A Data Assessment Method for A Specific Table

JUHENG YANG* and GAIKE WANG*, New York University, USA

A new semi-automated method which can partly accurately classify a specific driver application information into following parts: NULL value, misspellings, valid values, outliers and some invalid values. We firstly use some big data means like Spark, OpenRefine to analyze data, and then apply some common but trivial statistical means and machine learning tools to our model to handle those data. Although there is some limitations on our method, Undoubtedly, it is successful. It can accurately find some dirty data and then give people a forecast for the true value.

1 INTRODUCTION

As we all know, almost all the driver application status forms have three following common information: App No(Application Number), some dates(eg. App Date, Last Updated) and some specific test status(eg. drug test, driver exam). Based on these data, we can get some useful results of analyses or data-driven models. However, there are usually some quality issues on the data, which will negatively impact the quality of people's analyses. We list some errors at Table 1. Therefore, for the above three kinds of common data, we separately present an approach to find all the possible errors and notify them.

The rest of this paper is organized as follows, Section 2 discusses related work; Section 3 states the formulation of problem; Section 4 describes the proposed approach in detail; Section 5 shows the experiment results; Section 6 concludes our work.

2 RELATED WORK

2.1 Outlier Detection

Outlier was firstly considered by some clustering algorithms like CLARANS [6], DBSCAN [2], BIRCH [8] and CURE [3], and was only used to ensure that the clustering process is not interfaced. Later on, Knorr et al. defined distance-based outlier in [4], which is a point with at most k points within distance d of the point. Knorr and Ng used a simple nested-loop algorithm and a cell-based approach which can reduce the time complexity to compute outliers. Ramaswamy et al. proposed a novel formulation for distanced-based outliers in [7]. They rank rank each point on the basis of its distance to its k^{th} nearest neighbor and declare the top points in this ranking to be outliers.

Recently, B. Angelin et al. proposed a K-median clustering algorithm to detect outliers and compared it with the weight-based K-mean grouping in [1].

2.2 Misspellings Detection

As we all know, there are two kinds of misspellings. One is non-word misspellings, the other one is real-word misspellings. Right here, we only consider non-word misspellings. Kukich et al. surveys some common techniques like efficient pattern-matching and n-gram analysis for detecting misspellings in [5].

3 PROBLEM FORMULATION

Our object is to give a classification for the values in chart. The category includes NULL value, valid value(including misspellings), invalid values(all kinds of outliers). As mentioned above, our data

*Both authors contributed equally to this research.

2020. XXXX-XXXX/2020/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

	Error	Example
valid	misspelling	complete ->coomplete
	date format error	11/27/2020 ->2020/27/11
invalid	digit or word outlier	11/27/2020 ->20202/27/11
		(0,500) ->1000
		{a,b,c,d} ->e
	semantic outlier	(date,number) ->(100,11/27/2020)
	co-occurrence error	some logic errors

Table 1. Some examples of errors

	category		
App No	NULL value	invalid value(outlier(semantic, digit))	valid value
Date	NULL value	format error	valid value
	co-occurrence error	invalid value(outlier(semantic, digit))	
Status	NULL value	misspelling	valid value
	co-occurrence error	invalid value(outlier(semantic, digit))	

Table 2. data classification - data error category

Class	Column name					
NNUMBER	App No					
DATE	App Date	FRU inter	Last Update			
STATUS	Type	Status	Drug Test	WAV course	Def Driving	MedForm
OTHER	Other Req					

Table 3. Divide input data into four classes

can be classified into App No, dates and some status(string). Now, we will give a corresponding relation between data classification and data error category, which is listed in Table 2. Next, what we should do is to build a specific approach for each data class to accurately tell from each kind of data error and give a forecast for some valid error, such as misspelling error and format error.

4 METHOD, ARCHITECTURE AND DESIGN

4.1 Input and Output

At the very beginning, we use panda as tools to handle input and out. As we know, pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. It can help us read .csv profile as a dataframe. We recognize that in the table of TCL New Driver Application Status, there are 12 columns. We divided them into four classes: Number, Date, Status and Other. The details are in Table 3.

