CS 6513 Big Data Final Project
NYC Open Data Profiling, Quality, and Analysis

Xiahao Zhang
New York University
xz2456@nyu.edu

Yuchuan Huang
New York University
yh2834@nyu.edu

Wenxuan Ma
New York University
wm1065@nyu.edu

ABSTRACT

In this project, we derived metadata of 1900 datasets from NYC Open Data using Python, Spark, and Pandas. We also extracted more detailed information from 264 selected columns and used different methods to predict their semantic types. In addition, we explored and experimented with several ways to identify and resolve data quality issues.
We analyzed and visualized the results with the help of applications such as Jupyter Notebook.
In order to deal with very large data sets, we also designed multiple optimizations to improve efficiency.

KEYWORDS

data profiling, metadata, semantic profiling, data quality, datasets, Dataframe, HDFS, Spark, Pandas

1 INTRODUCTION

Profiling data to generate and analyze metadata about a given data set is an essential and frequent activity for any professional and researcher[1]. It contains various methods for examining data set information and generating metadata. In Task1 Generic Profiling, we profiled 1900 datasets from NYC Open Data and derived metadata information that can be used for data discovery, querying and identification of data quality issues. In Task2 Semantic Profiling, we extracted semantic types of 264 columns and for each column, we counted how many values belong to each semantic type it has. In Task3, we explored and experimented with methods to identify null values and outliers in datasets.

In Generic Profiling Task, we first extracted the paths of all datasets on HDFS and saved them to a text file. Then we wrote a simple program to check if we can continually submit spark jobs until all datasets have been traversed once. After that, we tried a variety of data structures to read the contents of the dataset, such as the dataframe of pyspark, RDDs, and the dataframe of the pandas library. The dataframe of pandas has many useful build-in methods and is very efficient, significantly faster than PySpark's dataframe and RDDs. Most operations of Task1 can be done by performing the functions of the dataframe itself, without the need to execute complicated SQL queries and transformations of RDDs. To speed up the processing of large data sets, we used a multi-threaded approach. Firstly, we divided the large data sets into multiple subsets, then created multiple processes to process them, and finally merged the results together. In order to optimize the use of memory space, we only read one column of the data set at one time and estimated the time overhead by the number of columns and rows of the data set in advance to compare performance. Finally, we obtained metadata information for all 1900 datasets, and counted frequencies of each data type and generated their frequent itemsets.

In Semantic Profiling Task, we used a variety of methods to predict semantic types of columns, including referencing column names, regular expressions, keywords, frequent values, and iterating learned values. We also improved classification accuracy and recall by combining multiple methods. Some library functions can also help identify the semantic type of values. For example, Stanford's NLTK library can determine whether a string is a person's name. However, such methods usually require extra training time.

In Data Analysis Task, we tried to identify Null values by observation and auto-detection(with the help of other libraries(like Pandas, NumPy)). We also combined frequency-based and distance-base methods to help recognize outlier values.
The code of this project can be found at the following Github repository: https://github.com/wm1065/Big-Data-Final-Project/
The profiling results can be found at the following HDFS directory: /user/yh2834/2019-BigDataResults/

2 GENERIC PROFILING

Since open data often comes with little or no metadata, profiling a large number of open datasets and deriving metadata can be very useful for data discovery, querying, and identifying data quality issues. There are 1900 datasets and for each column in a dataset, we extract the number of non-empty cells, empty cells, distinct values, top frequent values, and data types information. Specifically, we identified 4 data types: integer, real/float, DateTime, and text for each distinct column values. We counted the number of values of each type. For columns that contain at least one value of integer or real, we calculated the maximum value, the minimum value, the mean, and the standard deviation. For columns that contain at least one value of DateTime, we found the maximum DateTime and the minimum DateTime. For columns that contain at least one value of text/string, we extracted the top-5 shortest values and longest values and calculated the average length.

2.1 Architecture

The architecture of Task1 consists of 4 parts:

- datasets partitioning scripts: used commands and script programs to divide the 1900 datasets on HDFS into 4 categories according to size, and store them in 4 different directories. Each directory also has a list of corresponding data sets, which contains the absolute path of each data set on HDFS.
- a script program that traverses multiple datasets: used Shell to write some script programs. For each time, read a row in the list of datasets and submitting a spark job.
- a Python program that generates metadata: for each dataset, extracted various metadata information and saved it in a JSON file. Besides, we used some scripts to adjust the order

of JSON information and merged 1900 JSON files of each data set into a single one.
- a JupyterNotebook: counted the frequencies of data types and the frequent itemsets.

