Automated Data-Model Creater

By: Arjun and Naveen

Setup Steps

This Project Is written in python 2.7.x and requires prior installation of the language . Further following libraries are required for the proper functioning of the project.

Packages/Libraries Used:

---

1. **Messy Tables**(Used For Improving Type Detection):A library for dealing with messy tabular data in several formats, guessing types and detecting headers.**Minimum requirement version 0.15.1**.

DOWNLOAD LINK: HTTPS://PYPI.PYTHON.ORG/PYPI/MESSYTABLES

---

2. **Pandas:***pandas* is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.**Minimum requirement version 0.18.1**.

DOWNLOAD LINK: HTTP://PANDAS.PYDATA.ORG/PANDAS-DOCS/STABLE/INSTALL.HTML

---

3. **Numpy:**NumPy is the fundamental package for scientific computing with Python.**Minimum requirement version 1.11.1**.

DOWNLOAD LINK: HTTP://WWW.SCIPY.ORG/SCIPYLIB/DOWNLOAD.HTML

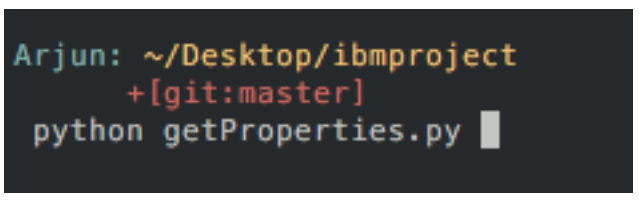
Execution Steps

Make Sure The Libraries and packages and packages are installed on your workstation.

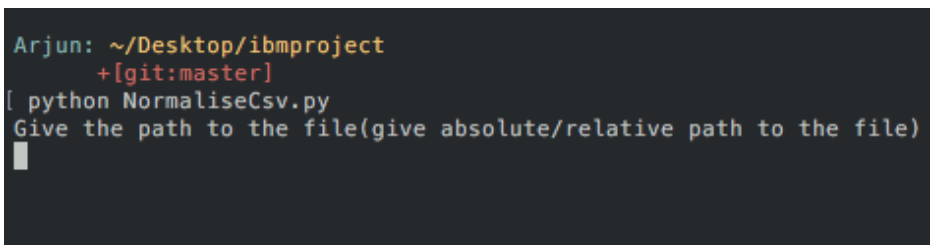
Open Terminal(Or any compiler that complies Python Scripts).

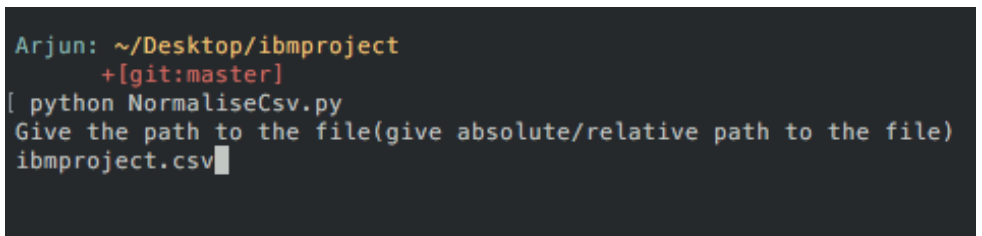
Go to the directory where the project is present,using cd command in terminal.

Now to Normalise csv file(single table) type **python NormaliseCsv.py** and to get various properties related to the file like getting the Column types type **python getProperties.py** and press enter/ return.



After running any of the above programs the program will prompt you to give the file path(path to csv file) Let me show it for python NormaliseCsv.py .Type the path to the file like in this case ibmproject.csv(this is relative path as the file is in the same folder as the script.





