BIG DATA PROJECT REPORT - Citywide Payroll Data (Fiscal Year)

RUI WANG RW3068, YUTING PENG YP2212, and XIXUAN WANG XW2579*, New York University, US

Github link

ACM Reference Format:
Rui Wang rw3068, Yuting Peng yp2212, and Xixuan Wang xw2579. 2021. BIG DATA PROJECT REPORT - Citywide Payroll Data (Fiscal Year). 1, 1 (December 2021), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Data is extremely important in a variety of fields, which plays an unshakable role in making informed decisions, finding solutions to problems, and making people's lives much easier. However, the quality of data is further more essential than the data itself, and there are numbers of cases that awful data quality leads to severe consequences such as substantial money waste and fatal accidents. Therefore, data cleaning is deemed to be an indispensable pre-process before actually making the best use of those data. It's hence reasonable to ask: what is data cleaning?

Data cleaning is the process of eliminating or fixing incorrect, corrupted, incorrectly formatted, duplicate, or incomplete data within a dataset, which has significant meaning in increasing the overall productivity and allowing for the highest quality information in decision makings.

However, data cleaning is usually so difficult and time-consuming for several reasons. First, a variety of data formats must be dealt with. Second, there are so many different data quality issues as databases try to fit a complex world into a simple abstraction. Finally, cleaning the data requires domain knowledge.

Fortunately, openclean saves us a lot of work. Openclean is a Python library for data profiling and data cleaning, which will be discussed further in the later Related Work section. Openclean, together with spark, a unified engine for large-scale data analytics, supported most of our work in part one and part two of this project respectively.

2 PROBLEM FORMULATION

The project has two part. The first part is to profile and clean data for the Citywide Payroll Data (Fiscal Year), and the second part is to apply the clean method to a volume of datasets, testing the scale ability of our clean method.

2.1 project part one

2.1.1 *profile data.*

We mainly used OpenClean for data profile, for instance, from the basis use of profiles.stats(), profiles.minmax(), and profiles.column().get() to more deep methods helping us to find the dirty data like the similarity library and create

*All authors contributed equally to this research.

Authors' address: Rui Wang rw3068; Yuting Peng yp2212; Xixuan Wang xw2579, New York University, US.

Manuscript submitted to ACM

cluster function. The formulation of this part of work is as follow:

functions	what we found from it
profiles.stats()	overview the attribute of dataset
profiles.minmax()	see the numerical data's range
profiles.column().get('topValues')	see the top 10 values for each attribute and count their frequency
different kinds of cluster	to check the spelling error and no consistent data

For different kind of data, we design different methods to find the error or inconsistent data.

Attribute	The method we use
Agency Name	distinct statistic and Key-collision cluster
Last and First Name	uppercase
Mid Init	distinct and invalid character check
Agency Start Date	invalid date check and impossible date check
Work Location Borough	distinct statistic and key-collision cluster
Title Description	distinct statistic and KNN clustering
other number values	negative dealing

by doing the profile using the above method, we find a lot of data problems like data defected, data inconsistent, data error, and extra, the below just show a small part of them:

(1) Empty values in 'Last Name', 'First Name'
(2) characters in 'Mid Init'
(3) 'Agency Name' value : police department, board of correction
(4) 'Agency Start Date' value : 12/31/9999, 10/16/2049
(5) 'OT hours', 'Regular Hours' negatives:
(6) 'Work Location Borough' : MANHATTAN manhattan....
(7) 'Title Description': many typos

2.1.2 clean data. We design clean method for each type of data. and the main function is as below:

functions	what we found from it
Agency Name	uppercase and specific change
Last and First Name	uppercase
Mid Init	using ASCII to select the A to Z character
Agency Start Date	judge the invalid days for each month and year, delete the future time
Work Location Borough	uppercase and empty data to 'UNKNOWN'
Title Description	uppercase and specific change
other number values	change negative value to null

2.2 project part two

2.2.1 scale method and analysis.

We used spark to clean the scale datasets and analysis our clean method in the scale datasets. To better focous on different attribute, for each column, we find 10 datasets from the NYC open data source having the same attribute and apply our clean method to those datasets, while refine the clean method to make the cleaning be more comprehensive

and precise.

refine method for each attribute

functions	the refine method we added
Agency Name	space eliminated, inconsistent refine and uncompleted data supplement
Mid Init	using spark
Agency Start Date	to date function
Work Location Borough	space eliminated
Title Description	using spark

2.2.2 *data references.*

For the String attribute: Agency Name, Work Location Borough and Title description, we used the final cleaned 10 datasets for each attribute to create a reference data. Since the 10 datasets volume is large enough and have a lot of duplicate, which can double make sure our final data are clean, we UNION all the data through the 10 datasets and get the result of the reference data.

3 RELATED WORK

3.1 Openclean

Openclean is a Python library for data profiling and data cleaning, whose goal is to bring together data cleaning tools in a single environment that is easy and intuitive to use for a data scientist, allowing users to compose and execute cleaning pipelines that are built using a variety of different tools. Moreover, it is flexible and extensible to allow easy integration of new functionality.

