The Australian National University **Research School of Computer Science, CECS**

COMP3430/COMP8430 – Data Wrangling – 2019

Lab 4: Comparison for Record Linkage Week 6

Overview and Objectives

In today's lab we will continue working on the record linkage program that we began in lab 3. However, we will be moving onto the next step in the linkage process, record pair comparison (especially how to compare strings). The basic idea of string comparison functions is to provide a numerical measurement of how similar two strings of characters are. In practice this is very important, because nicknames, typographical and data entry errors, variations, etc. are all common in many databases, and so only looking for exactly the same string (such as through a database join) may give very poor results.

As with lab 3, we provide you with a starting point `comparison.py` (in the `comp3430_comp8430_reclink-lab-3-6.zip` archive available in Week 5) and two simple string comparison techniques, and we ask you to implement some additional ones yourself.

Lab Questions

Before you begin, please take a moment to go back over the work from lab 3 and remind yourself what we were doing and how the overall program is structured.

Once you are comfortable with the program structure, have a look at the code already provided in `comparison.py` for the functions `exact_comp` and `jaro_comp` so you can see how the string comparison functions work (inputs, return values, etc.). You can experiment by running either function on some of the example data sets to see what the output looks like and how it performs. Maybe add some print statements to see what strings are being compared and what their corresponding calculated similarities are.

As a group, manually calculate the Jaro similarities between the following pairs of names:

- jones / johnson
- michelle / michael
- shackleford / shackelford

Hint: See Section 5.5 in the *Data Matching* book for a description of this algorithm as well as example calculations.

Next, begin to implement the following three string comparison functions. Assume A and B are two list of values (such as q-grams extracted from Strings).

1. Jaccard similarity, which is calculated as,

$$Sim_{Jacc}(A, B) = \frac{|Set(A) \cap Set(B)|}{|Set(A) \cup Set(B)|}$$

2. Dice similarity, which is calculated as,

$$Sim_{Dice}(A, B) = \frac{2 * |Set(A) \cap Set(B)|}{|Set(A)| + |Set(B)|}$$

3. Bag distance, which is calculated as,

$$Dist_{Bag}(A, B) = max(|Bag(A) - Bag(B)|, |Bag(B) - Bag(A)|),$$

   where the function $Bag()$ returns the multiset of values in an element (see lecture 16 for more details).

Discuss, as a group, what the functions $Set()$ and $Bag()$ should return, for example for the q-gram lists $[jo, oh, hn]$ (from 'john') and $[ba, ar, rn, na, ar, rd]$ (from 'barnard').

The basic idea of each of these string comparison techniques was outlined in lecture 16. You may find additional details on the Internet and there are several academic works that have compared different string comparison techniques (see for example the *Field and Record Comparison* chapter in the *Data Matching* book).

Once you have finished implementing each technique, please experiment with them. Some questions to explore are:

1. How does each comparison function compare to the others on different types of data (names, dates, postcodes, etc.)?
2. Are they all equally fast or slow? (You may need to run them on the larger data sets to get a good indication of this). You can use the Python *time.time()* function from the *time* module to measure how long some code is running.
3. Are there some problems / errors that none of these string comparison techniques can deal with?

If you finish the three techniques described above, please also have a look at the Winkler modifications to the Jaro string comparison technique, as well as the edit distance function discussed in lecture 16. Note that for edit distance, you will need to use a dynamic programming implementation.