Data cleaning at Scale - by ITA Group

Shourie Sai*
New York University
New York, USA
sk9247@nyu.edu

Huimin Zhang
New York University
New York, USA
hz2466@nyu.edu

Xinyu Zhang
New York University
New York, USA
xz3369@nyu.edu

ABSTRACT

We present an approach for cleaning big datasets containing geographical information. Our focus is on identifying and solving data quality issues, like typos, missing values, and inconsistent values in the Motor Vehicle Collisions - Crashes dataset. We mainly used the Openclean library, GeoJson, and Turfpy to clean geographical data found in the dataset and manually measured the effectiveness with precision and recall method. We also scaled our solution to clean ten similar data files and measure their effectiveness. Our experimental results indicate that our approach is practically useful for the Motor Vehicle Collisions - Crashes dataset, it cleans geographical data with an average of 76.714% precision and 99.629% recall.

1 INTRODUCTION

Our report details the steps to discover data quality issues in the dataset provided by NYC Open Data and the methods we employed to resolve these data inconsistency issues. We worked on the Motor Vehicle Collisions - Crashes dataset which comprised 1,839,610 records of data about vehicle crashes ranging from 2012 to 2021. This dataset was organized into 29 columns including information such as the borough, street names, zip code, ordinates of the site of the crash, and also the number of people affected by the crash.

In the first part of this project, we worked exclusively on the dataset to determine and resolve data quality issues presented in it. We used the python packages Openclean and pandas to profile the data and clean the data to remove the inconsistencies that we discovered. In the second part of the project, we improved our previous solution to remove dependency violations and then we scaled our cleaning script to handle other datasets similar to the vehicle crashes data.

2 PROBLEM FORMULATION

While profiling the data with the Openclean library, we discovered that the Motor Vehicle Collisions dataset had various data quality issues including missing values, wrong data types, typos„ and inconsistent values.

2.1 Missing values

Most of the features in the dataset had missing values. Although it was valid for some columns to have null values, it was invalid for other columns to have null values.

Table 1 is the summary of the missing values for each column.

Table 1:Effectiveness Measurements Result of Motor Vehicle Collisions - Crashes

Column Name	missing values	NULL values allowed
CRASH DATE	0	Not allowed
CRASH TIME	0	Not allowed
BOROUGH	570882	Not allowed
ZIP CODE	571104	Not allowed
LATITUDE	215644	Not allowed
LONGITUDE	215644	Not allowed
LOCATION	377988	Not allowed
ON STREET NAME	377988	Allowed
CROSS STREET NAME	665336	Allowed
OFF STREET NAME	1561716	Allowed*

* If OFF STREET NAME exists, ON STREET NAME and CROSS STREET NAME could be NULL. If ON STREET NAME and CROSS STREET NAME exist, OFF STREET NAME could be NULL.

2.2 Wrong Data Type

	date	float	int	str
CRASH DATE	3418	0	0	0
CRASH TIME	0	0	0	1440
BOROUGH	0	0	0	5
ZIP CODE	0	0	231	1
LATITUDE	0	122314	1	0
LONGITUDE	0	95949	2	0
LOCATION	0	0	0	243078
ON STREET NAME	86	0	16	15865
CROSS STREET NAME	2	1	26	19209
OFF STREET NAME	30	0	1	178661
NUMBER OF PERSONS INJURED	0	0	27	0
NUMBER OF PERSONS KILLED	0	0	7	0
NUMBER OF PEDESTRIANS INJURED	0	0	13	0
NUMBER OF PEDESTRIANS KILLED	0	0	4	0
NUMBER OF CYCLIST INJURED	0	0	5	0
NUMBER OF CYCLIST KILLED	0	0	3	0
NUMBER OF MOTORIST INJURED	0	0	27	0
NUMBER OF MOTORIST KILLED	0	0	6	0
CONTRIBUTING FACTOR VEHICLE 1	0	0	2	59
CONTRIBUTING FACTOR VEHICLE 2	0	0	2	59
CONTRIBUTING FACTOR VEHICLE 3	0	0	2	48
CONTRIBUTING FACTOR VEHICLE 4	0	0	0	39
CONTRIBUTING FACTOR VEHICLE 5	0	0	0	29
COLLISION_ID	0	0	1839256	0
VEHICLE TYPE CODE 1	0	0	12	1298
VEHICLE TYPE CODE 2	0	0	12	1426
VEHICLE TYPE CODE 3	0	0	0	202
VEHICLE TYPE CODE 4	0	0	0	84
VEHICLE TYPE CODE 5	0	0	0	55

Figure 1: Data Type Analysis

*Three authors contributed equally to this research.