This is it you have to do if things go smoothly you should not see any error and you should see something like the figure below in case of NormaliseCsv.py

```
Arjun: ~/Desktop/ibmproject
      +[git:master]
[ python NormaliseCsv.py                                                                                                                                                                  ]
Give the path to the file(give absolute/relative path to the file)
ibmproject.csv

Number of Missing Values in Each Column:--

Ticket Number        0
Assigned             0
Status               0
Client               0
Category             0
Severity             1
Date Created         0
Date Last Modified   0
Response Date        0
Date Closed          0
Time Spent (min)     0
Parent Ticket        0
dtype: int64

primary_key ----------------------> ['Date Last Modified']
Main Table after Normalistion-------> ['Severity', 'Date Last Modified', 'Response Date', 'Date Closed']
Extra Tables after Normalisation----> [['Date Created', 'Ticket Number'], ['Category', 'Ticket Number'], ['Status', 'Parent Ticket'], ['Ticket Number', 'Time Spent (min)'], ['Ticket Number', 'Client'], ['
Assigned', 'Ticket Number']]
Primary Key In Main Table-----------> ['Date Last Modified']
```

And for the case of getProperties.py it should look something like the following,this is just a part of the output.

```
Arjun: ~/Desktop/ibmproject
      +[git:master]
[ python getProperties.py                                                                                                                                                                  ]
ibmproject.csv

Guessed datatypes of tables:---->

Ticket Number -------------> Bool
Assigned -------------> String
Status -------------> String
Client -------------> String
Category -------------> String
Severity -------------> String
Date Created -------------> Integer
Date Last Modified -------------> <type 'datetime.datetime'>
Response Date -------------> <type 'datetime.datetime'>
Date Closed -------------> <type 'datetime.datetime'>
Time Spent (min) -------------> Integer
Parent Ticket -------------> Integer


column name :  0
data type :  Bool
possible values :  2
possible values are :  [0 1]


column name :  1
data type :  String
possible values :  2
possible values are :  ['Manjit' 'Naveen']
Max length of the string :  6


column name :  2
data type :  String
possible values :  2
possible values are :  ['Closed' 'Open']
Max length of the string :  6


column name :  3
data type :  String
possible values :  2
possible values are :  ['Google' 'IBM']
Max length of the string :  6


column name :  4
data type :  String
possible values :  2
possible values are :  ['New Change' 'Bug']
Max length of the string :  10
```

EXPLANATION OF OUTPUT AND CODE

Program 1:Get Properties Of the Columns And Get Column to Column Relation

---

First Thing You see In the output is The Guessed Datatypes of Each Column of the CSV Table where the left column contains the column name and it is mapped to its predicted dataType,Format: Column——->type. This type guessing is achieved with the use of messy tables,but messy tables had some bug issues in date time detectection so that had to be done using another method for this expression testing(hit and trail) is used.The use of messy tables can be seen in the image below.The Code explanation of each line is in the Image.(also in the code).In the Image below that getTypes functions is shown that is used for type detection.The first three lines of the code are for file path input.

```
#CASE:1 from Messy Tables #specify The Path where the file is
filename = raw_input()
datafile = file(filename)
fh = open(filename, 'r')
table_set = CSVTableSet(fh)

# A table set is a collection of tables:
row_set = table_set.tables[0]

# guess header names and the offset of the header:
offset, headers = headers_guess(row_set.sample)
row_set.register_processor(headers_processor(headers))

# add one to begin with content, not the header:
row_set.register_processor(offset_processor(offset + 1))

# guess column types:
types = type_guess(row_set.sample, strict=True)

# and tell the row set to apply these types to
# each row when traversing the i:
row_set.register_processor(types_processor(types))
```

Use of Messy Tables

```
#type detection for datetime formats,for tests Tests array defined above is used
def getType(value):
    for typ, test in tests:
        try:
            test(value)
            return typ
        except ValueError:
            continue
    # No match
    return str
```

Get Types function used for date detection

Now this csv data is inserted/processed in a pandas dataframe.This data frame creates a table for storing the csv data.For the getting various information relating to the data,a class data and some methods have been defined in it. The figure in the right shows Class Data,and its constructer _init_.Some of the basic information related to the tables like number number of rows(nrows),number of columns(ncolumns) and list_col(list of columns) is initialised in _init_.This method takes one argument that is dataframe(df).

```
class Data:

    list_col=None
    ser_dtype=None
    ncolumns=None
    nrows=None
    matrix=None

    def __init__ (self,df):
        Data.list_col=list(df.columns)
        Data.ser_dtype=df.dtypes
        Data.nrows,Data.ncolumns=df.shape
        Data.matrix=np.zeros((Data.ncolumns,Data.ncolumns))
```

