Final Project Report - Team CreaTAK

Ankush Jain
aj2885@nyu.edu
NYU Tandon School NYU
of Engineering
New York, NY

Theodore Hadges
th2313@nyu.edu
NYU Tandon School
of Engineering
New York, NY

Ruinan Zhang
rz1109@nyu.edu
NYU Tandon School
of Engineering
New York, NY

Abstract

The purpose of this report is to explain the findings and conclusions made by our team during the final project for the Big Data (CS-GY 6513) fall 2019 course. It includes two tasks (1) {**generic profiling**} of NYC Open Data (2) {**semantic profiling**} of a subset of NYC Open Data. For task1, we sampled in total 150 out 1900 (a total of 1159 were processed) number of files and identified an average of 10.2 integer types, 11.4 real(double) types, 1.07 data_time types and 4.71 text types. We found there are an average of 175460.3 out of all values are empty in the dataset which is approximately 22%. For task 2, we analyzed 260 columns and we were able to identify the semantic types for 210 columns with a precision of 72.40%

1. Introduction

The entire idea of generic data profiling is to produce metadata about the different datasets on the basis of different information that can be mined within the dataset so as to enable us in making better business decisions. In our case, for generic profiling (task 1), we processed all XXX datasets along all of its columns.. Our goal was to find the number of non-empty cells, number of empty cells, number of distinct values, the top-5 most frequent value(s), and the different data types (integer, real, string and date-time). For different data types, we performed different statistical operations (min, max, most frequent values and so on). At the same time, we also identified the columns that are candidates for being keys of that specific table. In essence, our approach for task one was to iterate over every column of each dataset. We first used ***pyspark.sql.DataFrameReader*** *and its* **InferSchema** function to check the default data types. Once, a baseline was set with the default types, we began interpreting the string type columns for mining various other data types such as date-time, integers and real values. Python's *dateutil.parser* was utilized for interpreting dates however, special care was taken to prevent false positive string identifications by employing Regular Expressions.

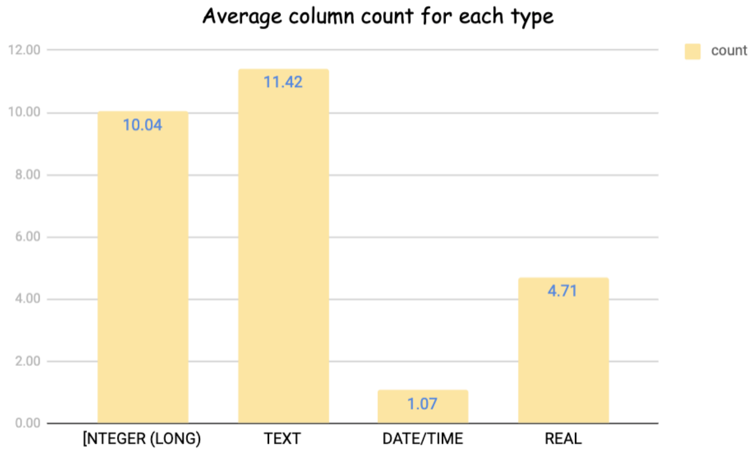
Next step towards a successful understanding of the dataset is by studying the underlying semantics of the dataset. Semantic profiling enables us to gather knowledge about the domain of the data. If we are aware of what the domain of the data source is, we can profile it better and gain useful insights. For semantic profiling (task 2), we worked with a subset of total 260 columns of the NYC Open Data and identified and summarized the semantic types of these columns. For different types, we used different methods to identify its types.

