The Biggest Data:

an exploratory overview of Big Data principles
and techniques in profiling and semantic labeling

Alex Hong
Computer Science and Engineering
New York University
ah4927@nyu.edu

Vincent Nikolayev
Computer Science and Engineering
New York University
vn500@nyu.edu

Harman Singh Chawla
Computer Science and Engineering
New York University
hsc367@nyu.edu

ABSTRACT

Our report focuses on highlighting the fundamental skills intrinsic to profiling large amounts of open access data. Throughout Task 1, we focused on optimizing our code and devising efficient designs that maximized speed without sacrificing functionality. We were able to implement a relatively fast reader that relies on a generator and obtained significant processing speedups by implementing MapReduce for parallel processing in Spark. Task 2 was more challenging and required different set of strategies to generate our desired output. Instead of focusing on efficiency, we opted for greater precision and recall as we deemed these factors to be the more important metric in semantic labeling. Lastly, we prepared an analysis of 311 complaints in an attempt to explain which factors if any led to demonstrable changes in complaint distribution over time. These findings led us down interesting avenues and produced some thought-provoking discussions. We supplement these tasks with visualizations where appropriate and justify decisions and outcomes when explanation is warranted.

CCS CONCEPTS

• Information systems~Extraction, transformation and loading • Information systems~Data cleaning • Information systems~Data mining

ACM Reference format:

Alex Hong, Vincent Nikolayev and Harman S Chawla. 2019. The Biggest Data: an exploratory overview of Big Data principles and techniques in profiling and semantic labeling.

Introduction

Throughout the project, our group collaborated intimately through every step of the assignment. We were often required to make deliberate design and implementation choices depending on the aims and purpose of each task. In many ways, this gave us crucial insight into the varied considerations that data scientists and engineers typically undergo when attempting to process large sets of unprofiled data. Exposure to the underlying challenges of processing Big Data efficiently and accurately, turned out to be a valuable experience. Admittedly we did not always achieve the results that were originally intended; however, we felt that this experience was still valuable as it provided hands-on opportunities to engage as a part of this process. An important understanding gained through this experience is realizing that profiling and analysis for Big Data would be an incredible undertaking without access to Spark and high-performance computing clusters, such as DUMBO. Accordingly, our approaches took advantage of these platforms as well as making use of the MapReduce paradigm when applicable. All in all, the project proved a fruitful learning experience in collaborative data profiling. We have attached our GitHub repository below:

https://github.com/deusalexmachina/theBiggestData

1 Generic Profiling

1.1 Generator Function – ds_reader.py

For the first stage of profiling, we implemented a reader that takes in a file directory consisting of NYC Open Data datasets and converts each file into Spark SQL data frames. The reader takes advantage of lazy evaluation by outputting data frames via a Python generator.

The reader is implemented as a generator function: a lazily-evaluated iterator that processes a number of datasets to be read and stored in memory at request. We opted for this method over a naïve implementation of simply returning a list, as the latter approach means building the entire list of data frames in memory. While list building may be okay for normal applications, it would quickly run out of memory for extremely large datasets without making changes to Spark's memory configurations.

Our reader offers improved performance by targeting efficiency on the basis of lower memory usage, as well as for storage and processing. For instance, one advantage is that we do not need to wait for the reader to iterate over all datasets before we can begin processing them.

1.1.1 Timing Interface – timing.py

In order to assess the reader's performance, we developed a standardized timing interface that calculates the average runtime of the generator as shown in Figure 1. The timing.py file can also be generalized for any function that makes use of the reader.