We found that many of the columns had unexpected data types (data type that is inconsistent with the domain of the column). There are also instances of a column having values with different data types.

Figure 1 summarizes data type information for columns that contain data with different data types. Each number in the table represents unique values for each data type.

As it can be noted from the above table, features such as ON STREET NAME have values in different data types, hence it must be must typecast to the required data type.

2.3 Typographical error

The primary string-valued columns used in this dataset are columns for address and street names such as 'ON STREET NAME, 'OFF STREET NAME', etc. These features had a very large amount of typographical errors. For instance, the feature 'ON STREET NAME' had about 4937 typos. There were different types of typographical errors and the most frequent ones are listed below:

- **Type 1** Multiple representations of the same street name with prefixes or suffixes in a different order. For example, the street name '134 AVE & SPRINGFIELD BLVD' was represented with values such as 'SPRINGFIELD BLVD && 134 AVE', '134 AVE && SPRINGFIELD BLVD' etc.
- **Type 2** Same street Names have different representations of the same suffixes or prefixes. For example, the street name 1st Avenue was represented with values 'FIRST AVENUE', 'FIRST AVE', '1 AVE', etc

2.4 Consistency Issues

All the columns representing geographical data such as zip code, latitude, longitude, street name, etc, should be consistent with each other. Below are the dependencies that should hold for geographical data.

- A single instance of zip code should only have one borough, for example, zip code 11220 should only belong to one borough(Brooklyn).
- A particular geographical ordinate (latitude, longitude) should only point to a single instance of zip code, borough, street name.
- An instance of OFF Street Name should only have one zip code.
- An instance of ON Street Name and Cross Street Name should belong to a single borough.

In this dataset, however, there were many instances where these consistency checks were failing. There were multiple boroughs for the same zip code, and there were multiple zip codes for the same Off Street Name and borough.

3 RELATED WORKS

3.1 Related Papers

Despite the fact that there has been an abundant amount of tools developed for data cleaning, we found it complicated to work with a different set of tools to perform various tasks in data cleaning. The python library openclean [1] alleviated this complexity by combining tools involved in data profiling and data cleaning into a single environment. As we were using very large datasets, The data stream feature in Openclean allowed us to work with large datasets without worrying about the limited memory overhead of main memory. We utilized the profiling operators provided in Openclean to explore metadata that helped us detect anomalies in our datasets. The transformation operations helped us perform data cleaning effectively by enabling data frame operations such as filtering, insertion, updating data, etc.

We also found that the paper [2] gave us helpful insights into various phases of the Data cleansing process and about cleaning big datasets qualitatively in general. It also introduced us to Cleanix [3]; A parallel big Fata cleansing system that strives to handle problems pertaining to variety and the volume of big data. Cleanix primarily handles data quality issues such as duplication, incomplete data filling, conflict resolution, and abnormal value detection. Since Cleanix is incorporated with features such as unification, scalability, and usability, it can perform data quality reporting and data cleansing tasks in parallel. There are four main stages in the data flow of Cleanix, 1) correct abnormal data after reading data, 2) deal with missing values, 3) broadcast the updated value in local gram 4) perform conflict resolution and solve duplication. As It is very user-friendly and has a simple graphical user interface, the users are not required to have expert-level domain knowledge.

3.2 Related Technologies

- **PySpark** PySpark is a Python API for Apache Spark to process larger datasets in a distributed cluster. We used PySpark here to perform the data cleaning part, like fixing typos, fixing outliers, filling up missing values.
- **Geojson and Turfpy** GeoJSON is an open standard format designed for representing simple geographical features. The features here include points, lines, polygons, and etc. turfpy is a Python library for performing geospatial data analysis. Here, we utilized turfpy and Geojson to infer the zip code from coordinates to fill up missing values.

4 METHODOLOGY/DESIGN

We basically follow the following steps to clean the datasets:

- Fix columns
- Clean original dataset
- Find similar datasets
- Scaling our methods to other datasets
- Measure the effectiveness

4.1 Dealing with Data Type Errors and Outliers

4.1.1 Data type Errors. Most of the Data type inconsistencies that we found in the dataset belonged to integer-valued columns such as 'NUMBER OF PERSONS INJURED', 'NUMBER OF PEDESTRIANS INJURED', etc. and it required a simple typecasting of the column from string to int.