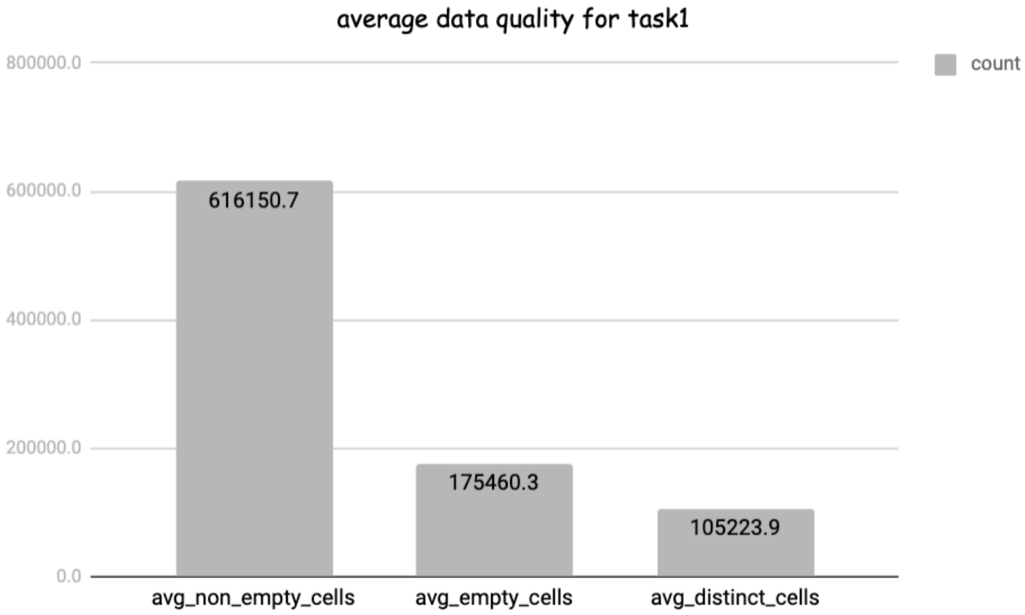
Regular Expressions & Patterns were used to identify phone number, website, zip code, building codes, locations, addresses, house numbers, school names, parks etc. These types are either very distinct patterns e.g. phone number(ddd-ddd-dddd) or share a lot of similar words e.g. most of the street names have St, Ave and so on. For the types that don't have these features like person names, we took one of two approaches, depending on the type. For categories deemed to contain mostly distinct values but with context, we used **Named-Entity Recognition** (NER), **Soundex** and natural language processing to label them. Otherwise, for sets with many repetitive items, we generated **Frequent Item Lists** against which a given column's row value can be compared. If the item under consideration is in a given set, then the row containing that item is classified as the set's category.

2. Task 1 - Generic Profiling

2. 1 Data summary & Evaluation

From the output of all the data-sets we processed from NYC Open data, we found out the following average stats:

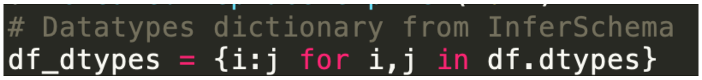




2. 2 Method

2.2.1 Method Description

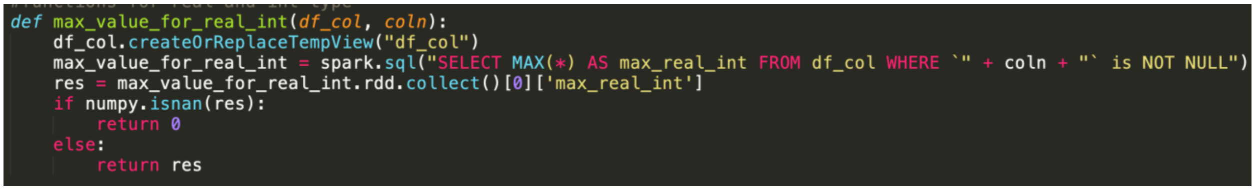
* Github Link for Task 1 Code:
https://github.com/theodorehadges/big_data_course_project/tree/develop/task1
For task 1, we first iterate over every column for each data set in the NYC open data, and used ***pyspark.sql.DataFrameReader(spark)*'s InferSchema** to check its data type：



```
# Datatypes dictionary from InferSchema
df_dtypes = {i:j for i,j in df.dtypes}
```

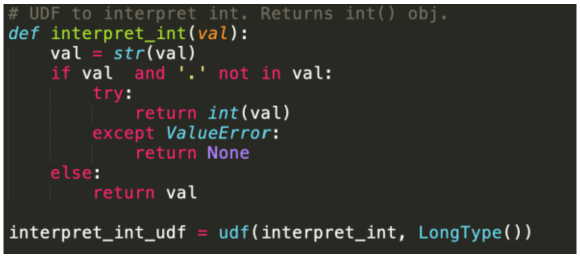
This might return ['int', 'bigint', 'tinyint', 'smallint']for Integer type, ['float', 'double' , 'decimal'] for Real type, ['timestamp', 'date', 'time', 'datetime'] for Date_Time type and ['string'] for Text type. However, since InferSchema will go over the column at once and if even one of the rows is 'string' or invalid value, it will return 'string' as the datatype for the whole column. So, just using InferSchema wasn't sufficient to identify the column types that might be dirty or have multiple data types and invalid values or outliers.