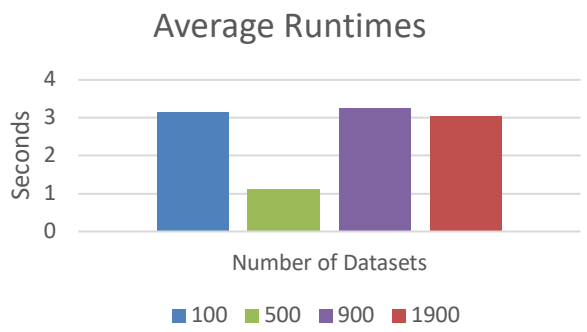


Figure 1: Average runtimes of generator function on various dataset sizes

Initially, we benchmarked our reader on 100, 500, and 900 datasets, which on average yielded a running time of 3 seconds (under optimal cluster load). Running 500 datasets gave us a much faster time, but we discovered this to be the result of skewed dataset sizes. For future benchmarks, we added an optional parameter to allow us to pass in randomized datasets rather than reading files sequentially from the directory.

Loading all 1900 datasets resulted in a completion time of 96.44 minutes, or roughly 3 seconds per dataset. This consistency across multiple tests suggests that our generator scales with volume.

1.1.2 Limitations and Considerations

We considered the possibility of parallelizing reads by spawning multiple concurrent processes. The idea would be to allocate each process to a dataset, which in theory could distribute datasets more efficiently to individual processes.

However, one deciding factor in opting not to implement this approach is due to the complexity of debugging. We reasoned that it would be difficult to troubleshoot a particular dataset if we were to delegate in this manner. As such, an undetected bug could adversely impact our results later down the processing pipeline. The generator, on the other hand, is much simpler and predictable in its output.

In the interest of time, we decided to forego this strategy, though we do acknowledge that this approach could result in greater efficiency. Ultimately, we decided that the generator method would be a much safer and reliable option, and that our energies could be better spent optimizing more critical bottlenecks.

1.2 Profiling Metadata – basic_metadata.py

For metadata profiling, it seemed logical to approach the task using single-column profiling techniques since they fit our use-case for extracting cardinalities and classifying data types as opposed to a multiple-column context [1]. With this strategy in mind, we designed our code to work within the Spark runtime model as much as possible while prioritizing built-in Spark functions and primitives to optimize for speed.

One such method is to utilize the MapReduce model to process data frames individually by columns. Our idea is to convert each column in a given dataset into key-value pairs and generate metadata statistics via subsequent calls to a reducer function. The running the MapReduce model with Spark allows us to take advantage of fast in-memory computations.

We implemented a mapper that creates a flattened output of key-value pairs such that column names are mapped to each value contained in that column. The mapper then outputs a list of columns that can be fed into custom reducer functions (built upon Spark's `groupBy()`) to generate desired metadata. After each call to a reducer function, the output is updated on a master dictionary according to the tasks' metadata specifications.

1.2.1 Cardinalities

Generating the required cardinalities for each column was relatively a straightforward process which relied on the MapReduce paradigm. We were able to count the number of distinct values in each column using Spark's `countDistinct()` function which is passed as a parameter in a custom reducer. Using a similar approach, we checked empty values by calling `isnan()` and `isnull()` through lambda reduce functions.

Processing the top-5 frequent values proved to be more challenging to optimize. We were able to improve speed by sorting by count value rather than by key. The sorted values are then put into a list where the top 5 values are extracted and appended to the master dictionary.

1.2.2 Data Types – basic_profile.py

Data type identification strategies turned out to be more nuanced, since our approach relies on Boolean 'checker' functions. We made sure to leverage built-in functions when applicable and continued to use mapped columns as input to avoid iterating over a dataset except when absolutely necessary.

We decided that each value should go through deliberate sequential try-except blocks. Failing a certain checker function would pass it down to the next checker in the following order: REAL, DATE, INT, TEXT. We reasoned that a value that fails identification would likely be a TEXT, with a few defined exceptions.

The first checker determines whether a value is a REAL type by attempting to cast it to a float and returning true if it was castable. Should this fail, we applied a regex strategy which checked for decimal placement and/or scientific notation. Otherwise, we return false and pass it to the next checker function.

Identifying dates turned out to be the more challenging task that required a two-step process to ensure accuracy. For step 1, we examined the column name of each value for references to keywords, such as "date", "period", "year", "month", "day", and "time". If so, we proceed to step 2 which involves attempting to parse the value into a DATE/TIME object under various conditions. We realized dates could be missing specificity in terms of month or

day and decided to parse dates as long as a year and month can be identified. The rationale for this is that dates without a year and/or month could not be reliably dated with any degree of certainty. Thus, we would omit dates that cannot be reasonably contextualized.