The following code presents where and how we mainly used the openclean library. As shown below, openclean provides numbers of APIs which significantly reduced the complexity of operations, making us be able to concentrate and focus more on the data cleaning core logic.

```
from openclean.data.source.socrata import Socrata
dataset = Socrata().dataset("k397-673e")

from openclean.pipeline import stream
ds_full = stream(datafile)

from openclean.profiling.column import DefaultColumnProfiler
profiles = ds.profile(default_profiler=DefaultColumnProfiler)
profiles.stats()

for col in ds.columns:
    p = profiles.column(col)
    print("␣␣'{}'␣({})".format(col, p['datatypes']['distinct'].most_common(1)[0][0]))

profiles.minmax('Fiscal␣Year')
profiles.column('Agency␣Name').get('topValues')

from openclean.cluster.knn import knn_clusters
from openclean.function.similarity.base import SimilarityConstraint
from openclean.function.similarity.text import LevenshteinDistance
from openclean.function.similarity.text import HammingDistance
from openclean.function.similarity.text import JaroSimilarity
from openclean.function.similarity.text import StringSimilarityFunction
```

```
from openclean.function.value.threshold import GreaterThan
```

3.2 Clustering

The concept of clustering is about using statitic methods for processing data by organizing items into groups, or clusters, on the basis of how closely associated they are. In this project, we mainly used two methods related to clustering including the KNN clustering method and the key collision method.

KNN is short for K-nearest neighbors, and it is one of the most popular learning algorithms. The purpose of it is to use a database in which the data points are separated into several classes to predict the classification of a new data point. In our project, we don't need to make predictions for new data points, so we actually used KNN to cluster the data of existing columns so that we are able to find out similar data fields. For example, by utilizing the KNN clustering method, we are able to distinguish "ADM MANAGER-NON-MGR" and "ADM MANAGER-NON-MGRL", or "DIRECTOR OF MANAGEMENT PLANNING" and "DIRECTOR OF MANAGEMENT PLANNING SS" ,etc. This really helps use to make judgement on which data fields are incorrectly formatted or needed to be unified to be the same data field.

```
Cluster 1
  NON-TEACHING ADJUNCT III (x 3419)
  NON-TEACHING ADJUNCT I (x 8407)
  NON-TEACHING ADJUNCT V (x 1047)
  NON-TEACHING ADJUNCT IV (x 1279)
  NON-TEACHING ADJUNCT II (x 3774)
Cluster 2
  EDUCATIONAL ADMINISTRATOR UFT (x 58)
  EDUCATIONAL ADMINISTRATOR (x 1056)
  EDUCATIONAL ADMINISTRATOR CSA (x 6281)
Cluster 3
  SUPERVISOR II (x 1503)
  SUPERVISOR I (x 3390)
  SUPERVISOR III (x 490)
Cluster 4
  ADM MANAGER-NON-MGR (x 39)
  ADM MANAGER-NON-MGRL (x 5406)
Cluster 5
  INTELLIGENCE RESEARCH MANAGER (x 6)
  INTELLIGENCE RESEARCH MANAGER-PD (x 39)
Cluster 6
  HEALTH SERVICES MANAGER NON MANAGERIAL LEVEL II (x 92)
  HEALTH SERVICES MANAGER NON MANAGERIAL LEVEL I (x 301)
Cluster 7
  DIRECTOR OF MANAGEMENT PLANNING (x 3)
  DIRECTOR OF MANAGEMENT PLANNING SS (x 23)
......
```

As metioned above, KNN clustering helps us be able to discover similar data fields. They are considered similar due to one or more character differences. The other clustering method - key collision, is also used to help us find out similar data fields, whereas they are considered similar because of special character differences or the words within two data fields are in different orders. For example, " *ADM SCHOOL SECURITY MANAGER-U" and "ADM SCHOOL SECURITY MANAGER-U" are basically the same except for the leading "*" character. Also, "ASSISTANT DEPUTY

COMMISSIONER" and "DEPUTY ASSISTANT COMMISSIONER" are quite similar with the only difference of word order. We can assert that "DEPUTY ASSISTANT" and "ASSISTANT DEPUTY" both refer to the same thing.

```
Cluster 1
  *ADM SCHOOL SECURITY MANAGER-U (x 16)
  ADM SCHOOL SECURITY MANAGER-U (x 3)

Cluster 2
  *ADMIN SCHL SECUR MGR-MGL (x 4)
  ADMIN SCHL SECUR MGR-MGL (x 1)

Cluster 3
  *ADMINISTRATIVE ATTORNEY (x 24)
  ADMINISTRATIVE ATTORNEY (x 5)
...
Cluster 19
  ASSISTANT DEPUTY COMMISSIONER (x 9)
  DEPUTY ASSISTANT COMMISSIONER (x 326)

Cluster 20
  *CUSTODIAL ASSISTANT (x 147)
  CUSTODIAL ASSISTANT (x 1340)
......
```

3.3 Apache Spark

Apache Spark is an open-source, distributed processing system used for big data workloads, which utilizes in-memory caching and optimized query execution for fast analytic queries against data of any size. It provides development APIs in Java, Scala, Python and R, and in the part two of our project, we used the APIs in Python named Pyspark. For example:

```
%spark.pyspark
work_location_borough1 = dataset1.map(lambda x : (x[7], 1)).countByKey()
for key, value in work_location_borough1.items():
    print(key, value)

work_location_borough2 = dataset2.map(lambda x : (x[2], 1)).countByKey()
for key, value in work_location_borough2.items():
    print(key, value)

%spark.pyspark
work_location_borough3 = dataset3.map(lambda x : (x[4], 1)).countByKey()
for key, value in work_location_borough3.items():
    print(key, value)
... ...
```

By using Pyspark, we can easily obtain much faster speed to even concurrently find out the frequencies of each data field of each dataset. Even if our datasets become extremely large, Apache Spark or Pyspark still performs quite well in dealing with them with as fast speed as possible.

4 METHODS, ARCHITECTURE AND DESIGN

4.1 'Mid Name Initial' Attribute

Initial is an attribute that everyone has reached a consensus: only capital letters are permitted to appear. Therefore, the first thing we did was to find out the distinct values of this column. Since there should be at most 26 possible values, it is quite quick to locate the problems. Our results for part one are shown as follow:

initial	frequency
null	1,596,166
A	348,499
M	330,479
...	...
Q	2,181
.	131
-	86
1	71
x	39
2	30
...	...