The values in the Zip Code column were in float, so we also had to get rid of the decimal point after typecasting the zipcode value to a string. Finally, The Latitude and Longitude values in some instances contained punctuation mark characters(such as ','), after removing such characters, we converted latitude and longitude columns to float.

4.1.2 Outliers. The outliers in the datasets are primarily in the Zipcode, Latitude, and Longitude columns. There are many instances of Zip Code where it belonged to locations outside New York City. We used a CSV file containing all the zip codes to NYC to filter out the outlier zip codes and replaced them with an empty string.

The columns Latitude and Longitude had values that were not in latitude(40.4961154,40.91553278) and longitude range(-74.25559136,-73.70000906) of NYC. We had checked if the latitude and longitude values were in this range and removed rows that were outliers.

4.2 Fixing Typos

The typographical error present in the string-valued columns is due to different representations of the same data. This could include spelling mistakes and also have suffixes or prefixes in different orders. To find out the similar texts which represent the same data we employed the clustering technique, which is the operation of finding groups of different values that are representing the same value.

The method we employed to group similar texts is key collision clustering, wherein the operations are employed on string data to create an alternative version that only contains the meaningful part of the string to create a key. Then it groups these values based on their key. The general operations performed in order to create a key are:

- Remove all the trailing and leading whitespaces
- Change the string to lowercase
- Remove punctuation characters to normalize the string.
- Create tokens by dividing strings by whitespaces
- Perform sort operation on the token and remove duplicates
- Join the remaining tokens back together by adding a single white-space between them

We used the key clustering algorithm along with a custom key generator USStreetNameKey() provided by the python library Openclean. This function performs key clustering to find clusters and it also accounts for general abbreviations used in street names such as 'str' for street, 'AVE' for Avenue, etc.

We used the custom standardizer function StandardizeUSStreetName() (provided by Openclean library) to modify strings belonging to the same cluster and change their values to the standard key. The dataset still had typographical errors after using this function, so we stored the remaining cluster values in a dictionary and chose the most frequently used value in a cluster to be the key, and then mapped all the values in the cluster to the key, to standardize it. After this operation, the dataset did not have typos.

4.3 Dealing with Missing Values

Most of the columns in our dataset had missing values. As for the integer and float valued columns such as 'LATITUDE', 'NUMBER OF PERSONS INJURED', we replaced the missing value with integer -1. For the missing values in string-valued columns such as 'CONTRIBUTING FACTOR VEHICLE 1' and 'VEHICLE TYPE CODE 1', etc, we replaced them with a string 'unspecified'.

4.3.1 Fix missing Zip Code. If a record is missing a zip code but has latitude and longitude, we can use GeoJson(data structure containing all the ordinates belonging to a zip code as a polygon) and python library Turfpy(to perform spatial data analysis) to infer which zip code the location belongs to. To be specific, we used the bolean_point_in_polygon method provided by Turfpy library to determine if a coordinate is in a polygon of a zip code. The New York City GeoJson data we used here was created by GIS Lab, Newman Library, Baruch CUNY. [2]

4.3.2 Fix missing Borough. By referring to [5] which lists all the boroughs in New York City and the zip codes corresponding to them, we created a dictionary that maps a given zip code to the borough that the zip code belongs to. This dictionary contains zip codes as keys and their corresponding borough names as values. For example:

```
zipcode_borough_mapping = {
    10001: 'Manhattan'    ,
    10002: 'Manhattan'    ,
    10003: 'Manhattan',
    '..': '...',
    11201: 'Brooklyn',
    '..': '...'
}
```

Then if a column has a missing borough field but zip code exists, we can infer the borough field using this dictionary by mapping zip code to a borough name.

4.3.3 Infer Borough and Zip Code by Street Name. We inferred the missing zip code and borough values from the column 'OFF STREET NAME' value. We implemented Geopy(python client for geocoding service) to use the 'OFF STREET NAME' value to retrieve other geographical information such as the borough name and the zip code of the corresponding 'Off Street Name'. This method however worked fine on small datasets with about hundreds of rows, but it did not scale well with larger datasets(containing over 100 thousand rows). The providers of Geopy restricted its use by setting the query upper limit to an absolute maximum of 1 request per second. If we intended to put the program to sleep 1 second between 2 queries, it would have taken a long time to run over the whole dataset which has more than 1800,000 records. Thus we kept the empty borough fields and empty zip codes unchanged even though we can infer them based on the 'off street name'.