Should casting and parsing for DATE/TIME return false, we pass the value into the next checker which attempts to identify if it is an INTEGER type. This checker employs a similar strategy as the REAL checker function.

The last checker tests whether a value can be casted to a string. Failure to cast at this final step would mean that the data type has failed all defined criteria and would automatically return false as an uncategorizable element.

Overall, we felt that accuracy was important enough to warrant multiple try-except blocks to mitigate the number of false positives, and address data quality issues that make identification more difficult. Admittedly, our profiler made no generalizations about data types based on column name alone, but this was intended to address columns that contain heterogeneous datatypes.

We extracted data type statistics on REAL and INTEGER values by calling built-in spark functions `max()`, `min()`, `mean()`, and `stdev()`. DATE/TIME types were formatted and compared to find max and min values. TEXT types are put into an RDD and mapped by length, delaying costly transformations until all actions are completed. We perform these steps to extract shortest, longest, and average lengths.

1.2.3 Timing

One challenge in our methodology is determining a way to tackle extremely large datasets. In the interest of time, we opted to delay processing files that decompressed to over 2.5GB. We reasoned that it would be more efficient to process smaller files first and begin working on accessible data immediately.

Using the timing.py interface, we ran tests on increasing sets of files to benchmark the time to complete Task 1. This includes calculating cardinalities, identifying datatypes, and generating statistics.

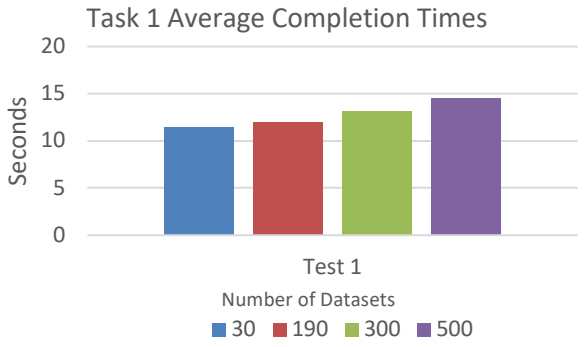
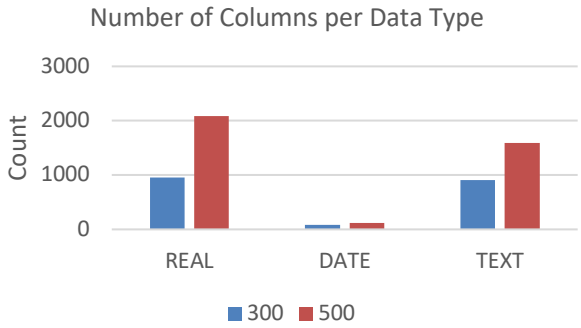


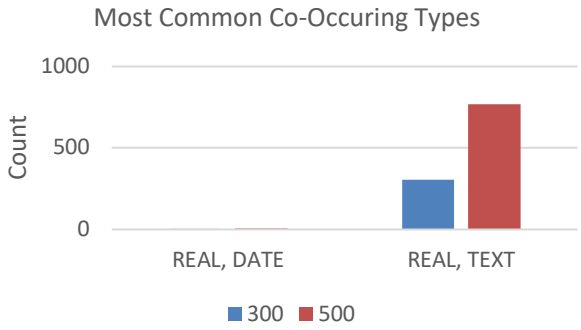
Figure 2: Average runtimes of deriving metadata on cardinalities and data type profiling on various dataset sizes.

Our results suggest that the average time to load and process each dataset is roughly 15 seconds, under variable server load.

1.3 Results

Testing on 300 and 500 datasets gave us the following results:





1.2.4 Considerations and Issues

One interesting result is that our code did not generate any counts for INTEGER types. We believe this is due to a bug in our data profiling code. We made note of this and attempted to fix it with varying results. As a result, the common occurring types were also impacted by this anomaly.

