Data Cleaning Strategies for DOB House Permit Datasets

Github: https://github.com/sid2305/BigDataProject

Siddhant Lahoti
sl6981@nyu.edu

Vijay Krishnaa
vk2200@nyu.edu

Kevin Ye
ky572@nyu.edu

Abstract

With the evolution of new technologies, the production of digital data is constantly growing. These massive amounts of data are available for the organization which will influence their business decision. The data gathered through different sources, such as sensor networks, social media, business transactions, etc. are inherently uncertain due to noise, missing values, inconsistencies, and other problems that impact the quality of big data analytics. Data cleansing offers better data quality which will be a great help for the organization to make sure their data is ready for the analyzing phase. The data cleansing process mainly consists of identifying the errors, detecting the errors, and correcting them. This paper reviews the data cleansing process, the challenge of data cleansing for big data, and the available data cleansing methods.

1. Introduction

The growing need for data-driven decision-making has created the importance of accurate and precise prediction over recent years. Incomplete, incorrect, inaccurate, and irrelevant records in a database will cause uncertainties during data analysis, and these have to be handled within the data cleansing process else can lead to a misguided decision. Data cleaning aims to improve the quality of data by identifying and removing errors and inconsistencies. Due to the enormous amount of data, manual cleansing takes a long time and is prone to errors, and traditional data cleansing systems cannot be scaled very easily.
So in this paper, we have defined data cleaning techniques on a dataset that are also generalized to be used for any similar datasets. We have also measured the effectiveness of our techniques and the results have been explained in this paper. The four most significant qualitative parameters that we are using to evaluate every data cleaning method are as follows:

- Scalability: The data cleaning mechanism's ability to detect and repair dirty data often involves processing large and different types of datasets without compromising data quality. We are evaluating this by using our techniques on 10 different datasets.
- Efficiency: We are determining the efficiency in terms of precision and recall.
- Data quality and accuracy: This is detected with various parameters like data incompleteness, out-of-range values, obsolete data, and values in the wrong field.

- Usability: Does the technique allow ease of use via a user-friendly programming interface?

1.1 Technologies Used

- Google Colab
- Jupyter Notebook
- Zeppelin Notebook
- Open Clean
- Pyspark

1.2 Datasets Used

- Historical DOB Permit Issuance Housing & Development (dataset UID: bty7-2jhb)
- DOB Certificate Of Occupancy (bs8b-p36w)
- DOB Job Application Filings (ic3t-wcy2)
- Electrical Permit Applications (dm9a-ab7w)
- DOB Cellular Antenna Filings (iz2q-9x8d)
- Housing Maintenance Code Complaints (uwyv-629c)
- Housing Litigations (59kj-x8nc)
- Housing Maintenance Code Violations (wvxf-dwi5)
- DOB License Info (t8hj-ruu2)
- DOB Stalled Construction Sites (i296-73x5)
- DOB After Hour Variance Permits (g76y-dcqj)
- 1995 Street Tree Census(kyad-zm4j)

2. <u>Problem Overview</u>

2.1 Objective

We are going to perform our initial exploration on the Historical DOB Permit Issuance Housing & Development dataset. This dataset includes nearly 2.5 million records related to construction permits filed with and issued by the NYC Department of Buildings between the years 1989 and 2013.

2.2 Metrics

We start with a “dirty” dataset with a variety of errors and anomalies. Cleaning strategies are applied to the dataset with the objective of obtaining consistent and correct data as the output. The effectiveness of the cleaning strategies will thus be the degree by which data quality is improved through cleaning. We define two metrics that benchmark the effectiveness of data cleaning strategies :

1.) Precision:
It is defined as the percentage of records correctly identified as a wrong record, i.e.

$$\frac{No\ of\ correctly\ identified\ wrong\ records}{Total\ no\ of\ identified\ wrong\ records\ by\ our\ method} \times\ 100\ \%$$

2.) Recall:
This is also known as percentage hits. It is defined as the percentage of wrong records being correctly identified.

