Flexible and Extensible Generation and Corruption of Personal Data *

Peter Christen and Dinusha Vatsalan
Research School of Computer Science, The Australian National University,
Canberra ACT 0200, Australia
{peter.christen, dinusha.vatsalan}@anu.edu.au

ABSTRACT

With much of today's data being generated by people or referring to people, researchers increasingly require data that contain personal identifying information to evaluate their new algorithms. In areas such as record matching and deduplication, fraud detection, cloud computing, and health informatics, issues such as data entry errors, typographical mistakes, noise, or recording variations, can all significantly affect the outcomes of data integration, processing, and mining projects. However, privacy concerns make it challenging to obtain real data that contain personal details. An alternative to using sensitive real data is to create synthetic data which follow similar characteristics. The advantages of synthetic data are that (1) they can be generated with well defined characteristics; (2) it is known which records represent an individual created entity (this is often unknown in real data); and (3) the generated data and the generator program itself can be published. We present a sophisticated data generation and corruption tool that allows the creation of various types of data, ranging from names and addresses, dates, social security and credit card numbers, to numerical values such as salary or blood pressure. Our tool can model dependencies between attributes, and it allows the corruption of values in various ways. We describe the overall architecture and main components of our tool, and illustrate how a user can easily extend this tool with novel functionalities.

Categories and Subject Descriptors

H.2.6 [**Database management**]: Database Applications—*Data mining*; H.2.4 [**Database management**]: Systems—*Textual data-bases*

General Terms

Experimentation

*This work was partially funded by the Australian Research Council (ARC) under Linkage Project LP100200079; and by Fujitsu Laboratories (Japan).

CIKM '13 San Francisco, California USA

http://dx.doi.org/10.1145/2505515.2507815.

Keywords

Synthetic data; data generator; data matching; duplicates

1. INTRODUCTION

Much of the data generated today either refer to people or they are generated by people. Examples include shopping and travel transactions; electronic health and financial records; taxation, social security, and census records; emails, tweets, blog posts, and so on. For researchers who work on the development of novel algorithms that process, integrate, or analyze data that contain personal identifying information, it is often difficult to obtain real data because of privacy and confidentiality concerns. These concerns prohibit the publication of data collections that contain personal information. Public data collections have shown to be highly valuable resources for research in areas such as information retrieval and machine learning [2, 8]. Without real data, it is however difficult to properly evaluate the performance and accuracy of novel algorithms. Only a few 'real' data sets that contain personal information are available for research [1, 4]. The intrinsic details of real data, such as data entry errors, phonetic variations, typographical mistakes, nicknames, measurement variations, random noise, dependencies between values, and values changing over time, can however crucially influence the performance of algorithms that work on data that contain personal information [1].

The alternative to using real sensitive data is to create synthetic data based on real data. Such synthetic data should exhibit similar characteristics as real data with regard to distributions of values, errors, noise, variations, and dependencies between attribute values. Generating synthetic data is a non-trivial process, and as a result many researchers implement their own simple ad-hoc methods for data generation and/or corruption (the process of adding noise and errors to data). While several data generators that can create personal data have been developed in the past (summarized in Section 2), all of them are limited in their functionality to generate data with certain characteristics only, their inability to be extended to novel types of data, and their lack of capability to generate data for Unicode character sets other than ASCII. Our tool is a first that overcomes these limitations. It is freely available at: `http://dmm.anu.edu.au/geco`

2. RELATED WORK

Several tools specifically aimed at generating data that contain personal information have been developed in the past. A first such generator was presented by Hernandez and

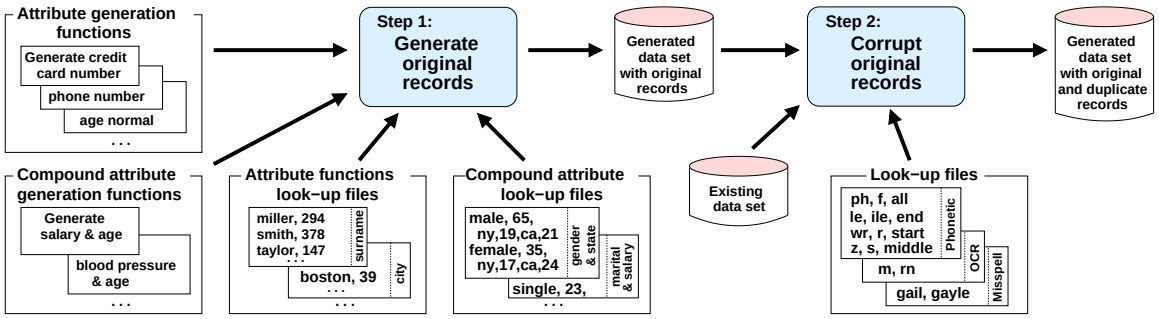

Figure 1: Overview of our data generation and corruption process, shown with simplified look-up files.

