spark)'s InferSchema to check its data type:

NEED TO INSERT IMAGE OR CODE HERE This might return ['int', 'bigint', 'tinyint', 'smallint']for Integer type, ['float', 'double' , 'decimal'] for Real type, ['timestamp', 'date', 'time', 'datetime'] for date_time type and ['string'] for Text type. However, since InferSchema will go over the column at once and if even one of the rows is 'string' or invalid value, it will return 'string' as the datatype for the whole column. So, just using InferSchema wasn't sufficient to identify the column types that might be dirty or have multiple data types and invalid values or outliers.

For Integer, Real and Date_Time(timestamp) returned by inferSchema(), we compute the relevant statistics using Spark SQL. An example for this is the below function to compute max and min for integer:

NEED TO INSERT IMAGE OR CODE HERE Next, we especially picked the 'string' type columns returned by the inferSchema and tried to interpret these particular columns as int, real and date_time and strings by performing explicit type casting. If it can be returned as these three types, we return that type's object, if not, we keep it as string and move to the next item for interpretation. This entire process is applied using Spark UDFs so it is distributed. An example for interpreting integer is:

NEED TO INSERT IMAGE OR CODE HERE Specifically, for date_time and string types, we had three different versions of methods to interpret and parse their values since the formats of date and time can be very various and using libraries such as Python's dateutil.parser is not accurate as it can result in false positive date-time interpretation which can cause harm to the authenticity of the datasets.

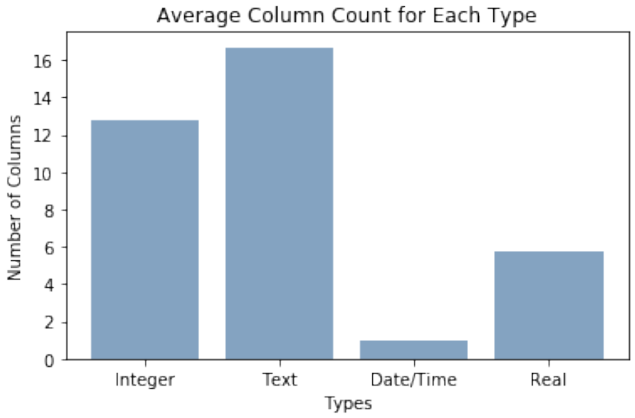

Figure 2: Average Column Count for Each Type

2.2 Task 2: Semantic Profiling

For task 2, our approach can be divided into three parts for finding different semantic types:

(1) Regex for checking types with very distinct patterns; e.g., phone number(ddd-ddd-dddd)

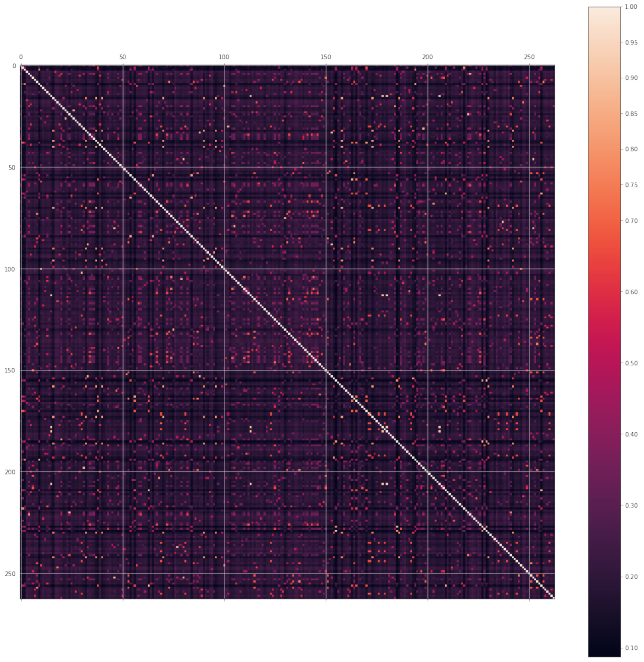

Figure 3: Filename n x n Cosine Similarity Matrix

Table 1: Generic Profiling: Data Quality

Type	Count
Average non-empty cells	917715.4
Average empty cells	212337.6
Average distinct cells	164253.8

(2) Named-entity recognition (NER) and natural language processing for checking types that don't have a distinct pattern or similar words but can be identified with NER tags. The SpaCy implementation 'en_core_web_md' has been used. This technique is used for identifying cities and people's names. The PERSON and GPE tags were used.

(3) Comparing row values against frequent item lists, where those lists contain the most frequent items for each category. If a row value is contained in the frequent item list, then the row is classified as the category from which the frequent item list was derived. This is a useful approach when there are many repeated values in a category.

For each method, we defined different functions to identify different types, and in the function we return the percentage of how many rows were identified as such types, if it's larger than a certain threshold, we return the found_type and total count of rows that identified as such type. An example can be found in the Method (1) Regex section's re_find_street_address function.

2.2.1 Method 1:Regex. The following semantic data types were found using regex:

- Phone number
- website
- zip code
- building code
- lat lon
- street
- address
- school name
- house number
- school subjects
- school level

```
\begin{teaserfigure}
  \includegraphics[width=\textwidth]{similarity.png}
  \caption{figure caption}
  \Description{figure description}
\end{teaserfigure}
```


ACKNOWLEDGMENTS

Thank you to Professor Julia Stoyanovich and Professor Juliana Freire for challenging us with this project and allowing us to discover and experience the things that go along with it.

REFERENCES