2.2 Data Partition

The first and foremost thing to consider is how to read the data. Our team wrote some shell scripts to iterate the datasets on HDFS. We extracted the path of each dataset as an argument and then passed it to the spark-submit command. At first, most datasets were small and we got the results very quickly. But at some point, we read a very large datasets and we didn't know the size of it. We waited for half a day and the program was interrupted because of a memory error. So we wrote a script that separated the datasets into the following 4 categories:

- small(<50MB)
- medium(50-100MB)
- large(100MB-1GB)
- extra-large(>1GB)

There are 1825 small datasets, 17 mid datasets, 49 large datasets, and 9 extra-large datasets. For each size of datasets, we wrote a shell script to open a .txt file with all datasets paths, and then read each line and submitted a spark job. The output will be saved in a directory. Now the program could run through the whole datasets without considering the time cost and memory usage. We will discuss how to handle the large datasets later.

2.3 Methods

After solving how to traverse the small datasets, we need to consider a method to load the datasets into memory. We used both RDDs and dataframe to load a dataset at first. If using dataframe, we can use SQL query to count the number of non-empty cells, empty cells, and distinct values. For the top frequent values, one intuitive way is to count their frequency and sort them by descending order, and extract the first 5 values.

If using RDDs, it is necessary to segregate the headline (first line) with the data. Header is a set of the column names, so we stored them into an array and used it with a for loop to iterate through the whole dataset by column names. Next, we removed the header line and used different transformations and actions of RDDs to implement the same things as what dataframe did. Using RDDs is a little bit faster than using dataframe, therefore we decided to use RDDs operations to extract more information about data types metadata.

To classify the values into 4 categories: integer, float, DateTime, and string, we performed different methods. For integer and float, we tried to convert the data type to integer or float. For example, if we can convert a value into an integer, then we consider it as an integer. It is also reasonable to judge if the class/type of the value is same with integer or float by "isinstance" method. After discussion we chose the first method. Although we could convert some data like IDs to a big integer, but we can extract more integer values from strings which only consist of digits.

The challenge of this method is that at some time, the semantic type of the value is not numbers. The system can convert data like "2E101" to a float because the format is scientific notation and the value of "2E101" is 2×10^{101} . If a string is consists of only numbers, it will be converted into a long integer. It may also lead the program to failure because the intermediate result can overflow when calculating the average and standard deviation. Even if we use a built-in method, it will return us "infinity". To solve this problem, it is necessary to add some conditions to control. We need to check the column name to determine whether it should be a number or a string, and also check if the value contains "inf" or "infinity" since it can be converted into infinity. Last but not least, in the output JSON files, the infinity or NaN value doesn't have double quotation marks, validating the format of the JSON file.

To identify if a value belongs to DateTime, we consider 2 methods: parsing the string and determine if it contains DateTime information, or using some hard-coded, conventional DateTime formats to validate if a string is a DateTime object. For the first method, it can extract all possible DateTime information from a string and convert it to a DateTime object, but the time cost is a big problem. Imagine the parser uses 100 different DateTime formats to match the data. There would be no difference by identifying DateTime value by hard-coding 100 formats manually. So we chose the second method. It worked very fast but the accuracy was not very good at first. To get more DateTime formats, we wrote a program to select all columns which name implies DateTime. It didn't take us too much time to observe a large number of different DateTime data and summarize their format and add to a DateTime formats list. Nonetheless, some data containing dates or times are classified as strings. But for task1, time is more valuable than accuracy, because there are 60 datasets larger than 100MB and 10 of them are even larger than 1GB.

If a value doesn't belong to integer, float, or DateTime, we identify it as a string. It is faster to generate the metadata of numbers than DateTime, string. For example, if an integer column with 10 million rows takes 30 seconds to calculate the minimum, maximum, mean and standard deviation, but it would take 3 times of that time to calculate something such as the shortest, longest string or the smallest, largest DateTime.