For Integer, Real and Date_Time(timestamp) returned by InferSchema, we compute the relevant statistics using Spark SQL. An example for this is the below function to compute max and min for integer:



```
def max_value_for_real_int(df_col, coln):
    df_col.createOrReplaceTempView("df_col")
    max_value_for_real_int = spark.sql("SELECT MAX(*) AS max_real_int FROM df_col WHERE `" + coln + "` is NOT NULL")
    res = max_value_for_real_int.rdd.collect()[0]['max_real_int']
    if numpy.isnan(res):
        return 0
    else:
        return res
```

Next, we especially picked the ‘string’ type columns returned by the InferSchema and tried to interpret these particular columns as int, real and date_time and strings by performing explicit type casting. If it can be returned as these three types, we return that type’s object, if not, we keep it as string and move to the next item for interpretation. This entire process is applied using Spark UDFs so it is distributed. An example for interpreting integer is:



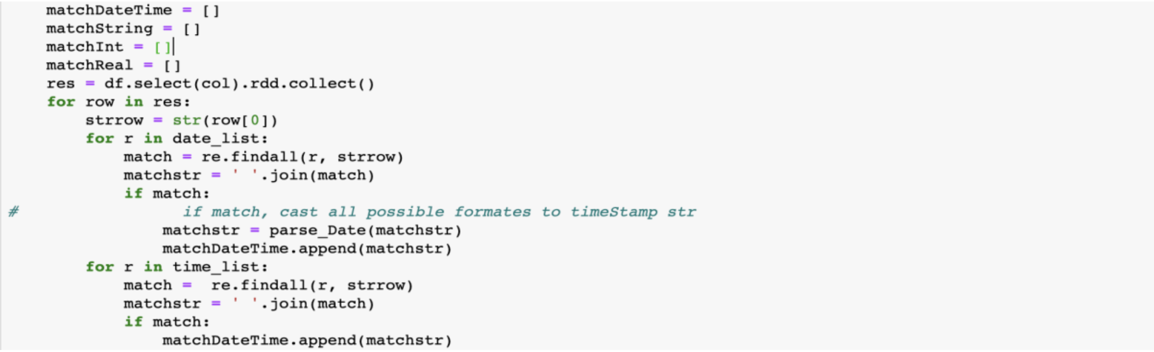
```python
# UDF to interpret int. Returns int() obj.
def interpret_int(val):
    val = str(val)
    if val  and '.' not in val:
        try:
            return int(val)
        except ValueError:
            return None
    else:
        return val

interpret_int_udf = udf(interpret_int, LongType())
```

Specifically, for date_time and string types, we had three different versions of methods to interpret and parse their values since the formats of date and time can be very various and using libraries such as ***Python’s dateutil.parser is not accurate as it can result in false positive datetime interpretation which can cause harm to the authenticity of the datasets***. We will explain these two types further in the next section Challenges and Optimizations

2.2.2 Challenges and Optimizations

Initial Failure: At the very beginning stage of the project, we tried iterating over all the columns of the dataset (row wise). But this resulted in an extremely slow script because of losing the Spark scalability to sequential execution.

To solve this problem, the Spark dataframes and RDDs were strictly used for processing. Further, another problem popped up : (1) using InferSchema has a limit in identifying columns with multiple values (2) date_time type has many different formats, so it’s hard to distinguish them from strings , we tried to iterate every cell and check its types using regex. However, after executing a few files, we soon realized this method will be too time consuming as it has to iterate every value and compare it to four different regex lists.

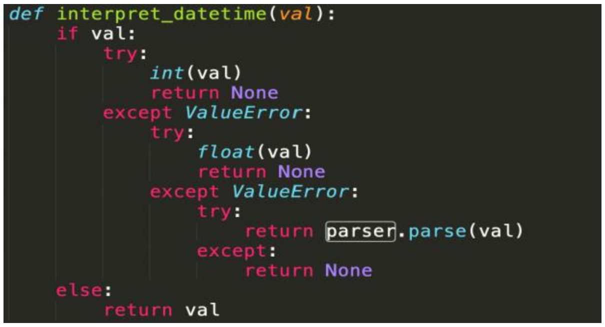


```python
    matchDateTime = []
    matchString = []
    matchInt = []
    matchReal = []
    res = df.select(col).rdd.collect()
    for row in res:
        strrow = str(row[0])
        for r in date_list:
            match = re.findall(r, strrow)
            matchstr = ' '.join(match)
            if match:
#                  if match, cast all possible formates to timeStamp str
                matchstr = parse_Date(matchstr)
                matchDateTime.append(matchstr)
        for r in time_list:
            match =  re.findall(r, strrow)
            matchstr = ' '.join(match)
            if match:
                matchDateTime.append(matchstr)
```

Optimization: Then, we come up with the idea to try to interpret values in all the columns that the return type is ‘string’ after implementing InferSchema.

