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

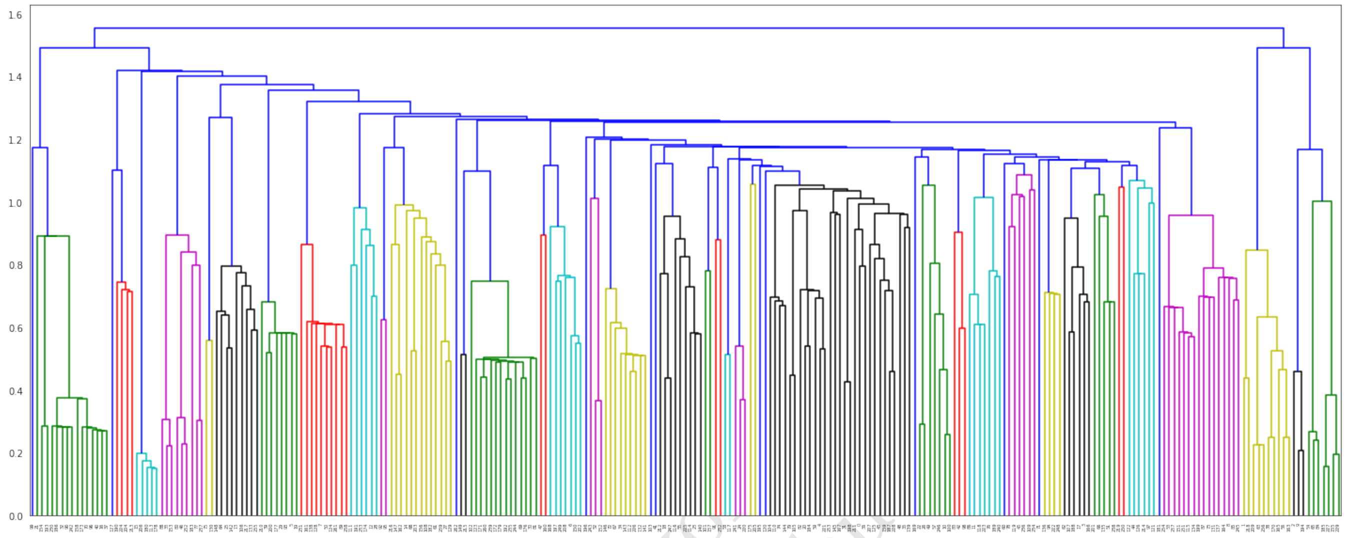

Figure 1: 260 Filenames Clustered Using Cosine Similarity

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 many 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 subset, 1159 datasets, with the disclaimer that the full collection consists of 1900 datasets and our results describe only this subset. However, the methods we defined are designed for big datasets and can be used for the entire 1900 dataset collection.

The two-step process, generic profiling and semantic profiling, can be broken down into two tasks, Task 1 and Task 2, respectively. Of the 1159 files we profiled in Task 1, we found 11674 integer columns, 13646 text columns, 1137 date/time columns, and 4527 real number columns. 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

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

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. 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 these columns. 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. For example, 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 appropriate 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.

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.

Next, we selected 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.

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 vary greatly. Using libraries such as Python's dateutil.parser is not accurate as it can result in false positive date_time interpretation, skewing our understanding of the data.

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)
(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.

2.2.1 Method 1:Regex. The following semantic data types were found using regex. We found this to be a suitable approach for types such as these, which have either an identifiable format (five ints suggests zipcode), or a few key words like HS, Academy, and School for the school_name category.

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

This seemed like a viable approach for this set of categories. However, Street and Address were tricky in that these two types share many similar words such as Road, Street, St, Ave and so on. Therefore, during the execution, they were always labeled together. In observing the data, we noticed that address types usually are longer than street names in length. Therefore, to solve the problem of Street and Address being indistinguishable, we compute the average length for all the columns. We tested and found optimal thresholds: if the string is less than or equal to 15 characters, it's a street; otherwise, it is an address. In this way, we successfully distinguished these two types.

2.2.2 *Method 2:Named Entity Recognition (NER) and Soundex.*

The following semantic data types were found using NER and Soundex:

- city (NER)
- person name (NER)
- car make
- car model
- color

Named Entity Recognition is one of the sub-tasks of information retrieval. It can be used to identify various information regarding the semantics of the text. It can give information about people, places, time, quantities, money, organizations, and more. We employed the Spacy implementation of NER in Task 2 to identify people and cities. The people have been identified using the PEOPLE tag and the cities have been identified using the GPE (Geo Political Entity) tag.