Stolfo [6]. It allowed the generation of records using lists of names, cities, states and zipcodes (without frequency information) and their corruption using edit-based modifications and swapping of categorical values. In 2003, Bertolazzi et al. [3] described an improved generator which allowed values to be randomly set to missing, and which provided a larger number of corruption functions. Arehart and Miller [1] described how they created a ground truth data set of 70,000 culturally diverse names drawn from different sources and manually incorporated diverse name variations.

The *Febrl* record linkage system provides a data generator that allows inclusion of frequency information, as well as look-up files for nicknames and misspellings [4]. A recent extension of this generator enables the modeling of some specific attribute dependencies, and allows the generation of groups of records that correspond to a family [5]. These dependencies and groups are however hard-coded and not extensible. Talburt et al. [7] recently presented the *SOG* tool for creating temporal records that represent people's occupancy. *SOG* can model the address history of individuals and couples by generating sequences of records with address changes based on collected real data sources.

Even though synthetic data have various advantages, the main challenge faced when developing a data generator is that the created data must represent the intrinsic characteristics of real data as much as possible. Our approach, described next, aims to tackle this challenge.

3. OVERALL ARCHITECTURE

Figure 1 provides a conceptual overview of our approach to data generation and corruption. In the first step, presented in the following section, we generate *original* records based on look-up files and attribute generation functions. In the second step, the original records are corrupted to form new records, called *duplicate* records. Different types of corruptions can be applied, as will be described in Section 5. More than one duplicate record can be generated for one original record. Alternatively to corrupting the original records, an existing data set, possibly containing real data, can be corrupted by only invoking the corruption step.

Each generated original record is given a unique identifier, and each duplicate record is also given a unique identifier that designates the original record it is based on, as can be seen in Figure 5. The original and duplicate records are written into a comma separated values (CSV) text file which allows easy portability and further processing. The number of records that can be generated is unlimited (subject to available memory size only).

```
def gen_ssn():  # Return a random social security number
  num1 = str(random.randint(1,999)).zfill(3)
  num2 = str(random.randint(1,99)).zfill(2)
  num3 = str(random.randint(1,9999)).zfill(4)
  return num1+'-'num2+'-'+num3
```

Figure 2: Python code to generate a random social security number (without checking its validity).

The key novelties of our data generation and corruption tool are that it (1) allows flexible definition of dependencies between up-to three attributes; (2) is easily extensible with new functionalities (via Python programs and look-up files); and (3) is Unicode compatible, therefore data in any character set can be generated and/or corrupted. We show Python code snippets to illustrate how our tool can be configured.

4. DATA GENERATION

A user configures the data generation process by defining the attribute types to be generated, their names, and type specific parameter settings. The values in an attribute are either generated independently from values in other attributes, or sets of attributes can be generated in a compound way with values depending on each other.

As shown in Figure 3, the main parameters to be set in the generation process are the name of the output file to be written, if a header line (with attribute names) is written into the output file, the number of *original* records that are to be generated, the details of the attributes to be generated and their names, as well as the Unicode encoding to be used.

4.1 Generating individual attributes

Two types of individual attributes can be generated. The first is based on frequency look-up files that contain categorical values and their relative frequencies, where the probability that a certain value will be generated depends upon these frequencies.

The second type of individual attribute is based on a user defined function which randomly generates values. Any function can be used, as long as it returns a value (as a string) each time it is called. Functions can be designed to generate a specific type of data, such as a credit card number, a phone number, a date, or a social security number (as Figure 2 shows); or they can be more general functions that for example generate uniformly or normally distributed numerical values in a certain range (functions can have parameters to be flexible). Users can easily extend our tool by writing their own Python functions according to their needs.

```
sname_attr = GenFreqAttr(attr_name = 'last_name',
                         freq_file = 'last_name_freq.csv')

ssn_attr = GenFuncAttr(attr_name = 'ssn',
                       function = attrgenfunct.gen_ssn)

gender_city_attr = GenCateCateAttr(
                       cate1_attr_name = 'gender',
                       cate2_attr_name = 'city',
                       lookup_file = 'gender_city.csv')

attr_name_list = ['last_name', 'gender', 'city', 'ssn']

attr_data_list = [sname_attr, ssn_attr, gender_city_attr]

test_gen = GenDataSet(output_file_name = 'test_file.csv',
                       write_header_line = True,
                       rec_id_attr_name = 'rec_id',
                       number_of_records = 10000,
                       attribute_name_list = attr_name_list,
                       attribute_data_list = attr_data_list,
                       unicode_encoding = 'ascii')
```