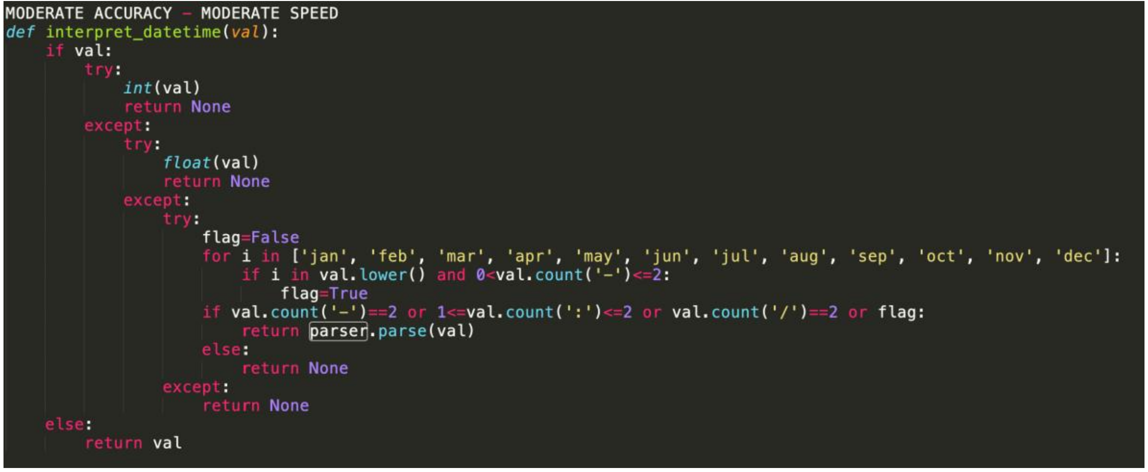
However, we faced the third problem (3) using library to parse the date_time might be fast but it’s not as accurate as checking with a regex list, but checking regex list will be too time consuming especially for large data-sets. (We have about 6 data sets that’s larger than 1 GB)
So we came up with three versions of interpreting sting and date_time :

(a) Use Python’s dateutil.parser, which is low accuracy but high speed:



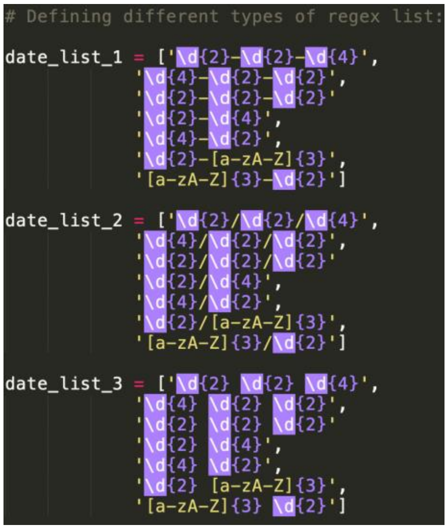
```
def interpret_datetime(val):
    if val:
        try:
            int(val)
            return None
        except ValueError:
            try:
                float(val)
                return None
            except ValueError:
                try:
                    return parser.parse(val)
                except:
                    return None
    else:
        return val
```

(b) Use both Python’s dateutil.parser and a single regex check, which is a moderate speed and accuracy:



```
MODERATE ACCURACY - MODERATE SPEED
def interpret_datetime(val):
    if val:
        try:
            int(val)
            return None
        except:
            try:
                float(val)
                return None
            except:
                try:
                    flag=False
                    for i in ['jan', 'feb', 'mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct', 'nov', 'dec']:
                        if i in val.lower() and 0<val.count('-')<=2:
                            flag=True
                    if val.count('-')==2 or 1<=val.count(':')<=2 or val.count('/')==2 or flag:
                        return parser.parse(val)
                    else:
                        return None
                except:
                    return None
    else:
        return val
```

(c) Use both dateutil.parser a list of more accurate regex, which is low speed but high accuracy (shown only the regex list we defined):