After running through all the small datasets, we moved on to the mid datasets. However, for a mid dataset which contains 2 million rows and 20 columns, it took us 3 hours. It is only larger than 50MB. So we need to figure out another way to generate the metadata of large files.

2.4 Improvements

An improvement for identifying DateTime values is that we use a general format to represent DateTime. To achieve that, we extract the year, month, date, hour, minute, and second information from a sentence and try to match them to a DateTime. If this succeeds, we consider it as a DateTime object and do the same thing for the following data, otherwise skip this judgment and consider it as a string. Something to notice is that to avoid noise data disturbing the result, we need to sample some data and check whether most of them are DateTime information.

To speed up the processing of large datasets, we use dataframe from pandas library and do the same operations. To avoid reading the whole datasets into the memory, we read the columns one by one. Firstly, we only read the header row and calculate how many columns in this dataset. Then when we read the first column, we get the number of rows of the dataset. To get the number of empty cells, we drop all the missing value and the length of the remaining column is the number of non-empty cells, and we use subtraction to get the number of empty cells. Similarly, to get the number of distinct values, we use “unique” function to further reduce the size of the original column and it can save a lot of time on the subsequent operations.

Now, the process of identifying data type splits into 2 parts. In the first part, we try to identify if this column may contain DateTime data because this is the most difficult and complicated part in task 1. After sampling some data, if most of them can provide DateTime information, then we perform a parsing method to rebuild each cell to a DateTime object. In the second part, if this column doesn’t contain DateTime values, we apply different judgment functions on the column and separate the data into 3 parts: integers, float numbers, and strings. Then we use similar logic to generate the metadata for each part. For the medium size dataset mentioned above, now it only takes about 2 minutes to generate the metadata. For the large datasets which contain more than 10 million rows, it takes less than an hour. For the extra-large datasets, we create multiple threads on Dumbo and run multiple tasks at the same time. In addition, we enlarge the memory size of the executors to prevent memory out of usage.

2.5 Results

After analyzing the 1900 data sets, we first adjusted the format of these files and then merged them into a single JSON file. Next, we started to count the frequency of data types in all columns, and we found that integer and text appeared the most frequently. In addition to the characteristics of the datasets itself, the result is also affected by the classification method we chose. Firstly, there are a large number of string-type numbers classified as integers. Secondly, in order to save time and improve the classification speed, we did not choose to parse the time information that may exist in all strings but chose to use pattern matching to determine whether a string is a DateTime object. Therefore, the number of integers and strings is naturally very high.

Because integers and text data appear very frequently, frequent items of data types usually include integers and text data, and the frequency of both is also very high. In the experiment, the minimum support we chose is 0.01 for the total frequency of data types and we got 8 frequent items. In addition, integers and floats nearly always appear together. The most likely reason is that data accuracy is not uniform. Given a column that contains percentages, ’1.0’ will be classified to real but ’1’ will be classified into an integer. Moreover, ’1’ may be an outlier if its frequency is very low, or much lower than the frequency of ’1.0’. There are also some other reasons: for example, in a column with the number of single men and single women, in addition to the integer data that should be, there are actually many floating numbers around 0.9. Even if it represents a percentage, the column name should have a percent sign. Lastly, text data often come together with integer or float because sometimes if we can’t express the data appropriately then we will use a string like "No Data", "s", "unspecified" to represent it.

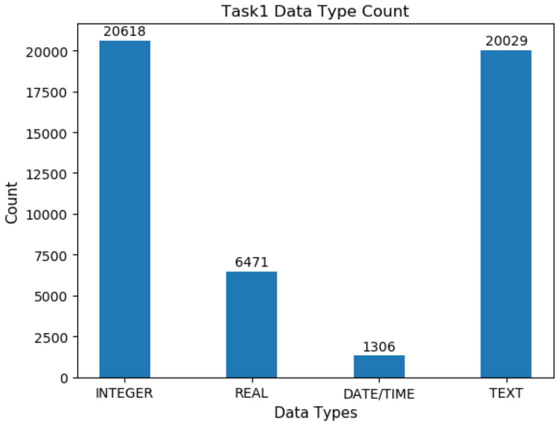
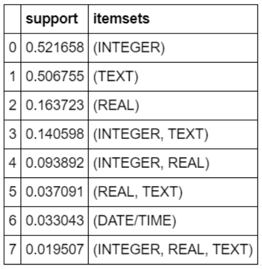


