NYC Open Data Analysis & Profiling

Zili Xie
Computer Science
New York University
New York, NY, US
zx979@nyu.edu

Zihan Li
Computer Science
New York University
New York, NY, US
zl2795@nyu.edu

Longtao Lyu
Computer Science
New York University
New York, NY, US
ll3056@nyu.edu

ABSTRACT

This project is consisted of two parts. The first part is the generic profiling of the 1900 datasets in NYC Open Data, where we will analyze the composition of each data column and obtain metadata such including numbers of empty and non-empty cells, frequently occurred values, and data types in each column. In the second part, more sematic analysis will be involved. We will try to interpret the meaning of each column.

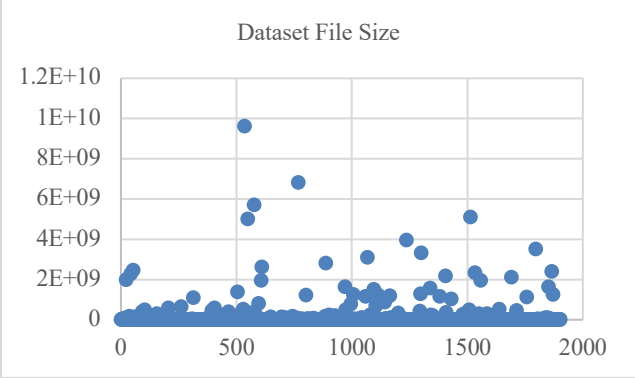
1. GENERIC PROFILING

1.1 A Brief Introduction to NYC Open Data:

The NYC Open Data is a free public dataset that contains information collected and maintained by the New York city government. An important goal of this open project is to extract valuable knowledge from massive data and formulate good policies that benefits all New Yorkers. NYC Open Data provides comprehensive data that spans a large range of city operations, including education, traffic conditions, business, buildings, social services, cultural affairs and more.

1.2 Dataset Files

The data that we are dealing with consists of over 1900 small datasets, where all of the files in dataset are in *.tsv* file format. The plot below shows how file size varies in the Open Data. As the scatter diagram indicates, the largest file from dataset is about 1×10^{10} bytes large, while most of the files are far smaller than that, being less that 5×10^{8} bytes.



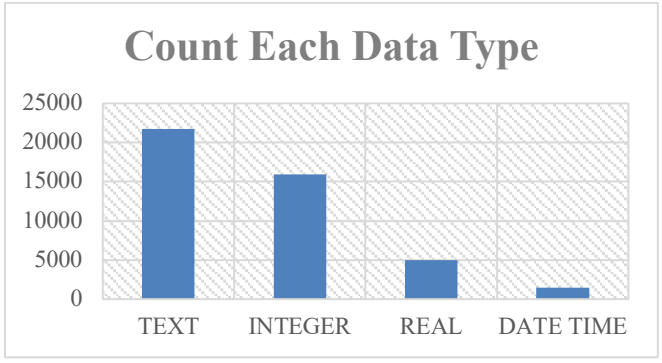
1.3 Goals

For each column in dataset, extract the metadata that consists of
1. Number of non-empty cells
2. Number of empty cells
3. Number of distinct values
4. Top 5 frequent values
5. Data types (INT, REAL, DATE/TIME, TEXT) exist in column

For INT and REAL data type, we will also provide max/min value, mean and standard deviation for each column. For DATE TIME, we will output the max/min of time, and For TEXT, average text length and max/min text length will be calculated.