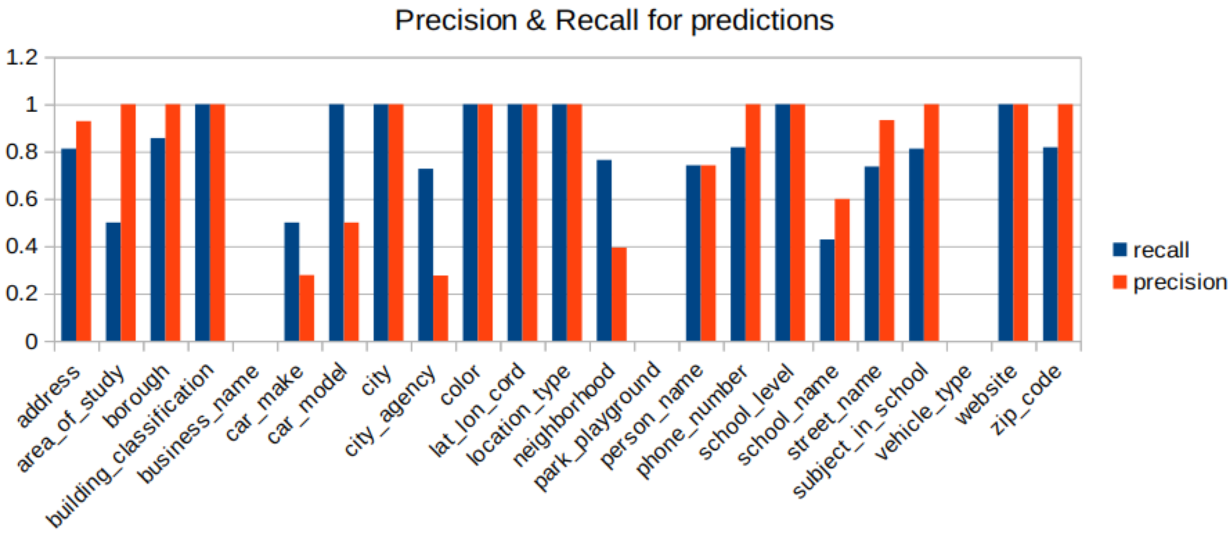
```
# Defining different types of regex list:

date_list_1 = ['\d{2}-\d{2}-\d{4}',
               '\d{4}-\d{2}-\d{2}',
               '\d{2}-\d{2}-\d{2}'
               '\d{2}-\d{4}',
               '\d{4}-\d{2}',
               '\d{2}-[a-zA-Z]{3}',
               '[a-zA-Z]{3}-\d{2}']

date_list_2 = ['\d{2}/\d{2}/\d{4}',
               '\d{4}/\d{2}/\d{2}',
               '\d{2}/\d{2}/\d{2}'
               '\d{2}/\d{4}',
               '\d{4}/\d{2}',
               '\d{2}/[a-zA-Z]{3}',
               '[a-zA-Z]{3}/\d{2}']

date_list_3 = ['\d{2} \d{2} \d{4}',
               '\d{4} \d{2} \d{2}',
               '\d{2} \d{2} \d{2}'
               '\d{2} \d{4}',
               '\d{4} \d{2}',
               '\d{2} [a-zA-Z]{3}',
               '[a-zA-Z]{3} \d{2}']
```

The idea behind this is to execute different sizes of data-sets using different methods, for larger data-sets, we switched to the (a) method and for smaller data-sets, we used (b) or (c) methods. In that way, we managed to find a balance between the speed and accuracy which we think is also applicable for other applications and projects.

3. Task 2 - Semantic Profiling

3. 1 Data summary & Evaluation

The dataset was manually labelled and then the semantic classification of the columns from various dataset was performed. A list of the 23 most suitable labels was created for the datasets, as given in the below visualization. An average recall of 71.38% was seen and an average precision of 72.40% was observed .



3. 2 Methods

3.2.1 Method Description

*Github code link for task 2:
https://github.com/theodorehadges/big_data_course_project/tree/develop/task2
For task 2, our approach can be divided into three parts for finding different semantic types:

(a) **Regex** for checking types with very distinct patterns; e.g., phone number(ddd-ddd-dddd) or sharing lots of similar words
(b) **Named-entity recognition (NER)** and natural language processing for checking types that don’t have a distinct pattern or similar words but can be identified with NER tags. The SpaCy implementation ‘en_core_web_md’ has been used. This technique is used for identifying cities and people’s names. The PERSON and GPE tags were used.
(c) Comparing row values against **frequent item lists**, where those lists contain the most frequent items for each category. If a row value is contained in the frequent item list, then the row is classified as the category from which the frequent item list was derived. This is a useful approach when there are many repeated values in a category.

For each method, we defined different functions to identify different types, and in the function we return the percentage of how many rows were identified as such types, if it’s larger than a certain threshold, we return the found_type and total count of rows that identified as such type.
An example can be found in the **Method (a) Regex section’s re_find_street_address** function.

3.2.1.1 Method (a) Regex:

(i) Data types found by this method are:
Phone number, website, zip code, building Code, lat_lon, street, address, school name, house number, school subjects, school level
(ii)Method description: From observation by doing the manual labels of all the columns assigned to our group, we found out that we are able to identify types of distinct patterns and having lots of similar words through regex.
Types of distinct patterns includes:

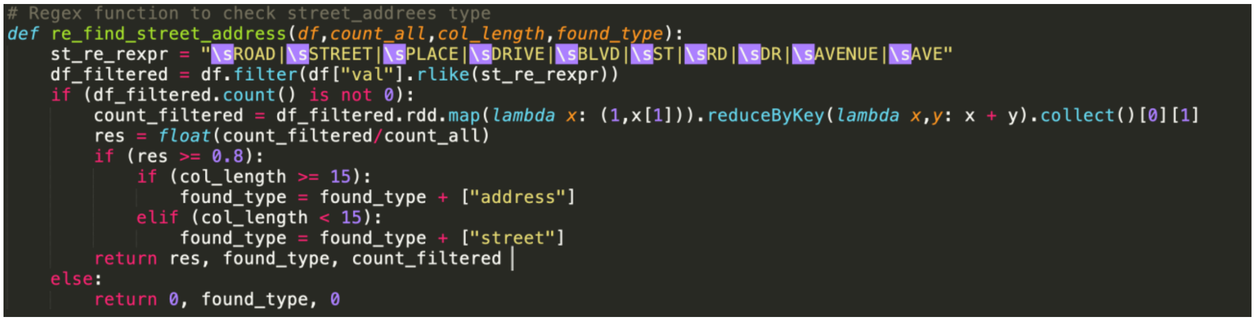
1. Phone number, which can be found by matching it with
   phone_re_rexpr = "^\d{10}?$|^\(\d\d\d\)\d\d\d\d\d\d\d$|^\d\d\d\-\d\d\d\-\d\d\d\d$"
2. Zip code: ip_re_rexpr = "^\d{5}?$|^\d{5}?-\d\d\d$|^\d{8}?$"
3. Building code: "([A-Z])\d\-"
4. Website: web_re_rexpr = "WWW\.|\.COM|HTTP\:|HTTP"
5. Lat_lon:ll_re_rexpr = "\([-+]?[0-9]+\.[0-9]+\,\s*[-+]?[0-9]+\.[0-9]+\)"
6. House number: houseNo_re_rexpr = "^\d{2}?$|^\d{3}?$|^\d{4}?$"

Types having many similar words includes:

1. Stress and Address:
   st_re_rexpr"\sROAD|\sSTREET|\sPLACE|\sDRIVE|\sBLVD|\sST|\sRD|\sDR|\sAVENUE|\sAV"

Especially for Street and Address, these two types share many similar words such as Road, Street, St, Ave and so on. So during the execution, it always labeled them together. To fix this problem, we compute the average length for all the columns. Because from observation, we found out address types usually are longer than street names in length. So after a few tests, we set a threshold for the length of column for address to >15 and for street names <=15. So in this way, we successfully distinguished these two types.

Example of regex function **re_find_street_address:**



```
# Regex function to check street_addrees type
def re_find_street_address(df,count_all,col_length,found_type):
    st_re_rexpr = "\sROAD|\sSTREET|\sPLACE|\sDRIVE|\sBLVD|\sST|\sRD|\sDR|\sAVENUE|\sAVE"
    df_filtered = df.filter(df["val"].rlike(st_re_rexpr))
    if (df_filtered.count() is not 0):
        count_filtered = df_filtered.rdd.map(lambda x: (1,x[1])).reduceByKey(lambda x,y: x + y).collect()[0][1]
        res = float(count_filtered/count_all)
        if (res >= 0.8):
            if (col_length >= 15):
                found_type = found_type + ["address"]
            elif (col_length < 15):
                found_type = found_type + ["street"]
        return res, found_type, count_filtered
    else:
        return 0, found_type, 0
```

2. School name:
   school_re_rexpr =
   "\sSCHOOL|\sACADEMY|HS\s|ACAD|I.S.\s|IS\s|M.S.\s|P.S\s|PS\s|ACADEMY\s"
3. School subjects:
   school_subj_re_rexpr = "^ENGLISH$|^ENGLISH\s[0-9]?$|^MATH\s[A-Z$]|^MATH$|^SCIENCE$|^SOCIAL\STUDIES$|^ALGEBRA\s[A-Z]$|\^CHEMISTRY$|...etc"
4. School Level:
   schlvl_re_rexpr = "^[K]\-\d?$|^HIGH SCHOOL$|^ELEMENTARY$|^ELEMENTARY SCHOOL$|^MIDDLESCHOOL$|...etc”

3.2.1.2 Method (b) NER & Soundex:

(i) Data types found by this method are:

NER: City, Person_names

Soundex: car_make, car_models, colors

(ii)Method description:

Named Entity Recognition - It is one of the sub tasks in information retrieval. It can be used to identify various information regarding the semantics of the text. It can give information about people, places, time, quantities, money, organizations etc. In our application of task 2. We have used NER to identify people and cities. The people have been identified using the PEOPLE tag and the cities have been identified using the GPE (Geo Political Entity) tag. The Spacy implementation of NER is used. It’s the 'en_core_web_md', the medium sized, web scraped model that has been used to perform NER.

Soundex - This algorithm has been used to identify similar or phonetically similar sounding words together. This is a good approach for identifying the color names and the car makers. It helps in converging the misspelled words together.