Now we move on to the next method that is prop_data(),this method takes two arguments data frame(df) and types,now types is a array returned by messy tables,based on this array each column is assigned to various Type methods some of them are numeric_col,String Column,Date time Column,Generic Column(Its a column whose datatype is not handed)

```
#main funtion that calls all other type functions like numeric etc
def prop_data(self,df,types):
    for i in xrange(0,len(Data.list_col)):
        data_type=types[i]
        if(str(data_type) == "Integer"  or str(data_type) == "Decimal"):
            print "column name : ",i
            print "data type : ",str(data_type)
            self.numeric_col(df[Data.list_col[i]])
            print "\n"
        elif (str(data_type) == "String"):
            print "column name : ",i
            print "data type : ",str(data_type)
            self.string_col(df[Data.list_col[i]])
            print "\n"
```

The code for the numeric_col (datatype method) can be seen on the right it uses some inbuilt methods defined in pandas like min,max,mean,std,unique,these method are used to find minimum, maximum,mean,standard deviation,number of unique values in column.If the number of unique values in the table is less than or equal to twelve then the array of those values are displayed on screen.

```
#Part 2:Finding various properties related to the columns
    #Done Finding Mean,maximum,minimum,std using inbuilt dataframe functions
    def numeric_col(self,s):
        minimum=s.min()
        maximum=s.max()
        mean=s.mean()
        std=s.std()
        uniq=s.unique()
        uniq_val=len(uniq)
        print "possible values : ",uniq_val
        if(uniq_val<=10):
            print "possible values are : ",uniq
        print "mean : ",mean
        print "std : ",std
        print "maximum : ",maximum
        print "minimum : ",minimum
        return
```

Now Moving on too the last main function that is relation main this function finds the relation between two columns(every two columns from the table).The relationship can be one to one,many to one and many to many.This is checked by mapping each value from column one to column two and storing the relation in the dictionary in this case dic.This Relation is printed in form of a matrix containing integers 11/12 in each cell.if a column has one to one relation with a column b then 11 is used,else 12 is used for one to many relation.example of the output matrix is shown in the next page

```
#main function that Finds the relation between a pair of columns
def relation_main(self,df):
    m=Data.matrix
    for i in range(0,Data.ncolumns):
        for j in range(0,Data.ncolumns):
            if(i!=j):
                col_1=df[Data.list_col[i]]
                col_2=df[Data.list_col[j]]
                dic={}
                for a in range(0,Data.nrows):
                    if(col_1[a] not in dic):
                        dic[col_1[a]]=col_2[a]
                        if(a==Data.nrows-1):
                            m[i][j]=11

                    else:
                        if(dic[col_1[a]]!=col_2[a]):
                            m[i][j]=12
                            break
    return
```

```
[[  0.   0.  12.   0.   0.  12.   0.  12.  12.  12.   0.  12.]
 [  0.   0.  12.   0.   0.  12.   0.  12.  12.  12.   0.  12.]
 [ 12.  12.   0.  12.  12.  12.  11.  12.  12.  12.  12.  11.]
 [  0.   0.  12.   0.   0.  12.   0.  12.  12.  12.   0.  12.]
 [  0.   0.  12.   0.   0.  12.   0.  12.  12.  12.   0.  12.]
 [ 12.  12.  12.  12.  12.   0.   0.  12.  12.  12.  12.  12.]
 [ 12.  12.  12.  12.  12.  12.   0.  12.  12.  12.  12.  12.]
 [ 11.  11.  11.  11.  11.  11.  11.   0.  11.  11.  11.  11.]
 [ 11.  11.  11.  11.  11.  11.  11.  11.   0.  11.  11.  11.]
 [ 11.  11.  11.  11.  11.  11.  11.  11.  11.   0.  11.  11.]
 [  0.   0.  12.   0.   0.  12.   0.  12.  12.  12.   0.  12.]
 [ 12.  12.  11.  12.  12.  12.  11.  12.  12.  12.  12.   0.]]
```

For each i,j in the matrix if matrix[i][j] is 11 the relation i->j is one-one and if matrix[i][j] = 12 then relation i->j one to many and so if matrix[i][j] = 12 and matrix[j][i] = 12 then it has both relation ship i.e. many to many.