As for data quality issues, we kept null and nan values, during datatype profiling for consistency since this was requested as a metric for the cardinality sections.

2 Semantic Profiling

2.1 Semantic Labeling

Our primary focus on Task 2 was developing strategies that produced adequate precision without significantly compromising recall. Due to inherent design tradeoffs (i.e. sacrifice precision for greater recall, and vice-versa), we decided to approach this problem using Jaccard Distance Measures and heuristic matching techniques [2].

2.1.1 Manual Labeling and Generating Reference Set

The preparatory stage in devising a balanced labeling strategy is to create a reference set of values based on known semantic labels. This involved manual labeling our columns to identify "true" semantic types. Doing so would enable us to compute how our predictions actually stack against true types.

Logically, it follows that we should use our true column results to apply string matching and derive the necessary semantic labels. However, this is too general of an approach that may miss the mark in terms of both precision and recall. Instead, it makes more sense to use this technique as an initial filtering strategy among other more sophisticated methods.

We opted to elaborate upon true labelling by generating frequent values for all true labelled columns. Rather than simply sorting and extracting some arbitrary number of frequent values, we reasoned that it would be better have a unique distribution of values that is also proportionate to the size of the column. For example, a larger column set with 50 unique elements should have more representative values than a small column of only 5 unique elements. Implementing this frequent values generator would later aid us with string matching.

2.1.1 Column Matching and Jaccard Distance – similarity.py

We decided to employ similarity algorithms to generalize semantic labelling. In other words, we attempt to derive a semantic type using labels found in our reference set. To begin, we converted each column name and value into a string, and map them into key-value pairs. We then determine an optimal k-shingle based on values from our reference set. Through trial and error, we found that using a k-shingle that is half the length of the longest string produced suitable results for measuring Jaccard distance.

Using Jaccard distance, we devised a general strategy that matches a column value of unknown semantic type to any other column name or value of known types. The algorithm can be broken down by cases as follows:

Case 1 - Substring + Jaccard Distance Matching:

In the event of a likely match between our target substring value and a column name, Stage 1 iterates through the column's values in the reference set and check for matches with our target string. If there are positives, we add the matched reference values to a new list, or to an existing list that the target string is in. We return this list when used for processing Jaccard distance.

During Stage 2, we compare the Jaccard distance of our target string with the Jaccard distance of all values that are contained in the list. Stage 3 returns a tuple consisting of the predicted true label, the matched value, and the minimum distance (or most similar). Case 1 basically performs high-precision matching using our true semantic labels.

Case 2 - Jaccard Distance Matching:

For values that do not return a match, and therefore have an unknown semantic label, we apply Stages 2 and 3 and process the similarity using Jaccard distance only. This is most error prone and is more likely to generate false positives when similar strings are encountered. Case 2 is much broader and does not rely on exact string matching.

Integration via String Preprocessing:

As the two cases seem to skew towards high-precision or overly general distance comparison, we devised a way to combine the two matching functions via string preprocessing. Our solution is to apply increasing degrees of string transformations to our target, and check the new converted target string for possible substring matches. Should a match fail, we then resort to applying distance matching to return a string with the smallest Jaccard distance.

This integrated approach combines the best of both worlds by attempting high-precision matching first, followed by Jaccard distance. At each iteration, an unmatched string would start off as alphanumeric, then lowercase, and finally numerics replaced with "*", such that we terminate with a value that either loosely matches a reference value or achieves a close minimum Jaccard distance. This approach seems to yield consistent results across our datasets with adequate similarity matching.

2.1.2 Matching Semantic Values – Heuristic Labeling

The last technique in our balanced labeling algorithm seeks to identify semantic types independently from the true label. By adopting a heuristic approach of examining column names and values for similarity, we can make informed generalizations that avoid mislabeling or dissimilar comparisons.