4.4 Deal with consistency check

4.4.1 Functional Dependency-Check. We applied the "fd_violations" method in Openclean to find the functional dependency violations in the dataset. As for the columns we picked, there should be a functional dependency from zip code to borough as a zipcode can only belong to a single borough. Following is an example that shows how we applied this method, and what we did to clean the dataset based on the violations we found.

We wanted to find violations of the functional dependency: ZIP CODE -> BOROUGH for Motor Vehicle Collision - Crashes dataset. When we applied "fd_violation", we got the results showing all the violations (shown by Figure 2). We then used a dictionary to find the right mapping from zip code to borough and update the records with the wrong borough name.

```
ZIP CODE | BOROUGH
=============|===============
11237        | 8547 x BROOKLYN
             | 325 x QUEENS
-------------|---------------
11385        | 18473 x QUEENS
             | 34 x BROOKLYN
-------------|---------------
11222        | 12351 x BROOKLYN
             | 1 x MANHATTAN
-------------|---------------
10301        | 7664 x STATEN ISLAND
             | 2 x QUEENS
-------------|---------------
11208        | 18743 x BROOKLYN
             | 171 x QUEENS
-------------|---------------
11421        | 6189 x QUEENS
             | 7 x BROOKLYN
-------------|---------------
```

Figure 2: Functional Dependency Violations from zip code to borough

4.4.2 Geographical Data Consistency Check. Since our dataset has columns representing geographical information such as latitude, longitude borough, etc. We need to make sure that there is geographical consistency between these columns. We establish the geographical consistency between the columns borough name and zip code by using a dictionary containing all the possible mappings of zip codes in NYC to borough names and we used this dictionary to check if zipcodes are consistent with borough names. To establish geographical consistency with ordinates (latitude and longitude) and zip code, we used a JSON containing all the zip codes in NYC and also having all the ordinates belonging to a particular zip code(stored as a polygon for each zip code). We were then able to map an ordinate to a zip code using this JSON file. Since we established geographical consistency between Borough and zip code, as well as consistency between ordinate to zip code, It can be implied that there is also geographical consistency between ordinate and borough.

4.5 Find similar datasets

We utilize Jaccard containment to find 10 other datasets in NYC Open Data whose columns overlap with the Vehicle Collision - Crashes.

The formula of Jaccard containment is |QX|/|Q|, in which Q is a query column and our goal is to find all columns X in collection C. Jaccard containment is more suitable for similar columns research than Jaccard similarity because Jaccard similarity favors domains with small size and Jaccard containment is agnostic to the difference in the sizes of the domains. When |X| increases, Jaccard containment remains unchanged but Jaccard similarity tends to 0.

First, obtain all column names of each NYC Open dataset which is available on Peel HDFS using pyspark. Next, regard all column names in one dataset as a string and convert the string to sets containing strings with two-word-shingle. For example, twoWordshingle("Primary Artist Last") = ['Primary Artist'], ['Artist Last']. Then, apply Jaccard containment to column name strings. In the formula of Jaccard containment, Q is a set obtained from the string "BOROUGH ZIP CODE LATITUDE LONGITUDE STREET NAME". Finally, select the top 10 datasets as similar datasets according to the Jaccard containment.

4.6 Scaling our methods to other datasets

We manually defined the column name mapping dictionaries of other datasets to rename the column names of those datasets to a standard one as what we used in Motor Vehicle Collision - Crashes. The standard column names are "BOROUGH", "ZIP CODE", "LATITUDE", "LONGITUDE", "STREET NAME".

Here is an example of the column name mapping dictionary for another dataset:

```
column_name_mapping_gcaa = {
    'Borough': 'BOROUGH',
    'Zip Code': 'ZIP CODE',
    'Latitude': 'LATITUDE',
    'Longitude': 'LONGITUDE',
    'Address': 'STREET NAME'
}
```

Then we create a list containing the names of the ten datasets, and we create a python script to loop through the list and perform the methods we have created above to other datasets.

The following table shows an overview of how to apply methods to fix data type errors and outliers for other datasets. Each function name we defined here fixes one column and the operation's row in the table shows the details of how we fix those columns.