$$\frac{No\ of\ correctly\ identified\ wrong\ records}{Total\ no\ of\ wrong\ records} \times\ 100\ \%$$

Example : Suppose we have 7 records 1,2,3,4,5,6,7 with {1,2,3,4,5} being wrong records and {6,7} being correct records. If our cleaning strategy identifies {1,2,3,4} as wrong record out of which {1,2} is cleaned correctly, then our precision would be 2/4=50% and recall would be 2/5= 40%

2.3 Goals

- We aim to efficiently clean the DOB House Permit Dataset using python libraries like pyspark and openclean.
- To develop generalized data cleaning techniques that can help in cleaning similar datasets to DOB House Permit without much manual intervention.
- Evaluate the cleaning strategies by applying the techniques on 10 other datasets and calculating precision and recall for those datasets according to the above equations.

3. Related Work

In this section, we briefly review recent research directions on exploring and cleaning big data.

The sampling-based approximation has become a common approach for big data analysis in cluster computing frameworks. ApproxHadoop[1] uses online multi-stage sampling from HDFS blocks to get approximate results. BlinkDB [2] is a distributed approximate query processing engine that uses offline stratified sampling on frequently occurring columns and uniform sampling to support ad-hoc queries. IncApprox [3] uses online stratified sampling to produce an incrementally updated approximate result from streaming data. These frameworks operate directly on the data and compute error bounds without considering data errors.
SampleClean [4], on the other hand, combines sampling and cleaning to improve the accuracy of aggregate queries. A random sample of dirty data is created first, and then a data cleaning technique is applied to clean the sample. Next, the cleaned sample is used to estimate the results of aggregate queries. SampleClean supports closed-form estimates based on normal approximation. A query is formulated first as calculating the mean value so that the confidence interval of the result can be estimated according to the Central Limit Theorem. ActiveClean [5] is an iterative data cleaning framework that cleans a small sample of the data to produce a model similar to if the full data set were cleaned. It starts with a dirty model, then incrementally cleans

a new sample and updates the model. ActiveClean supports convex loss models and uses the model as a guide to identifying future data to clean by cleaning those records likely to affect the results

4. Methodology

4.1 Data Profiling
We first use the data profiler from the OpenClean library. It is a pre-configured tool to profile datasets that help to report some actionable metrics. . We used the default profiler to get basic statistics like min/max, entropy, distinct values, data types, etc. This gave us the basic idea of the dataset before cleaning.

4.2 Sampling
We used the OpenClean inbuilt function to generate small random samples of our big data set to explore the statistical properties of the entire data and define cleaning rules. We calculated the size of the sample using the following parameters:
Sample size - number of rows in the dataset
Confidence interval - 6
Confidence Level - 95%

4.3 Cleaning Process
After Sampling we started with the process of looking for more insights into the data. We started with the first dataset i.e. Historical DOB Permit Issuance Housing & Development dataset. We took this dataset column by column and tried to build a cleaning technique for all the significant columns. These techniques are discussed in detail in the later section. We have created these techniques as generalized as possible. To evaluate these generalized techniques we took 10 more similar datasets and used our cleaning strategies on the common columns within the datasets. We evaluated our results for each column and the cleaning technique to make those cleaning techniques more efficient and robust.

4.4 Initial Cleaning Strategies

Removal of Columns
The initial preprocessing of the dataset starts with analyzing every column for the empty values. The columns with high null values will not help much in analyzing so it is better to remove those columns. We first calculate the percentage of the empty values for every column and if the percentage is more than 70% then we remove those columns.

Following are the columns that we cleaned based on semantic meaning:

Dates
- After looking at various date columns like Filing date, Issuance date, Expiration date, Job start date, and more we realized that the date format contains time which is not an

important factor for analysis of this dataset so we have stripped the time part from every row which contains the date.
- Scalability: After running on other datasets we realized that there are some rows where the Issuance date is before the filing date but this is wrong because the application cannot be issued before it is filed and the application cannot expire before the filing date so our technique is scalable to add this constraint to these columns

Names

- For all the names-related columns we are checking if the name is pure alphabets and removing if any other character occurs in the name. We are removing any white space in and For the people's names, we were able to detect the wrong format of the name with high recall.