Figure 1: Task1 Data Types Count



	support	itemsets
0	0.521658	(INTEGER)
1	0.506755	(TEXT)
2	0.163723	(REAL)
3	0.140598	(INTEGER, TEXT)
4	0.093892	(INTEGER, REAL)
5	0.037091	(REAL, TEXT)
6	0.033043	(DATE/TIME)
7	0.019507	(INTEGER, REAL, TEXT)

Figure 2: Task1 Frequent items

2.6 Key Discovery

We designed and implemented some simple algorithms and approaches to complete this task.

- Methodologies
  - Naive method: obviously, a column can be a primary key if it has n distinct values (where n is the number of rows in the dataset). Since we have our data profiled in task1(general profiling) before, we can simply use our generated “task1.json” to quickly select the columns that has exact n distinct values, and make any of them to be the primary key. This method is naive but efficient. According to our statistics, over half of the datasets (839, to be exact) has such columns, and therefore we can easily get desired results for a large amount of datasets.
  - On the other hand, if a dataset has duplicate rows, in our method, the key column candidate field would be an empty list, because we must be careful when we handle duplicate rows (the duplication could be intended, thus, we decide to keep these duplication before running our algorithms).
  - Brute Force method: if the number of columns is small, say it is no more than 5, we can easily get all combinations of the columns and try to find if the data can be uniquely identified by those column combinations. Even if the total

number of columns in a dataset is large, we can still try some small combinations.

- Randomization: for the datasets that have more than 10 columns, we just randomly choose some combinations of the columns (using python's own random library), and test if these are good candidates.
- Sampling: another significant improvement of the performance is that we do not run the program based on the whole dataset(this is inefficient in both time and memory). For large datasets, instead of loading the whole table, we only load the first few thousands of rows, and apply the algorithms above.
- Pruning: note that for some columns A and B, if the distinct valuse count of A times the distinct values count of B is less than distinct rows in the table, then the combination of A and B together can't not be our candidate.

- Implementation and Results
  The final implementation is a combination of all the methods mentioned above, and the code is written separately from other parts of the task1 code, because we need the result from task1.json as mentioned in method1. Please see the github for the code and relevant part in task1.json for the results.

3 SEMANTIC PROFILING

In the second task, we extracted semantic metadata from 264 different columns. We used a variety of methods, including regular expressions, some foreign libraries that can identify person names, parse addresses, and company names, and also reference column names to aid classification. We counted how many types of semantic data appear in each column, calculated the frequency, and output predicted semantic type labels. Finally, we compared the prediction results with the manual labeling results to calculate the accuracy and recall.

3.1 Methodologies

The methodologies of this task have mainly 4 directions: Columns' name, regular expression, keywords and iteration of learned values. As well as a combination of 2 or more of them.

Intuitively, classifying semantics by columns' name is a convenient and reasonable way. And from our test and observation, the columns' name has a strong correlation with the semantic type of values in the columns. Some may be obvious, like person name, color. Some may not be that obvious and require some translation, like dba of business name. We can either use them as strong suggestions or just as a reference and combining it with other classification methods.

Regular expression is exceptionally suitable for those values having strict rules for their format. In our project, we've identified 4 of them which are phone numbers, Lat/Lon Coordinates, zip codes, and websites. Take phone numbers as an example. They have strictly 9 digits. They may be combined together using parentheses or dashes like (xxx)xxx-xxxx or xxx-xxx-xxxx. They may also be simple 9-digit numbers without any other symbols. By considering all these cases, we construct a simple way to correctly detect them.

Classification by key words helps us save a huge amount of time from iterating all possible values. The first step is running word count to extract frequent values of the target semantic type. By observing the result as well as the original data, we have found some important words that may decide what semantic category the whole string belongs to. For example, a business name may contain 'corp', 'llc', 'inc', etc. A park name may contain 'playground', 'square', 'garden', etc.

