Big data Fall 2021 Project - Group 3

XIAOZHE WANG(WX2083)*, JINGXUAN FENG(JF4179), and GUANLIANG YAO(GY2023)

Additional Key Words and Phrases: NYC open data, data cleaning, borough, date, city

ACM Reference Format:
Xiaozhe Wang(wx2083), Jingxuan Feng(jf4179), and Guanliang Yao(gy2023). 2021. Big data Fall 2021 Project - Group 3. 1, 1 (December 2021), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

With the rapid development of information technology, more and more types of data begin to appear in people's lives, which means more and more difficult data problems occur, which can cause a significant obstacle to the work of data analysts. Diverse data cleaning and data profiling techniques are essential for improving data processing efficiency. In this project, we start by processing different types of data in a typical dataset, analyze their data structures, and derive a more efficient data processing strategy, then evaluate and improve the efficiency of this strategy by processing data in other highly similar datasets and finally obtain a generalized data processing strategy.
This project implements and demonstrates more efficient and diverse data cleaning techniques. In this project, we use the DOB Job Application Filings dataset as our typical dataset and use the similarity query to get the dataset with high similarity and complete the analysis.

2 CLEANING BOROUGH DATA

2.1 Column Introduction

Borough, a town or district which is an administrative unit, is a division of the country's regions for hierarchical management. Our selected dataset consists mainly of the five boroughs of Brooklyn, Bronx, Queens, Staten Island, and Manhatten in New York City, and the Datatype of Borough is Text type.

2.2 Work of Part1

2.2.1 Original Dataset Description. Our original dataset is 'DOB Job Application Filings.' This dataset contains all job applications submitted through the Borough Offices; it has 1.77M rows and 96 columns in different data types. As a result, we choose the column 'Borough' in 'Text'data type to forward data cleaning. And the data format in 'Borough' is like MANHATTEN.

*All authors contributed equally to this research.
Github Repository: https://github.com/JungFF/Big_Data_Report

Authors' names: Xiaozhe Wang(wx2083); Jingxuan Feng(jf4179); Guanliang Yao(gy2023).

2.2.2 Original Problem Formulation. Since the data type of column 'Borough' is text and the value of this column is limited to five values: BROOKLYN, BRONX, QUEENS, STATEN ISLAND, and MANHATTEN, there may exist some problems in our original dataset:

- Borough value of some data in the dataset is empty;
- The Borough values of some data in the dataset do not correspond to the case.

2.2.3 Original Related Work (Our Original Strategy). In our original dataset, 'DOB Job Application Filings,' we only found dirty data with null borough values and letter case errors, so we used our initial strategy. We choose two strategies to solve the two problems mentioned above, just like the screenshot below. Because there are null values in borough, null values are illegal data in borough. So we replace all null values in borough with 'UNKNOWN' by writing functions and lambda expressions. At last, we update them to the original data.

- Since there is a problem of case error in the values in the column 'borough', we change the spelling of all data values in the column 'borough' to all uppercase and update the original data.
- Finally, we use the precision formula to calculate the precision and call of the original strategy and use value_counts() to check the data cleaning results. We conclude that our data cleaning strategy for this dataset is successful.

Apply Original Method

```
# For our original method, we firstly use value_counts method to check if there is any empty value,
# then we repleace them with 'UNKNOWN'.
# Then we make all the values in this column uppercase
def getNoneIdx(df, column):
    res = set()
    idx = 0
    for name in df[column]:
        if name is None or (len(name) == 0):
            res.add(idx)
        idx += 1
    return res


def original_cleaning_method(df, column):
    mask = df[column] != df[column].str.upper()
    upper_rows_index = set(df.loc[mask].index.to_list())
    df = update(df, columns=column, func=str.upper)
    empty_rows_index = getNoneIdx(df, column)
    df = update(df, columns=column, func=lambda x: 'UNKNOWN' if is_empty(x) else x)
    affected_count = len(upper_rows_index.union(empty_rows_index))
    return df, affected_count

def calc_precision_recall(remain_problems_rows_count, cleaned_rows_count):
    total = remain_problems_rows_count + cleaned_rows_count
    precision = (cleaned_rows_count - remain_problems_rows_count) / cleaned_rows_count if cleaned_rows_count != 0 else
    recall = (cleaned_rows_count - remain_problems_rows_count) / total if total != 0 else 0.0
    print(f'The precision value should be {precision} and the recall value should be {recall}')

(df_original_clean, count) = original_cleaning_method(df, 'BOROUGH')
```

Fig. 1. Original strategy of 'Borough'

2.3 Work of Part2

2.3.1 New Datasets Description. After original strategy processing, we obtained nine datasets that contain both column 'Borough' and have high similarity to the data in the original dataset 'DOB Job Application Filings' by querying NYC

Open data. All nine datasets obtain the same column Borough and have the same 'Text' data type. The nine similar datasets and their different Borough data formats are listed below:

- DOB_Certificate_Of_Occupancy : Manhatten
- DOB_NOW__Build___Approved_Permits : MANHATTEN
- DOB_Stalled_Construction_Sites: Manhatten
- Interagency_Coordination_and_Construction_Permits_Data__MOSYS_ : Manhatten
- Active_Projects_Under_Construction : MANHATTEN
- FY19_BID_Trends_Report_Data : MN
- DOB_NOW__Electrical_Permit_Applications : MANHATTEN
- DOB_NOW__Safety___Facades_Compliance_Filings : MANHATTEN
- DOB_Sign_Application_Filings : MANHATTEN

2.3.2 New Problem Formulation. Since the data in column Borough is 'Text,' the data format in column Borough is different in each dataset. So there will occur a lot of other problems.
For all nine different new datasets, we extract different 'Borough' columns from nine different datasets and merge them into a new data frame, which contains data from all 'Borough' columns in the nine datasets.
Then, we use our original strategy to clean the new dataset. We firstly replace all null values in the borough with 'UNKNOWN' by writing functions and lambda expressions. And we also change the spelling of all data values in the borough to all uppercase. At last, we calculate the value_counts() of the new data frame and show all different data values. We can easily differentiate the outlier data.
Like the screenshot below, there are many outliers like 'MN,' 'MANHATTAN;#QUEENS' and 'K.' So we need a new strategy to modify all the outliers, which our original strategy can't change.

```
df_original_clean['BOROUGH'].value_counts()
```

BROOKLYN	581124
QUEENS	478945
MANHATTAN	443935
BRONX	177071
STATEN ISLAND	133217
STATEN IS	393
BROOKLYN	355
QUEENS	335
BRONX	206
K	183
Q	182
X	98
M	93
R	51
MN	25
BK	23
BRONX;#BROOKLYN;#MANHATTAN;#QUEENS;#STATEN ISLAND	14
QN	13
UNKNOWN	13
BX	10
MANHATTAN;#QUEENS	7
SI	4
STATEN ISLAND;#QUEENS;#MANHATTAN;#BROOKLYN;#BRONX	4
BRONX;#MANHATTAN	4
MANHATTAN;#BRONX	3
QUEENS;#BROOKLYN	3
BROOKLYN;#STATEN ISLAND	3
MANHATTAN;#BROOKLYN	3
BRONX;#BROOKLYN;#MANHATTAN;#QUEENS	3
BRONX;#QUEENS	3
BROOKLYN;#MANHATTAN	3
QUEENS;#BRONX	3

Fig. 2. Result of original strategy

The problems are listed below:

- Borough values of some data in the dataset contain spaces;
- Borough values of some data in the dataset are abbreviated;
- Borough values' spellings of some data in the dataset are wrong;
- Borough values of some data in the dataset do not meet the requirements (i.e., they contain multiple Boroughs at the same time and contain illegal punctuation marks like '#')

2.3.3 New Related Work (Our New Strategy). By doing value_counts() on the new data frame, we found much dirty data with the wrong borough data format. Using our original strategy could not modify the different format errors, so we decided to use some more format modification strategies to keep the format uniform and use the KNN algorithm to merge similar values.

2.4 Methods, architecture and design

2.4.1 Load datasets. For all nine different new datasets, we extract different 'Borough' columns from nine different datasets and merge them into a new data frame, which contains data from all 'Borough' columns in the nine datasets. Furthermore, we will use this data frame to design our new strategy. If we could get the correct result by using this new data frame and new strategy, we then modify all nine datasets by using this strategy.

2.4.2 New strategy design. To solve the problems mentioned above, we need to figure out new strategies. Our design for the different four problems comes out easily:

- To modify the 'Borough' values containing multiple 'Borough' values and some illegal punctuation marks like '#,' we choose to change them to 'UNKNOWN' because we can't make sure which borough name they stand for.
- To modify the 'Borough' values are the abbreviations of the borough names; we create a mapper based on NYC Open Data to make one-to-one targeted changes to these abbreviations, such as changing BK to BROOKLYN.
- To modify the 'Borough' values that contain spaces and have wrong spellings, we decide to use the KNN algorithm to analyze their similarity and merge the data that match the similarity factor [5]. For example, merge 'BROOKLYN' with 'BROOKLYN' and 'STATEN IS' with 'STATEN ISLAND.'