4.2 Date Pre-handling in the First Iteration

We create some lists, dictionaries and sets to save the key features of each columns. For Number column, we use a dictionary to save all appeared number in the above columns. For Date columns, we use a list to save all appeared values because it is easy to convert it into NumPy and then get some important parameters in the step 3. For STATUS columns, we use dictionary to save all

appeared values as well. The Fig 1 shows the details of all variables and basic logic of first iteration.

```
Procedure pre-handle of the table(df)
Begin
  appDate = df['App Date']
  fRUInterviewScheduled = df['FRU Interview Scheduled']
  lastUpdated = df['Last Updated']
  list_appDate = []
  list_fRUInterviewScheduled = []
  list_lastUpdated = []
  status = df['Status']
  dic_status = {}
  drugTest = df['Drug Test']
  dic_drugTest = {}
  wAVCourse = df['WAV Course']
  dic_wAVCourse = {}
  defensiveDriving = df['Defensive Driving']
  dic_defensiveDriving = {}
  driverExam = df['Driver Exam']
  dic_driverExam = {}
  medicalClearanceForm = df['Medical Clearance Form']
  dic_medicalClearanceForm = {}
  dtype = df['Type']
  dic_dtype = {}
  appNo = df['App No']
  dic_appNo = {}
  set_ocappNo = set()
  for each i in rows:
    DateHandle(i,appDate,list_appDate)
    DateHandle(i,fRUInterviewScheduled,list_fRUInterviewScheduled)
    DateHandle2(i,lastUpdated,list_lastUpdated)
    StatusHandle(i,status,dic_status)
    StatusHandle(i,drugTest,dic_drugTest)
    StatusHandle(i,wAVCourse,dic_wAVCourse)
    StatusHandle(i,defensiveDriving,dic_defensiveDriving)
    StatusHandle(i,driverExam,dic_driverExam)
    StatusHandle(i,medicalClearanceForm,dic_medicalClearanceForm)
    StatusHandle(i,dtype,dic_dtype)
    NumberHandle(i,appNo,dic_appNo)
End
```

Fig. 1. First Iteration to Pre-handle Data

To handle the NUMBER column, we check the value if it is a nonvalue and then check if it is a number, finally we save the number frequency of the number length in the dictionary. The procedure **PreNUMBERHandle** is in Fig 2 to show details. Input parameters cnt means the rows in df, num means the number column in df, d is the dictionary we will use and nonedic provides all

formats of NULL value.

```
Procedure PreNUMBERHandle (cnt, num, d)
Begin
  Value = num[cnt]
  if Value in noneDic:
    Value = Value + '$' + NONEVALUE
    return
  if value is not a number
    Value = Value + '$' + INVALID
    return
  d[Value] = len(Value)
  Value = Value + '$' + VALID
End
```

Fig. 2. Pre-handle of NUMBER column

To pre-handle the status data, we first check if it is a nonvalue and then save all appeared values into the dictionary. The procedure **PreSTATUSHandle** is in Fig 3 to show details.

```
Procedure PreSTATUSHandle (cnt, status, d)
Begin
  value = status[cnt]
  if status[cnt] in noneDic:
    status[cnt] += '$' + NONEVALUE
    return
  d[status[cnt]] += 1
  value = value + '$' + VALID
End
```

Fig. 3. Pre-handel of STATUS column

To pre-handle the DATE data, we first check if it is a nonvalue and then use Regular expression to detect the format of the date, we label the format error if the date is in different format of standard date format. Then we us some library function to determine if the date is a valid date. For example 13th month is not valid. The procedure **PreDATEHandle** is in Fig 4 to show details.