Permit Type / Work Type / Permit SubType

- For all the permit and work type-related columns, we are checking if the length of the content is equal to 2 because according to the data dictionary these columns can only have a 2 letter code and hence replacing it with N/A if that type of code is not valid.

Permit Status/ Filing Status:

- We already know the predefined values for these two status columns. So we are just looking in the list of those predefined values and comparing it with the content of the cell. If it doesn't match then we make it N/A.
- Scalability: As we take different datasets we can add the predefined values for these 2 columns in the existing code.

After our initial work on the Historical DOB Permit Issuance dataset, we focused on cleaning a set of attributes related to addresses and scaling this cleaning to other datasets with address data. This set of attributes included borough, street names, state names, and postal codes. Additionally, we examined a domain-specific constraint related to construction job applications. In the following sections, we discuss the problems we encountered when scaling our initial strategies for the above problems to other NYC open datasets. We then discuss the refinements made to our strategies to accommodate these datasets. Finally, we conclude each section with a discussion of the results of these refinements and the new obstacles that arose during the process.

5. <u>Borough</u>

5.1 Applying Initial Strategies

Our strategy was to examine (Latitude, Longitude) -> Borough functional dependency. Then we resolved the conflict of those records which were violating this dependency by finding which borough is the highest occurring(Mode) for each (Latitude, Longitude) pair and updating that value.

5.2 Refinements

The initial strategy will not work if the latitude and longitude are wrong and also if the mode of the pair is the wrong value. So, we used a reference dataset containing the mapping between zip code and borough to clean the Borough column.[1]

5.3 Results and Scalability

When we refined our method to use zip code, precision and recall was higher because the initial method relied on latitude and longitude which were incorrect sometimes. There is, however, one notable exception. The neighborhood of Marble Hill is located off Manhattan Island, uses a Bronx-area zip code, but is legally a part of Manhattan. For addresses in this neighborhood, using zip code alone is not sufficient to determine the borough.

6. <u>Street Names</u>

6.1 Applying Initial Strategies

We saw that there were many street names that were alternate representations of other street names. So using key collision clustering we found groups of different street names that might be alternative representations of the same street. The key collision technique will remove leading and trailing whitespace, convert string to lowercase, normalize string by removing punctuation and control characters, tokenize string by splitting on whitespace characters, sort the tokens and remove duplicates and concatenate remaining (sorted) tokens using a single space character as the delimiter.

Ex: ST NICHOLAS AVE (2451)
ST. NICHOLAS AVE (125)
St Nicholas Ave (23462)
ST, NICHOLAS AVE (1)
ST NICHOLAS AVE (9)
ST NICHOLAS   AVE (1)
ST NICHOLAS AVE (4)
ST. NICHOLAS AVE (1)
Suggested value: St Nicholas Ave

6.2 Refinements

Issues with the clustering method are :

- Different abbreviations for street types, e.g., 125 St vs. 125 Str vs. 125 Street.
- Missing whitespace between street number and street type, e.g, 125 St vs. 125 St.
- Text representations of street numbers, e.g., Fifth Ave vs. 5th Ave vs. 5 Ave.

[1] https://github.com/erikgregorywebb/nyc-housing/blob/master/Data/nyc-zip-codes.csv

To address the issues, the refined code uses the custom USStreetNameKey key generator. The generated clusters in general are significantly larger than the clusters that were generated using the default fingerprint algorithm.

6.3 Results and Scalability

The refined method can be used to form clusters and detect the missing values correctly with better precision and good recall. This method is easily scalable to any dataset as it forms clusters and standardizes the name of the street from the data generator.

7. <u>State</u>

7.1 Applying Initial Strategies

When we did profiling, we saw there were 57 distinct values for the State column. But it is abnormal since the US has only 50 states. So, we used a reference dataset containing US states to clean and remove the data which were not in the reference dataset.