2.4.3 Methods Implementation.

- We first perform the value_counts() method on the newly obtained data frame, and according to the result, we can see that many data are containing multiple borough names at the same time and many data containing illegal symbols '#.' Then we use a lambda expression to traverse the entire data frame. If the data includes an illegal character '#' in a line, we change the data in that line to 'UNKNOWN.' Then we could use unique() to see all different values in the data frame now. So we could get a result like a screenshot below to figure out that all data containing multiple 'borough' names and '#' simultaneously are converted to 'UNKNOWN.'
- Next, we perform the KNN algorithm to detect spelling and space errors in the data frame. We determine the cluster size to limit the number of clusters and the similar factor to get the similarity. Then we run the KNN algorithm and get similar results and suggested values. Just like the screenshot shown below.
- Then, we change the value in the data frame to the suggested value according to the KNN algorithm result, including spelling and space errors.
- At last, We noticed that abbreviations for the borough, such as 'Q' (for 'QUEENS'), 'M' (for 'MANHATTAN'), and 'BK' (for 'BROOKLYN'), were also included in the data frame. Then we create a mapper to map the unformatted

```
# We firstly convert all words containing # to unknown
df_clean = update(df_clean, columns='BOROUGH', func=lambda x : 'UNKNOWN' if '#' in x else x)
```

```
df_clean['BOROUGH'].unique()
```

```
array(['QUEENS', 'BROOKLYN', 'STATEN ISLAND', 'MANHATTAN', 'BRONX',
       'UNKNOWN', 'BROOKLYN ', 'QUEENS   ', 'BRONX    ', 'STATEN IS', 'Q',
       'K', 'R', 'X', 'M', 'MN', 'BX', 'QN', 'BK', 'SI'], dtype=object)
```

Fig. 3. Result of new strategy I (Borough)

```
def print_cluster(cnumber, cluster):
    print('Cluster {} (of size {})\n'.format(cnumber, len(cluster)))
    for val, count in cluster.items():
        print('{} ({})'.format(val, count))
    print('\nSuggested value: {}\n\n'.format(cluster.suggestion()))

# Minimum cluster size. Use ten as default (to limit
# the number of clusters that are printed in the next cell).
def run_knn(df, column, t=0.5, minsize = 2):
    dba = df.select(column).distinct()
    clusters = knn_clusters(
        values=dba,
        sim=SimilarityConstraint(func=LevenshteinDistance(), pred=GreaterThan(t)),
        minsize=minsize
    )
    print('{} clusters of size {} or greater'.format(len(clusters), minsize))
    # Sort clusters by decreasing number of distinct values.
    clusters.sort(key=lambda c: len(c), reverse=True)
    for i, cluster in enumerate(clusters):
        print_cluster(i + 1, cluster)

run_knn(df, 'BOROUGH')
```

```
BROOKLYN  (355)
BROOKLYN (144814)

Suggested value: BROOKLYN


Cluster 38 (of size 2)

QUEENS    (335)
QUEENS (111171)

Suggested value: QUEENS


Cluster 39 (of size 2)

STATEN IS (393)
STATEN ISLAND (27847)
```

Fig. 4. Result of new strategy II (Borough)

value obtained from the initial dataset to the formatted reference data. Including the KNN suggested value 'STATEN IS'(for 'STATEN ISLAND'), we could quickly get the final data cleaning results that contain the correct answer. Just like the screenshot below.

```
mapper = {'K': 'BROOKLYN', 'Q': 'QUEENS', 'X': 'BRONX', 'M': 'MANHATTAN', 'R': 'STATEN ISLAND', 'MN': 'MANHATTAN',
          'BK': 'BROOKLYN', 'STATEN IS': 'STATEN ISLAND', 'QN': 'QUEENS', 'BX': 'BRONX', 'SI': 'STATEN ISLAND'}
df_clean = update(df_clean, columns='BOROUGH', func=lambda x : mapper.get(str(x)) if x in mapper else x)
```

```
df_clean.value_counts()
```

```
BOROUGH
BROOKLYN         581685
QUEENS           479475
MANHATTAN        444053
BRONX            177385
STATEN ISLAND    133665
UNKNOWN              94
dtype: int64
```

Fig. 5. Result of new strategy III (Borough)

2.4.4 Encapsulate the new strategy to a function. We encapsulate our advanced method in a function and apply this function to all nine different datasets to see the results. We do the data cleaning step by step. Change the null value to 'UNKNOWN.'

- Merge similar data by deleting space and modifying spelling errors
- Convert letters to uppercase
- Change the data contains '#' to 'UNKNOWN'
- Create a mapper between the abbreviation and change them one by one

```
# Now we encapsulate our refined method in a function
def refined_method(df, column):
    mapper = {'K': 'BROOKLYN', 'Q': 'QUEENS', 'X': 'BRONX', 'M': 'MANHATTAN', 'R': 'STATEN ISLAND', 'MN': 'MANHATTAN',
    df = update(df, columns=column, func=lambda x: 'UNKNOWN' if is_empty(x) else x)
    df = update(df, columns=column, func=lambda x: str(x).strip())
    df = update(df, columns=column, func=str.upper)
    df = update(df, columns=column, func=lambda x : 'UNKNOWN' if '#' in x else x)
    df = update(df, columns=column, func=lambda x : mapper.get(str(x)) if x in mapper else x)
    df = update(df, columns=column, func=lambda x : 'UNKNOWN' if x == 'NAN' else x)
    return df
```

```
BOROUGH
BROOKLYN         581685
QUEENS           479475
MANHATTAN        444053
BRONX            177385
STATEN ISLAND    133665
UNKNOWN              94
dtype: int64
```

Fig. 6. Code and result of new strategy (Borough)

2.5 Results

2.5.1 Results of our refined strategy. In this part, we apply this new strategy to evaluate all datasets and generate the Pie Chart of the result. If the result's value counts and the Pie Charts' labels only have six BROOKLYN, QUEENS, BRONX, STATEN ISLAND, MANHATTEN and UNKNOWN. We could conclude that this strategy is successful and can be applied to all datasets which contain the column 'Borough.'

- For dataset DOB_Certificate_Of_Occupancy, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
df[columns[0]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

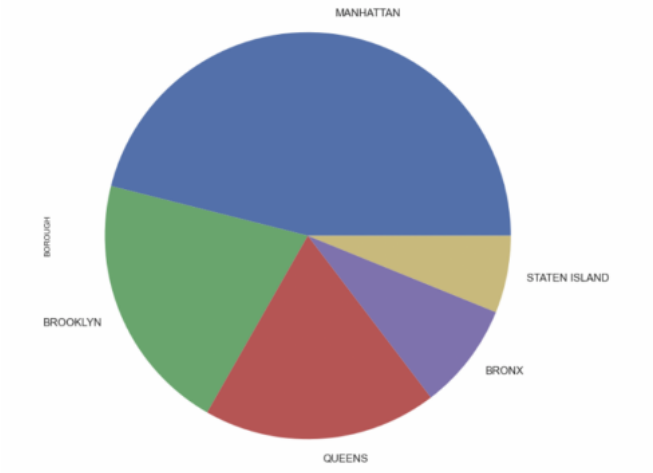

Fig. 7. Code and result of dataset DOB_Certificate_Of_Occupancy (Borough)

- For dataset DOB_NOW__Build___Approved_Permits, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
# After cleaning the data, show the visualization.
df[columns[1]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

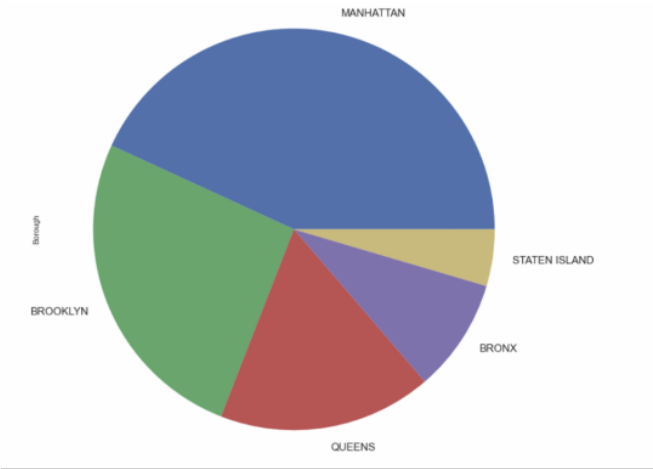

Fig. 8. Code and result of dataset DOB_NOW__Build___Approved (Borough)

- For dataset DOB_Stalled_Construction_Sites, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
# After cleaning the data, show the visualization.
df[columns[2]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

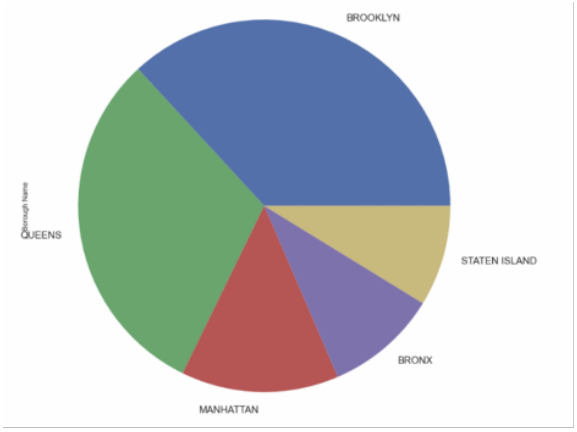

Fig. 9. Code and result of dataset DOB_Stalled_Construction_Sites (Borough)

- For dataset Interagency Coordination and Construction Permits Data MOSYS_, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
# After cleaning the data, show the visualization.
df[columns[3]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

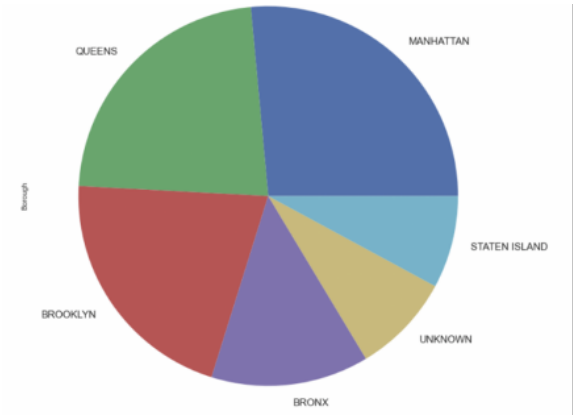

Fig. 10. Code and result of dataset Interagency Coordination and Construction Permits Data MOSYS_ (Borough)