Iteration is an inefficient and awkward way to do semantic recognition. But sometimes it is necessary. Especially when it contains domain knowledge or there are some existing lists including all of them. While they don't have any regulated format or fixed keywords (or they themselves are the keywords), we cannot simply recognize them using the above methods. To construct the list of known values, we can either extract all distinct values from columns with that semantic label (we identify them by simlarity) or take them from some official source. Sometimes tokenization, converting lowercase and edit-distance will help to remove unwanted match, which will be further discussed in later section. Semantics like colors, boroughs, building classification codes are all requiring this kind of method.

For about a combination of the above methods, the address is a good example. By observation, we found that an address always has a digit at the front most, followed by a street name. We have keywords to classify street names, so we combined a pattern recognition method with keywords detecting function to decide whether it is officially recognized as an address. The result is good and the efficiency is high.

Besides all the methods listed above, another tradeoff we made is deciding the sequence of semantic checks. We will place those with higher confidence to earlier stages, and the ambiguous ones later. For example, we can decide whether a value is a subject with little false positive or false negative, so we can place it at the front most. And if any value is recognized as that semantic, it will be removed from the list and thus avoid to be classified again to other wrong categories. To gain higher confidence, the earlier a semantic type judged, the more strict it should fit its given rules.

3.2 Challenges/Limitations

Ambiguity is the largest limitation of semantic classification. Mostly, ambiguity appears because a value duplicates through multiple labels(For example, Brooklyn can be both 'borough' and 'city'). There are many cases that may lead to such Ambiguity.

First, the shorter value is, the higher the possibility it may appear in multiple labels. A classic example is 'B', it may appear in colors, representing 'Black'. It may also appear in borough representing 'Brooklyn' or 'Bronx'. It still can show up in a person's middle name. To deal with such cases, we have set up a soft rule to filter out those values that are too short when performing classification. They will not be counted unless they lie in a column with a reasonable column name.

Second, some given labels have ambiguity themselves. Without proper domain knowledge of official regulation, people cannot even clearly distinguish them manually. City and borough is one of such cases. Borough like 'Brooklyn', 'Manhattan' also always appear in column of city and are still among the most frequent values. And some values naturally appears in multiple labels due to similarity of the labels themselves in semantic. It's not strange when a value appears in both a street name and a park name. To solve this, we decided to place the more precise label to the front. Like borough will be evaluated earlier than city and removed if match.

Third, outliers will lead to cross-label duplication. Some values like 'others', '-', 'N/A' are leading to such duplication. We used strategies in first case. When encounter with outliers, we will turn to column names. If we first identify the outliers, and then get some information from column name, if they do not match, we will ignore these outlier values.

In addition to ambiguity, when creating our 'knowledge list', we need to be careful with string matching. For every type that requires our 'known value iteration' method, a simple string matching does not work fine. Sometimes a tested value just contains a proportion of the corresponding value in knowledge list, which may still belong to that category. Therefore, we need fuzzy match. Checking string contain relationship is one way. But we still have the case that a value is just part of a word. It can still be regarded as part of the knowledge string, but that's not what we want. So tokenization is a another solution. And for those which need harsh evaluation standard, we use edit-distance and adjust the threshold to approach desired results.

3.3 Results

We present the results of Task2 in the following Figures:

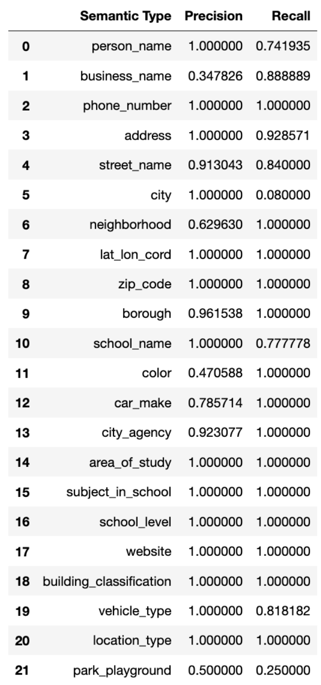
Figure 3: First Precision and Recall Result Form
Figure 4: Best Precision and Recall Form
Figure 5: Best Precision and Recall - Graph
Figure 6: Number of Columns by Types
Figure 7: Heterogeneous Columns - Manually Labeled
Figure 8: Heterogeneous Columns - Predicted

These figures can also be reproduced in Jupyter Notebook file task2_notebook.ipynb.