Table 2:Function names and Description of the methods implemented to fix the columns

Function Name	Operation
fix_borough	Define a mapping to perform borough name standardization
fix_zipcode	If not valid, set it to -1
fix_street_name	Perform street name standardization by the method in openclean
fix_latitude	If not valid, set it to -1
fix_longitude	If not valid, set it to -1

4.7 Measure the Effectiveness

The methodology we used to measure the effectiveness is as follows:

- **We define the confusion matrix by**
  Table 3:Definitions of evaluating terms

Terms	Definitions
True Positive (TP)	The original is right, we didn't change it.
True Negative (TN)	The original is wrong, we changed it correctly
False Positive (FP)	The original is wrong, we didn't change it.
False Negative (FN)	The original is right, we changed it wrongly .

- **Manually Perform Measurement** We manually measure each value in a row to see if this value is changed and see if this value is changed correctly. Then count the number of these types

- **Calculate Precision Score and Recall Score**

$$PrecisionScore = \frac{TP}{FP + TP}$$

$$RecallScore = \frac{TP}{FN + TP}$$

5 RESULT

5.1 Environment

5.1.1 *Running Procedure.*

(1) We used Jupyter Notebook("Motor Vehicle Collisions - Group ITA.ipynb") and implemented data profiling and cleaning techniques using Openclean library and pandas for part1 of this project.
(2) We refined our previous work and generated a new jupyter notebook("Refined part1.ipynb") to clean the Motor Vehicle Collisions dataset for part2.
(3) We then wrote a pyspark program("search_data.py") to extract all column names of datasets on the Peel HDFS and wrote a jupyter notebook("Jaccard.ipynb") to select 10 similar datasets.
(4) We extracted the main methods from "Refined part1.ipynb" and wrote a python script("main.py") which ran on the Peel HDFS to clean ten other datasets. We used "hfs -get" to copy ten datasets to our own directory since we could not access /user/CS-GY-6513 using python and tested them using python. Besides, we also turned our methods to a pyspark program("pyspark_main.py") and tested a dataset on the peel.

5.1.2 *Packages and Dependencies.*

- openclean 0.2.1
- openclean_geo 0.1.0
- turfpy 0.0.7
- pandas 1.3.4
- geojson 2.5.0
- pyspark 2.4.0-cdh6.3.4

5.2 Result

The following tables show the evaluating result of the datasets we have analyzed:

5.2.1 Motor Vehicle Collisions - Crashes Result. Table 4 show the result of original version in part 1:

Table 4:Motor Vehicle Collisions - Crashes Effectiveness Measurement Result - original version

	Boro	Zip	Lat	Lon	On	Cross	Off
TP	632	632	890	890	847	785	93
TN	249	249	0	0	0	0	0
FP	118	118	110	110	153	215	907
FN	1	1	0	0	0	0	0
Precision	0.843	0.843	0.890	0.890	0.847	0.785	0.093
Recall	0.998	0.998	1	1	1	1	1

The average precision is 74.157% and the average recall is 99.943%.

Table 5 shows the result of the refined version in part 2:

Table 5:Motor Vehicle Collisions - Crashes Effectiveness Measurement Result - refined version

	Boro	Zip	Lat	Lon	On	Cross	Off
TP	792	739	921	921	815	710	154
TN	0	180	68	68	0	0	0
FP	208	61	11	11	185	290	846
FN	0	20	0	0	0	0	0
Precision	0.792	0.923	0.988	0.988	0.815	0.71	0.154
Recall	1	0.974	1	1	1	1	1

The average precision is 76.714% and the average recall is 99.629%.

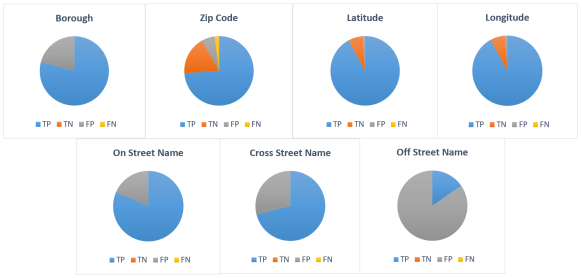

Figure 3: Pie Charts of the Effectiveness Measurement for the Refined Version

Compared with the original method, the refined method has an improvement in total average precision and could clean zip codes, street names, latitude, and longitude more efficiently. But for other geographical data, the results are similar.