Heuristic labeling is by far the more sophisticated of the two approaches. The important point is that the heuristic approach only triggers when the Jaccard distance measure from the integrated preprocessing is sufficiently dissimilar (i.e. 0.9 from a scale of 0.0 to 1.0). We can characterize the heuristic stages as follows:

Stage 1 requires us to first run the integrated preprocessing algorithm to return a Jaccard distance measure. Should the average distance prove sufficiently dissimilar compared to our cutoff, we run the heuristic approach in Stage 2 to see if we can produce a better measure.

Stage 2 involves manually examining the frequent data from our reference set without referring to the true semantic labels. It is similar to the integrated approach in that it checks the column name for a substring match. A basic check minimizes semantic misclassifications and limits us from comparing websites to phone numbers. If we arrive at a match on the column name, we move on to Stage 3; otherwise, we check the values of the column for similar substring. Similar substring values are then added to a separate list for subsequent processing.

Stage 3 reruns the integrated preprocessing algorithm on the column name. Cutoff average is checked once again. If it is inadequate, we rerun preprocessing for the last time on either the known semantic type values, or the values in the list appended by the heuristic in Stage 2. No further preprocessing happens after this, so we default to the semantic type of the column name.

Stage 4 checks that stage 3 results are significantly better than stage 1 results according to a predefined ratio. If stage 3 is better, we keep it; otherwise, we use stage 1 semantic labeling.

Lastly for Stage 5, we check a value's minimum distance for some similarity cutoff and add it to the reference set it isn't already there. If our initial matches are correct, the later matches should be more accurate as a result of this update. The reference set does however top out at some sufficient size in order to prevent slow down and redundant matching.

2.1.3 Manually Labeling "True" definitions

In the following section, we justify some of our decisions when manually labeling columns by their "true" semantic type. For clarity, we denote a particular semantic label using the `courier` font face.

`City` vs `Borough`

It comes to mind that there may be semantic ambiguity in how the label `city` is actually used in the context of NYC Open Data. Presumably, the only or most common "city" would be New York, but we found that applying a "true" semantic definition imposed too strict of a standard on most of our datasets. We illustrate with the following example:

In a column named "EMPCITY" containing [*New York, Manhattan, Queens, Brooklyn, Staten Island, and Bronx*], it would be semantically consistent to label the column as both `borough` and `city`. Technically, the only city in the list is *New York* though it appears to be the case that city and borough are used interchangeably in the above set. We attempted to be regular in labeling in accordance with the column's given name and its contextual usage.

`Address` vs `Street Name`

By definition, an `address` is a complete description necessary to identify a specific location. We will consider a label to be an address if a street number and street name is included. Whereas a `street name` is a subset of an address which may or may not include a street number. As such, every `address` is a `street_name` but not every `street_name` is an `address`.

2.1.4 Known Issues and Challenges

In rare instances, we encountered what appears to be an unidentifiable parsing error in Spark that inexplicably populates a column with values from a different column. We thought it may be related to the way Spark handles \t delimiters, though we weren't able to pinpoint it to any particular instance to confirm. The exact cause remains elusive and it seems to be quite inconsistent. We acknowledge that this is bug.

Another issue we faced is predicting whether a column value of a single letter is a person's middle initial or an abbreviation for a borough. We were able to apply specific checks to ensure correct categorization.

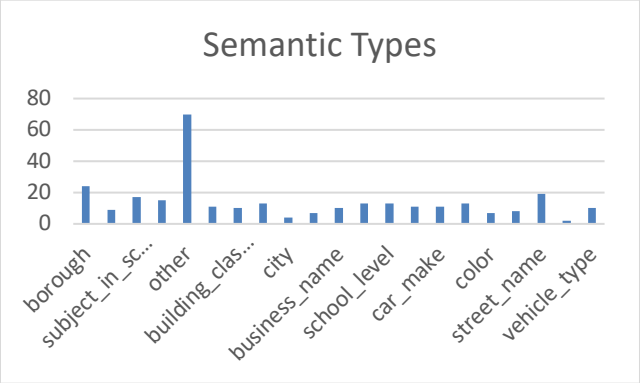
We feel it is important to mention once again that encountering extremely large datasets resulted in long processing times. One of the dataset columns assigned to us had 1569426 rows, which resulted in a major bottleneck when running our similarity algorithms. We reason that because Task 2 is primarily concerned with precision and recall, that our code did not necessarily prioritize efficiency as compared to Task 1.