4 DATA QUALITY

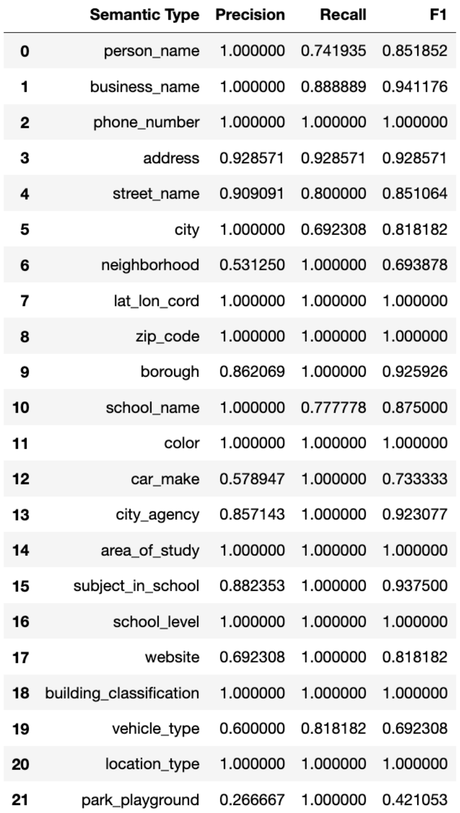
In Data Analysis Task, we tried to identify Null values and recognize outlier values.

To recoginze Null values, we used methods that combine observation by eyes and detection by programs. We observe common values such as N/A, unknown, not applicable, etc. With the help programs and results of profiling, we can discover Null values in those frequently repeated short values('*', '-', 's', etc.) Libraries like Pandas and Numpy can also auto-detect some Null values. To identify outliers, we used intuitive ways such as type check, length check and range check. Other than this, we mainly use two methods frequency-based method and distance-base method. With the metadata generated in Task1, we can identify frequent values of a dataset and its frequency, we then do type check, length check and range check to conclude if it is an outlier. For distance-base method, by carefully choose p and D, we can considered a value an outlier if at least p fraction of the values in a column lies greater than distance D from the value. There are also other simple but effective methods. For example, for data which satisfy normal distribution, we consider values that are 3σ (or 4σ) an outlier.



	Semantic Type	Precision	Recall
0	person_name	1.000000	0.741935
1	business_name	0.347826	0.888889
2	phone_number	1.000000	1.000000
3	address	1.000000	0.928571
4	street_name	0.913043	0.840000
5	city	1.000000	0.080000
6	neighborhood	0.629630	1.000000
7	lat_lon_cord	1.000000	1.000000
8	zip_code	1.000000	1.000000
9	borough	0.961538	1.000000
10	school_name	1.000000	0.777778
11	color	0.470588	1.000000
12	car_make	0.785714	1.000000
13	city_agency	0.923077	1.000000
14	area_of_study	1.000000	1.000000
15	subject_in_school	1.000000	1.000000
16	school_level	1.000000	1.000000
17	website	1.000000	1.000000
18	building_classification	1.000000	1.000000
19	vehicle_type	1.000000	0.818182
20	location_type	1.000000	1.000000
21	park_playground	0.500000	0.250000

Figure 3: Task2 First Precision and Recall Result Form



	Semantic Type	Precision	Recall	F1
0	person_name	1.000000	0.741935	0.851852
1	business_name	1.000000	0.888889	0.941176
2	phone_number	1.000000	1.000000	1.000000
3	address	0.928571	0.928571	0.928571
4	street_name	0.909091	0.800000	0.851064
5	city	1.000000	0.692308	0.818182
6	neighborhood	0.531250	1.000000	0.693878
7	lat_lon_cord	1.000000	1.000000	1.000000
8	zip_code	1.000000	1.000000	1.000000
9	borough	0.862069	1.000000	0.925926
10	school_name	1.000000	0.777778	0.875000
11	color	1.000000	1.000000	1.000000
12	car_make	0.578947	1.000000	0.733333
13	city_agency	0.857143	1.000000	0.923077
14	area_of_study	1.000000	1.000000	1.000000
15	subject_in_school	0.882353	1.000000	0.937500
16	school_level	1.000000	1.000000	1.000000
17	website	0.692308	1.000000	0.818182
18	building_classification	1.000000	1.000000	1.000000
19	vehicle_type	0.600000	0.818182	0.692308
20	location_type	1.000000	1.000000	1.000000
21	park_playground	0.266667	1.000000	0.421053