4.3 Model for Learning the Similarity of Two Strings

How to find a misspelling and the word which should not be misspelled? There must a relationship between them. They should be "similar". Therefore, the problem is how to define "similar". For word a and word b, we use three arguments to evaluate how similar a and b is, which includes:

- length difference L_{ab}:

$$L_{ab} = \frac{\Delta L}{\max(L_a, L_b)} \tag{1}$$

- Jaccard distance between a and b J_{ab}:

$$J_{ab} = 1 - \frac{a \cap b}{a \cup b} \tag{2}$$

```
Procedure PreDATEHandle (cnt, date, l)
Begin
  value = date[cnt]
  if value in noneDic:
    return NONEVALUE
  foramt = re.findall(r'[0-9]+',s)
  if not isFormatValid(format):
    return FORMATERROR  + '@' + standard_foramt_date
  if not isValidDate(value):
    return INVALID
  l.append(standard_foramt_date)
  return VALID + '$' + standard_foramt_date
End
```

Fig. 4. Pre-handel of Date column

- The edit distance of a and b D_{ab}:
  There are three operations, which are **INSERT**, **DELETE**, and **REPLACE**

For a and b, we generate a three-dimension vector (L_{ab}, J_{ab}, D_{ab}), and label $y \in \{0, 1\}$ (1 means a is similar to b and 0 means a is not similar to b). Obviously, we find that there are some common parts among three dimensions. Therefore, we use PCA to get three linearly independent dimensions for our training. Firstly, we build our train set Xtr (after PCA on vector generated from a word pair) and ytr which contains 0 and 1. Then we use them as input to train a logistic classifier. What can this classfier do? It can judge whether two words is similar. Finally, we build a test set and predict labels for the set. Fortunately, the accurate rate is 100%. To be honest, the training result is very sensitive to the training set. For example, for a long word, if we change three or more words in it, we can also say that these two words is similar. However, for a short word, whose length maybe less than five, if we change three words in it, generally, we can't say that they are similar. Therefore, more general train data is better. This is the limitation of our model.

4.4 Parameters Calculation using pre-handled dictionary

In this step, we will find all misspelling error dictionary and outlier error set of all classes including DATE, STATUS and NUMBER. The misspelling error dictionary saves the $originalvalue$, $correctedvalue$ as key, $value$ and the outlier set save all outlier value.

At the very beginning, we create standard dictionaries for all STATUS columns. Then we create many empty dictionaries and sets to save all error values and use Handle Procedure to attain those values like in the second step. The procedure **MidHandle** is in Fig 5 to show details.

To handle NUMBER middle data, we first create a new dictionary to save all number length frequency, then we choose the most frequent length as standard length. Finally, we compare all length with standard length, if the difference is more than two, we define it as outlier error. The procedure **MidNUMBERHandle** is in Fig 6 to show details.

To mid-handle the status data, we first check all value in the dictionary we get in the first step if it is in the correct set, if not, we calculate the similarity between this value and standard value in correct set. If this value match any standard value in correct set, we label it as mis-spelling, otherwise we label it as outlier. The procedure **MidSTATUSHandle** is in Fig 7 to show details.

```
Procedure MidDATEHandle ()
Begin
  set_correctValue_dtype = {'HDR','PDR','VDR'}
  set_correctValue_status = {'Incomplete','Approved - License Issued','Under Review','Pending Fitness
Interview'}
  set_correctValue_drugTest = {'Complete','Needed','Not Applicable'}
  set_correctValue_wAVCourse = {'Complete','Needed','Not Applicable'}
  set_correctValue_defensiveDriving = {'Complete','Needed','Not Applicable'}
  set_correctValue_driverExam = {'Complete','Needed','Not Applicable'}
  set_correctValue_medicalClearanceForm = {'Complete','Needed','Not Applicable'}
  dic_mis_dtype, set_outlier_dtype = StatusHandleStep3(dic_dtype,set_correctValue_dtype)
  dic_mis_drugTest, set_outlier_drugTest =
StatusHandleStep3(dic_drugTest,set_correctValue_drugTest)
  dic_mis_wAVCourse, set_outlier_wAVCourse =
StatusHandleStep3(dic_wAVCourse,set_correctValue_wAVCourse)
  dic_mis_defensiveDriving, set_outlier_defensiveDriving =
StatusHandleStep3(dic_defensiveDriving,set_correctValue_defensiveDriving)
  dic_mis_driverExam, set_outlier_driverExam =
StatusHandleStep3(dic_driverExam,set_correctValue_driverExam)
  dic_mis_status, set_outlier_status = StatusHandleStep3(dic_status,set_correctValue_status)
  dic_mis_medicalClearanceForm, set_outlier_medicalClearanceForm =
StatusHandleStep3(dic_medicalClearanceForm,set_correctValue_medicalClearanceForm)
  set_outlier_appDate,barl_appDate,barh_appDate = DateHandleStep3(list_appDate)
  set_outlier_fRUInterviewScheduled,barl_fRUInterviewScheduled,barh_fRUInterviewScheduled =
DateHandleStep3(list_fRUInterviewScheduled)
  set_outlier_lastUpdated,barl_lastUpdated,barh_lastUpdated = DateHandleStep3(list_lastUpdated)
  set_outlier_appNo = set_outlier_appNo
  End
```