What see below now is the output of the program according to different column types.

```
column name :  7
data type :  <type 'datetime.datetime'>
possible values :  3
possible values are :  ['2014-03-31 17:44.31' '2016-03-31-17.44.31' '2014-03-31-17.44.31']
```

Date time type

```
column name :  6
data type :  Integer
possible values :  1
possible values are :  [1996]
mean :  1996.0
std :  0.0
maximum :  1996
minimum :  1996
```

Integer type

```
column name :  5
data type :  String
possible values :  2
possible values are :  ['Sev 4' nan]
Max length of the string :  5
```

String type

Now moving onto part:2 Normalisation and FD(Funtional Dependencies) finding:

This program Does Normalisation 2nf(2nd normal form) and 3nf(3rd normal form) and breaks the file into smaller tables.**Normalisation** is the process of taking data from a problem and reducing it to a set of relations while ensuring data integrity and eliminating data redundancy. Data integrity - all of the data in the database are consistent, and satisfy all integrity constraints,the data is expected to be in 1nf(1st normal form) from the start(which means that the program assumes the each cell in the table contains only one value).The output format is given below.

Output

When you run NormaliseCsv.py you will first see the table given in the right,Now the first function of this program is to find the number of missing values in each column of the csv table,the column name and its corresponding number of missing values is shown in the image example number of missing values in Severity column is one and number of missing values in Parent Ticket Column is 0 and so on.

```
      +[git:master]
[ python NormaliseCsv.py
Give the path to the file(give absolute/relative path to the file)
ibmproject.csv

Number of Missing Values in Each Column:--

Ticket Number          0
Assigned               0
Status                 0
Client                 0
Category               0
Severity               1
Date Created           0
Date Last Modified     0
Response Date          0
Date Closed            0
Time Spent (min)       0
Parent Ticket          0
dtype: int64
```

Now the rest of things you see is the final output of normalisation you can see the output of this part below.It includes finding primary key of the main table,columns left in the main Table after Normalisation,The extra Tables after Normalisation,now this is array of arrays and to access the first of these tables can be accessed form the code with indexing of zero,i.e. to get first table do array[0] in the code.The output shown below is just a part of the output.

```
primary_key ----------------------> ['Date Last Modified']
Main Table after Normalistion-------> ['Severity', 'Date Last Modified', 'Response Date', 'Date Closed']
Extra Tables after Normalisation----> [['Date Created', 'Ticket Number'], ['Category', 'Ticket Number'],
Assigned', 'Ticket Number']]
Primary Key In Main Table-----------> ['Date Last Modified']
```

This program outputs another thing but to see that you will have to remove comments from the last few lines of code,they have been commented so that output doesn't look too long,what this part of the code outputs is

```
['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Assigned']
['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Status']
['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Client']
['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Category']
['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Severity']
['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Date Created']
['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Time Spent (min)']
['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Parent Ticket']
['Ticket Number', 'Assigned', 'Status', 'Category', 'Date Closed', 'Time Spent (min)'] -----> ['Ticket Number']
```

relationship between many columns ——> to a single column,the output for this part looks something like what is shown in the above figure,it will be too long even for number of columns being equal to 12.This part is what is called finding the functional dependencies,a line of the program looks like ['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] -----> ['Assigned'],what this means is that set of these columns ['Ticket Number', 'Assigned', 'Status', 'Category', 'Response Date', 'Parent Ticket'] is one to one,or many to one related to 'Parent Ticket'.(What the relation is will be written in the start of the output.).

Code Explanation for part 2:

```
#-------All functions Used-----------#

from itertools import combinations
#Input is a List and Output is a list of all possible combinations,
#We have to Ignore the first and Last Combination,empty and all
#to get all combinations of columns may make the code a bit lengthy but is required
def GetAllCombinations(input):
    return sum([map(list, combinations(input, i)) for i in range(len(input) + 1)], [])

#----finds Number of Missing Values(NaN in each column)-----#
def num_missing(x):
  return sum(x.isnull())
```

Mainly two functions have been used in this code,these are shown here,the first of them is GetAllCombinations,It takes one argument,an array,and returns all combinations of the elements of the array,in a array of arrays format.