Among all values above, null value has the greatest frequency. It means that most people don't have a middle name or they just don't want to write it down. What's more, there are some weird characters, including numbers, dashes, parentheses etc. We need to do judgements and turn them to null. Last but not least, there is a 'x' value. It's a tiny problem but it can be deadly. We need all the letters to be uppercase, just to be on the safe side.

4.2 'Title Description' Attribute

It's a little tricky when we are dealing with those columns which are stored as strings. First, a couple of typos exist such as 'RADIO AND TEVEVISION OPERATOR'. Second, the same data may be written as several different forms. For example, 'SERGEANT D/A SPECIAL ASSIGNMENT' and 'SERGEANT-D/A SPECIAL ASSIGNMENT', 'SECRETARY TO COMMISSIONER' and 'SECRETARY OF COMMISSIONER', 'CHAUFFEUR ATTENDANT' and 'CHAUFFEUR-ATTENDANT'... It's difficult to make a decision. We are not sure which one is the correct answer, because we don't have reference data. Under such circumstances, we calculate the frequency of values in each similar pairs and choose the value with the highest frequency as the correct one. Furthermore, we find some cultural issues which are really interesting.

title description	frequency
AGENCY ATTORNEY INTERN	110
AGENCY ATTORNEY INTERNE	698

It's strange that the frequency of word 'INTERNE' is much higher than 'INTERN' which we thought would be the only right spelling of this word. As people from remote China, we are surprised to find 'INTERNE' is also right as a NORTH AMERICAN idiom. This makes the cleaning strategy more complicated. More factors should be taken into account.

4.2.1 *Original Method in Part1.*

In part1, since the dataset is relatively small, we used reduceBeKey() on each value. Then, we did clustering to find similar strings by the methods we discussed above(3.2). The results are clear:

```
Cluster 28
  SECRETARY TO THE CHAIR (x 1)
  SECRETARY TO THE CHAIRMAN (x 11)
...
Cluster 30
  SECRETARY TO DEPARTMENT (x 7)
  SECRETARY OF DEPARTMENT (x 1)
...
Cluster 34
  SUPV OF HOUSING EXTERMINATORS (x 5)
  SUPV OF HOUSING EXTERMINATOR (x 23)
...
Cluster 8
  AGENCY ATTORNEY (x 8921)
  *AGENCY ATTORNEY (x 9)
...
```

Then, We clean data manually according to the rules we discussed above.

4.2.2 *Apply to new datasets.*

When we apply our methods to new datasets, we find some new issues. Some values are truncated suddenly: 'PROGRAM OFFICER (DEPT FOR THE'. It means details are lost. We need to find the correct information and fix them. This phenomenon always happen across a whole single dataset. It's impossible to do completion on data by do clustering on the same dataset(each value lost its details more or less). If we can get detail from another dataset, then the problem can be solved. In conclusion, we hope the datasets can complement each other.

4.2.3 *Data cleaning on scale.*

We find ten datasets that have a 'title' attribute. We find it is difficult to work with openclean and spark together in hpc. Therefore, we need to write our own clusters.

First, we read all the 'title' data.

```
%spark.pyspark
# title description

# Read All the data
# NYC Civil Service Titles 1
ds1 = sc.textFile('hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.nzjr-3966.csv').mapPartitions(lambda x :
      reader(x)).map(lambda x: [x[1], 1]).filter(lambda line : len(line)>1 and 'Title_Description' not in line)
# Civil Service Titles Subject to Investigation 1
ds2 = sc.textFile('hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.h7fs-cqma.csv').mapPartitions(lambda x :
      reader(x)).map(lambda x: [x[1], 1]).filter(lambda line : len(line)>1 and 'TITLE_DESCRIPTION' not in line)
...
...
# Appeals Closed In 2016 7
ds9 = sc.textFile('hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.wbtw-zkex.csv').mapPartitions(lambda x :
      reader(x)).map(lambda x: [x[7], 1]).filter(lambda line : len(line)>1 and 'Title' not in line)
# Civil Service List (Terminated) 7
```

```
ds10 = sc.textFile('hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.qu8g-sxqf.csv').mapPartitions(lambda x :
        reader(x)).map(lambda x: [x[7], 1]).filter(lambda line : len(line)>1 and 'List_Title_Desc' not in line)
```

union

```
all = ds1.union(ds2).union(ds3).union(ds4).union(ds5).union(ds6).union(ds7).union(ds8).union(ds9).union(ds10)
all.count()
 amount: 2672736
```

Currently. we have got more than 2.5 million rows of title data. Next, we use spark to do mapreduce.

```
%spark.pyspark
title = all.reduceByKey(add)
title.count()
 amount: 2818
```

We are excited to find the data amount is reduced to a very significant extent(2672736->2818). Now, it's time to do clustering. We think Levenshtein distance will help us to acheive the best results because our samples are all strings.

```
def LevenshteinDistance(s1, s2):
    d = [[x for x in range(len(s1)+1)] for _ in range(len(s2)+1)]
    for y in range(1,len(s2)+1):
        d[y][0] = d[y-1][0] + 1
    for x in range(1, len(s1)+1):
        for y in range(1, len(s2)+1):
            if s1[x-1] == s2[y-1]:
                d[y][x] = d[y-1][x-1]
            else:
                substute = d[y-1][x-1] + 1
                add = d[y][x-1] + 1
                delete = d[y-1][x] + 1
                d[y][x] = min(add, substute, delete)
    return d[-1][-1]
```