Fig. 5. Mid-handle of all middle data

```
Procedure MidNUMBERHandle (dic_appNo)
Begin
  dic_length_freq_appNo = {}
  set_outlier_appNo = set()
  for value in dic_appNo.values():
    dic_length_freq_appNo[value] += 1
  most_length_appNo = 0
  tmp = -float('inf')
  for k,v in dic_length_freq_appNo.items():
    if v > tmp:
      most_length_appNo = k
      tmp = v
  for k in dic_appNo.keys():
    if abs(len(k)-most_length_appNo) >= 2:
      set_outlier_appNo.add(k)
End
```

Fig. 6. Mid-handle of NUMBER data

To mid-handle the DATE data, we first convert the list got from step 1 to numpy, and then get the mean value and standard deviation. We use mean value and stand deviation to calculate two bars, low bar and high bar. Then we iterate all value in the list, if the value is beyond neither low bar nor high bar, we label it as outlier. The procedure **MidDATEHandle** is in Fig 8 to show details.

```
Procedure MidSTATUSHandle (d, s_correct)
Begin
  set_outlier = set()
  dic_mis = {}
  for k in d.keys():
    if k in s_correct:
      continue
    for m in s_correct:
      if similarity(k, m) == 1:
        dic_mis[k] = m
        break
    else:
      set_outlier.add(k)
  return dic_mis, set_outlier
End
```

Fig. 7. Mid-handle of STATUS data

```
Procedure MidDATEHandle (d, s_correct)
Begin
  set_date_outlier = set()
  n = np.array(l)
  ave = np.mean(n)
  std = np.std(n)
  barl = ave-3*std
  barh = ave+3*std
  for year in l:
    if barl <= year <= barh:
      continue
    set_date_outlier.add(year)
  return set_date_outlier,barl,barh
End
```

Fig. 8. Mid-handle of NUMBER data

4.5 Cross-validation and Co-occurrence Error Verification

In this step, we iterate the whole table again find read it line by line. In every row, we use cross-validation to find if this value is valid in other columns. We label those errors as semantic error. At the same time, we find all co-occurrence error when there are logical errors between each columns. What's more, we use misspelling dictionaries and outliers sets gained from step 4 to label the error values.

By accessing every value line by line, column by column, we add outlier label and misspelling label. Then we use cross-validation to get if this value is valid in other line. If yes, we label it as semantic error and gave the qualified column. In the end we verify if there is co-occurrence error in this line. The following procedure **FinalDATAHandle** in Fig 9 shows the details.

To verify if the invalid col is valid in other column, which also means semantic error, we use three function including IsOtherNumber, isOtherStatus, isOtherDate to confirm if this value is valid in other columns. If any of columns matches, we label it as semantic error and break the function. The following procedure **Compare** in Fig 10 shows the details.

```
Procedure FinalDATAHandle (dic_misspelling,set_outliers)
Begin
  for index, row in df.iterrows():
    line = row
    for col in cols:
      if col in dic_misspelling:
        label col as MISSPELLING
      if col in set_outliers:
        label col as OUTLIERS
      if col is not valid and Compare(col):
        label col as SEMANTICeRROR
    if Co_OccurrenceError(line):
      label line as COoCCURRENCEeRROR
End
```

Fig. 9. Final-handle of all data

```
Procedure Compare (value)
Begin
  if isOtherNumber(value):
    return MATCH,AppNo
  if isOtherStatus(value):
    return MATCH,Status
  if isOtherDate(value):
    return MATCH,Date
  return False
End
```