- For dataset Active_Projects_Under_Construction, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
# After cleaning the data, show the visualization.
df[columns[4]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

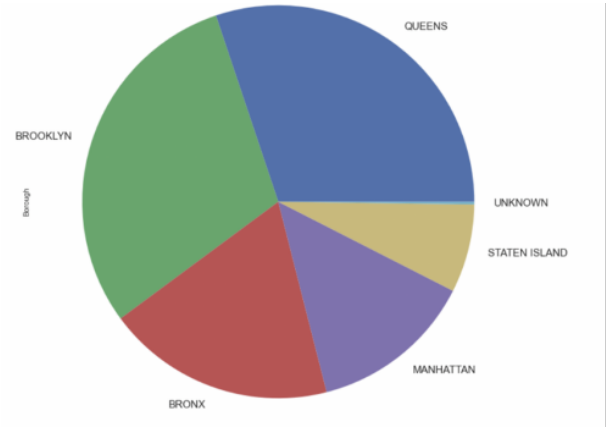

Fig. 11. Code and result of dataset Active_Projects_Under_Construction (Borough)

- For dataset FY19_BID_Trends_Report_Data, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
# After cleaning the data, show the visualization.
df[columns[5]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

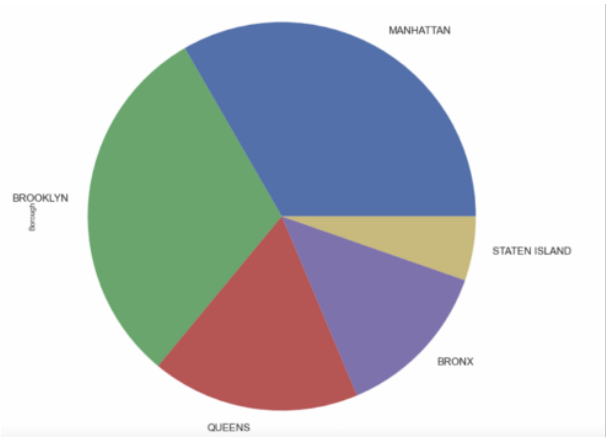

Fig. 12. Code and result of dataset FY19_BID_Trends_Report_Data (Borough)

- For dataset DOB_NOW__Electrical_Permit_Applications, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
# After cleaning the data, show the visualization.
df[columns[6]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

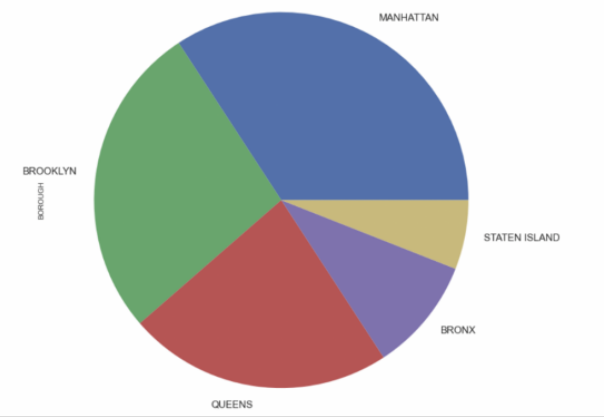

Fig. 13. Code and result of dataset DOB_NOW__Electrical_Permit_Applications (Borough)

- For dataset DOB_NOW__Safety___Facades_Compliance_Filings, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
# After cleaning the data, show the visualization.
df[columns[7]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

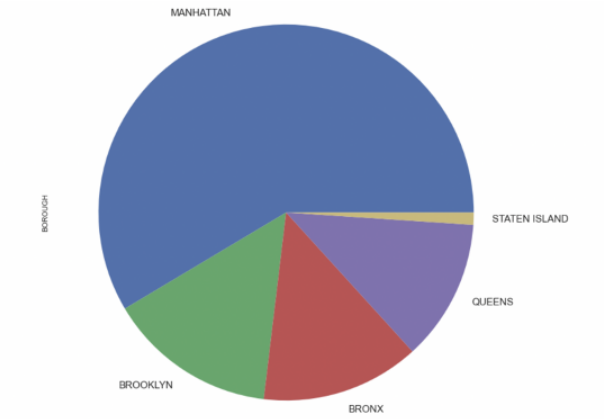

Fig. 14. Code and result of dataset DOB_NOW__Safety___Facades_Compliance_Filings (Borough)

- For dataset DOB_Sign_Application_Filings, we apply the refined_method and then Draw the value_counts() results into a pie chart, then we find that the result and the pie chart are both correct.

```
# After cleaning the data, show the visualization.
df[columns[8]].value_counts().plot(kind='pie', figsize=(15, 13), fontsize=15)
plt.show()
```

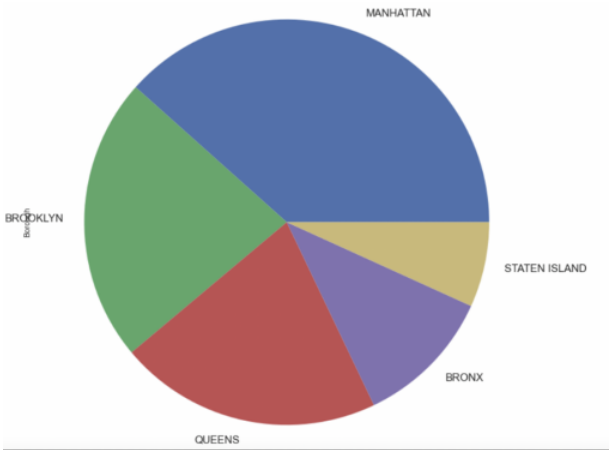

Fig. 15. Code and result of dataset DOB_Sign_Application_Filings (Borough)

2.5.2 Conclusion. From all the pie charts and results shown above, we could draw a conclusion that the new strategy we designed for the column 'Borough' is right, and we could apply this strategy to all datasets.

3 CLEANING DATE DATA

3.1 Column Introduction

Date-related data is prevalent in any dataset, so it is critical to clean it well. In this part, we apply our strategy to other datasets to validate and improve our original strategy. The final output is a refined method applicable to as many datasets as possible.

3.2 Work of Part1

3.2.1 Original Problem formulation. Our original dataset is DOB Job Application Filings, the date format in this dataset is MM/DD/YYYY. Potential problems are empty data, incorrectly formatted data such as MM-DD-YYYY, YYYY/MM/DD.

3.2.2 Original related work (Original strategy). Our original strategy is to use a regex to filter the data of this date format. Match all date data with regex ("$^(0[1-9] \mid 1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.](19 \mid 20)\d\d\$$") to find all rows that do not match this regex. The date data that do not match regex is only 1/10,000 of the total, and we choose to remove them from the original dataset.

- Before cleaning:

```
10/13/2017     792
05/18/2017     774
01/25/2017     744
02/12/2018     739
11/30/2016     734
               ...
06/03/2017       1
2018-10-03       1
2018-05-07       1
2016-12-13       1
12/15/2007       1
Name: Latest Action Date, Length: 7371, dtype: int64
```

Fig. 16. Raw data of 'Date'

- After cleaning:

```
mask = ds["Latest Action Date"].str.match("^(0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])[- /.](19|20)\d\d$")
```

```
ds = ds.loc[mask, :]
```

```
ds["Latest Action Date"].value_counts()
```

```
10/13/2017    792
05/18/2017    774
01/25/2017    744
02/12/2018    739
11/30/2016    734
              ...
03/21/2015      1
04/04/2015      1
01/18/2015      1
01/10/2015      1
12/15/2007      1
Name: Latest Action Date, Length: 6990, dtype: int64
```

Fig. 17. Original strategy of 'Date'

3.3 Work of Part2

3.3.1 New Problem formulation. Apply original strategy on other datasets.

We randomly selected three other datasets on the NYC Open Data website that contain the date column. We found that they have different date formats:

- DOB ECB Violations: 20180607
- DOB Complaints Received: 01/04/1989
- DOB NOW: Build — Approved Permits: 01/04/1989 12:17:00 PM

Only date data in DOB Complaints Received have the same format as our original dataset. We cannot apply the original strategy to the other two datasets.

3.3.2 New related work (New strategy). We found the date format of the new dataset to be representative. The date formats of other datasets are also basically the same as the above three datasets. Our new strategy is to modify the original regex and increase its matching range to all three date formats [4]. We will discuss it in detail in the next section.

3.4 Methods, architecture and design

```
pd.set_option('display.max_rows', 300)
pd.set_option('display.min_rows', 300)
root = '../Dataset/'

files = [
    'Date/DOB_NOW__Build___Approved_Permits.csv',
    'Date/DOB_Complaints_Received.csv',
    'Date/DOB_Violations.csv'
]

#Data column of Borough in different files
columns = [
    'Approved Date',
    'Disposition Date',
    'ISSUE_DATE'
]
```

Current Profile

```
# add all date columns to one dataframe, then generate the data profile
df = pd.DataFrame(columns=['Date'])
for file in files:
    path = root + file
    ds = dataset(path, encoding='utf-8')
    print("Loading... ", file)
    # rename all date cols to Date
    ds.rename(columns={'Approved Date': 'Date', 'Disposition Date': 'Date', 'ISSUE_DATE': 'Date'}, inplace=True)
    date = ds[['Date']]
    df = df.append(date)
print(len(df))
```

```
Loading...  Date/DOB_NOW__Build___Approved_Permits.csv
Loading...  Date/DOB_Complaints_Received.csv
Loading...  Date/DOB_Violations.csv
5487839
```

Fig. 18. Load all data of three datasets

3.4.1 Extract the date columns from three datasets and put them together into a new column (hereinafter called new_dataframe) for testing the refined strategy later.