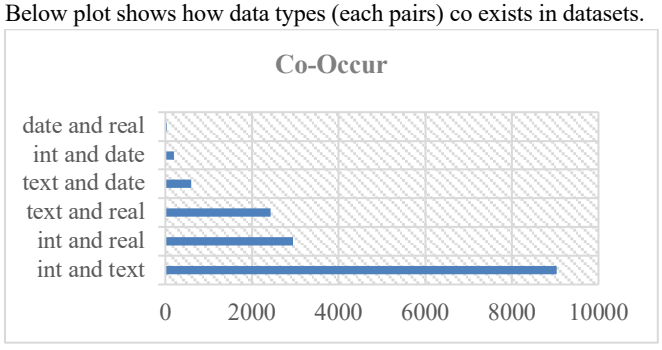
1.4 Result Summary

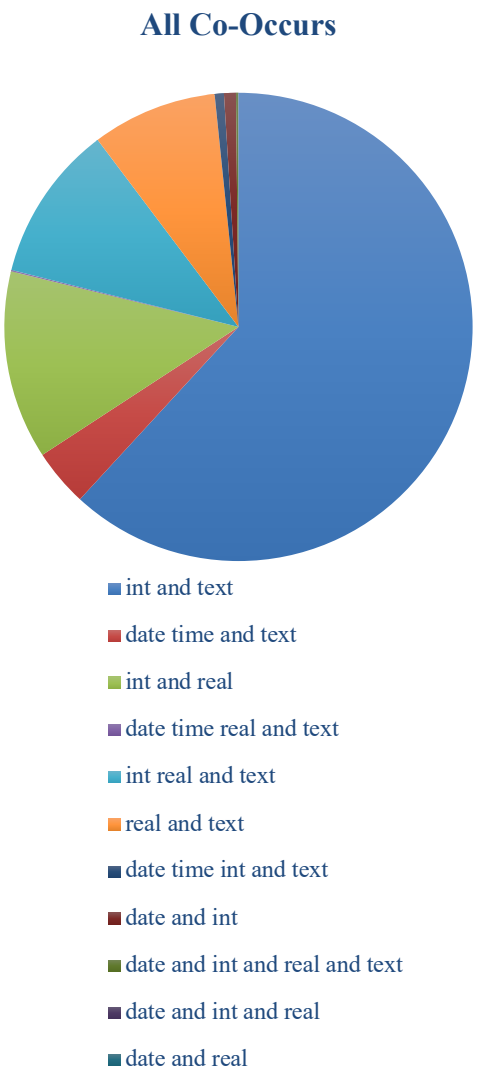
By the time when this report is composed, 1660/1900 datasets have been profiled. The histogram below shows for each data type, the number of columns that contains it.



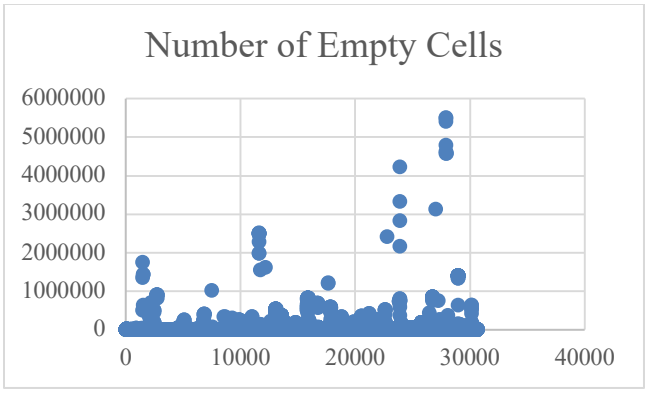
As expected, TEXT and INTEGER types are more frequently occurred than REAL and DATE TIME.

Below plot shows how data types (each pairs) co exists in datasets.





The two chart both describe the fact that INTEGER type and TEXT type are commonly co-exist. It exists in more than 60% of all the heterogeneous columns.



1.5 Data Quality Issues

(1) Mix of record and summary:

A very representative example is that some dataset includes the total value of a column in last line. It doesn't make the table wrong, but it introduces error if we want to compute the average value or standard deviation in this column. If the data is massive and it is hard to check whether a table has a row for the total value, then this can be problematic.

88QLC1 - QUEENS	10	8	8
88RLC1 - BROOKLY	5	5	5
88XLC1 - BRONX A	9	8	8
Grand Total	3890	2682.5	2556

(2) "Line Breaks and Tabs":

Analysis of the column names or headers is usually the first step of generic profiling dataset, since it takes up the first line of most datasets. So, the number of columns is actually fixed by the length of header. The start of a row of record is led by a newline character and every two fields (columns) are separated by a tab. Consider the case when the record value itself contains tabs or newline characters, then it will interfere the way we read the data.

It's a little hard to present the problem, but in the screenshot below (2zbg-i8fx) a cell that contains a newline character was recognized as two different lines, and will be therefore read as two different records. (the second field should contain location info which combines address and coordinate, but the line break makes the coordinate part become the value in the first column)

```
5-03191-0061  "1160 RICHMOND ROAD Staten Island, NY
(40.606674000105, -74.162401000006)"  5 10304 40.598436 -74.091972
(40.605386778303, -74.076177344182)"  ROSEBANK  D1-ELEVATOR 136 1966
```

1.4 Challenges & Problems

(1) Use of Data Structure

Our first attempt on this project is to read each dataset into a dataframe and then try to extract all metadata using pyspark's SQL. The positive things about this approach is that the syntax of SQL is familiar to all of us, and SQL is a more "structed" language which make a lot of convenience for coding.