Fig. 10. Cross-validation

To verify if the value is valid in Number column, we use NumberCheck function in step2 to confirm it is a number and use standard number length in step3 to constrain the value. If the value is a number and match the length, then we recognize it as a semantic error and it's qualified column is Number column. The following procedure **IsOtherNumber** in Fig 11 shows the details.

```
Procedure IsotherNumber (value)
Begin
  if (not NumberCheck1(value)) or abs(len(value)-length) >= 2:
    return NOTFIT
  return FIT
End
```

Fig. 11. Cross-validation of NUMBER

To verify if the value is valid in STATUS column, we check if the value in the correct set of STATUS column, and then check the similarity between this value and other values in correct set. The following procedure **IsOtherStatus** in Fig 12 shows the details.

To verify if the value is valid in DATE column, we check if the value is a date using a library function and then check if the value between low bar and high bar gained in the second layer. The following procedure **IsOtherDate** in Fig 13 shows the details.

```
Procedure IsOtherStatus (value)
Begin
  if s in set_correctValue:
    return FIT
  for m in set_correctValue:
    if similarity(s, m) == 1:
      return FIT + '!' + MISSPELLING_OTHER + '@' + m
  return NOTFIT
End
```

Fig. 12. Cross-validation of STATUS

```
Procedure IsOtherDate (value)
Begin
  if not ValidDate(value):
    return not fit
  if not barl <= date <= barh:
    return NOTFIT
  return FIT
End
```

Fig. 13. Cross-validation of DATE

In the end of the step5, we check if there is a co-occurrence error. we notify that some column are in logical relationship, for example, the finish time must be after the application time. Besides, the status is depended on other STATUS column. Therefore, we check co-occurrence error between the DATE columns and the STATUS columns. We first split the whole line to get values in every column and then check the relationship between each values. The following procedure **Co-OccurrenceError** in Fig 14 shows the details.

```
Procedure Co_OccurrenceError(line)
Begin
  if status[1].startswith(VALID) and drugtest[1].startswith(VALID) and wav[1].startswith(VALID) and
defn[1].startswith(VALID) and driverexam[1].startswith(VALID) and med[1].startswith(VALID):
    flag = 1 if (drugtest[0] == Complete or drugtest1[-1] == Complete) and (wav[0] == Complete or
wav1[-1] == Complete) and (defn[0] == Complete or defn1[-1] == Complete) and (driverexam[0] ==
Complete or driverexam1[-1] == Complete) and (med[0] == Complete or med1[-1] == Complete) and
other == NOTAPPLICABLE else -1
    flag *= -1 if (status[0] == Incomplete or status1[-1] == Incomplete) else  1
    if flag== -1:
      label line[3] as COOCCURRENCEERROR

  if status[1].startswith(VALID) and fru[1].startswith(VALID):
    flag = 1 if status[0] == UnderReview or status1[-1] == UnderReview else -1
    flag *= 1 if fru[0] == NOT_APPLICABLE or fru1[-1] == NOT_APPLICABLE else -1
    if flag== -1:
      label line[3] as COOCCURRENCEERROR
End
```

Fig. 14. Co-Occurrence Error Detection

5 RESULTS

Our proposed method is used to handle new driver application status tables. The dataset we choose is TLC New Driver Application Status table. In our experiments, the results indicate the correctness of our approach in find some common data errors.

5.1 Analyzing and Generating Test Data

Firstly, we use OpenRefine and Spark to analyze the table. After analysis, according to the similarity of columns, we separate them into four categories, as is shown in Table 3, and we find all the data errors of each column, as is shown in Table 2. As is shown in Table 1, there are some examples. Secondly, for testing our method's stability, we manually generate 68 common value errors in 37 rows. Examples are shown in Fig 15.