3.4.2 Write corresponding regexes for each of the three date formats and compose them into a complete regex.

- Format1: 20180607
  Regex1: \d4(?:0[1-9]|1[0-2])(?:0[1-9]|[12][0-9]|3[01])
- Format2: 01/04/1989
  Regex2: (?:0[1-9]|[12][0-9]|3[01])/(?:0[1-9]|1[0-2])/\d4
- Format3: 01/04/1989 12:17:00 PM
  Regex3: (?:0[1-9]|1[0-2])/(?:0[1-9]|[12][0-9]|3[01])/\d4 (?:0[1-9]|1[0-2])(?::[0-5][0-9])2 [PA]M

Compose these 3 regexes into a complete regex:
\d4(?:0[1-9]|1[0-2])(?:0[1-9]|[12][0-9]|3[01])| (?:0[1-9]|[12][0-9]|3[01])/(?:0[1-9]|1[0-2])/\d4|(?:0[1-9]|1[0-2])/(?:0[1-9]|[12][0-9]|3[01])/\d4 (?:0[1-9]|1[0-2])(?::[0-5][0-9])2 [PA]M

3.4.3 Apply refined strategy on new_dataframe to verify it.

```
mask1 = df_clean.Date.str.match(reg)
mask2 = df_clean.Date == 'UNKNOWN'
mask = mask1 | mask2
```

```
df_clean.loc[~mask]
```

	Date
3279544	19505536
3520715	0000207
3600249	0306
3824849	000000
4062842	0808
4190778	0519
4541241	18110561
4594776	20067328
4633455	1230
4658283	0808
4752243	0 0612
4763349	0808
4766648	0808
4925158	0320
5017969	Y9990120
5018139	1016
5091190	T90517
5186331	33218262
5188814	10772841
5257103	0 1224
5277861	19108124
5301505	Y30819
5396400	10809293

Fig. 19. Code and result of refined strategy (Date)

3.4.4 *Apply refined strategy separately on every dataset and find the content outlier by drawing pictures.*

- Refined strategy works on DOB NOW: Build — Approved Permits dataset

For dataset DOB_NOW__Build___Approved_Permits.csv

```
df = pd.read_csv(root + files[0], low_memory=False)
```

```
# Apply refined method to this dataset
df = refined_method(df, columns[0])
```

```
# Extract years, months and days
years = []
months = []
days = []
for date in df[columns[0]]:
    if date == 'UNKNOWN':
        continue
    MDY = date.split()[0]
    strs = MDY.split('/')
    month = int(strs[0])
    day = int(strs[1])
    year = int(strs[2])
    months.append(month)
    days.append(day)
    years.append(year)
```

Fig. 20. Code of dataset DOB NOW: Build — Approved Permits (Date)

```
# After cleaning the data, show the years visualization.
draw(years, 'Year', 'Frequency', 'Year-Frequency')
```

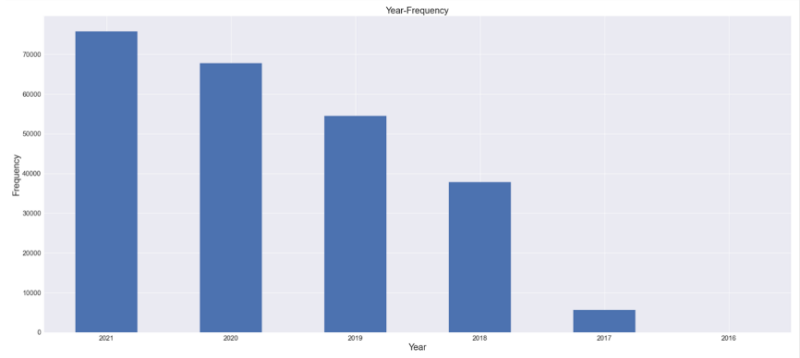

Fig. 21. Code of dataset DOB NOW: Build — Approved Permits (Date)

```
# After cleaning the data, show the days visualization.
draw(days, 'Day', 'Frequency', 'Day-Frequency')
```

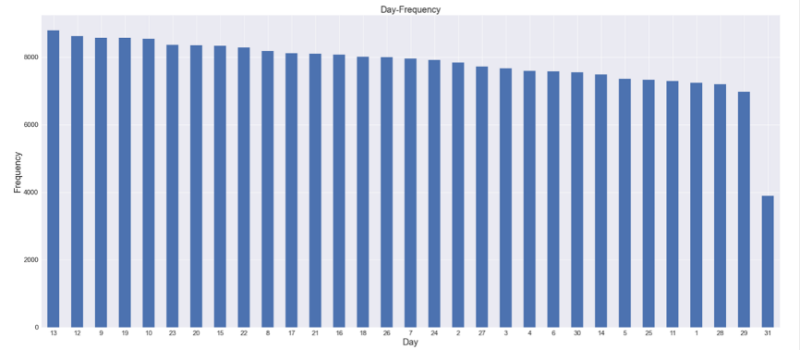

Fig. 22. Day result of dataset DOB NOW: Build — Approved Permits (Date)

```
# After cleaning the data, show the months visualization.
draw(months, 'Month', 'Frequency', 'Month-Frequency')
```

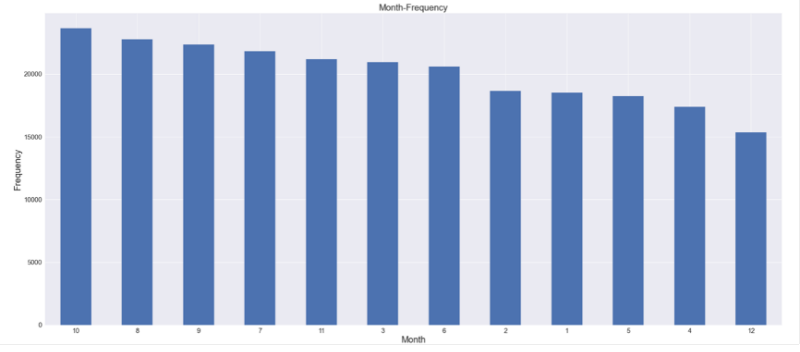

Fig. 23. Month result of dataset DOB NOW: Build — Approved Permits (Date)

- Refined strategy works on DOB Complaints Received

For dataset DOB_Complaints_Received.csv

```
df = pd.read_csv(root + files[1], low_memory=False)

# Apply refined method to this dataset
df = refined_method(df, columns[1])

# Extract years, months and days
years = []
months = []
days = []
for date in df[columns[1]]:
    if date == 'UNKNOWN':
        continue
    strs = date.split('/')
    month = int(strs[0])
    day = int(strs[1])
    year = int(strs[2])
    months.append(month)
    days.append(day)
    years.append(year)
```

Fig. 24. Code of dataset DOB Complaints Received (Date)

```
# After cleaning the data, show the years visualization.
draw(years, 'Year', 'Frequency', 'Year-Frequency')
```

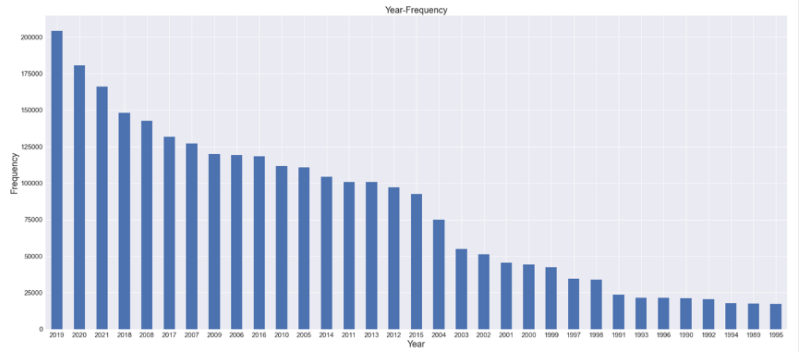

Fig. 25. Year result of dataset DOB Complaints Received (Date)

```
# After cleaning the data, show the days visualization.
draw(days, 'Day', 'Frequency', 'Day-Frequency')
```

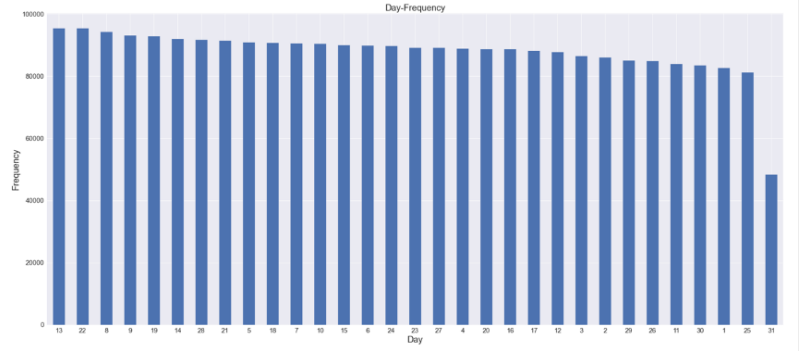

Fig. 26. Day result of dataset DOB Complaints Received (Date)

```
# After cleaning the data, show the months visualization.
draw(months, 'Month', 'Frequency', 'Month-Frequency')
```

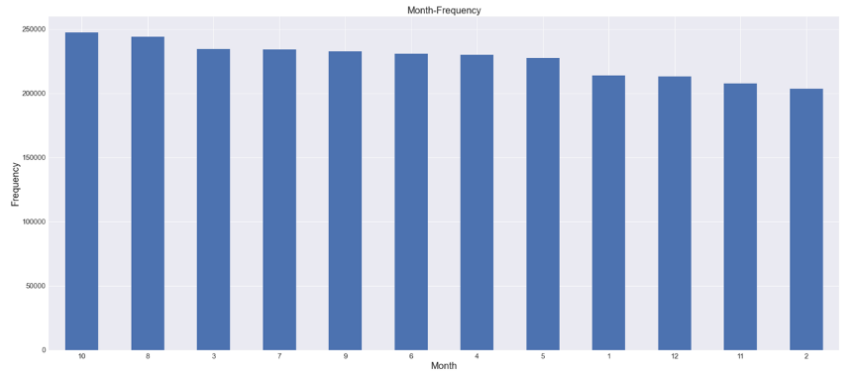

Fig. 27. Month result of dataset DOB Complaints Received (Date)