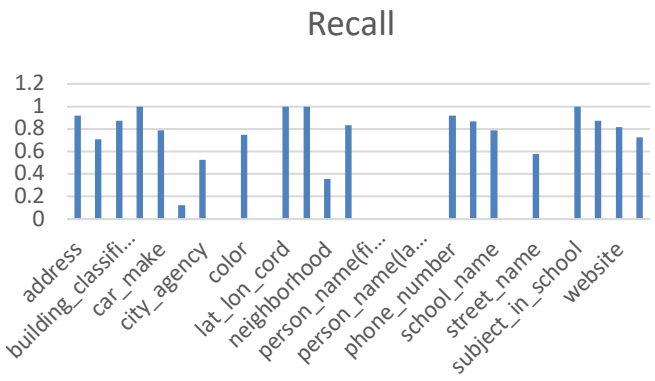
2.2 Results

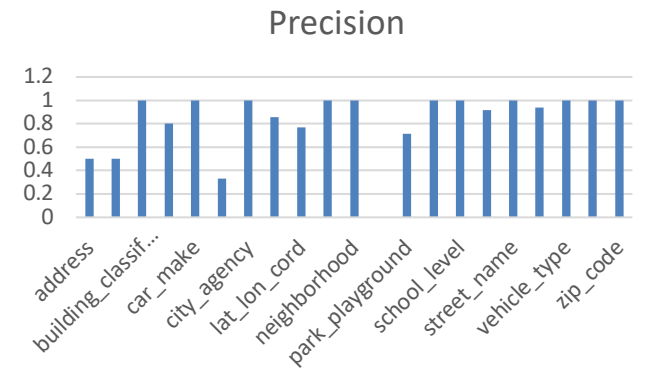
We uncovered some interesting results from running our similarity strategies. The goal was to balance recall with precision; but as it turns out, the easiest solution would be to fine-tune for an F1-score based on the harmonic mean of precision and recall.

We highlight some of the more interesting and/or unexpected results below. `Borough` and `Lat/Lon coordinates` were predicted to be related based on our similarity algorithms. While this is not a correct true label, we consider this to be in the domain of location labels which we found interesting.

Another interesting find is that `School Name` and `Parks/Playgrounds` are also predicted as interchangeable semantic labels. One thought of a possible reason for this being shared landmark and locational names in both types.







Evidently, our recall turned out to be worse overall, though we did not determine this to be a surprising discovery. One of the major limitations we encountered is the sheer size of some of the datasets which prevented us from sampling as much as we needed.

3 311 Data Analysis

The task at hand was to analyze the 311-reports, observe patterns in changes with respect time and locations and comprehend the possible reasons to explain the factors that might account for these observed disparities.

The dataset was generously provided to us in the project's repository hosted on DUMBO. Due to the extraordinary size of this dataset, we employed a sampling mechanism to explore the data without loading the entire set. Going over the data for 311 reports for years 2010-2019, we observed that there has not been any unreasonable increase in the number of complaints. The marginal increase can be credited to the increase in the population over the last decade. Furthermore, we see certain complaint types stay prominent throughout the time period (for instance, noise and heat).

Borough	Number of Complaints	Estimated Population	Population Density (per sq km)	Normalized for Estimated Population
MANHATTAN	1221062	1629000	25846	0.749
QUEENS	4920996	2359000	8354	2.086
BROOKLYN	6412785	2648771	14,182	2.421
BRONX	3752661	1471000	13,231	2.551
STATEN ISLAND	1082175	479458	3,132	2.257

The first part of the task is to get the top complaints for each borough in New York City. This can be achieved using a `groupBy()` statement. We saved the entire dataset as a csv for visualizations, but the top complaints for each borough are listed below in Table 2. Although, they are not quite the same, we do see a trend in population density and noise complaints. Furthermore, if we decide to group together subcategories like Noise and Noise-Residential, we see a different set of top complaints with much more variation.