Soundex has been used to identify similar or phonetically similar sounding words. This is a good approach for identifying the color names and the car makers. It helps in converging the misspelled words together.

2.2.3 *Method 3: Similarity and Frequent Item Comparison.*

The following semantic data types were found using Similarity and Frequent Item Comparison:

- business names
- areas of study
- city agency
- location type
- school subject
- parks playgrounds

The pipeline for this method is as follows: First, we need to determine the categories of each file. To achieve this, we decided to group similar files together. In similarity.py, we used file names as a proxy for file content similarity and applied cosine similarity with 3-grams onto the filenames to create a m x m similarity matrix, where m is the list of filenames (Fig 2). In the figure similar files are represented by a bright dot where the higher their similarity, the brighter the dot. Thus, the presence of some bright dots in the similarity matrix motivated us to find out if files could be clustered by file name.

We passed the similarity matrix into the scipy cluster hierchy linkage() function to generate a linkage matrix, Z. Finally, Z was passed into scipy's fcluster() function to form flat clusters from Z. We made a simple python dictionary using the resulting fcluster structure with key value pairs: clusterN: [filenameR, filenameS], where all filenames in clusterN are similar.

In Figure 3, Vertical lines represent filenames. Neighbored lines (color-coded) were mapped to same cluster. The lower portions represent the resulting files which were mapped to the same clusters. As you can see (from their similar heights), many file names were correctly mapped to clusters containing files with similar file names. We wrote this final dictionary to a JSON file.

Next, once the file names were clustered we identified files which were mapped to the same clusters as being the same type. Of course, since this clustering was unsupervised, it was not perfectly clustered. However, we were lucky in that all of the data types we set out to classify for this part were mapped to homogeneous clusters.

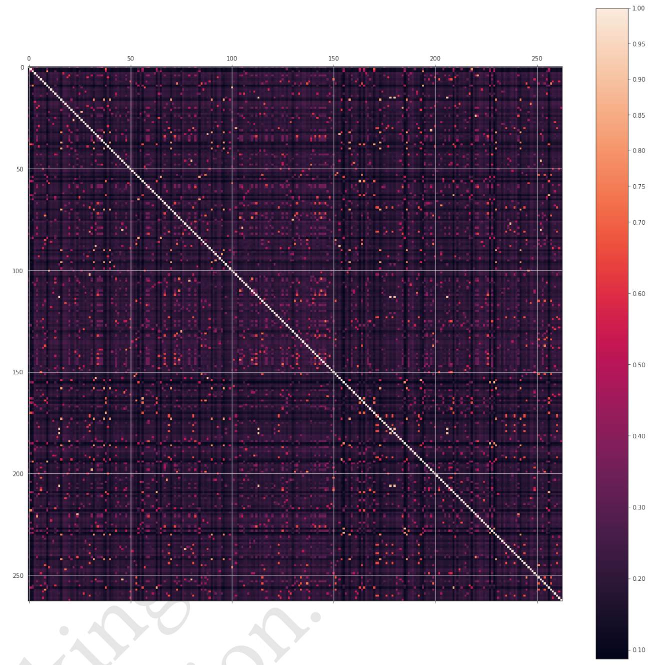

Figure 2: Filename n x n Cosine Similarity Matrix. The presence of some bright (similar) dots in the similarity matrix motivated us to find out if files could be clustered by file name.

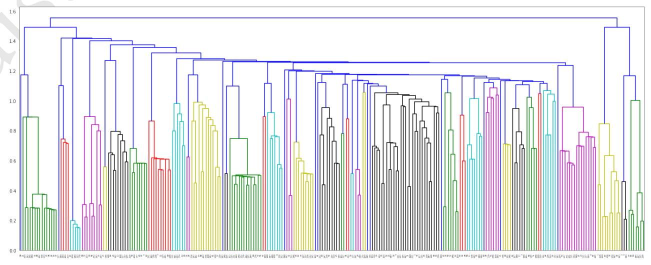

Figure 3: Linkage Matrix. The vertical lines in this dendrogram represent filenames. Neighbored lines were mapped to same cluster.