- Refined strategy not works on DOB ECB Violations dataset

For dataset DOB_Violations.csv

```
df = pd.read_csv(root + files[2], low_memory=False)
```

```
# Apply refined method to this dataset
df = refined_method(df, columns[2])
```

```
# Extract years, months and days
years = []
months = []
days = []
for date in df[columns[2]]:
    if date == 'UNKNOWN':
        continue
    year = date[0:4]
    month = date[4:6]
    day = date[6:]
    months.append(month)
    days.append(day)
    years.append(year)
```

Fig. 28. Code of dataset DOB ECB Violations (Date)

We can find from the figure that this dataset has many unsuccessfully processed outliers (such as 0996, 1012, 1014)

```
# After cleaning the data, show the years visualization.
draw(years, 'Year', 'Frequency', 'Year-Frequency', figsize=(50, 15))
```

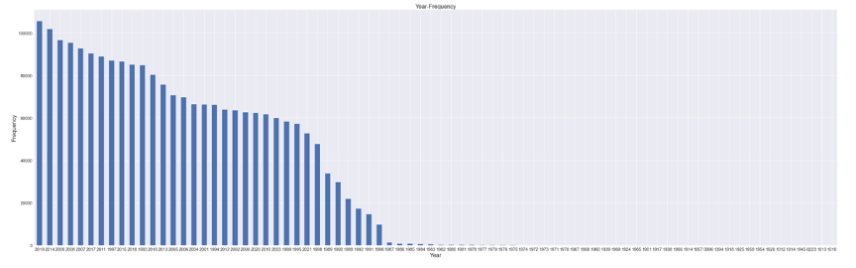

Fig. 29. Year result of dataset DOB ECB Violations (Date)

Fig. 30. Wrong year result of dataset DOB ECB Violations (Date)

3.4.5 Modify the refined strategy for outliers to get the final refined method. With the previous figure, we can see that the current regex cannot filter illegal dates which have illegal years, such as 0996, 1012, and 1014. In the new regex, we want to handle this case further. Modify the regex so that all dates outside the year 1900-2029 can be filtered

- Format1: 20180607
  New regex1: (?:19[0-9][0-9]|20[0-2][0-9])(?:0[1-9]|1[0-2])(?:0[1-9]|[12][0-9]|3[01])
- Format2: 01/04/1989
  New regex2: (?:0[1-9]|1[0-2])/(?:0[1-9]|[12][0-9]|3[01])/(?:19[0-9][0-9]|20[0-2][0-9])
- Format3: 01/04/1989 12:17:00 PM
  New regex3: (?:0[1-9]|1[0-2])/(?:0[1-9]|[12][0-9]|3[01])/(?:19[0-9][0-9]|20[0-2][0-9]) (?:0[1-9]|1[0-2])(?::[0-5][0-9])2 [PA]M

Compose these 3 new regexes into a complete regex:
(?:0[1-9]|1[0-2])/(?:0[1-9]|[12][0-9]|3[01])/(?:19[0-9][0-9]|20[0-2][0-9])|(?:0[1-9]|1[0-2])/(?:0[1-9]|[12][0-9]|3[01])/(?:19[0-9][0-9]|20[0-2][0-9]) (?:0[1-9]|1[0-2])(?::[0-5][0-9])2 [PA]M|(?:19[0-9][0-9]|20[0-2][0-9])(?:0[1-9]|1[0-2])(?:0[1-9]|[12][0-9]|3[01])

3.4.6 Apply final refined method separately on every dataset to verify it. We will discuss the result in detail in the next section.

3.5 Results

3.5.1 Apply final refined method to each dataset and drawing pictures to verify it.

- Final refined strategy works on DOB NOW: Build — Approved Permits dataset

For dataset DOB_NOW__Build___Approved_Permits.csv

```
df = pd.read_csv(root + files[0], low_memory=False)
# Apply refined method to this dataset
df = refined_method(df, columns[0])
# Extract years, months and days
years = []
months = []
days = []
for date in df[columns[0]]:
    if date == 'UNKNOWN':
        continue
    MDY = date.split()[0]
    strs = MDY.split('/')
    month = int(strs[0])
    day = int(strs[1])
    year = int(strs[2])
    months.append(month)
    days.append(day)
    years.append(year)
```

Fig. 31. Code of final method of dataset DOB NOW: Build — Approved Permits (Date)

```
# After cleaning the data, show the years visualization.
draw(years, 'Year', 'Frequency', 'Year-Frequency')
```

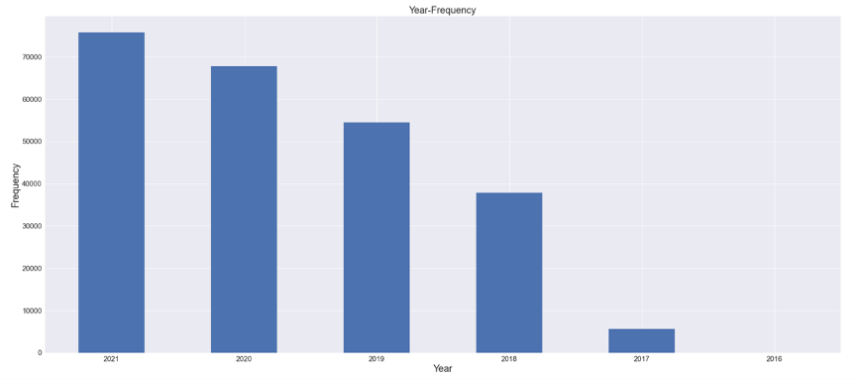

Fig. 32. Year result of dataset DOB NOW: Build — Approved Permits (Date)

```
# After cleaning the data, show the days visualization.
draw(days, 'Day', 'Frequency', 'Day-Frequency')
```

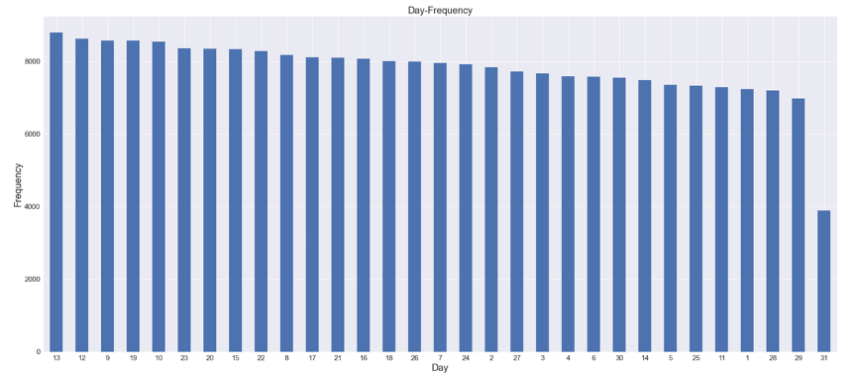

Fig. 33. Day result of dataset DOB NOW: Build — Approved Permits (Date)

```
# After cleaning the data, show the months visualization.
draw(months, 'Month', 'Frequency', 'Month-Frequency')
```

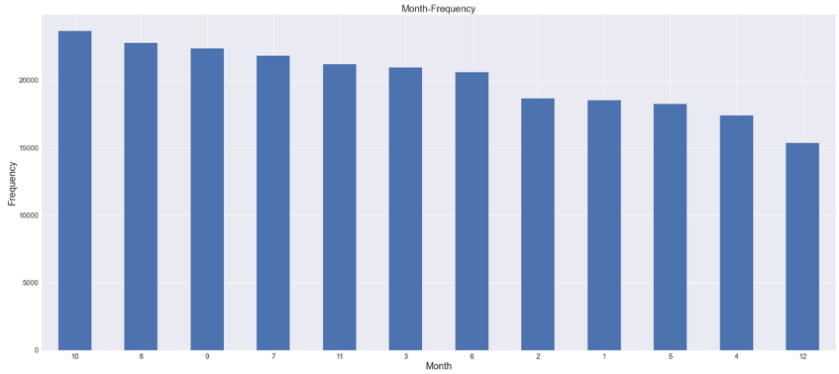

Fig. 34. Month result of dataset DOB NOW: Build — Approved Permits (Date)

- Final refined strategy works on DOB Complaints Received

For dataset DOB_Complaints_Received.csv

```
df = pd.read_csv(root + files[1], low_memory=False)
# Apply refined method to this dataset
df = refined_method(df, columns[1])
# Extract years, months and days
years = []
months = []
days = []
for date in df[columns[1]]:
    if date == 'UNKNOWN':
        continue
    strs = date.split('/')
    month = int(strs[0])
    day = int(strs[1])
    year = int(strs[2])
    months.append(month)
    days.append(day)
    years.append(year)
```

Fig. 35. Code of final method of dataset DOB Complaints Received (Date)

```
# After cleaning the data, show the years visualization.
draw(years, 'Year', 'Frequency', 'Year-Frequency')
```