Among all the cluster algorithms, we think Affinity Propagation is the most suitable one, because we don't know how many groups we may need at the very beginning. We calculate the LevenShtein Distance between each string in the title sets. We use it as our similarity matrix.

```
%spark.pyspark
import numpy as np
import sklearn.cluster
title = title.collect()
similarity = -1*np.array([[LevenshteinDistance(t1,t2) for t1 in title] for t2 in title])
print(similarity)
affprop = sklearn.cluster.AffinityPropagation(affinity="precomputed", damping=0.5)
affprop.fit(similarity)
    [[  0 -38 -23 ... -28 -26 -39]
     [-38   0 -39 ... -46 -42 -44]
     [-23 -39   0 ... -19 -19 -40]
     ...
     [-28 -46 -19 ...  0 -16 -35]
     [-26 -42 -19 ... -16  0 -33]
     [-39 -44 -40 ... -35 -33  0]]
```

We still can't figure out how to use spark to do machine learning, so we have to ask sklearn for help. Although it takes some time to run the model, the results are nice. We do find the datasets can help each other! The data that lost details are always be clustered into the same group with the data with full details. So finally we know how to correct the dirty data!

data that lost detail	complement data
PROGRAM OFFICER (DEPT FOR THE	PROGRAM OFFICER (DEPARTMENT FOR THE AGING)
GENERAL SUPERINTENDENT (SANITA	GENERAL SUPERINTENDENT (SANITATION)
AREA SUPERVISOR (HIGHWAY MAINT	AREA SUPERVISOR (HIGHWAY MAINTENANCE)
SECRETARY OF THE DEPARTMENT (D	SECRETARY OF THE DEPARTMENT (DBS)
PUBLIC HEALTH NURSE (SCHOOL HE	PUBLIC HEALTH NURSE (SCHOOL HEALTH)
SUPERVISOR (WATER & SEWER SYST	SUPERVISOR (WATER AND SEWER SYSTEMS)
CONSULTANT (PUBLIC HEALTH-SOCI	CONSULTANT (PUBLIC HEALTH SOCIAL WORK)
MULTIPLE DWELLING SPECIALIST (	MULTIPLE DWELLING SPECIALIST (BUILDINGS)
...	...

Still, we have to admit that we can not fix all the problems, for example, we can't find the corresponding value of 'DEPUTY COMMISSIONER (SPECIAL S', 'ASST COMMISSIONER FOR PUBLIC &', 'DIRECTOR OF INFORMATION AND RE' and so on. But we believe if the dataset is large enough, it is possible to solve all the problems.

4.3 'Agency Start Date' Attribute

4.3.1 *Original Method in Part1.*

In the NYCPayroll dataset, we used the following steps to do cleaning on the date column. Finding out the minimum and the maximum of dates can help us get a picture of the data and narrow the range. Furthermore, since it's an attribute that represents an event happened before, it cannot be a date in the future. And Of course, we should filter impossible dates as well.

(1) find the min date and the max date
(2) check whether there is a date in the future
(3) check impossible date, E.g. 2021-13-01, 2021-2-30
(4) for each abnormal value, do cleaning manually

```
agency_start_dates = ds.distinct('Agency_Start_Date')
for rank, val in enumerate(agency_start_dates.most_common()):
    dt, freq = val
    if dt == '':
        continue
    if int(dt.split('/')[-1]) > 2021:
        print(dt)
    elif int(dt.split('/')[1]) > 31 or int(dt.split('/')[1]) < 1:
        print(dt)
    elif int(dt.split('/')[1]) > 30 and int(dt.split('/')[0]) in [4,6,9,11]:
        print(dt)
    elif int(dt.split('/')[1]) > 28 and int(dt.split('/')[0]) == 2 and int(dt.split('/')[-1]) % 4 != 0:
        print(dt)
    elif int(dt.split('/')[1]) > 29 and int(dt.split('/')[0]) == 2 and int(dt.split('/')[-1]) % 4 == 0:
        print(dt)
    elif int(dt.split('/')[0]) > 12 or int(dt.split('/')[0]) < 1:
        print(dt)
```

Using the openclean package and the code above, we are able to find out abnormal value:

12/31/9999

10/16/2049

4.3.2 *Apply to new datasets.*

There is no doubt that this method works well on other datasets, because date is a kind of attribute which is relatively stable. But If we take efficiency into account, this method may not that good. For each date, we determine whether the day is valid, whether the month is valid and whether the year is valid. It's really time-consuming. What's more, we have to clean the dirty data manually like this:

```
def agency_start_date(x):
    if x == '12/31/9999':
        return ''
    elif x == '10/16/2049':
        return ''
    else:
        return x
ds = update(ds, columns='Agency Start Date', func = agency_start_date)
```

It is lucky we just meet with two problematic dates in this NYCPayroll dataset. What if there are a lot? We need a general and effiecnt method to do this.

So, we decide to use Spark. We are happy to find several amazing convenience Spark brings us during this process. If we use Spark to read data from a .csv file. The date attribute will be parsed as a String, which is not what we want. So we need to turn them back into date type. Spark offers us the to_date() method to do this. We use the following code to do this(just an example).

```
test = spark.createDataFrame([('1-30-2021',)], ['date'])
test.select(to_date(test.date, 'MM-dd-yyyy').alias('date')).show()
```

By observing the results, we will get some interesting facts (p.s.:2020 is a leap year):

string	format	to_date()
01-32-2021	'MM-dd-yyyy'	null
02-29-2021	'MM-dd-yyyy'	null
02-29-2020	'MM-dd-yyyy'	datetime.date(2020,2,29)
02/29/2020	'MM-dd-yyyy'	null
02/29/2020	'MM/dd/yyyy'	datetime.date(2020,2,29)
02/29-2020	'MM/dd-yyyy'	datetime.date(2020,2,29)
2-29-2020	'MM-dd-yyyy'	datetime.date(2020,2,29)

Amazing! Spark help us do date checking automatically when we call the to_date() function. All we need to do is to match a suitable format String to the data, then all data will be edited to 'yyyy-MM-dd'.