In generate_keyword_lists_from_columns.py, we use the file clusters to generate the most frequent items in each. Each file in the cluster is loaded into a dataframe. Each row in the data frame is exploded such that each word is now in its own row. This is useful so that we can perform some filtering.

For every file in the cluster, we take the most frequent n words (n depends on category) and union it with the n most frequent values of the next file's most frequent n words. This results in a concise set which is representative of the majority of that category (assuming there are many repeat items in that category). We manually removed some list entries which may be present in another category to avoid ambiguity; however, this could also have been done by finding the intersection of all category sets and removing the resulting values from each set.

Finally, in task2.py, we use these frequent item lists to check if values in a given row contain any of the items in the frequent itemset for each category. If it does, then that row is classified as the type from which the itemset was derived.

3 RESULTS

3.1 Task 1: Results

The first thing to notice is that by looking at Figure 4 and Figure 5 we can see that most data is classified as either integer or text. However, upon performing much more precise and resource intensive mining, it may be possible to extract more Date/Time data from the Text fields. Below, in Table 1 and Table 2, we can see the raw values from which the diagrams were derived.

In Figure 6, we can see that most of the data, about 82%, contained values, while the remaining 18% contained empty data cells.

Table 1: Number of Columns with Identified Types

Type	Count
INTEGER (LONG)	11674
TEXT	13646
DATE/TIME	1137
REAL	4527

Table 2: Number of Cells with Identified Types

Type	Count
INTEGER (LONG)	675048313
TEXT	957412154
DATE/TIME	405729706
REAL	435655955

In Table 3, we can see a summary of the 1159 datasets. One thing to observe is that the empty cells row has a median of 0.0 and a mode of 0. Therefore it must be the case that most of the datasets do not have empty cells.

Finally, we observed candidate key. The results are shown in Table 4. We found datasets with the largest potential and smallest potential candidate key. 587 datasets don't have a candidate key, hence a composite key or indexing is required

Table 4: Candidate Key Counts

Type	Count
Largest	255
Average	2.55
Lowest	0
Number of datasets without keys	587

587 datasets do not have a candidate key; composite/index may be required.

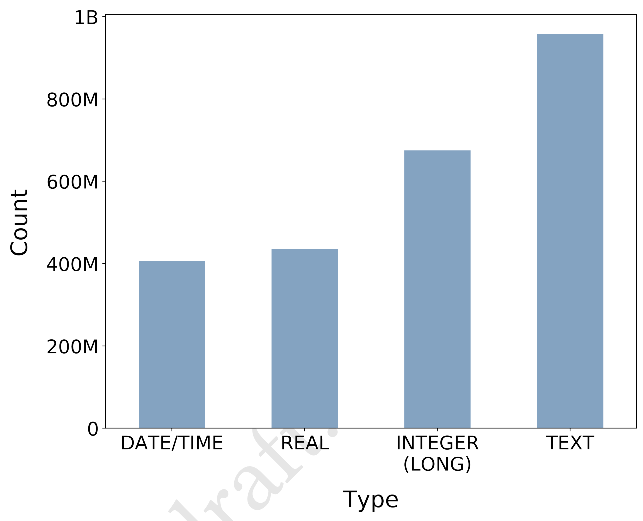

Figure 4: Total Number of Columns for Each Type

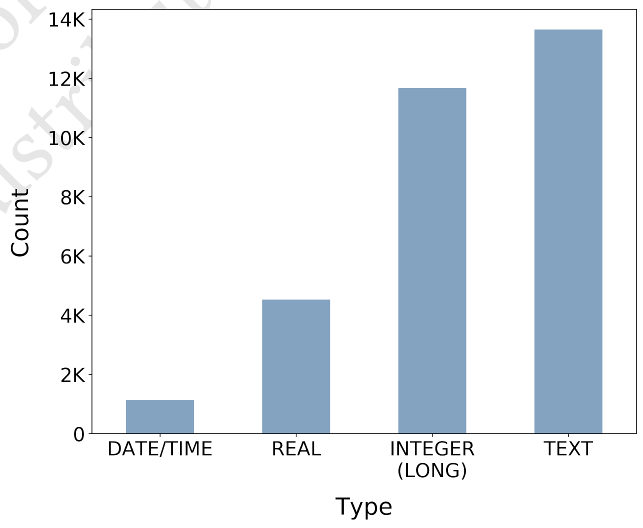

Figure 5: Total Number of Cells for Each Type

3.2 Task 2: Results

The dataset was manually labelled and then the semantic classification of the columns from various datasets were performed. A list of the 23 most suitable labels was created for the datasets, as shown n Figure 7. An average recall of 71.38% was seen and an average precision of 72.40% was observed.