However, our input data is not "structed". Some dataset contain massive empty values, some columns has multiple types and some erroneous characters exist in some values can be problematic for SQL syntax.

Our wanted output is also not "structed". It is hard to put results like Top-5 values and all Data Types of a column into one result table. An intuitive answer to this would be writing multiple queries, but that will cause a lot of duplicate filtering and projection which will be time consuming.

Considering all the problem above, RDD is a better choice of data structure over dataframe in this project for the following reason:

a. RDD is more flexible. We can easily create customized helper function for filtering or type casting in RDD, instead of creating complex UDF in dataframe.

b. RDD benefits a lot from Python module when identifying data type (more on this in the Correctness and Stability section)

(2) Efficiency & Optimization:

Quite unexpectedly, efficiency problem is the problem that we spend most of the time.

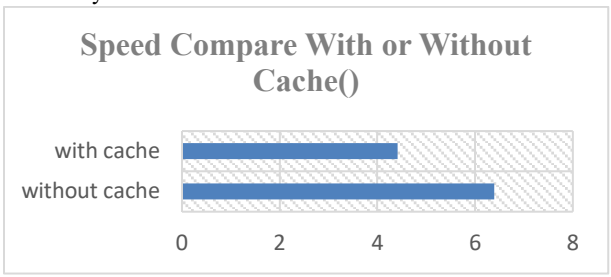
1. Caching the intermedia result

Why is caching so important? Consider the case when we want to count how many integers are there in a column and then find out the maximum integer. First, we read the dataset into an RDD, split each line by tabs, project the column we are looking for, filter all the records that are **not** INTEGER type, then finally we have an RDD that has all records that can be identified as INTEGER (let's call it *int_rdd*). The last step will be calling the *count()* action on *int_rdd*. This process can give the number we want and it consists of several transformations and one action.

Now, when we want to continue and find out the maximum value, it should be fast, because we already have an RDD of all integers, we can just call *max()* on int_rdd, but it is not! It will take approximately the same amount of time as the previous *count()*. This can slow down the processing speed a lot! especially, when the data amount is huge.

But why is that happening? The reason is that the RDDs we used in program are not the actually data loaded in memory (in other word, they are not performed at all). They are just data with queued up transformations. When actions like *count()* or *max()* are called, it will force all the transformations in the queue to perform, and after the action returns, all changes are discarded. So, in our case, when the *max()* is called, all the transformation happened before head are performed again. That is why *cache()* so important, since it stores transformed data in RDD to memory, and save time when the RDD will be reuse many times.

Below diagram is a speed test between with *cache()* and without *cache()* on the first file in our dataset: sm48-t3s6.tsv.gz. Its file size is 2715 bytes.



2. Transformation vs Action

There are two most commonly used operation in RDD: Transformation and Action. Even though, sometimes it might be more straight forward to use Action other than Transformation because Action can produce objects in python (either an actural number or a list), while Transformation only produces another RDD. It is still more preferable to use more Transformation because Tranformations in RDD are much faster than Actions.

The problem we encountered in this project is that we try to call *collect()* on some RDDs and try to manipulate the data in a more "Python" way, which make the program very slow.

The following fragment of code is from our old program.

```
data_types = rdd_type_value.map(lambda x:x[0]).distinct().collect()
if any(keyword in data_types for keyword in ['INTEGER']):
    rdd_value = rdd_type_value.filter(lambda x:x[0] == 'INTEGER')
    rdd_value = rdd_value.map(lambda x:x[1])
    count_int = rdd_value.count()
if any(keyword in data_types for keyword in ['REAL']):
    rdd_value = rdd_type_value.filter(lambda x:x[0] == 'REAL')
    rdd_value = rdd_value.map(lambda x:x[1])
    count_int = rdd_value.count()
```

We tried to call collect() and do condition judgement in each iteration, which is straight forward but not efficient enough, because actions in RDD is slow. Instead, we use try:… catch:… (try catch could also hurt performance but in this case it doesn't slow thing down and it fix the speed problem here).

```
try:
    rdd_value_int = translated_value.filter(lambda x: x[0] == 'INTEGER')
    rdd_value_int = rdd_value_int.map(lambda x:x[1]).cache()
except:
    pass
try:
    rdd_value_real = translated_value.filter(lambda x: x[0] == 'REAL')
    rdd_value_real = rdd_value_real.map(lambda x:x[1]).cache()
except:
    pass
```

We also avoid duplicated computations (especially duplicated actions)

```
min_int = int_col_data.min()
max_int = int_col_data.max()
count_int = int_col_data.count()
s = int_col_data.sum()

mean_int = s/count_int
s2 = int_col_data.map(lambda x: pow(x - mean_int, 2)).sum()
sd_int = math.sqrt(s2/count_int)
```

Take the computation of standard deviation as example, the formula of standard deviation is the square root of mean square error.

```
int_col_data.stddev()
```

RDD has its own function to compute standard deviation. We don't use it because the count() action should happened internally, which we have already computed in our program, remember that actions in RDD are slow.