Thanks to the great job by Spark, the only thing we need to do next is to check whether there exists future dates. To achieve the goal, we can use the sparksql:

```
spark.sql("select * from ds where date > current_date()")
```

4.3.3 *Refined method.*

Through all the above analysis, we can get the final improved cleaning plan.

(1) read raw data with pyspark
(2) figure out what the date format is
(3) turn string to date type
(4) use sparksql to find whether future dates exist
(5) do cleaning according to requirements(drop null records or leave them alone)

Following is the code, take NYCPayroll dataset for instance:

```
%spark.pyspark
ds = spark.read.csv('hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.k397-673e.csv', header=True,
      inferSchema=True)
ds = ds.withColumn('Agency Start Date', to_date(ds['Agency Start Date'], 'MM/dd/yyyy'))
#ds.printSchema()
#ds.show()
ds.createOrReplaceTempView('ds')
spark.sql("select * from ds where `Agency Start Date`>current_date()").show()
```

4.4 'Agency Name' Attribute

4.4.1 *Original Method in Part1.*

In the NYCPayroll dataset, we used the following steps to do cleaning on date column. We tried different kinds of knn clusters to find data problems. Specifically, We use LevenshteinDistance, HammingDistance, and JaroSimilarity

(1) Cluster for Agency Name used kNN cluster and the Levenshtein distance as the similarity measure.
(2) Cluster for Agency Name used key collision and multiple threads (4) to generate value keys in parallel.
(3) We change all the agency name to the upper case to make the data consistence
(4) According to the outcome of the cluster, we change the same agency name to the same description

KNN cluster by using Levenshtein distance

```
minsize = 2
clusters = knn_clusters(
    values=agency_names,
    sim=SimilarityConstraint(func=JaroSimilarity(), pred=GreaterThan(0.9)),
    minsize=minsize
)

def print_k_clusters(clusters, k=5):
    clusters = sorted(clusters, key=lambda x: len(x), reverse=True)
    val_count = sum([len(c) for c in clusters])
    print('Total number of clusters is {} with {} values'.format(len(clusters), val_count))
    for i in range(min(k, len(clusters))):
        print('\nCluster {}'.format(i + 1))
        for key, cnt in clusters[i].items():
            if key == '':
                key = "''"
            print(f'  {key} (x {cnt})')

print_k_clusters(clusters,13)
```

key collision with 4 threads

```
    from openclean.cluster.key import key_collision
    minsize = 2

    clusters = key_collision(values=agency_names, minsize=minsize, threads=4)

    print('{} clusters of size {} or greater'.format(len(clusters), minsize))
    print_k_clusters(clusters)
```

due to the problem we find, we design the clean function as follow:

```
def agency_name(x):
    if x != '':
        x = x.upper()
    if x == 'BOARD OF CORRECTIONS':
        x ='BOARD OF CORRECTION'
    return x
```

However, if the dataset becomes larger and larger, we will face the problem that the dataset can not fit in to the memory for us to clean it. So for the part two of the project, we change the way by using pyaspark to find the dataset problems and clean the column of the Agency Name.

4.4.2 *Apply to New Datasets in Part 2.*

In order to make sure if the techniques for cleaning this kind of data columns are general and can be used to generate reference data, we need to find 10 datasets that include 'Agency Name' columns. the dataset for the Agency Name is as below:

```
dataset1 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.ye3c-m4ga.csv").mapPartitions(lambda
        x: reader(x))
dataset2 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.423i-ukqr.csv").mapPartitions(lambda
        x: reader(x))
dataset3 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.gzfs-3h4m.csv").mapPartitions(lambda
        x: reader(x))
dataset4 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.mdcw-n682.csv").mapPartitions(lambda
        x: reader(x))
dataset5 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.k397-673e.csv").mapPartitions(lambda
        x: reader(x))
dataset6 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.mwzb-yiwb.csv").mapPartitions(lambda
        x: reader(x))
dataset7 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.a9md-ynri.csv").mapPartitions(lambda
        x: reader(x))
dataset8 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.vx8i-nprf.csv").mapPartitions(lambda
        x: reader(x))
dataset9 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.xrwg-eczf.csv").mapPartitions(lambda
        x: reader(x))
dataset10 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.qbjq-atxv.csv").mapPartitions(
       lambda x: reader(x))
```

In the first part, the main clean method for the Agency Name is:

```
def clean(x):
    x = toUpperCase(x)
    return x
def toUpperCase(x):
```

```
    if x != '':
        return x.upper();
    else:
        return "␣"
```

we apply it to the 10 dataset and get the 1st version of the clean data for each dataset. In order to do analysis and compare the result with the refine way, we named the 1st version of the clean dataset as dataset old.

the spark funtion for the clean is as below:

```
agency_name_old1 = dataset1.map(lambda x: (clean(x[3]),1)).countByKey()
for key, value in agency_name1.items():
    print(key, value)
```

4.4.3 *refine the part1 clean method for new datasets.*

there are some new problems in the new datasets

(1) some of the dataset have a specific length for the attribute, if the Agency Name length is small, there will be a space in the String to make the same Agency Name count more than twice when we do analysis

(2) some of the dataset have a specific length for the attribute, if the Agency Name length is large, some part of the String will be lost, and the information is not complete

(3) there will be different way to represent a same Agency Name for example, the following two tables have the same type of data but with completely different present way.