3204	5966185$INVALID	HDR$VALID	Needed$INVALID(SEMANTIC OUTLIER)#Drug Test	Incomplete$VALID	Not Applicable$VALID
3205	5966185$INVALID	HDR$VALID	11/06/2020$VALID	Incmoplete$VALID(MISSPELLING)@Incomplete	Not Applicable$VALID
3206	5966185$INVALID	HDR$VALID	11/06/2020$VALID	Incompleteee$VALID(MISSPELLING)@Incomplete	Not Applicable$VALID
3207	5966185$INVALID	HDR$VALID	11/06/2020$VALID	woeodsji$INVALID	Not Applicable$VALID
3208	5966185$INVALID	HDR$VALID	11/06/2020$VALID	Complete$VALID(MISSPELLING)@Incomplete	Not Applicable$VALID
3209	5966185$INVALID	HDR$VALID	11/06/2020$VALID	nan$NONEVALUE	Not Applicable$VALID

Fig. 15. Generated Value Error

5.2 Algorithm Implemented

Our method's completion includes two main techniques. One is Three-Layer-Analyzing Algorithm, the other is String Vector Extraction and Similarity Calculation.

- **Three-Layer-Analyzing Algorithm:** This algorithm was described in Section 4. In order to handle table date in an efficient way, we use this algorithm to analyze TLC New Driver Application Table. In the first layer we iterate the whole table to find Nonvalue errors and storage some important features of all columns like frequency and length. In the second layer we processed those key features and get some dictionary to separate misspelling errors and outlier errors from original data set. In The third layer, we compare every column with other columns to find semantic outliers. Then we use Regular expression to find if there is a co-occurrence error in each line.
- **String Vector Extraction and Similarity Evaluation:** For finding whether two strings are similar, we extract vector from each word pair according to there metrics described in section 4.3. Then we use a logistic classifier to judge whether two words in a word pair are similar.

5.3 Performance Results

There are 3229 rows with 12 columns in the TLC New Driver Application Table. In all the rows, there are 68 errors in total in 37 rows including misspelling errors, Nonvalue errors, outliers and

Error type	Number
Misspelling	28
Nonvalue	12
Semantic Outliers	8
Co-occurrence error	5
Invalid	15

Table 4. Detected Errors

co-occurrence errors. After running the whole program including the learning part and TLA part, we use less than 8s totally. We have detected all 68 errors in 5 types. The details are showed in Table.4.

The results showed our algorithm is perfect to handle those data in TCL New Driver Application Table with only two iterations of whole tables in 8s.

5.4 Method Drawback

For a specific table, there usually are some parts which need to be manually handled. Therefore, our method's defects nearly come from these. Firstly, our model for judging the similarity between two strings is defected. We need to manually label all the string pairs, and because of this, the training set is limited. Secondly, we can't handle 'Other Req' column's values because they are some arbitrary notes, which means they may have totally different syntactic structures.

REFERENCES

[1] B. Angelin and A. Geetha. 2020. Outlier Detection using Clustering Techniques – K-means and K-median. 373–378. https://doi.org/10.1109/ICICCS48265.2020.9120990

[2] M. Ester, H.-P. Kriegel, and X. Xu. 1995. A database interface for clustering in large spatial databases. In *Proceedings of the 1st international conference on Knowledge Discovery and Data mining (KDD'95)*. AAAI Press, 94–99.

[3] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. 1998. CURE: an efficient clustering algorithm for large databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Laura Haas, Pamela Drew, Ashutosh Tiwary, and Michael Franklin (Eds.). ACM Press, 73–84. http://doi.acm.org/10.1145/276304.276312

[4] Edwin M. Knorr and Raymond T. Ng. 1998. Algorithms for Mining Distance-Based Outliers in Large Datasets. 392–403. http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=9A0ED8BD4E41A9DC15F6E02185FD3B8D?doi=10.1.1.55.8026&rep=rep1&type=pdf

[5] Karen Kukich. 1992. Techniques for Automatically Correcting Words in Text. *ACM Comput. Surv.* 24, 4 (Dec. 1992), 377–439. https://doi.org/10.1145/146370.146380

[6] Raymond T. Ng and Jiawei Han. 1994. Efficient and Effective Clustering Methods for Spatial Data Mining. 144–155.

[7] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. 2000. Efficient algorithms for mining outliers from large data sets. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (Dallas, Texas, United States). ACM, New York, NY, USA, 427–438. https://doi.org/10.1145/342009.335437

[8] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: an efficient data clustering method for very large databases. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (Montreal, Quebec, Canada). ACM, New York, NY, USA, 103–114. https://doi.org/10.1145/233269.233324