3.2.1.3 Method (c) similarity and frequent item comparison:

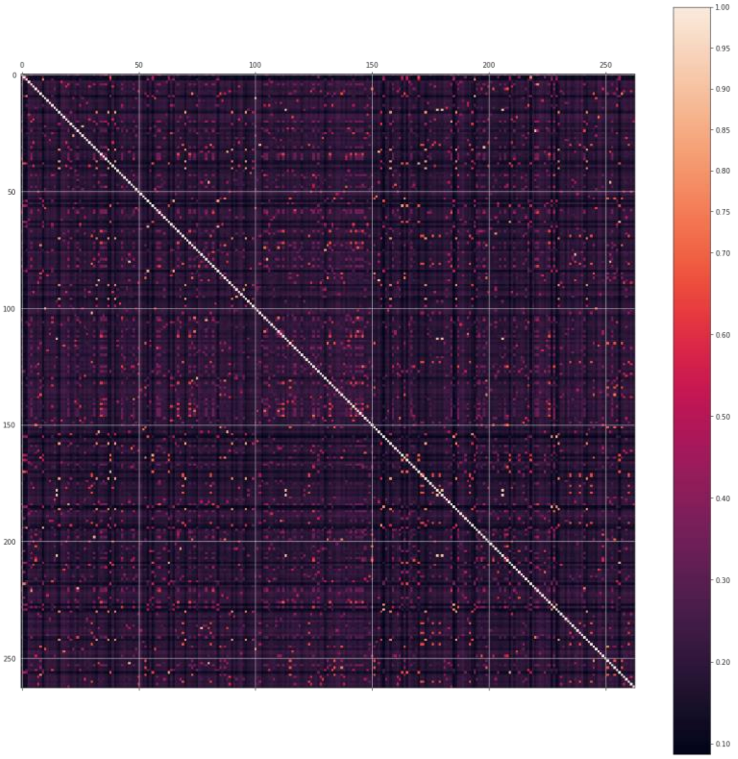
(i) Data typtages found by this method are:

Business names, areas of study, city agency, location type, school subject, parks & playgrounds

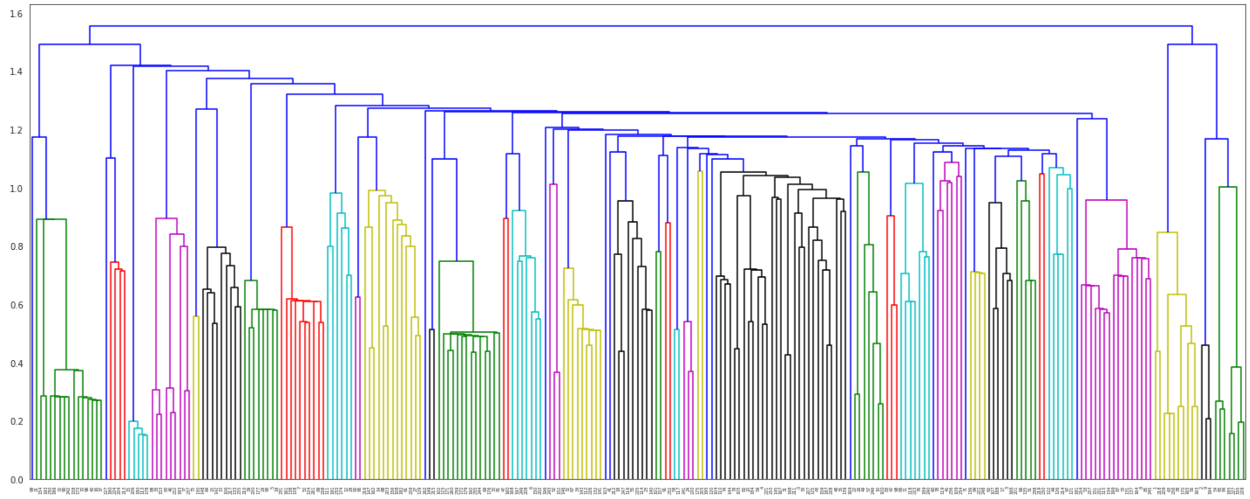
(ii)Method description:

The pipeline for this method is as follows:

First, we need to determine the categories of each file. To achieve this, we decided to group similar files together. In similarity.py, we used file names as a proxy for file content similarity and applied cosine similarity with 3-grams onto the filenames to create a m x m similarity matrix. Once the matrix was computed, we passed the similarity matrix into the scipy cluster hierchy linkage() function to generate a linkage matrix, Z. Finally, Z was passed into scipy's fcluster() function to form flat clusters from Z. We made a simple python dictionary using the resulting fcluster structure with key value pairs: clusterN: [filenameR, filenameS], where all filenames in clusterN are similar. In Fig x. We can see the similarity matrix (with axes file_names x file_names). Since there are many bright dots, there are many pairs of similar file names. This motivated us to continue to the clustering step.