1st dataset	2nd dataset
BRONX COMMUNITY BOARD 2	COMMUNITY BOARD NO.4-MANHA
BRONX COMMUNITY BOARD 3	COMMUNITY BOARD NO.9-QUEEN
BRONX COMMUNITY BOARD 4	COMMUNITY BOARD NO.17-BROO
BRONX COMMUNITY BOARD 5	COMMUNITY BOARD NO.8-BRONX
BRONX COMMUNITY BOARD 6	COMMUNITY BOARD NO.5-MANHA
BRONX COMMUNITY BOARD 7	COMMUNITY BOARD NO.10-BROO
BRONX COMMUNITY BOARD 8	COMMUNITY BOARD NO.11-QUEE
BRONX COMMUNITY BOARD 9	COMMUNITY BOARD NO.2-QUEEN
BRONX COMMUNITY BOARD 10	COMMUNITY BOARD NO.7-QUEEN
BRONX COMMUNITY BOARD 11	COMMUNITY BOARD NO.12-MANH
BRONX COMMUNITY BOARD 12	COMMUNITY BOARD NO.4-BROOK
BRONX COMMUNITY BOARD 1	COMMUNITY BOARD NO.6-BROOK

We check lots of other datasets and find that the first version to present it is more frequently, and the second version with incomplete district name. Therefore, to make the data more consistent, we decided to change the 2nd version to the 1st one.

According to the new problems we found, we design a new clean method to refine the original one

(1) we delete the space at the front and end of each String

```
    def elimateBlank(x):
        x = x.strip()
    return x;
```

(2) we design a identical function to make two types of representations become the same

```
def identical(x):
    if len(x.split('␣')) == 3 and x.split('␣')[0] == 'COMMUNITY' and x.split('␣')[1] == 'BOARD':
        number = x.split('.')[1].split('-')[0]
        reg = x.split('-')[1]
        print(len(reg))
        reg = reg[0:4]
        print(reg)
        if reg == 'QUEE':
            c_reg = 'QUEENS'
            x = c_reg + '␣' + 'COMMUNITY␣BOARD␣#' + number
        if reg == 'BROO':
            c_reg = 'BROOKLYN'
            x = c_reg + '␣' + 'COMMUNITY␣BOARD␣#' + number
        if reg == 'BRON':
            c_reg = 'BRONX'
            x = c_reg + '␣' + 'COMMUNITY␣BOARD␣#' + number
        if reg == 'MANH':
            c_reg = 'MANHATTAN'
            x = c_reg + '␣' + 'COMMUNITY␣BOARD␣#' + number
        if reg == 'RICH':
            c_reg = 'RICHMOND'
            x = c_reg + '␣' + 'COMMUNITY␣BOARD␣#' + number
    return x
```

(3) we manually complete the Agency Name for the specific rows to let the dataset become more complete, then we can get the references data

```
def completeTheDatas(x):
    if x == 'DEPARTMENT␣OF␣CITYWIDE␣ADM':
        x = 'DEPARTMENT␣OF␣CITYWIDE␣ADMINISTRATIVE␣SERVICES'
    if x == 'HRA/DEPARTMENT␣OF␣SOCIAL␣S':
        x = 'HRA/DEPARTMENT␣OF␣SOCIAL␣SERVICES'
    if x == 'CUNY␣QUEENSBOROUGH␣COMMUNI':
    #extra...
    return x
```

the new clean method is as below

```
def clean_new(x):
    x = toUpperCase(x)
    x = elimateBlank(x)
    x = identical(x)
    return x
```

Then, we apply the new clean method to the dataset and do the celan through spark to get the 2nd version of the clean data. For each dataset, we named it as dataset new

```
agency_name_new1 = dataset1.map(lambda x: (clean_new(x[3]),1)).countByKey()
for key, value in agency_name_new.items():
    print(key, value)
```

For some dataset, we needed manually update. We usee the clean method as below, through spark to get the 2nd version of the clean data. For each dataset, we named it as dataset d. And in the analysis part, we assumed the dataset d as the correct dataset.

```
def clean_for_dataset1(x):
    x = toUpperCase(x)
    x = elimateBlank(x)
    x = identical(x)
    x = completeTheDatas(x)
    return x
```

```
agency_name_d1 = dataset1.map(lambda x: (clean_for_dataset1(x[3]),1)).countByKey()
for key, value in agency_name_d1.items():
    print(key, value)
```

4.4.4 *Create the reference dataset for Agency Name.*

by combining the ten datasets from above, we can get a Agency Name reference data

```
reference_data_agency_name = set(agency_name_d10).union(set(agency_name_d9).union(set(agency_name_d8).union(set(
    agency_name_d7).union(set(agency_name_d6).union(set(agency_name_d5).union(set(agency_name_d4).union(set(
    agency_name_d3).union(set(agency_name_d2).union(set(agency_name_d1)))))))))
```

4.5 'Work Location Borough' Attribute

4.5.1 *Original Method in Part 1.*

In the NYCPayroll dataset, we used the following steps to do cleaning on the date column. Finding out the most frequent values in column 'Work Location Borough' gave us an overall picture of this data column, letting us realize 'MANHATTAN', 'QUEENS', 'BROOKLYN' and 'BRONX' are among the top four most popular work location boroughs. In addition, getting a set of distinct values for this column made us know how many different work locations in total are there as well as their counts respectively. Finally, we used different kinds of knn cluster methods to find out more data issues and analyze them accordingly.