7.2 Refinements

The initial strategy will result in empty values for records that have wrong values. But why leave it empty when we can get the data from the Zip Code? So, we filled the missing values by getting the state code from the zip code since they have a direct relationship.

7.3 Results and Scalability

We can use a Python package called pyzipcode to get State Code based on Zipcode and can be used to verify the state using the zip code. Also, this method is easily scalable as zipcode allows us to get accurate information about the state.

8. <u>Postal Code</u>

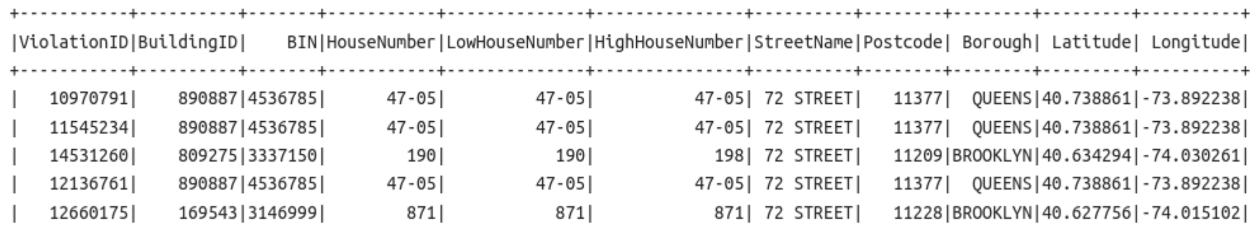
8.1 Applying Initial Strategies

We did not have any initial cleaning strategies for postal codes in the Historical DOB Permit Issuance dataset. However, given how many of our refinement strategies relied on it, the postal code became a valuable target for cleaning.

8.2 Refinements

Reverse geocoding, given latitude and longitude, can potentially yield postal codes. However, some datasets, such as the Housing Maintenance Code Complaints dataset, do not include latitude and longitude. Additionally, those datasets that do include columns for latitude and longitude may have missing geocoding data for some records. The Historical DOB Permit Issuance dataset includes 761 records that are missing both a postal code and geocoding data. The Housing Maintenance Code Violations dataset includes 3758 such records.

For these records, one strategy we considered was to find similar addresses that do have postal codes associated with them. For example, consider a record that is missing a postal code, but includes a house number and street name: 1754 Anthony Ave. We can try to find similar addresses by looking for exact matches on the street name and fuzzy matches on the house number. The reasoning behind this is that houses that are near each other on the same street will have similar house numbers. The numbers generally have the same format. Neighbouring zip code houses will also differ only by a simple increment.

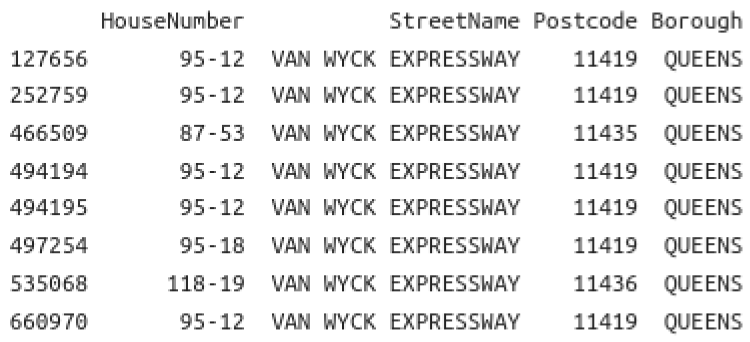


```
+-----------+----------+-------+-----------+--------------+---------------+----------+--------+--------+---------+----------+
|ViolationID|BuildingID|    BIN|HouseNumber|LowHouseNumber|HighHouseNumber|StreetName|Postcode| Borough| Latitude| Longitude|
+-----------+----------+-------+-----------+--------------+---------------+----------+--------+--------+---------+----------+
|   10970791|    890887|4536785|      47-05|         47-05|          47-05| 72 STREET|   11377|  QUEENS|40.738861|-73.892238|
|   11545234|    890887|4536785|      47-05|         47-05|          47-05| 72 STREET|   11377|  QUEENS|40.738861|-73.892238|
|   14531260|    809275|3337150|        190|           190|            198| 72 STREET|   11209|BROOKLYN|40.634294|-74.030261|
|   12136761|    890887|4536785|      47-05|         47-05|          47-05| 72 STREET|   11377|  QUEENS|40.738861|-73.892238|
|   12660175|    169543|3146999|        871|           871|            871| 72 STREET|   11228|BROOKLYN|40.627756|-74.015102|
```