(3) Identifying data type

There are multiple ways that we can choose to identify the data types exist in each column. The strategy can regular expression, keyword matching, column name inspection or other methods. But unfortunately, there is not a specific method that can do the identification for all types and all columns from datasets in NYC Open Data. So, our method of identifying data type is to use the cast function built in Python. We can get 3 benefits from this strategy:

1. Data cleaning happens when detecting data types. An obvious example is DATE TIME whose formats can be very different. "2019/12/10" and "2019-12-10 17:00:00" can both represent DATE TIME in data records. Our

method uses the *dateutil.parser* in python, and this problem can be fixed in an elegant way.

2. After the values in column have been casted, it will be a lot easier to compare values marked in same type. Use REAL type as an example, regex or keyword matching will only filter the values in string format that do not meet the expression or requirements, but it can be still difficult to find out the maximum or minimum value.
3. Lastly, this method can be integrated in try catch clauses. If the an occur errors when program is casting a value from RDD to be INT, it will not collapse and stop, instead, the program now knows that the value probably cannot be interpreted as INT type, and it will just give up on INT and keep trying on other types such as REAL or DATE TIME.

(4) Avoid misclassification
Some values can be interpreted as multiple types. "2016" can be either an INTEGER or a DATE TIME. It can also be TEXT in some cases, but how can we determine what type is the value? The answer is that we should have different orders of checking given a column name. If the column name for value "2016" contains word like "year", "date", etc. then it should be more likely to be DATE TIME instead of INT or TEXT. Similarly, if a column is named as "num", "amount", then we should not check DATE TIME first, because the dateutil.parser is super powerful, and there are chances that these numbers be cast to some crazy DATE TIME (B.C or year 3000).

2. SEMATIC PROFILING

2.1 Goals

For each column in dataset our program will classify the values in column as one or more sematic types from person name, business name, phone number, address, street name, city, neighbor, coordinate, zipcode, borough, school name, color, agency, study area, car make, subject, school level, website, building code, vehicle type, locations and parks.

2.2 Our Strategies

(1) External Resources: external resources are extremely useful for identifying sematic types. Just as an example, all existing names of cities, neighbors, boroughs and even towns, hamlets in New York state can be used as keywords for identifying district. Also, there are some specific code that are defined by collector of NYC Open Data. These specific code can represent different types of vehicles, or different types of color, or type of buildings. The code and its corresponding description are available on the website.

(2) Keywords: Common knowledges can also help identify sematic types. Small business can be restaurants, hotel, food market…(Though there are a lot of such types, it is still countable) Restaurant names could contain words like "restaurant", "ristorante"(french), "bistro", "kitchen"…
Any names that contain foods such as "chicken", "beef", "ice cream" have great chances of becoming business name, considering that no person would like to put these word in their names or name a school using theses words.
There are a lot of names that are cultural related: "Dragon", "Sushi", "Greek", "Mexico"… These keywords are also likely to be a part of business name.

(3) Cleaning:
Now assume that we have enough keywords, the following question is: How do we use these keywords? The data we are dealing with are noisy, un-uniformed and has a lot of typos.