Table 3: Quantitative Summary of Task 1 Results

Type	Mean	std	mincount	maxcount	median	mode
empty cells	20682.11	336203.51	0	16385532	0.0	0
non-empty cells	94945.66	1150276.58	0	50846562	1381.0	0
distinct values	8389.23	201663.02	0	12099654	33.0	2

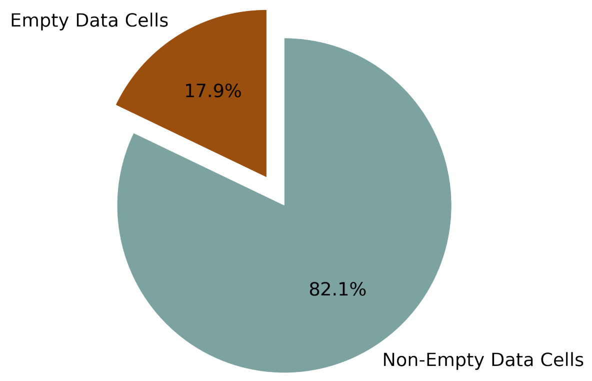

Figure 6: Average Completeness of Data

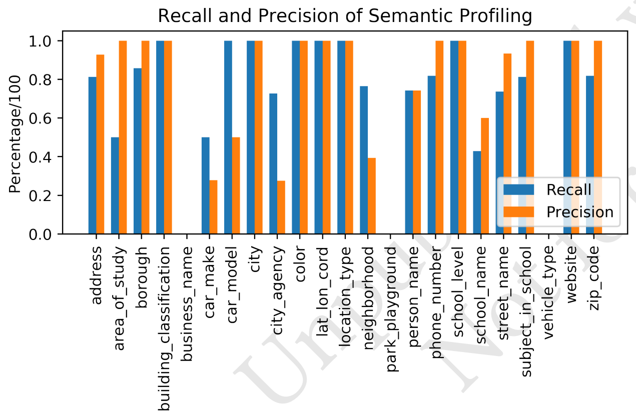

Figure 7: Recall and Precision of Semantic Profiling

4 CHALLENGES AND OPTIMIZATIONS

4.1 Task 1: Challenges and Optimizations

At the very beginning stage of the project, we tried iterating over all the columns of the dataset (row wise). But this resulted in an extremely slow script because of losing the Spark scalability to sequential execution.

To solve this problem, the Spark dataframes and RDDs were strictly used for processing. Further, another problem arose: (1) using inferSchema() has a limit in identifying columns with multiple values (2) date_time type has many different formats, so it's hard to distinguish them from strings , we tried to iterate every cell and check its types using regex. However, after executing a few files, we soon realized this method will be too time consuming as it has to iterate every value and compare it to four different regex lists.

Then, we come up with the idea of trying to interpret values in all the columns which have a return type of 'string' after implementing inferSchema(). However, we faced the third problem (3) using library to parse the date_time might be fast but it's not as accurate as checking with a regex list, but checking regex list will be too time consuming especially for large data-sets. (We have about 6 data sets that's larger than 1 GB). So we came up with three versions of interpreting sting and date_time:

(1) Use Python's dateutil.parser, which is low accuracy but high speed.
(2) Use both Python's dateutil.parser and a single regex check, which is of moderate speed and accuracy.
(3) Use both dateutil.parser a list of more well-defined regex, which is low speed but high accuracy.

The idea behind this is to execute different sizes of data-sets using different methods, for larger data-sets, we switched to the (1) method and for smaller data-sets, we used (2) or (3) methods. In that way, we managed to find a balance between the speed and accuracy which we think is also applicable for other applications and projects.

4.2 Task2: Challenges and Optimizations

4.2.1 NLP and NER. The biggest challenge for NER implementation was the inability to perform NLP in a distributed manner due to infrastructural restrictions of the Dumbo platform. To use NLP toolkits like SpaCy and NLTK in a distributed manner, these libraries must be installed on each of the worker nodes. Doing so with restricted permissions or with custom environment was not possible so a sequential approach had to be taken.