Figure 1: A sample of address records with the same street name

Figure 1 shows a sampling of addresses from the Housing Maintenance Code Violations dataset which all have the street name “72 STREET.” One instance of this street name is located in Queens and the other in Brooklyn. However, the house numbers along each respective street are distinguishable by format. An address on 72nd St that used a dashed house number would most likely be located in Queens, and if the house number is adjacent or near 47-05, it potentially would share a postal code with the three Queens addresses in the figure. To implement this analysis, we could group records by street name (relying on the previous section’s street name standardization) and find clusterings of house numbers in each group.

There are several problems with this approach. One is that the fuzzy matcher would be inexact. A second is that the shape of the geographical areas that share postal codes is not regular. A third is that the strategy would run into problems along the borders between different postal codes. An illustrative example can be seen in Figure 2, in which it is unclear which postal code to choose from among many similar addresses.

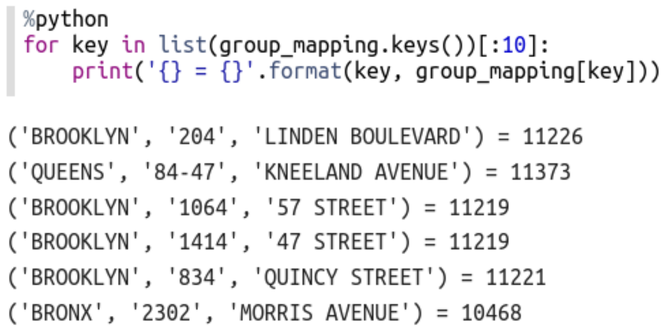


```
       HouseNumber          StreetName Postcode Borough
127656       95-12  VAN WYCK EXPRESSWAY    11419  QUEENS
252759       95-12  VAN WYCK EXPRESSWAY    11419  QUEENS
466509       87-53  VAN WYCK EXPRESSWAY    11435  QUEENS
494194       95-12  VAN WYCK EXPRESSWAY    11419  QUEENS
494195       95-12  VAN WYCK EXPRESSWAY    11419  QUEENS
497254       95-18  VAN WYCK EXPRESSWAY    11419  QUEENS
535068      118-19  VAN WYCK EXPRESSWAY    11436  QUEENS
660970       95-12  VAN WYCK EXPRESSWAY    11419  QUEENS
```

Figure 2: Similar addresses with different postal codes

Given an address like “109-55 VAN WYCK EXPRESSWAY,” the above strategy would most likely choose 11436 as the closest match postal code from the sample in Figure 2. The correct postal code, however, would be 11435. Indeed, all three of the candidate postal codes in the figure border each other.

A safer strategy, then, would be to find exact matches for both street names and house numbers. To deal with situations where two different addresses have exact matches on both street name and house number, we need to augment the data with another attribute that is likely to be distinct per house. We considered using BIN, the permanent building identification number assigned by the DOB. However, in the data sample, we tested with, many records that were missing postal codes were also missing BINs. We ended up using Borough and applying a functional dependency constraint on house number, street name, borough, and postal code. That is, records with the same house number, street name, and borough should have the same postal code.



```
%python
for key in list(group_mapping.keys())[:10]:
    print('{} = {}'.format(key, group_mapping[key]))
```

```
('BROOKLYN', '204', 'LINDEN BOULEVARD') = 11226
('QUEENS', '84-47', 'KNEELAND AVENUE') = 11373
('BROOKLYN', '1064', '57 STREET') = 11219
('BROOKLYN', '1414', '47 STREET') = 11219
('BROOKLYN', '834', 'QUINCY STREET') = 11221
('BRONX', '2302', 'MORRIS AVENUE') = 10468
```

Figure 3: Postal codes found

Figure 3 shows a set of postal codes found for a few records in our sampling of the Housing Maintenance Code Violations data. Originally, there were 378 (non-distinct) records with empty postal codes, and 6 postal codes were found. Upon inspection of the other records, we found that there were not enough matches in the sample to fill in the other postal codes.