BK 1	BELLEROSE 28
BKLN 5	BELLEVILLE 17
BKLYLN 3	BELLMAWR 1
BKLYN 754	BELLMORE 14

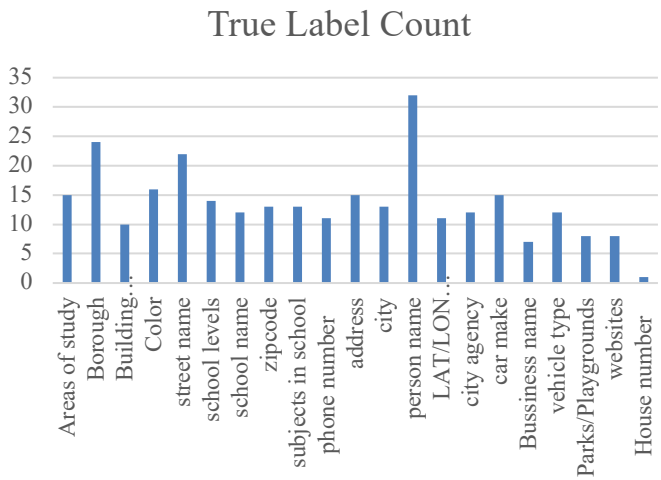
A Column name called EMPCITY can have values that are Borough name (abbreviation), hamlet, village, or even a country. How can we utilize the keyword to classify the column of these data? Some methods can be applied:

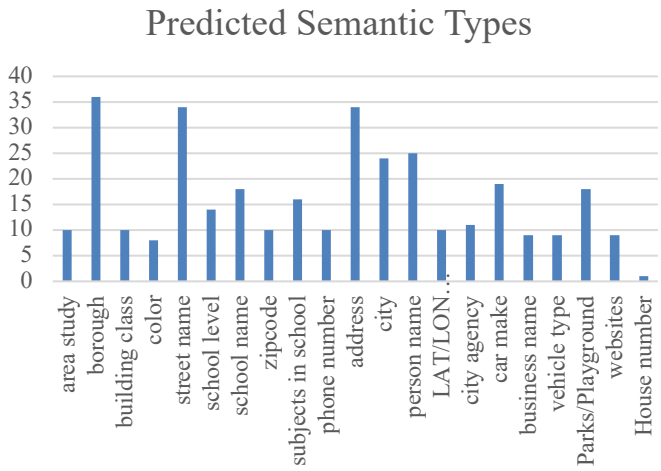
1. filter the noisy data. "Wrong Answers can be many, but there is only one that is correct." We can filter out the data that does not has big impact on the overall result. (such as outliers that only occur once or twice when the total data amount is massive.)
2. Matching Criteria. For different sematic types, our program uses different matching criteria.

- To match data column with business names we can check if a specific keyword is **in** the data.
- For data column such as coordinates, phone numbers, websites. Our program uses regular expressions.
- For sematic type like city name, neighbor names, boroughs levenshtein distance can produce accurate results.
- The original length of keyword and the length of data word to be match is critical in this process. If we find a good match for a keyword (the levenshtein distance is less than 4, but the keyword itself only has 2 or 3 characters), then this match can be unreliable. But consider the case when our keyword is long, even when the leveshtein distance is long, the matching can be good because this might be cause by long prefix or suffix in the word. "New York City (Uptown West)" should be a match to "New York"

2.3 Result

The result of our sematic profiling can be shown in the graphs below. As you may notice, the general frequency in each types is similar to our labels marked by hands.





There are some problems with the predicted semantic types. Firstly, prediction tends to interpret the values in a column to multiple types. Sometimes value can ambiguous. "14 West Street" can be either a street name or an address, and this can lead to multiple semantic meanings when profiling the column data while the true label is marked as single semantic meaning.

The Precision and recall statistics are presented below:

	precision	recall
Areas of study	1	0.666667
Borough	0.4	0.683333
Building	1	1
Color	1	0.8
Street name	0.617647059	0.954545
school_levels	1	1
school_name	0.555555556	0.833333
zipcode	1	0.769231
subjects	0.75	0.923077
phone number	0.9	0.818182
address	0.352941176	0.8
city	0.391304348	0.692308
city agency	0.818181818	0.75
LAT/LON Coordinates	1	1
person name	1	0.78125
car make	0.578947368	0.733333
Bussiness name	0.555555556	0.714286
vehicle type	1	0.75
Parks/Playgrounds	0.277777778	0.625
websites	0.888888889	1

GITHUB link: https://github.com/zilixie/big-data-project-submission

HDFS: /user/zx979/project-zl2795-zx979-ll3056/

REFERENCES

[1] List of cities in New York (state). (2019, December 6). Retrieved from https://en.wikipedia.org/wiki/List_of_cities_in_New_York_(state)
[2]Boroughs of New York City. (2019, November 14). Retrieved from https://en.wikipedia.org/wiki/Boroughs_of_New_York_City.
[3] Boroughs of New York City. (2019, November 14). Retrieved from https://en.wikipedia.org/wiki/Boroughs_of_New_York_City.
[4]https://data.ny.gov/Transportation/Vehicle-Makes-and-Body-Types-Most-Popular-in-New-Y/3pxy-wy2i