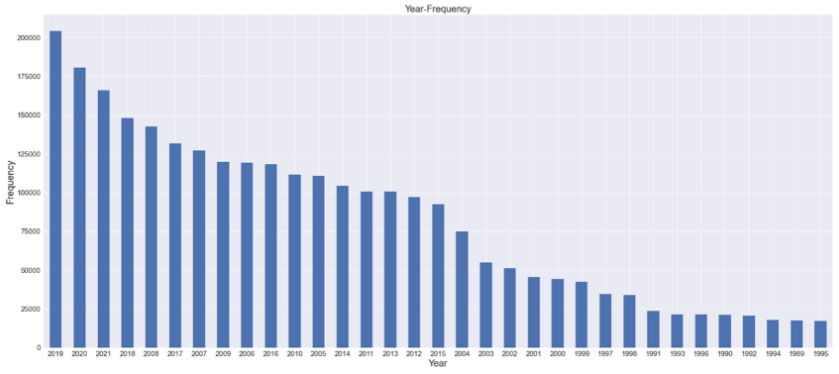

Fig. 36. Year result of dataset DOB Complaints Received (Date)

```
# After cleaning the data, show the months visualization.
draw(months, 'Month', 'Frequency', 'Month-Frequency')
```

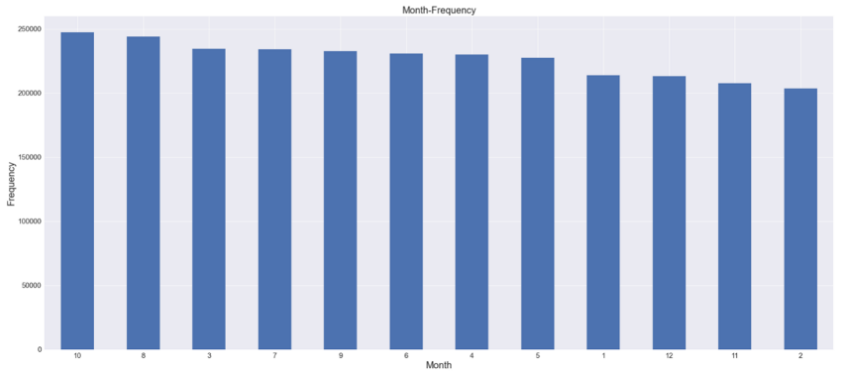

Fig. 37. Month result of dataset DOB Complaints Received (Date)

```
# After cleaning the data, show the days visualization.
draw(days, 'Day', 'Frequency', 'Day-Frequency')
```

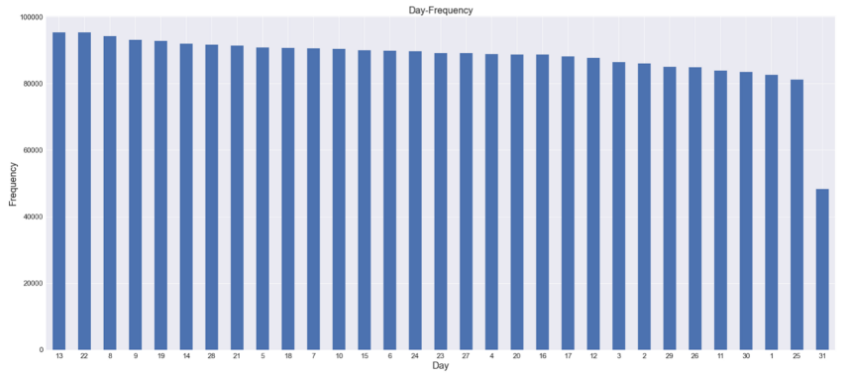

Fig. 38. Day result of dataset DOB Complaints Received (Date)

- Final refined strategy works on DOB ECB Violations dataset

For dataset DOB_Violations.csv

```
# Extract years, months and days
df = pd.read_csv(root + files[2], low_memory=False)
# Apply refined method to this dataset
df = refined_method(df, columns[2])
years = []
months = []
days = []
for date in df[columns[2]]:
    if date == 'UNKNOWN':
        continue
    year = date[0:4]
    month = date[4:6]
    day = date[6:]
    months.append(month)
    days.append(day)
    years.append(year)
```

Fig. 39. Code of final method of dataset DOB ECB Violations (Date)

```
# After cleaning the data, show the years visualization.
draw(years, 'Year', 'Frequency', 'Year-Frequency', figsize=(50, 15))
```

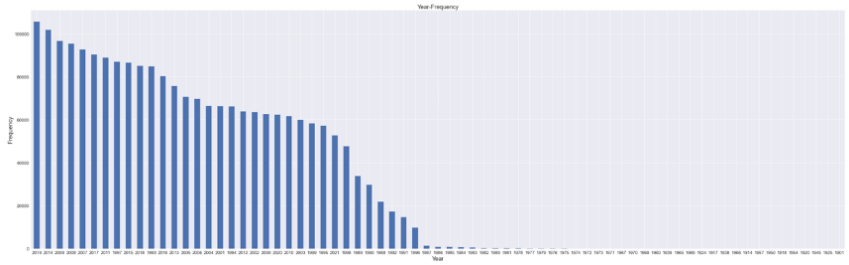

Fig. 40. Year result of dataset DOB ECB Violations (Date)

Compare with previous strategy:

Previous

Fig. 41. Previous wrong year result of dataset DOB ECB Violations (Date)

Now

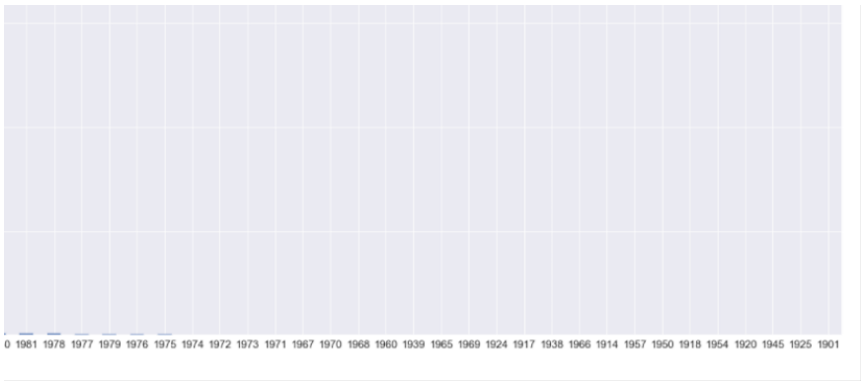

Fig. 42. Now correct year result of dataset DOB ECB Violations (Date)

```
# After cleaning the data, show the days visualization.
draw(days, 'Day', 'Frequency', 'Day-Frequency')
```

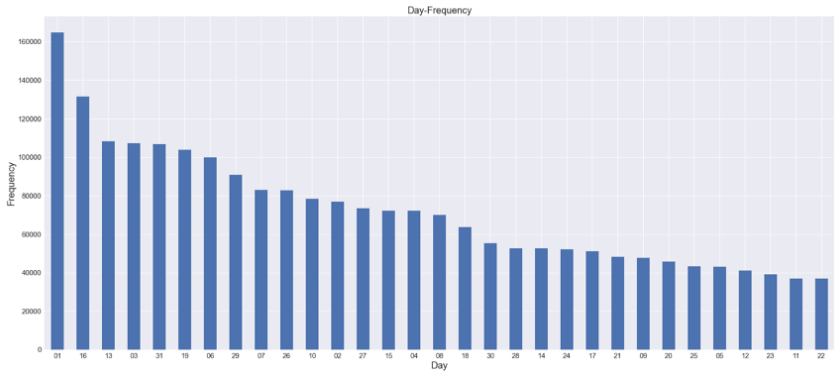

Fig. 43. Day result of dataset DOB ECB Violations (Date)

```
# After cleaning the data, show the months visualization.
draw(months, 'Month', 'Frequency', 'Month-Frequency')
```

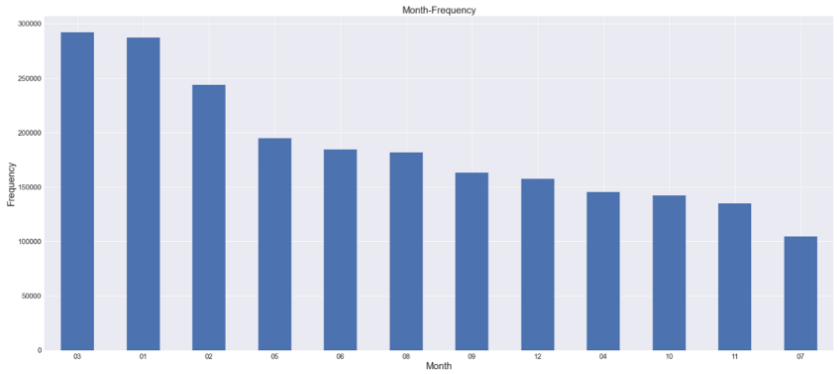

Fig. 44. Month result of dataset DOB ECB Violations (Date)

3.5.2 Conclusion. Through the correct use of regular expressions, we have achieved the desire to match all data sets with a single regular expression. We looked for a dozen new datasets, and their date formats all match one of the three above, indicating that our final refined method is an effective way to clean up date data.

4 CLEANING CITY DATA

4.1 Column Introduction

This column mainly consists of the city names related to Celluar Anthena Filings, Complaints of Illegal Parking, and Electrical Permits Applications. It contains cities in New York State, including Long Island City, Flushing, New York City, Richmond, and Buffalo, and other city names like Boston, Phoenix, and Los Angeles. And the data type of this column is Object.

4.2 Work of Part1

4.2.1 Original Problem formulation. Our original data set is DOB Job Application Filings, and some values within the city column are empty and some are composed of lower-case letters which make data collection much difficult to get the valid profiling and analyzing data.

4.2.2 Original related work (Original strategy). Our original method uses the value_counts method to check if there is an empty value. If the result of value_counts is too long to find all valid answers, we use a boolean array mask to represent the eligible rows, and then we just covert these rows to UNKNOWN.