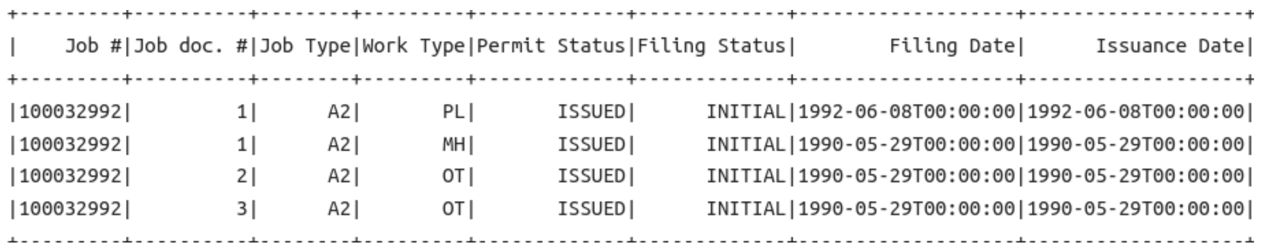
8.3 Scalability

The more data that is available, the more likely it will be that a matching address is found, and hopefully a postal code. Datasets that have a house number, street name, borough, and postal code attributes can be joined together via union. Since we are looking for functional dependency violations between addresses with exact matches on the borough, house number, and street name, we can parallelize the operations by partitioning the dataset on a key comprised of these attributes.

9. Job Number

9.1 Applying Initial Strategies

The job application number constraint is specific to the domain of the DOB and was explained in the data dictionary attached to our initial dataset. The dataset includes the attributes *Job #*, *Job doc. #*, and *Work Type*. *Job #* denotes a unique identifier for a job application submitted to the DOB. Each job application can have multiple documents, each of which has a different *Job doc. #*. A single document can include multiple *Work Types*. Figure 4 shows an example of a single job application in the Historical DOB Permit Issuance dataset.



```
+---------+----------+--------+---------+-------------+-------------+-------------------+-------------------+
|    Job #|Job doc. #|Job Type|Work Type|Permit Status|Filing Status|        Filing Date|      Issuance Date|
+---------+----------+--------+---------+-------------+-------------+-------------------+-------------------+
|100032992|         1|      A2|       PL|       ISSUED|      INITIAL|1992-06-08T00:00:00|1992-06-08T00:00:00|
|100032992|         1|      A2|       MH|       ISSUED|      INITIAL|1990-05-29T00:00:00|1990-05-29T00:00:00|
|100032992|         2|      A2|       OT|       ISSUED|      INITIAL|1990-05-29T00:00:00|1990-05-29T00:00:00|
|100032992|         3|      A2|       OT|       ISSUED|      INITIAL|1990-05-29T00:00:00|1990-05-29T00:00:00|
+---------+----------+--------+---------+-------------+-------------+-------------------+-------------------+
```

Figure 4: A sample job application

In this example, the job application number is 100032992. The application contains three documents, numbered from 1 to 3. Document 1 includes two work types: PL and MH. Document 2 and 3 each only include the OT work type. Each job application must have at least one document. The document numbers must start at 1, and each additional document should just increment the document number. In other words, the first document must have a *Job doc. #* of 1, the second document 2, the third document 3, and so on.

The query used to identify violations of this constraint is complex to express but conceptually simple. We identify all the distinct *(Job #, Job doc. #)* tuples in the dataset. We group these tuples by *Job #* and count the number of tuples in each group. This count is the number of documents found within that group's job application. Next, we identify the max *Job doc. #* in each group. This max value should match the count of documents in that group. Otherwise, that group violates the constraint.