And the other function used is num_missing,it is responsible for finding the number of missing values in the table,and in each column of the table,this uses an inbuilt function called isnull() its calling and printing is shown below.its applied to the complete data frame in which the table is in this case(dataFrame)

```
print "Number of Missing Values in Each Column:--"
print
Column_missing_values = dataFrame.apply(num_missing, axis=0)
print Column_missing_values
```

The following code is a part of the code responsible for finding FD's (functional dependencies),it takes as input all combinations obtained from using getAllcombinations() function on List Columns array.Listcolumn contains the list of names of columns of the dataframe(csv table).The first and Last members of ListOfCombinationsOfColumns is ignored as the first one is [](empty set) and the last one is the complete list of columns,This last column can be taken into account but its not necessary .What the inner for loop the one with parameter y does is it makes array of arrays of the full columns in a variable col_2 example if first column is [213,123] and the second column is(contains) [324,234] then col_2 = [[213,123],[324,234]] a map function is applied to column 2 this takes the transpose of col_2 so now col_2 = [[213,324],[123,234]].Now each Element is converted in tuples ,with the following code so now col_2_tuples = [(213,324),(123,234)].

```
for i in xrange(1,len(ListOfCombinationsOfColumns)-1): #taking all 2^n-2 combinations of columns
    col_2 = []
    for y in xrange(0,len(ListOfCombinationsOfColumns[i])):
        if(y==0):
            col_2 = [dataFrame[ListOfCombinationsOfColumns[i][y]].tolist()]
        else:
            col_2.append(dataFrame[ListOfCombinationsOfColumns[i][y]].tolist())
    col_2 = map(list, zip(*col_2))                #Transpose of original Col_2
    col_2_tuples = [tuple(l) for l in col_2]      #Taking a list of columns and tranforming their rows
    #Tuple to Compare
```

Now moving on to the second part of the above FD's code:this is a continuation of the for loop above,now work of first part of this code is same as the one in part one i.e. relationfinding,the only difference is that this finds only one-one and many to one relation instead of both,as FD's only involve one-one and many to one relations,rest all relations are invalid for this case.Now the explanation part same as in part one a dictionary is created to map a tuple of columns(for each row as created above in column one) to corresponding cell in col_1. example let col_1 = [2131,3442] and col_2 from the above relation was [(213,324),(123,234)]:Now in this case (213,324)->2131 and (123,234)->3442,and also all many to one cases must be ignored.For that let's take a complex example col_2 = [(213,324),(123,234),(213,324)] and col_1 = [2131,3442,3241]. and now in this case as can be clearly seen (213,324) is mapped to two values(different) if this case arises we have a flag for that its a variable that we convert to true if such a case arises and this dependency is ignored in the final ans.This part is also helpful afterwards and for finding the primary key,For finding the primary key if every column in the table is functionally dependent on a col_2 in that case that is a potential primary key so we push it in an array called super_keys/candidate_keys.Many to one and One to One relations are also stored in their respective arrays(many_to_one_relation and one_to_one_relation).Now all relations of the current tuple is stored in StoringAllRelationsInADictionaryOfTuples array,to be used later.

```
#Tuple to Compare
for_primary_key_and_file_breaking[i] = []
for j in xrange(0,len(ListColumns)):
    if(ListColumns[j] not in ListOfCombinationsOfColumns[i]): not requires as XAB doesn't mean XAB->X
    dic = {}
    dic2 = {}
    col_1 = dataFrame[ListColumns[j]].tolist()
    flag = False
    many_to_one = False
    for x in xrange(0,rw):
        if(col_2_tuples[x] not in dic):
            dic[col_2_tuples[x]] = col_1[x]
        elif(dic[col_2_tuples[x]] != col_1[x]):
            flag = True
            break
        if(col_1[x] not in dic2):
            dic2[col_1[x]] = col_2_tuples[x]
        elif(dic2[col_1[x]] != col_2_tuples[x]):
            many_to_one = True
    if(not flag):
        for_primary_key_and_file_breaking[i].append(ListColumns[j])
        if(len(for_primary_key_and_file_breaking[i]) == len(ListColumns)):
            super_keys.append(ListOfCombinationsOfColumns[i])
        if(many_to_one):
            many_to_one_relation.append([ListOfCombinationsOfColumns[i],[ListColumns[j]]])
        else:
            one_to_one_relation.append([ListOfCombinationsOfColumns[i],[ListColumns[j]]])
StoringAllRelationsInADictionaryOfTuples[tuple(ListOfCombinationsOfColumns[i])] = for_primary_key_and_file_breaking[i]
```