Table 2: Top complaints for each neighborhood

MANHATTAN	BROOKLYN	BRONX	QUEENS	STATEN ISLAND
Noise-Residential	Noise-Residential	Noise-Residential	Blocked Driveway	Street Condition
Heat/Hot Water	Heat/Hot Water	Heat/Hot Water	Noise-Residential	Street Light Condition
Noise	Blocked Driveway	Heating	Street Conditions	Water System
Noise-Street/Sidewalk	Illegal Parking	Street Light Condition	Street Light Conditions	Noise-Residential
Street Condition	Street Condition	Plumbing	Illegal Parking	Illegal Parking
Heating	Street Light Conditions	Blocked Driveway	Water System	Sewer

Some of these differences seen in Table 2 are fairly simple to explain. Queens, Brooklyn and Bronx both have Blocked Driveways as one of their top complaints, but Manhattan, due to its high population density, lacks driveways. On the flip side, Staten Island due to its low population density doesn't face this problem. Population density can also be partially credited for the high number of noise and general construction complaints seen in boroughs -- dropping as the population density drops.

We charted the top complaints each year from 2010-2019 to observe trends in the rise and fall of complaints. The graph is divided into two figures: Figure 1 and 2. This is due to a change in the type of complaints making a comparison on a single X axis unreasonable. Image 1 is a visualization of the same data.