We used Spark SQL to express this query using SQL syntax and output a file containing the job application numbers that violate this constraint. In our cleaning pass, we read the file into our OpenClean environment and remove these job applications from the dataset, so that our data is not polluted with incomplete job applications.

9.2 Refinements

Since it is straightforward to find violations of this constraint, we did not need to change the query to accommodate other datasets. However, given how domain-specific the constraint is, we could only test it on other datasets in the same domain and that include attributes for job application numbers and document numbers. We identified two other datasets that have the required attributes and are large enough to be interesting: DOB Permit Issuance and DOB Job Application Filings. For this cleaning task, we were more interested in performance than inaccuracy.

The query took around 55 seconds to run over the Historical DOB Permit Issuance dataset (~2.45M records), around 60 seconds to run over the DOB Permit Issuance dataset (~3.78M records), and around 45 seconds to run over the DOB Job Application Filings dataset (~1.77M records).

After converting the Spark SQL Dataframes program into an equivalent program using Spark RDDs, the runtime for each dataset improved to 29 seconds, 39 seconds, and 25 seconds respectively.

9.3 Scalability

This task is a good candidate for parallelization, as each job application is independent of the others. To have better control over the partitioning of the dataset in Spark, we can ensure that the data is structured as a keyed tuple, where the key is the job application number. We can also use Spark RDD's repartitioning mechanism. Since the job application number is numerical, we can use a range partitioner to ensure that each partition has a roughly equal load.

10. Results and Discussion

In this section, we discuss the performance of our cleaning strategies that we have applied on multiple datasets to make our techniques more efficient and scalable.

One of the recurring issues in data analytics is detecting and fixing dirty data, and failing to do so can result in skewed analyses and unreliable judgments. We have proposed generic cleaning algorithms in this research that can effectively clean data on corresponding columns. Our methods centered on qualitative data cleansing, which detects problems using restrictions, rules, or patterns.

Column	Runtime(s)	Wrong Records Found
Borough	104	2850
State	105	1266
Street Names	110	56287
Job Number	60	6950
Postal Code	100	180

Table 1: Results of processing on Historical DOB Permit Issuance Dataset

	Precision	Recall
Original Cleaning Method	0.82	0.323
Refined Cleaning Method	0.875	0.82

Table 2 : Metrics

Current limitations of our methods include:

- Different abbreviations for street types, e.g., 125 St vs. 125 Str vs. 125 Street.
- Missing whitespace between street number and street type, e.g, 125St vs. 125 St.
- Text representations of street numbers, e.g., Fifth Ave vs. 5th Ave vs. 5 Ave.
- Not enough matches in the sample to find postal code based on the similar address if latitude and longitude are not present
- Making the assumption that zip code is correct

11. **References:**

[1] Goiri Í, Bianchini R, Nagarakatte S, Nguyen TD. Approxhadoop: bringing approximations to MapReduce frameworks. In: Proceedings of the twentieth international conference on architectural support for programming languages and operating systems. ACM; 2015. p. 383–97.

[2]Agarwal S, Mozafari B, Panda A, Milner H, Madden S, Stoica, I. Blinkdb: queries with bounded errors and bounded response times on very large data. In: Proceedings of the 8th ACM European Conference on Computer Systems. ACM; 2013, p. 29–42.

[3]Krishnan DR, Quoc DL, Bhatotia P, Fetzer C, Rodrigues R. Incapprox: A data analytics system for incremental approximate computing. In: Proceedings of the 25th International Conference on World Wide Web. WWW '16. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland; 2016, p. 1133–44. https://doi.org/10.1145/2872427.2883026.

[4] Krishnan S, Wang J, Franklin MJ, Goldberg K, Kraska T, Milo T, Wu E. Sampleclean: fast and reliable analytics on dirty data. IEEE Data Eng Bull. 2015;38(3):59–75.

[5]Krishnan S, Wang J, Wu E, Franklin MJ, Goldberg K. Activeclean: interactive data cleaning for statistical modeling. Proc VLDB Endow. 2016;9(12):948–59. https://doi.org/10.14778/2994509.2994514.