4.3 Work of Part2

4.3.1 New Problem Formulation. After a random search in the nyu opendata database, we finally find three datasets that also contain City Column. They are:

- Complaints of Illegal Parking of Vehicles
- DOB NOW: Electrical Permit Applications
- DOB Cellular Antenna Filings

After using the value_counts method to these three datasets, we find that some city names are misspelled, some city names are abbreviations of the familiar city names, some even contain illegal spaces. Thus, in this condition, our original strategy does not work.

LAURELTON	333
HOLLIS	322
ROSEDALE	291
DOUGLASTON	271
BRONX	270
REGO PARK	246
LONG ISLAND	229
L.I.C	203
KEW GARDENS	180
STATEN ISLAND	171

Fig. 45. Value and frequency of three datasets (City)

4.3.2 New Related Work (New Strategy). We discovered a lot of incorrect city data when we ran value counts() on the new data frame. It was impossible to correct the different format errors using our original strategy, so we decided to use some additional strategies to keep the format uniform and merge similar values using KNN. We will talk about more about it in the next section.

4.4 Methods, architecture and design

4.4.1 Extract Columns. Each new dataset has a different 'City' column extracted from three different datasets and merged into a new data frame that contains all 'City' columns from the nine datasets, as shown in the following figure.

```
[4]: # add all city columns to one dataframe, then generate the data profile
df = pd.DataFrame(columns=['CITY'])
for file in files:
    path = root + file
    ds = dataset(path, encoding='utf-8')
    print("Loading... ", file)
    # rename all cols to CITY
    ds.rename(columns={'City': 'CITY'}, inplace=True)
    city = ds[['CITY']]
    df = df.append(city)
print(df.value_counts())
```

```
Loading...  All/Local_Law_8_of_2020___Complaints_of_Illegal_Parking_of_Vehicles_Operated_on_Behalf_of_the_
Loading...  All/DOB_NOW__Electrical_Permit_Applications.csv
Loading...  All/DOB_Cellular_Antenna_Filings.csv
CITY
BROOKLYN               83935
NEW YORK               32192
STATEN ISLAND          28649
BRONX                  25130
LONG ISLAND CITY       17870
FLUSHING               11844
ASTORIA                 9343
WOODSIDE                7635
JAMAICA                 5105
MASPETH                 4206
OZONE PARK              4046
SPRINGFIELD GARDENS     3495
RICHMOND HILL           2934
```

Fig. 46. Extract columns and merge (City)

4.4.2 Perform KNN. As we said above, the data in these three datasets have many similar or misspelled city names. Fortunately, the KNN algorithm can compute similar words based on the Levenshtein distance function and put them into a set while giving the suggested word, which is exactly the algorithm we need [3]. Therefore, we decided to use the KNN package in the openclean library to run the KNN algorithm on our dataset.

```
[51]: # Perform KNN to detect spelling errors
df = stream('../Reference Data/original_version_city.csv')
def print_cluster(cnumber, cluster):
    print('Cluster {} (of size {})\n'.format(cnumber, len(cluster)))
    for val, count in cluster.items():
        print('{} ({})'.format(val, count))
    print('\nSuggested value: {}\n\n'.format(cluster.suggestion()))

# Minimum cluster size. Use ten as default (to limit
# the number of clusters that are printed in the next cell).
def run_knn(df, column, t=0.8, minsize = 2):
    dba = df.select(column).distinct()
    clusters = knn_clusters(
        values=dba,
        sim=SimilarityConstraint(func=LevenshteinDistance(), pred=GreaterThan(t)),
        minsize=minsize
    )
    print('{} clusters of size {} or greater'.format(len(clusters), minsize))
    # Sort clusters by decreasing number of distinct values.
    clusters.sort(key=lambda c: len(c), reverse=True)
    for i, cluster in enumerate(clusters):
        print_cluster(i + 1, cluster)
    return clusters

clusters = run_knn(df, 'CITY')
```

```
60 clusters of size 2 or greater
```

Fig. 47. KNN Code (City)

```
Cluster 1 (of size 9)

NEW YORK,       (8)
NEW YORKER      (1)
NEW YORL        (1)
 NEW YORK       (1)
BEW YORK        (1)
MEW YORK        (1)
NEW YOEK        (1)
NEW  YORK       (6)
NEW YORK        (1497)

Suggested value: NEW YORK


Cluster 2 (of size 6)

S. RICHMOND HILL (115)
S.RICHMOND HILL (68)
RICHMOND HILLS (10)
RICHMOND HILL   (9)
RICHMOND.HILL (44)
RICHMOND HILL (2934)

Suggested value: RICHMOND HILL
```

```
Cluster 4 (of size 5)

BROOKLYN,       (1)
BROOKLY         (1)
BROKKLYN        (1)
BROOKYLN        (3)
BROOKLYN        (739)

Suggested value: BROOKLYN


Cluster 5 (of size 4)

RICHMOND HILL (2934)
RICHMOND HILL   (9)
RICHMOND.HILL (44)
RICHMOND HILLS (10)

Suggested value: RICHMOND HILL


Cluster 6 (of size 4)

SOUTH OZONE PK  (1)
SOUTH OZONE PAR (1)
SOUT OZONE PARK (49)
SOUTH OZONE PARK (625)

Suggested value: SOUTH OZONE PARK
```

Fig. 48. Some KNN Results (City)

4.4.3 Apply KNN Suggestions. To convert the city names to KNN suggested words, we firstly built two maps whose keys are both cluster numbers. Then we stored the index and its corresponding cluster keys(similar city names) into idx_to_set map and kept the index and the suggestion from its related cluster in the idx_to_sugg map. Therefore, we could connect the set and the suggested city names using the index. Then we loop through each city name in this column and find its corresponding cluster_num(index). After acquiring this index, we can know the correct city name and replace the old city name with the suggested one. The time complexity of this algorithm is $O(M * N)$, where M is the length of this city column, and N is the length of the index.

```
[52]: # First apply original method again
df, count = original_cleaning_method(df.to_df(), 'CITY')

def apply_knn_suggest(df, column, clusters):
    # Build the key-value pairs
    idx_to_set = {}
    idx_to_sugg = {}
    idx = 0
    for cluster in clusters:
        idx_to_set[idx] = cluster.keys()
        idx_to_sugg[idx] = cluster.suggestion()
        idx += 1
    # Apply knn suggestion and do strip
    idx = 0
    for city in df[column]:
        cluster_num = -1
        # Do strip
        df.loc[idx, column] = df.loc[idx, column].strip()
        for i in range(len(idx_to_set)):
            if city in idx_to_set[i]:
                cluster_num = i
                break
        if(cluster_num == -1):
            idx += 1
            continue
        suggestion = idx_to_sugg[cluster_num].strip()
        df.loc[idx, column] = suggestion
        idx += 1
    return df

# Apply knn suggestions to current dataframe
df = apply_knn_suggest(df, 'CITY', clusters)
```

Fig. 49. Apply KNN Suggestion Algorithm (City)

```
[53]: df.value_counts()

[53]: CITY
BROOKLYN                84680
NEW YORK                33717
STATEN ISLAND           28820
BRONX                   25401
LONG ISLAND CITY        17922
FLUSHING                11912
ASTORIA                  9375
WOODSIDE                 7645
JAMAICA                  5145
MASPETH                  4217
OZONE PARK               4059
SPRINGFIELD GARDENS      3495
RICHMOND HILL            3180
COLLEGE POINT            2670
QUEENS VILLAGE           2571
QUEENS                   2360
ST. ALBANS               2229
L.I.C.                   2111
WOODHAVEN                2104
RIDGEWOOD                1500
MIDDLE VILLAGE           1492
WHITESTONE               1378
PARSIPPANY               1217
UNKNOWN                  1147
```

Fig. 50. KNN Suggestion Algorithm Result (City)

4.4.4 Covert Remaining Incorrect Names. According to Figure 50, although we merged the similar city names according to the KNN result before, some were misspelled, or the KNN algorithm did not detect abbreviated city names. Therefore, we defined a mapper to restore all the abbreviated and misspelled city names based on the result of the value_counts function. Based on Figure 52 below, we could find that we solved all problems in this dataset city column.

```
[58]: # Based on the value_counts result, we designed another mapper to convert several incorrect names to standard representation
mapper = {'L.I.C.': 'LONG ISLAND CITY', 'LIC': 'LONG ISLAND CITY', 'L.I.C': 'LONG ISLAND CITY',
          'LONG ISLAND': 'LONG ISLAND CITY', 'L.I.CITY': 'LONG ISLAND CITY', 'BROOKYN': 'BROOKLYN',
          'BRONX,': 'BRONX', 'WEST NY': 'WEST NEW YORK', 'NEW CITY': 'NEW YORK',
         'NY': 'NEW YORK', 'S.I.': 'STATEN ISLAND', 'L.I. CITY': 'LONG ISLAND CITY'}
df = update(df, columns='CITY', func=lambda x : mapper.get(str(x)) if x in mapper else x)
```

Fig. 51. Covert Remaining Incorrect Name Code (City)

```
[59]: df.value_counts()
```

```
[59]: CITY
      BROOKLYN               84849
      NEW YORK               33751
      STATEN ISLAND          28825
      BRONX                  25522
      LONG ISLAND CITY       21059
      FLUSHING               11912
      ASTORIA                 9375
      WOODSIDE                7645
      JAMAICA                 5145
      MASPETH                 4217
      OZONE PARK              4059
      SPRINGFIELD GARDENS     3495
      RICHMOND HILL           3180
      COLLEGE POINT           2670
      QUEENS VILLAGE          2571
      QUEENS                  2360
      ST. ALBANS              2229
      WOODHAVEN               2104
      RIDGEWOOD               1500
      MIDDLE VILLAGE          1492
      WHITESTONE              1378
      PARSIPPANY              1217
      UNKNOWN                 1147
      EAST ELMHURST           1039
      FAR ROCKAWAY            1038
      FRESH MEADOWS           1025
      ROCKAWAY PARK           1002
      GLENDALE                 840
      BAYSIDE                  812
      JACKSON HEIGHTS          786
      SOUTH OZONE PARK         676
      SOUTH RICHMOND HILL      581
      BELLEROSE                564
      CORONA                   549
      HOWARD BEACH             536
      CAMBRIA HEIGHTS          470
      ELMHURST                 435
      BEDMINSTER               409
      LAURELTON                334
```