Now to find the primary key the super keys array is used,just one thing is left to check i.e. if an element is the potential super_keys is found in which no column has null elements i.e. number of missing values in all columns in that element of super keys array should be 0,that will be the primary key.In some cases no primary key is found in that case program outputs no primary key.Else the primary key to be used for the table is printed.

```
for x in xrange(0,len(super_keys)):
    CanBe = True
    for j in xrange(0,len(super_keys[x])):
        if(Column_missing_values[super_keys[x][j]] != 0):
            CanBe = False
            break
    if(CanBe):
        primary_key = super_keys[x]
        break
    else:
        CanBe = True

if(primary_key == None):
    print "No Primary Key Is Found"
else:
    print "primary_key ---------------------->",primary_key
```

```
#Now for the Last and Final Part Normalisation!
AllCombinationsOfPrimaryKeyFor2NFNormalisation = GetAllCombinations(primary_key)
ColumnsToRemove = []
ExtraTables = []
for i in xrange(0,len(ListColumns)):
    for x in xrange(1,len(AllCombinationsOfPrimaryKeyFor2NFNormalisation)-1):
        if(ListColumns[i] not in AllCombinationsOfPrimaryKeyFor2NFNormalisation[x]):
            if(ListColumns[i] in StoringAllRelationsInADictionaryOfTuples[tuple(AllCombinationsOfPrimaryKeyFor2NFNormalisation[x])
                ColumnsToRemove.append(ListColumns[i])
                A = list(AllCombinationsOfPrimaryKeyFor2NFNormalisation[x])
                A.append(ListColumns[i])
                ExtraTables.append(A)
                break
```

Now the Code above performs 2nf normalisation.A relation is in 2NF if, and only if, it is in 1NF and every non-key attribute is fully functionally dependent on the whole key. So the above code checks if the columns are directly dependent on a part/subset of columns of the primary key.Example if the primary key is [‘A’,’B’,’C’] where A,B,C are names of the columns in that case if a column ‘G’ is dependent on [‘A’,’B’] also in that case there is a discrepency. We remove it by removing column ‘G’ from the table and making another table [‘A’,’B’,G’] from it.

```
for i in xrange(0,len(ListColumns)):
    for x in xrange(1,len(AllCombinationsOfNonPrimaryKeyFor3NFNormalisation)-1):
        if(ListColumns[i] not in AllCombinationsOfNonPrimaryKeyFor3NFNormalisation[x]):
            if(AllCombinationsOfNonPrimaryKeyFor3NFNormalisation[x] not in super_keys):
                if(ListColumns[i] in StoringAllRelationsInADictionaryOfTuples[tuple(AllCombinationsOfNonPrimaryKeyFor3NFNormalisation[x])]):
                    ColumnsToRemoveAfter3NF.append(ListColumns[i])
                    A = list(AllCombinationsOfNonPrimaryKeyFor3NFNormalisation[x])
                    A.append(ListColumns[i])
                    ExtraTables.append(A)
                    break
```

Code for 3nf is similar code to 2nf,it is given in the figure above,A relation is in 3NF if, and only if, it is in 2NF and there are no transitive functional dependencies.It is even stricter normal form and remove most of the discrepancies the indirect meaning of this is we have to remove all relations with FD: non-key attribute -> non-key attribute.For this also we remove the relation from the table and make another table from the same,mostly same as above but with different relation.

Challenges:

- Date format parsing,their are still many date formats that are not handled
- File breaking this required a lot of time and analysis of functions but time is still a challenge that is how to optimise the algorithm so that It can work on bigger files.
- A subpart of point to i.e. finding functional dependencies if this is optimised overall algorithm will become fast because this is the slowest.

Lessons Learnt:

- Database Normalisation
- Difference between RDBMS and document based Databases.
- Type checking
- Working in various data science Libraries like pandas and bumpy
- Exposure to open source community like using libraries like messy tables in the project.

Things this project doesn’t handle

- Normalisation has many different forms this project only uses two.
- Date formats can be of various types,but this project accepts only some of them marks others as strings so this is still to be handled
- This project doesn’t cover many column to many column functional dependencies ,this when handled will further improve the efficiency of the code,this was not handled because of huge time complexity related to it.
- None values were not considered while normalisation.
- Testing this model in mysql and mongoldb etc still needs to be done.