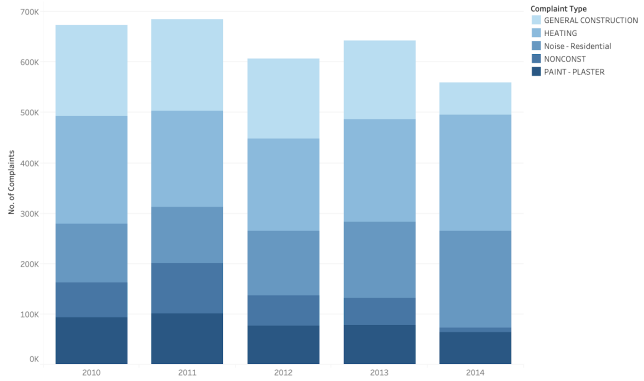


Figure 1: Top complaints for the years 2010-2014

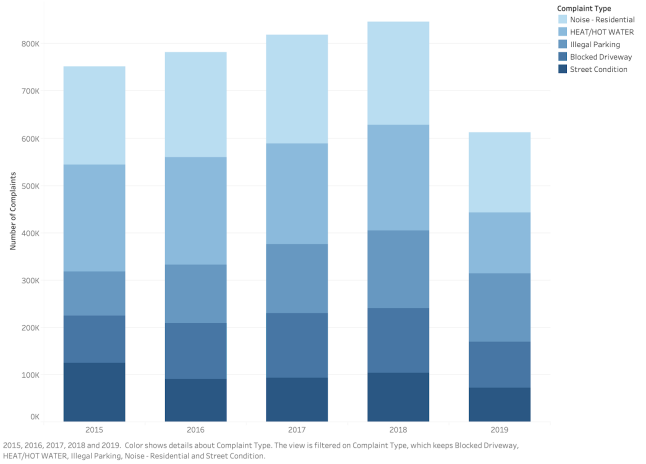
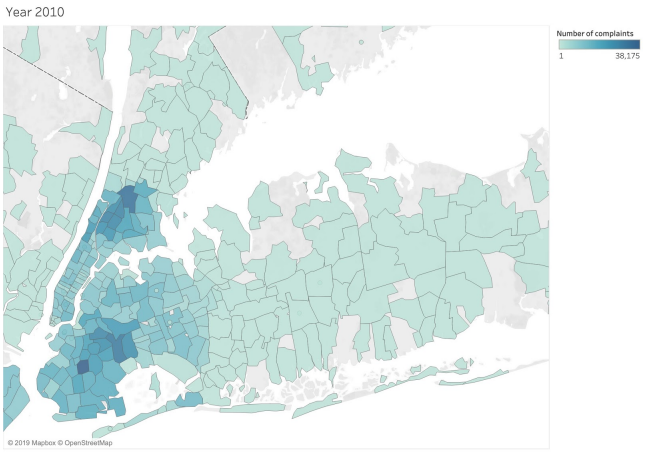
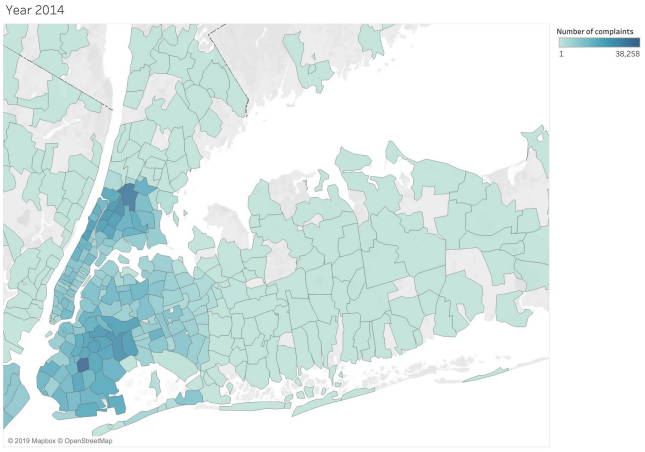
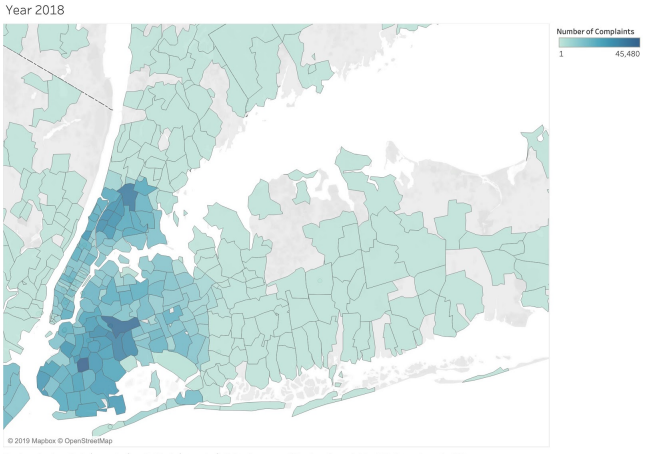


Figure 2: Top complaints for the years 2015-2019

For a more refined perspective, we analyzed the data and the variations in top complaints with respect to the neighborhoods in the city. Since the address attributes in the dataset lacked pre-defined order, we settled for zip codes for complaint types and counts. There are three areas we looked at to explain the top complaint at a neighborhood level: proximity to tourist attractions, seasonality and proximity to filming locations. The data generated is available as CSVs. For a better understanding, we imported the data into Tableau and generated heatmaps for the number of complaints for each zip code in the city. The figures represent the number of complaints.







Conclusion

As stated in the abstract, our report focuses on highlighting the fundamental strategies intrinsic to profiling large amounts of open access data. Throughout Task 1, we focused on optimizing our code and devising efficient designs that maximized speed without sacrificing functionality. We were able to obtain noticeable speedups by implementing MapReduce for parallel processing in

Spark. Task 2 required different set of strategies to generate our desired output. Instead of focusing on efficiency, we opted for greater precision and recall as we deemed these factors to be the more important metric in semantic labeling. An important consideration in Task 2 is that we encountered extremely large datasets that impacted the processing times of our algorithms. Lastly, we prepared an analysis of 311 complaints in an attempt to explain which factors if any led to demonstrable changes in complaint distribution over time.

Individual Contributions

All members contributed equally to the project.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *The VLDB Journal* 24, 4 (August 2015), 557–581. DOI:https://doi.org/10.1007/s00778-015-0389-y

[2] Cyril Goutte and Eric Gaussier. 2005. A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation. In *Advances in Information Retrieval*, 345–359.