(1) find out the frequencies of top values
(2) find out the frequencies of all distinct values
(3) cluster work location borough using key collision
(4) change all the data fields to be upper cases
(5) change empty values to be "UNKNOWN"

```
profiles.column('Work_Location_Borough').get('topValues')
wlb = ds.distinct('Work_Location_Borough')
for rank, val in enumerate(wlb.most_common()):
    wlb, freq = val
    print(f'{rank_+_1:<3}_{wlb}__{freq:>10,}')

def print_k_clusters(clusters, k=5):
    clusters = sorted(clusters, key=lambda x: len(x), reverse=True)
    val_count = sum([len(c) for c in clusters])
    print('Total_number_of_clusters_is_{}_with_{}_values'.format(len(clusters), val_count))
    for i in range(min(k, len(clusters))):
        print('\nCluster_{}'.format(i + 1))
        for key, cnt in clusters[i].items():
            if key == '':
                key = "''"
            print(f'__{key}_(x_{cnt})')
```

```

work_locations = ds.select('Work_Location_Borough').distinct()
clusters = KeyCollision(func=Fingerprint()).clusters(work_locations)

def work_location_borough(x):
    if x == '':
        return "UNKNOWN"
    else:
        return x.upper()

ds = update(ds, columns='Work_Location_Borough', func = work_location_borough)
```

By running the code above and utilizing the openclean package, we were able to find the following abnormal values:

BRONX

bronx

MANHATTAN

manhatton

QUEENS

queens

RICHMOND

Richmond

(null)

However, as mentioned before, if our dataset becomes increasingly larger, we will face the challenge that the dataset can't be fed into the limited memory. Therefore, we'll use spark/pyspark to continue our work on part two and do the data cleaning on columns which are similar to or the same as 'work location borough'.

4.5.2 *Apply to New Datasets in Part 2.*

In order to make sure if the techniques for cleaning this kind of data columns are general and can be used to generate reference data, we need to find as many datasets as possible that include 'borough' columns.

```
%spark.pyspark
dataset1 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.k397-673e.csv").mapPartitions(lambda
        x: reader(x))
dataset2 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.ic3t-wcy2.csv").mapPartitions(lambda
        x: reader(x))
dataset3 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.rbx6-tga4.csv").mapPartitions(lambda
        x: reader(x))
dataset4 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.vz8c-29aj.csv").mapPartitions(lambda
        x: reader(x))
dataset5 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.6khm-nrue.csv").mapPartitions(lambda
        x: reader(x))
dataset6 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.avir-tzek.csv").mapPartitions(lambda
        x: reader(x))
dataset7 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.xywu-7bv9.csv").mapPartitions(lambda
        x: reader(x))
dataset8 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.3qem-6v3v.csv").mapPartitions(lambda
        x: reader(x))
dataset9 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.ur7y-ziyb.csv").mapPartitions(lambda
        x: reader(x))
dataset10 = sc.textFile("hdfs:/user/CS-GY-6513/project_data/data-cityofnewyork-us.esmb-8zkm.csv").mapPartitions(
       lambda x: reader(x))
```

```

%spark.pyspark
work_location_borough1 = dataset1.map(lambda x : (x[7], 1)).countByKey()
for key, value in work_location_borough1.items():
    print(key, value)

work_location_borough2 = dataset2.map(lambda x : (x[2], 1)).countByKey()
for key, value in work_location_borough2.items():
    print(key, value)

%spark.pyspark
work_location_borough3 = dataset3.map(lambda x : (x[4], 1)).countByKey()
for key, value in work_location_borough3.items():
    print(key, value)

... ...
```

The above code locates ten datasets that contain columns related to New York boroughs. By using pyspark, we can easily obtain much faster speed to even concurrently find the frequencies of each data field of each dataset. After that, all the related issues become crystal clear merely by inspecting those data fields as well as their frequencies:

1. Characters of a location borough are not capitalized.
2. Empty string of a location borough.
3. Some white-spaces at the front or the tail of a location borough.

4.5.3 *Refine the Techniques Used in Part 1 for New Datasets.*

First, it is both reasonable to make all of the characters of a location borough to be lower cases or capitalize each character given a non-empty location borough, and we decided to choose the latter. Besides, there are multiple ways of dealing with empty string location boroughs, and we decided to change them to be "UNKNOWN", which is more reasonable because we actually don't know what data fields they are. Finally, in terms of the white-spaces, we decided to eliminate them with no doubt. Therefore, the refined cleaning techniques are as follows:

```
def clean(x):
    x = x.strip()
    if x != '':
        return x.upper()
    else:
        return "UNKNOWN"
```

For example, we'll apply the techniques to the dataset seven. Before applying the techniques:

```
%spark.pyspark
work_location_borough7 = dataset7.map(lambda x : (x[1], 1)).countByKey()
for key, value in work_location_borough7.items():
    print(key, value)

NYC Total 1
   Bronx 1
   Brooklyn 1
   Manhattan 1
   Queens 1
   Staten Island 1
```

After applying the refined techniques:

```
%spark.pyspark
%spark.pyspark
temp = dataset7.map(lambda x: (clean(x[1]), 1)).countByKey()
for key, value in temp.items():
    print(key, value)

NYC TOTAL 1
BRONX 1
BROOKLYN 1
MANHATTAN 1
QUEENS 1
STATEN ISLAND 1
```

It's now clear that the relevant issues with the dataset seven have been resolved.

5 RESULTS

For big data analysis, we need data to be as clean as possible to avoid the incorrect data record affecting our data research. So the outcome of the data clean results is extremely important. Since the above method we proposed cannot change correct data to the wrong data, we decided to use recall for the String part of the analysis. And for other data, we finally get a precise version to check the data problem and fix it to the right version.

5.1 The cleaning results of each attribtue

5.1.1 Mid Name initial.

It is easy to get the Right Mid Name initial by the above method, we can just pick all the characters form 'A' to 'Z' and make everyone is uppercase. It can have a reference data just be 'A,B,C,....,Z'.

5.1.2 Agency Start Date.

To dealing with the Date attribute, we can find the illegal date and future date by sql. So for each dataset, we can set all the illegal date and futrue data to the Null value. The only error data can exist in the dataset is that some mistaken input with the right date in the past. It just a tiny part of the all datasets, and will not have any big influence in the following data analysis.