Figure 3: Settings to generate a data set with 10,000 original records consisting of four attributes, to be written into the file `test_file.csv`.

```
edit_corr = CorrValEdit(pos_funct = pos_mod_normal,
                  char_set_funct = char_set_ascii,
                  insert_prob = 0.5,
                  delete_prob = 0.2,
                  substitute_prob = 0.3,
                  transpose_prob = 0.0)
miss_val_corr = CorrMissVal()
phon_corr = CorrValPhonetic(lookup_file = 'phonetic.csv')
sname_ms_corr = CorrCateVal(lookup_file = 'sname_miss.csv')

attr_corr_data = {'last_name':[(0.2, edit_corr),
                               (0.5, sname_ms_corr),
                               (0.3, phon_corr)],
                  'gender':[(1.0, miss_val_corr)],
                  'city':[(0.6, edit_corr),
                          (0.4, phon_corr)],
                  'ssn':[(1.0, miss_val_corr)]}

test_cor = CorrDataSet(num_of_org_rec = 10000,
                  num_of_mod_rec = 5000,
                  attribute_name_list = attr_name_list,
                  max_num_dup_per_rec = 5,
                  num_dup_dist = 'zipf',
                  max_num_mod_per_attr = 1,
                  num_mod_per_rec = 2,
                  attr_corr_prob = {'last_name':0.5,
                        'gender':0.1, 'city':0.3, 'ssn':0.1},
                  attr_corr_funct = attr_corr_data)
```

Figure 4: Settings to corrupt the records generated in Figure 3 and generate 5,000 duplicate records.

4.2 Generating compound attributes

Real-world data commonly show dependencies between attributes, and it is therefore important that such dependencies can be modeled. For example, a person's blood pressure might depend upon their gender, while their salary might depend upon their age. As an original record is generated, the value in one attribute can depend upon a randomly selected value in one or two other attributes. The configuration of these dependencies is based on special types of look-up files, and function definitions similar to the individual attribute generation functions described above. Our tool allows four types of attribute dependencies to be specified.[1]

(1) Categorical-categorical: This type of compound attribute generates categorical values in two attributes, where a value in the second attribute depends upon the value generated in the first attribute. This allows for example the generation of values in an attribute state that depend upon the values in a gender attribute, leading to different distributions of males and females in different states. Values in such compound attributes are generated based on look-up files that contain frequencies of combinations of categorical values. Figure 3 shows how such a compound attribute (gender_city_attr) is defined.

(2) Categorical-continuous: This type of compound attribute generates categorical values in the first attribute, and in the second attribute continuous values that depend upon the categorical values in the first attribute. This allows for example the generation of an attribute that contains continuous salary values that depend upon the values in a marital_status attribute. Values in such compound attributes are generated based on look-up files that contain categorical attribute values and their frequencies of occurrences, and the functions and their parameters used to generate the values in the continuous attribute.

(3) Continuous-continuous: This type of compound attribute generates two continuous values, where values in the second attribute depend upon the values generated in the first attribute. This allows for example the generation of values in an attribute blood_pressure that depend upon the values in an age attribute. Any user defined function that generates numerical values can be used for the second (dependent) attribute, while the values in the first attribute can be generated following either a uniform or normal distribution with parameters provided by the user, with further functions left for future work.

(4) Categorical-categorical-continuous: The last type of compound attribute generates values in two categorical and one continuous attributes, where the values in the second categorical attribute depend upon the values in the first categorical attribute, and the values in the continuous attribute depend upon the values in the second categorical attribute (and therefore indirectly also on the first categorical attribute). This allows for example the generation of an attribute that contains numerical blood pressure values that depend upon categorical gender and city values. Similar to the other compound attributes, look-up files are used to specify the categorical values and their frequencies, and the functions used to create the continuous values.

5. DATA CORRUPTION

To simulate real-world 'dirty' data [4, 6], researchers often not only need to generate data, but also corrupt data by applying various types of modifications to their data. Such corruptions simulate data entry errors that can occur during manual typing, scanning and OCR, or speech recognition of dictated information. Various combinations of these three major types of data entry errors can occur [5].

In the second step of our data generation and corruption process, the corruption module allows the specification of different types of modifications that are applied on selected attributes with certain probabilities, as is shown in Figure 4. The six types of corruption functions available are:

[1]Allowing dependencies between any number of attributes would require complicated look-up file formats, thus making it more difficult to configure our data generator.

(1) Missing value corruptor: This function simply replaces an attribute value with an empty string.

(2) Character edit corruptor: This function randomly selects a position in an attribute value (based on real-world studies more likely towards the middle or end of a value [4]), and applies one of four edits at that position: insert a new character, delete the character, substitute the character with a new character, or transpose the character with one of its neighbors. Different probabilities can be set for these edits.

(3) Keyboard corruptor: This function simulates a typing error and randomly replaces a character with a neighboring (same keyboard row or column) character according to a keyboard layout matrix. Different probabilities can be set for selecting a replacement in a row and or column.