Fig. 52. Some Results after Converting (City)

4.4.5 Encapsulation. To make it easier to use later, we encapsulated all the previous steps into a single function. We also designed and implemented the draw function for later drawing.

```
[77]: # Minimum cluster size. Use ten as default (to limit
      # the number of clusters that are printed in the next cell).
      def run_knn_without_print(df, column, t=0.8, minsize = 2):
          dba = df.select(column).distinct()
          clusters = knn_clusters(
              values=dba,
              sim=SimilarityConstraint(func=LevenshteinDistance(), pred=GreaterThan(t)),
              minsize=minsize
          )
          # Sort clusters by decreasing number of distinct values.
          clusters.sort(key=lambda c: len(c), reverse=True)
          return clusters


      def refined_method(df, column):
          mapper = {'L.I.C.': 'LONG ISLAND CITY', 'LIC': 'LONG ISLAND CITY', 'L.I.C': 'LONG ISLAND CITY',
                    'LONG ISLAND': 'LONG ISLAND CITY', 'L.I.CITY': 'LONG ISLAND CITY', 'BROOKYN': 'BROOKLYN', 'BRONX,': 'BRONX',
                    'WEST NY': 'WEST NEW YORK', 'NEW CITY': 'NEW YORK',
               'NY': 'NEW YORK', 'S.I.': 'STATEN ISLAND', 'L.I. CITY': 'LONG ISLAND CITY'}
          df.to_csv('../Reference Data/temp.csv')
          df = stream('../Reference Data/temp.csv')
          clusters = run_knn_without_print(df, column)
          # First apply original method again
          df, count = original_cleaning_method(df.to_df(), column)
          # Apply knn suggestions to current dataframe
          df = apply_knn_suggest(df, column, clusters)
          df = update(df, columns=column, func=lambda x : mapper.get(str(x)) if x in mapper else x)
          return df

      # Then we want to find the cotent outlier by drawing pictures
      def draw(city, xlabel, ylabel, start, end, title='City-Frequency', fontsize=15, rot=0, figsize=(50, 15)):
          if idx == 2:
              return
          vals = pd.Series(city)
          vals.value_counts()[start:end].plot(kind='bar', figsize=figsize, xlabel=xlabel, fontsize=fontsize, rot=rot)
          plt.title(title, fontsize=20)
          plt.xlabel(xlabel, fontsize=20)
          plt.ylabel(ylabel, fontsize=20)
          plt.show()
```

Fig. 53. Encapsulation and Draw (City)

4.5 Results

4.5.1 Results of Our Refined Strategy. A bar chart displaying the results of applying this new strategy to all datasets is generated in this section. For all datasets with the column 'City,' we can conclude that this strategy is successful and can be applied to all datasets that contain the column 'City.'

- Refined method and value counts() results were both correct for the dataset "Complaints of Illegal Parking of Vehicles," which we used to create the bar and pie charts.

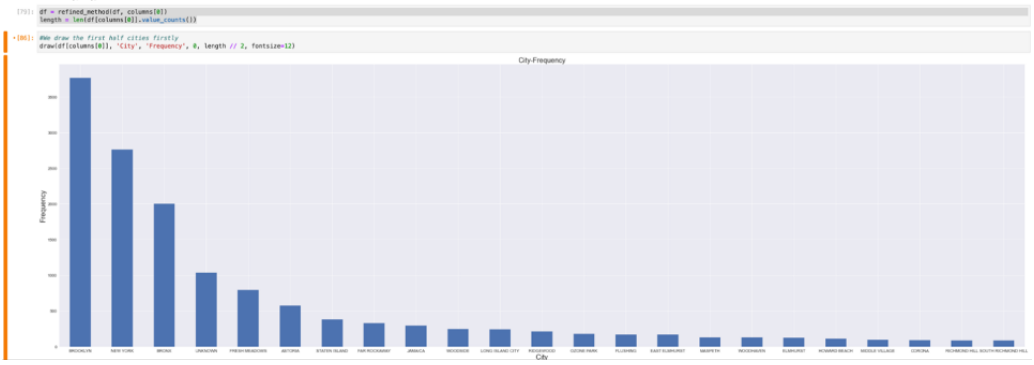

Fig. 54. Result of Dataset Illegal Parking of Vehicles (City)

- Refined method and value counts() results were both correct for the dataset “DOB_NOW_Electrical_Permit_Applications” which we used to create the bar and pie charts.

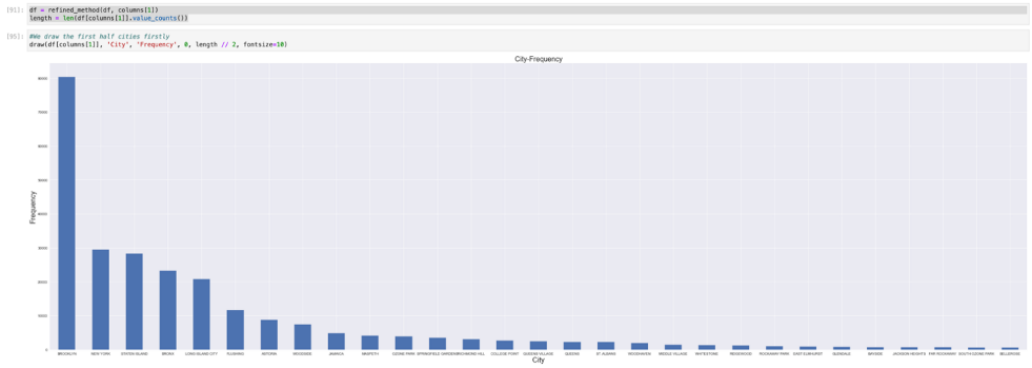

Fig. 55. Result of Dataset DOB_NOW_Electrical_Permit_Applications (City)

- Refined method and value counts() results were both correct for the dataset “DOB_Cellular_Antenna_Filings” which we used to create the bar and pie charts.

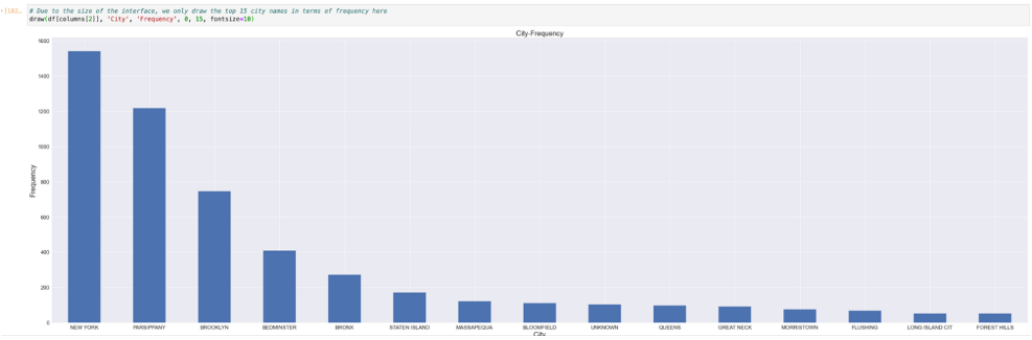

Fig. 56. Result of Dataset DOB_Cellular_Antenna_Filings (City)

4.5.2 Conclusion. We can conclude that the new strategy we devised for the column ‘City’ is correct, and we can apply this strategy to all datasets based on the bar charts and results shown above.

5 HOW WE WOULD RUN OUR APPROACH ON ALL DATASETS

To run our strategy on all NYC Open Data datasets. First, we extract the columns needed to be cleaned from all the datasets. Second, we merge them into a new data frame with only one column and contain all the data to be cleaned. Finally, we apply our strategy to this data frame and get the result.

REFERENCES

[1] Otmane Azeroual and Joachim Schöpfel. 2019. Quality Issues of CRIS Data: An Exploratory Investigation with Universities from Twelve Countries. *Publications* 7, 1 (2019), 14. https://doi.org/10.3390/publications7010014

[2] H.M. Abdul Aziz, Satish V. Ukkusuri, and Samiul Hasan. 2013. Exploring the determinants of pedestrian–vehicle crash severity in New York City. *Accident Analysis & Prevention* 50 (2013), 1298–1309. https://doi.org/10.1016/j.aap.2012.09.034

[3] Essatouti Basma, Khamar Hakima, El Fkihi Sanaa, Faizi Rdouan, and Oulad Haj Thami Rachid. 2018. Arabic sentiment analysis using a levenshtein distance based representation approach. In *2018 IEEE 5th International Congress on Information Science and Technology (CiSt)*. 270–273.

[4] T.L. KLEVELAND. 2015. The application of fuzzy text recognition and-manipulation technologies to clean-up, idealize, improve, and integrate sets of unstructured data.

[5] Abhinav Kumar and Jyoti Prakash Singh. 2019. Location reference identification from tweets during emergencies: A deep learning approach. *International journal of disaster risk reduction* (2019), 365–375.

[6] S. OZDEMIR. 2016. Principles of data science. *Packt Publishing Ltd* (2016).

[7] Uthayasankar Sivarajah, Muhammad Mustafa Kamal, Zahir Irani, and Vishanth Weerakkody. 2017. Critical analysis of Big Data challenges and analytical methods. *Journal of Business Research* 70 (2017), 263–286. https://doi.org/10.1016/j.jbusres.2016.08.001