5.1.3 Agency Name.

We use the recall to check the percentage of the dirty data and the numbers of the data we clean

(1) the whole number of data needed to be clean is: the precision data length to subtract the length of the intersection between the origin data and the precision data
(2) the original clean data volume is: the difference of the origin dataset and the 1st clean dataset
(3) the new clean data volume is: the difference of the origin dataset and the 2nd clean dataset

```
arr_origin = []
arr_old = []
arr_new = []
arr_pres = []
for key, value in agency_name1.items():
    arr_origin.append(key)
```

```
for key, value in agency_name_old1.items():
    arr_old.append(key)
for key, value in agency_name_new1.items():
    arr_new.append(key)
for key, value in agency_name_d1.items():
    arr_pres.append(key)

clean_recall = len(set(arr_old).difference(set(arr_origin))) / (len(arr_pres) - len(set(arr_origin).intersection(set(arr_pres))))
clean_new_recall = len(set(arr_new).difference(set(arr_origin))) / (len(arr_pres) - len(set(arr_origin).intersection(set(arr_pres))))

print(clean_recall)
print(clean_new_recall)
```

The result of the anylasis in the 10 dataset is as below:

dataset	clean recall	clean new recall
dataset1	0.0	0.6129
dataset2	1.0	1.0
dataset3	1.0	1.0
dataset4	1.0	1.0
dataset5	1.0	1.0
dataset6	1.0	1.0
dataset7	1.0	1.0
dataset8	0.5	0.5
dataset9	1.0	1.0
dataset10	0.0238	1.0

As we can see from the above table, the accuracy of the first clean method is lower than the new clean method for some datasets, which may have the problem of space in the end and front. By using the new clean method, we can get cleaner data, but there still will have some data inconsistent that need us to manually correct. And then we can use the final data to generate the Agency Name reference data.

5.1.4 Work Location Borough.

Inherently the refined techniques have merely one more operation than the original methods for cleaning the 'work location borough' column - eliminating the leading and tailing white-spaces. This is because when we did data cleaning on this column in part one, there was no such issues within that dataset. When we manage to experiment with more datasets in part two, such issues become clear.

Therefore, it becomes quite simple in this case to measure their effectiveness. The number of operations in the original techniques are m = 2; The number of operations in the refined techniques are n = 3; Suppose at most 3 operations are needed to finish data cleaning with regard to columns which are similar to Work Location Borough, we can see that m / n = 2/3, and thus 2/3 should be the overall performance evaluation of our original data cleaning techniques.

5.2 The reference data for some attributes

5.2.1 *Agency Name.*

The final reference data generated has 504 lines.

```
0
1       DEPARTMENT OF YOUTH AND COMMUNITY DEVELOPMENT
2                 CITY UNIVERSITY CONSTRUCTION FUND
3                   OFFICE OF COLLECTIVE BARGAINING
4                                        SANITATION
..                                              ...
501          PORT AUTHORITY OF NEW YORK & NEW JERSEY
502                          SMALL BUSINESS SERVICES
503                       DEPT OF PARKS & RECREATION
504  DISABILITIES, OFFICE FOR PEOPLE WITH DEVELOPME...

[505 rows x 1 columns]
FINISHED
```

5.2.2 *Work Location Borough Reference Data Generation.*

The final reference data generated are as follows:

```
BROOKLYN
MANHATTAN
BRONX
RICHMOND
QUEENS
WASHINGTON DC
ULSTER
WESTCHESTER
ALBANY
NASSAU
DELAWARE
SULLIVAN
ORANGE
SCHOHARIE
DUTCHESS
GREENE
PUTNAM
STATEN ISLAND
```

6 REFERENCES

https://data.cityofnewyork.us/City-Government/Civil-List/ye3c-m4ga

https://data.cityofnewyork.us/City-Government/Local-Law-18-Pay-and-Demographics-Report-Agency-Re/423i-ukqr

https://data.cityofnewyork.us/City-Government/Agency-Spending-by-Budget-Function/gzfs-3h4m

https://data.cityofnewyork.us/City-Government/Greenbook/mdcw-n682

https://data.cityofnewyork.us/City-Government/Expense-Budget/mwzb-yiwb

https://data.cityofnewyork.us/City-Government/Citywide-Payroll-Data-Fiscal-Year-/k397-673e

https://data.cityofnewyork.us/City-Government/Civil-Service-List-Active-/vx8i-nprf

https://data.cityofnewyork.us/City-Government/Civil-Service-List-Certification/a9md-ynri

https://data.cityofnewyork.us/City-Government/SCOUT-CORE/xrwg-eczf
https://data.cityofnewyork.us/City-Government/FY2018-PMMR-Spending-and-Budget/qbjq-atxv
https://data.cityofnewyork.us/City-Government/Citywide-Payroll-Data-Fiscal-Year-/k397-673e
https://data.cityofnewyork.us/Housing-Development/DOB-Job-Application-Filings/ic3t-wcy2
https://data.cityofnewyork.us/Housing-Development/DOB-NOW-Build-Approved-Permits/rbx6-tga4
https://data.cityofnewyork.us/Education/Borough-Enrollment-Offices/vz8c-29aj
https://data.cityofnewyork.us/Social-Services/Demographics-by-Borough/6khm-nrue
https://data.cityofnewyork.us/City-Government/New-York-City-Population-by-Borough-1950-2040/xywu-7bv9
https://data.cityofnewyork.us/Social-Services/Buildings-by-Borough-and-Community-District/3qem-6v3v
https://data.cityofnewyork.us/Social-Services/Associated-Address-by-Borough-and-Community-Distri/ur7y-ziyb
https://data.cityofnewyork.us/Education/School-Based-Programs-by-Borough/esmb-8zkm