There were some key decision making steps involved in making the best use of Spacy NER. During the initial trials we made use of the available en_core_web_sm 2.0, the semantic parser trained on a small web dataset that was available on the Dumbo Python 3.6.5 module. However, this being an outdated version, we made use of a custom environment with en_core_web_md 2.2 version of the parser which is trained on a much bigger and newer web datasets. There was a significant improvement in the results. With the en_core_web_sm 2.0 the word hits for NER were as low as 10-20%. With en_core_web_md 2.2, this rate increased to as low as about 40% of the words being mapped and as high as 70-80% words getting mapped.

The key decision making step with a sequential approach to NER was to choose the trade-off between precision and execution time. Initially we tried to improve times for sampling. However, sampling is prone to not being representative of the entire data so we decided to discard this approach. Since, we were working on much smaller datasets, we chose to trade-off higher execution times for higher precision by using the sequential processing approach. The best solution to this would have been applying NLP in a distributed manner.

4.2.2 Similarity and Frequent Item Comparison. We start this task by looking for easily classifiable categories such as city agency. We observed the data for city agency and noticed that it was very clean in that it does not contain any extra characters, and it is very similar to the official data on the nyc.gov website. Therefore we built a simple scraper, nyc_agency_scraper.py using BeautifulSoup and parsed the string names of an agency to one file and the abbreviations of that agency to another file. The idea was that we could do a simple contains() to see if a value is contained in this list. We moved on to other types in this section and realized that using sources such as these would be great for general classification, but since we can observe our datasets, there might be a smarter way to choose the list to compare against.

By smarter, we mean a smaller list which yields higher accuracy. We started observing the datasets and skimming through to see items which seemed frequent and made lists for comparison. However, we realized that this is not a scalable solution for building representative itemsets and took a step back. We generated the aforementioned similarity matrix, performed clustering, and built a list generator for frequent items for each category across all files in that category. File name clustering was not perfect at first. After writing a function for jaccard similarity and experimenting with different shingle sizes, we found that 3-grams worked best. However, the clustering was not ideal. We later found a cosine similarity function and were happy to find that the results were much better. After reading about it, we learned that cosine similarity tends to be a better metric than jaccard when measuring similarity between text sets. One challenge we still face is that we would like to use frequent itemsets of size 2 or 3 in some cases (rather than just a frequent itemset of singletons). In the future, we plan on adding this functionality so that we can have better accuracy during the row classification step.

5 INDIVIDUAL CONTRIBUTIONS

5.1 Task 1: Generic Profiling

Ankush Jain:

- Built data pipeline
- String data interpretation
- Code optimization

Theodore Hadges:

- Generic profiling
- Integer profiling
- Real profiling

Ruinan Zhang:

- String profiling
- Date_time profiling
- RegEx-based filtering functions

5.2 Task 2: Semantic Profiling

Ankush Jain:

- Named Entity Recognition classification
- Soundex classification
- Categories:
  - person_name
  - city
  - color
  - car_make
  - borough

Theodore Hadges:

- Cluster analysis
- Frequent item classification
- Categories:
  - area_of_study
  - neighborhood
  - city_agency
  - location_type
  - parks
  - businesses_name

Ruinan Zhang:

- Built data pipeline
- RegEx classification
- Categories:
  - address
  - street
  - website
  - phone
  - lat_lon_coord
  - building_class
  - zip code
  - school_subjects

ACKNOWLEDGMENTS

Thank you to Professor Julia Stoyanovich and Professor Juliana Freire for tasking us with this project. Our freedom to explore and experiment with different methods made for a challenging journey yet we are grateful to have had an opportunity to have such freedom; after all, the benefit of one taking a breadth-first path toward a leaf-node destination is that one learns something new at each node visited.

REFERENCES

[1] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (July 2009), 58 pages. https://doi.org/10.1145/1541880.1541882
[2] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, 226–231. http://dl.acm.org/citation.cfm?id=3001460.3001507
[3] Ming Hua and Jian Pei. 2007. Cleaning Disguised Missing Data: A Heuristic Approach. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '07)*. ACM, New York, NY, USA, 950–958. https://doi.org/10.1145/1281192.1281294
[4] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. 2000. Efficient Algorithms for Mining Outliers from Large Data Sets. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 427–438. https://doi.org/10.1145/342009.335437