Below, in Fig. x, a dendrogram of the clusters. The lower portions are the resulting files which were mapped to the same clusters. As you can see (from their similar heights), many file names were correctly mapped to clusters containing files with similar file names. We wrote this final dictionary to a JSON file.



Next, once the file names were clustered we identified files which were mapped to the same clusters as being the same type. Of course, since this clustering was unsupervised, it wasn’t perfectly clustered. However, we were lucky in that all of the data types we set out to classify for this part were mapped to homogenous clusters. In generate_keyword_lists_from_columns.py, we use the file clusters to generate the most frequent items in each. Each file in the cluster is loaded into a dataframe. Each row in the data frame is exploded such that each word is now in its own row. This is useful so that we can perform some filtering:

```
# split words in each row and create new df with one word per row, and count
# aos is area_of_study category
aos_df_split_words = aos_df_clean.withColumn('word', f.explode(f.split(f.col('clean_word'), ' ')))\
        .groupBy('word')\
        .count()\
        .sort('count', ascending=False)\
        .filter("word != "")
```

This results in a two-column dataframe with columns ‘word’ and ‘count’

For every file in the cluster, we take the most frequent n words (n depends on category) and union it with the n most frequent values of the next file’s most frequent n words. This results in a concise set which is representative of the majority of that category (assuming there are many repeat items in that category). We manually removed some list entries which may be present in another category to avoid ambiguity; however, this could also have been done by finding the intersection of all category sets and removing the resulting values from each set.

Finally, in task2.py, we use these frequent item lists to check if values in a given row contain any of the items in the frequent itemset for each category. If it does, then that row is classified as the type from which the itemset was derived. This task was performed by Ruinan Zhang.

3.2.2 Challenges and Optimizations

3.2.2.1 Challenges and Optimizations for NLP and NER

The biggest challenge for NER implementation was the inability to perform NLP in a distributed manner due to infrastructural restrictions of the Dumbo platform. To use NLP toolkits like SpaCy and NLTK in a distributed manner, these libraries must be installed on each of the worker nodes. Doing so with restricted permissions or with custom environment wasn't possible so a sequential approach had to be take.

There were some key decision making steps involved in making the best use of Spacy NER. During the initial trials we made use of the availabe 'en_core_web_sm 2.0', the semantic parser trained on a small web dataset that was available on the Dumbo Python 3.6.5 module. However, this being an outdated version, we made use of a custom enviroment with 'en_core_web_md 2.2' version of the parser which is trained on a much bigger and newer web datasets. There was a significant improvement in the results. With the 'en_core_web_sm 2.0' the word hits for NER were as low as 10-20%. With 'en_core_web_md 2.2', this rate incresed to as low as about 40% words being mapped and as high as 70-80% words getting mapped.

The key decision making step with a sequential approach to NER was to choose the trade-off between precision and execution time. Initially we tried to improve times for sampling. However, sampling is prone to not being representative of the entire data so we decided to discard this approach. Since, we were working on much smaller datasets, we chose to trade-off higher execution times for higher precision by using the sequential processing approach. The best solution to this would have been applying NLP in a distributed manner. This task was performed by Ankush Jain.

3.2.2.1 Challenges and Optimizations for similarity and frequent item comparison

We start this task by looking for easily classifiable categories such as city agency. We observed the data for city agency and noticed that it was very clean in that it does not contain any extra characters, and it is very similar to the official data on the nyc.gov website. Therefore we built a simple scraper, nyc_agency_scraper.py using BeautifulSoup and parsed the string names of an agency to one file and the abbreviations of that agency to another file. The idea was that we could do a simple contains() to see if a value is contained in this list. We moved on to other types in this section and realized that using sources such as these would be great for general classification, but since we can observe our datasets, there might be a smarter way to choose the list to compare against. By smarter, I mean a smaller list which yields higher accuracy. We started observing the datasets and skimming through to see items which seemed frequent and made lists for comparison. However, we realized that this is not a scalable solution for building representative itemsets and took a step back. We generated the aforementioned similarity matrix, performed clustering, and built a list generator for frequent items for each category across all files in that category. File name clustering was not perfect at first. After writing a function for jaccard similarity and experimenting with different shingle sizes, we found that 3-grams worked best. However, the clustering was not ideal. We later found a cosine similarity function and were happy to find that the results were much better. After reading about it, we learned that cosine similarity tends to be a better metric than jaccard when measuring similarity between text sets. One challenge we still face is that we would like to use frequent itemsets of size 2 or 3 in some cases (rather than just a frequent itemset of singletons). In the future, we plan on adding this functionality so that we can have better accuracy during the row classification step. This task was performed by Theodore Hadges.