(4) Optical character recognition (OCR) corruptor: This function is based on a look-up file that contains pairs of character sequences that have similar shapes, such as 5↔S or m↔rn. Such pairs can model OCR errors.

(5) Phonetic corruptor: This function simulates phonetic variations (for example for names that sound similar). It is based on a look-up file that contains pairs of phonetic variations of sub-strings, such as ph↔f, or rie↔ry.

(6) Categorical value swap corruptor: The corruption function can be applied on specific categorical attributes by using a look-up file that contains categorical values and their variations (such as misspellings and nicknames). This corruption function allows the creation of realistic modified values collected from real data.

Once the different corruption functions have been defined, as shown in Figure 4, they are attached to selected attributes (see attr_corr_data), and probabilities need to be specified which corruption function will be selected with a certain likelihood. A corruption function can be applied on several attributes, and several corruption functions can be applied on one attribute. An overall probability distribution (argument attr_corr_prob) needs to be set to determine how likely an attribute is selected for corruption. Other required parameters include the number of duplicated records to be generated, the maximum number of duplicated records generated based on a single original record, the distribution of how duplicated records are generated from an original record (following a poisson, uniform, or zipf distribution), the maximum number of corruptions applied on a single attribute, and the number of corruptions applied on one record.

6. EXAMPLE DATA AND DISCUSSION

Once all attributes, their settings, and the corruption functions have been specified, the generation and corruption of a data set can be started with a few simple function calls:

```
org_recs = test_gen.generate()
org_dup_recs = test_cor.corrupt_records(org_recs)
test_gen.write()
```

The write function saves the complete data set into a CSV text file. Figure 5 shows a small example data set generated based on the specifications given in Figures 3 and 4.

We have generated a large number of data sets of various sizes and many different parameter settings to test our tool and evaluate that the generated data follow the specifications given. Generating a million original records consisting of 8 attributes each took around 6 min on a desktop with a 3 GHz CPU and 4 GBytes of main memory, and creating one million duplicates for these records took 38 min on the same machine. To validate the Unicode compatibility of our tool we have generated data sets using three different Japanese character sets (Kanji, Katakana, and Hiragana), which were manually inspected by a Japanese colleague for validity.

```
rec_id,       last_name, gender, city,      ssn
rec-0-org,    wiegold,   male,   new york,  193-12-8750
rec-0-dup-0,  wigold,    ,      new york,  193-12-8750
rec-0-dup-1,  wiekold,   male,   newyork,   193-12-8750
rec-0-dup-2,  wiegld,    male,   nev yorke,
rec-1-org,    sheriff,   female, atlanta,   694-70-0582
rec-1-dup-0,  sherriff,  female, alanta,    694-70-0582
```

Figure 5: Example data with original records (identifiers of the form rec-*X*-org) and their (corrupted) duplicates (identifiers of the form rec-*X*-dup-*Y*, where *X* is the original record they are based on).

7. CONCLUSIONS AND FUTURE WORK

We have presented a flexible and extensible data generation and corruption tool that is capable of generating attribute values both of categorical and continuous types; that can model dependencies between up-to three attributes; and that can corrupt the generated data based on functions that model various real data entry processes. We believe our tool will be valuable for researchers that require realistic personal data to evaluate their algorithms with regard to performance, efficiency, and effectiveness. Our tool is freely available, and can be modified and extended easily.

There are two main avenues for extending our tool. The first is to allow data generation with dependencies between groups of records, such as records that represent a family or household. The second extension is to incorporate temporal aspects, similar to the generator developed by Talburt et al. [7]. Our overall aims are to facilitate both temporal data generation as well as dependencies within groups of records. We also work on a Web based implementation of our tool to allow easy generation and corruption of data sets online.

8. REFERENCES

[1] M. Arehart and K. J. Miller. A ground truth dataset for matching culturally diverse romanized person names. In *LREC*, 2008.

[2] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[3] S. M. Bertolazzi P, De Santis L. Automated record matching in cooperative information systems. In *DQCIS Workshop held at ICDT*, Siena, Italy, 2003.

[4] P. Christen. *Data Matching*. Data-Centric Systems and Applications. Springer, 2012.

[5] P. Christen and A. Pudjijono. Accurate synthetic generation of realistic personal information. In *PAKDD, Springer LNAI 5476*, Bangkok, 2009.

[6] M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. In *ACM SIGMOD*, 1995.

[7] J. Talburt, Y. Zhou, and S. Shivaiah. SOG: a synthetic occupancy generator to support entity resolution instruction and research. In *ICIQ*, Potsdam, 2009.

[8] E. M. Voorhees. The philosophy of information retrieval evaluation. In *Evaluation of cross-language information retrieval systems*, volume LNCS 2406. Springer, 2002.