Figure 4: Task2 Best Precision and Recall Form

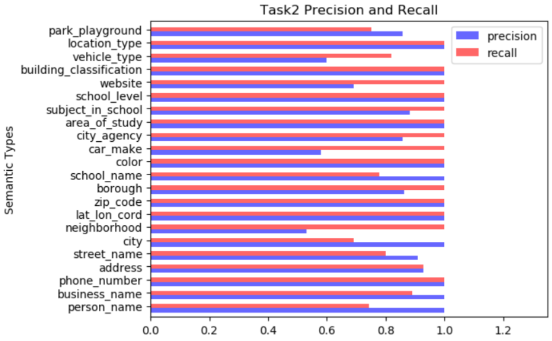


Figure 5: Task2 Precision and Recall

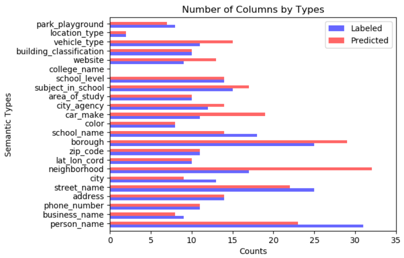


Figure 6: Task2 Number of Columns by Types

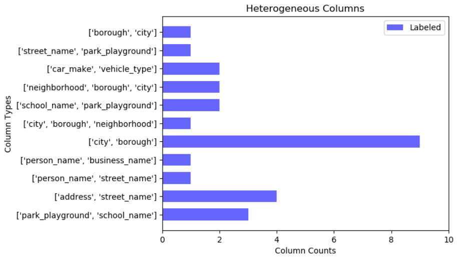


Figure 7: Task2 Heterogeneous Columns - Manually Labeled

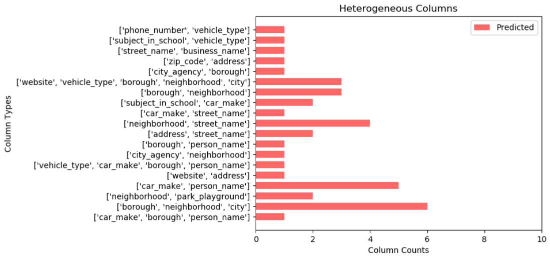


Figure 8: Task2 Heterogeneous Columns - Predicted

by observation and auto-detection(with the help of other libraries(like Pandas, NumPy)). We also combined frequency-based and distance-base methods

5 INDIVIDUAL CONTRIBUTIONS

Xiahao Zhang:

- Wrote Task1 program framework
- Used pyspark's RDDs and dataframe to extract metadata except for data types
- Wrote some scripts to format the output and combine small JSON files into a single one
- Visualized the results of Task1 and Task2

Yuchuan Huang:

- Extracted data type metadata of Task1
- Labeled the data of Task2 and preprocessed the training data
- Designed methodologies and implemented extracting semantic types

Wenxuan Ma:

- Wrote pandas code of Task1 which significant improvement of the efficiency
- Designed and implemented algorithm to discover keys in task1
- Help preprocess the training data in Task2
- Visualized some results of Task2
- Designed and implemented programs Task3

6 SUMMARY

In summary, data profiling is a complicated issue, and working on it needs tremendous amount of work, as well as creativity.
Through this project, we learned a variety of tools, technologies and methodologies to deal with big data, such as Hadoop, Spark, Pandas and Linux shell programming, etc.
We enjoyed the process of discovering and exploring new tools, programming languages, libraries and algorithms in this project. And we are certain that the experience we got can help us with lots of relevant topics in the future.

REFERENCES

[1] Abedjan, Ziawasch, Lukasz Golab, and Felix Naumann. *"Profiling Relational Data: A Survey."*. The VLDB Journal 24.4 (2015): 557–581.

[2] Knorr and Ng *"Algorithms for Mining Distance-Based Outliers in Large Datasets"*. VLDB 1998

[3] W. McKinney *"Pandas: a Foundational Python Library for Data Analysis and Statistics."*. scipy2010.

[4] Rajaraman, A., Leskovec, J., Ullman, J. *"Mining of massive datasets"*. 2012