This improvement can be attributed to two reasons:

(1) We found that our borough_zipcode.csv file used in part 1 was incomplete and lacked tens of zip codes. Therefore, we manually generated a new borough_zipcode.csv and applied it to part 2. We thought this csv file might be useful to others, and so uploaded the new file to the reference data repository.
(2) In the second part to the project, we changed the key generator from the generic FingerPrint() generator to a street name specific USStreetNameKey() key generator, which mapped street name specific abbreviations(such as 'Str' and 'street) to the same token, hence resulting in more accurate detection of typos.

5.2.2 *Average result of other 10 datasets*

. **Table 6**: Effectiveness Measurement Results of Other 10 Datasets

	Boro	Zip	Lat	Long	Street Name
Precision	0.987	0.881	1	1	1
Recall	1	0.999	1	1	1

*Details of each dataset could be seen in Appendix.

As we can see, the results of cleaning other datasets have high accuracy which indicates that our method is useful to clean general geographical data.

5.3 Limitations of our work

The following are the limitations of our work:

- We couldn't establish geographical consistency between columns 'ON STREET NAME', 'CROSS STREET NAME', and 'ZIP CODE'. Initially, we used geocoding services such as Geopy and Google Maps API to get address information by giving 'On Street Name' and 'cross street name' as inputs. Even though this method worked efficiently for smaller datasets it didn't scale well with larger datasets due to the query limits set by Geopy providers.
- The spark script that we used to find similar datasets used column names to find similar datasets, by finding the Jaccard distance between the names of the columns. However, using Jaccard distance on the column values instead of column names would have been much more accurate. But it was time-consuming to compare all the column values of our dataset to other datasets.

6 CONCLUSIONS

A core challenge in cleaning datasets is dealing with complicated geographical information. Our project presented a general method for profiling and cleaning geographical data. In the first part of this project, our cleaning techniques had an average precision of 74.157% and an average recall of 99.943%. Refining our techniques for the second part of this project resulted in an increase in average precision to 76.714% and average recall to 99.629%. The method was applied to 11 large datasets and reached high accuracy, which indicates that the approach is accurate enough to correctly deal with geographical data quality issues.

REFERENCES

[1] Heiko Müller, Sonia Castelo, Munaf Qazi, and Juliana Freire. 2021. The openclean Open-Source Data Cleaning Library. In *the VLDB Endowment.*

[2] Fakhitah Ridzuan and Wan Zainon. 2019. A Review on Data Cleansing Methods for Big Data. In *The Fifth Information Systems International Conference.*

[3] Hongzhi Wang, Mingda Li, Yingyi Bu, Jianzhong Li, Hong Gao, and Jiacheng Zhang. 2014. Cleanix: A Big Data Cleaning Parfait. In *the 23rd ACM International Conference on Conference on Information and Knowledge Management.* ACM.

A APPENDIX

A.1 Github repository

The link to the the Github code repository:

- https://github.com/kateinUs/BigDataFinalPorject

A.2 Results of other 10 datasets

The result of other 10 datasets are given below:

Table 7: Effectiveness Measurement Results of p6bh-gqsg dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	1	1	1	1
Recall	1	1	1	1	1

Table 8: Effectiveness Measurement Results of ipu4-2q9a dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	1	1	1	1
Recall	1	1	1	1	1

Table 9: Effectiveness Measurement Results of gjm4-k24g dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	1	1	1	1
Recall	1	1	1	1	1

Table 10: Effectiveness Measurement Results of fp78-wt5b dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	1	1	1	1
Recall	1	1	1	1	1

Table 11: Effectiveness Measurement Results of dpm2-m9mq dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	0	1	1	1
Recall	1	/	1	1	1

Table 12: Effectiveness Measurement Results of bty7-2jhb dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	1	1	1	1
Recall	1	0.999	1	1	1

Table 13: Effectiveness Measurement Results of 43nn-pn8j dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	0.95	1	1	1
Recall	1	1	1	1	1

Table 14: Effectiveness Measurement Results of 5ziv-wcy4 dataset

	Boro	Zip	Lat	Long	Street Name
Precision	0.87	0.86	1	1	1
Recall	1	1	1	1	1

Table 15: Effectiveness Measurement Results of 3ub5-4ph8 dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	1	1	1	1
Recall	1	1	1	1	1

Table 16: Effectiveness Measurement Results of 2pgc-gcaa dataset

	Boro	Zip	Lat	Long	Street Name
Precision	1	1	1	1	1
Recall	1	1	1	1	1