Andre Holder

HW 15 – Checking code similarities.

My code is designed to check similarities between 8 codes (from deid HW assignment) using two string similarity metrics (Jaro similarity and Absolute Levenshtein distance) and a Universal Sentence Similarity metric.

The string metrics were used to test how similar the raw text output is for 8 codes (converted from text files to strings, tabs substitute for new lines). The Jaro distance is a normalized value such that 0 equates to no **similarity** and 1 is an exact match. The Levenshtein distance between two words (or strings) is the minimum number of **single-character edits** (i.e. insertions, deletions or substitutions) required to change one word into the other. While these metrics are good measures of how similar literal text could be between two strings, it does not account for the similarity in the *meaning* behind them.

In addition to noting similarity in characters, the Universal Sentence Encoders (such as that used in Tensorflow) also creates a context in which two sentences with the similar meaning are judged to be “similar”. Because computer code can exhibit a similar behavior, I created a framework in which the intent of the code is similar. I chose to ignore the code comments as this was the most likely to have semantic variation, and contributed little to code functionality. After review of the code content for each, I broke the code down into seven major tasks: (1) Describe the identifier (since everyone chose different types of identifiers); (2) Name the compiler; (3) Find the iterations throughout each string (most commonly done using a for loop); (4) Create the string to write to a file; (5) Write the string; (6) Open the output file; and (7) Export/print the string values to the output file. I also included the different variable names to ensure that this variation was not a factor in determining code similarity. The differences were then visualized using a semantic textual similarity heat map.

My results are found below:

The Jaro similarity metric measures are:

code 1 vs code 2: 0.5636364720630022

code 1 vs code 3: 0.7693673490760204

code 1 vs code 4: 0.7741594158957942

code 1 vs code 5: 0.7791594891948573

code 1 vs code 6: 0.7482529603015388

code 1 vs code 7: 0.7090608120128867

code 2 vs code 3: 0.5867514442377398

code 2 vs code 4: 0.5947612031278235

code 2 vs code 5: 0.5867488786590064

code 2 vs code 6: 0.5679697031819618

code 2 vs code 7: 0.6245976809757386

code 2 vs code 8: 0.6073715186342309

code 3 vs code 4: 0.822566622827735

code 3 vs code 5: 0.8010374003352917

code 3 vs code 6: 0.7763666382428639

code 3 vs code 7: 0.7195949762116066

code 3 vs code 8: 0.6567340587947946

code 4 vs code 5: 0.8142641234970611

code 4 vs code 6: 0.779738456966431

code 4 vs code 7: 0.7243258176235227

code 4 vs code 8: 0.6622267258095486

code 5 vs code 6: 0.7819688398610026

code 5 vs code 7: 0.7291634696614113

code 5 vs code 8: 0.6694987258141047

code 6 vs code 7: 0.7116282243114193

code 6 vs code 8: 0.676308203734497

code 7 vs code 8: 0.7420013817826021

The Absolute Levenshtein distance metric measures are:

code 1 vs code 2: 15615

code 1 vs code 3: 2090

code 1 vs code 4: 1866

code 1 vs code 5: 1846

code 1 vs code 6: 3099

code 1 vs code 7: 3660

code 2 vs code 3: 15168

code 2 vs code 4: 14970

code 2 vs code 5: 15387

code 2 vs code 6: 16013

code 2 vs code 7: 14610

code 2 vs code 8: 14620

code 3 vs code 4: 261

code 3 vs code 5: 683

code 3 vs code 6: 2247

code 3 vs code 7: 2877

code 3 vs code 8: 6251

code 4 vs code 5: 427

code 4 vs code 6: 2053

code 4 vs code 7: 2638

code 4 vs code 8: 6217

code 5 vs code 6: 2250

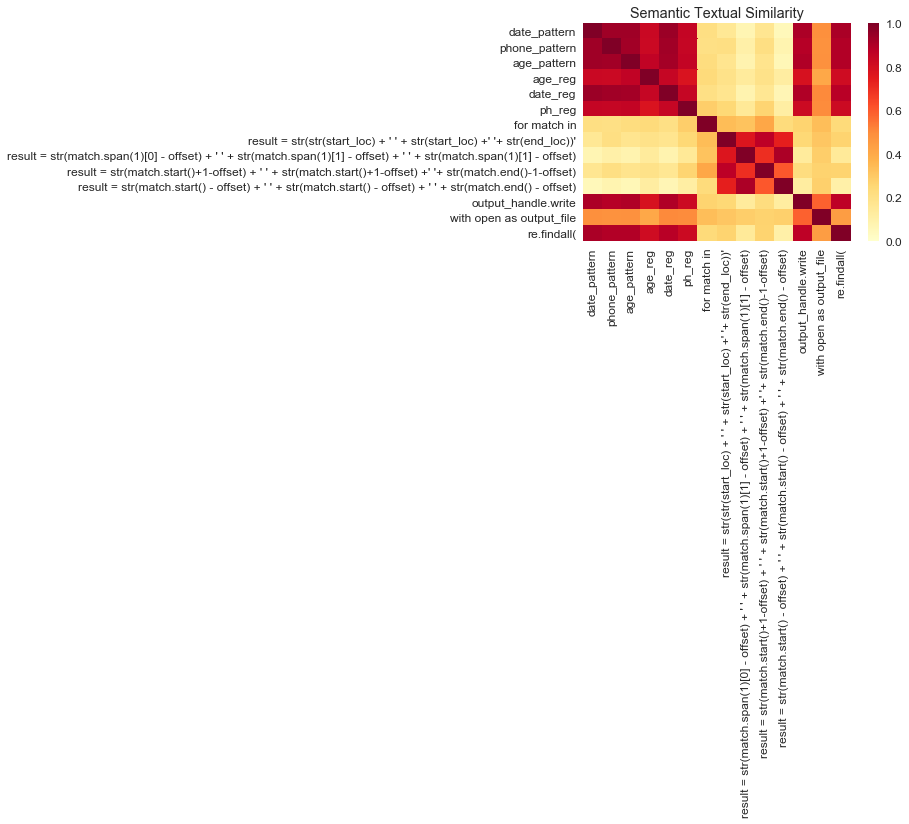
code 5 vs code 7: 2600

code 5 vs code 8: 6264

code 6 vs code 7: 3868

code 6 vs code 8: 6059

code 7 vs code 8: 6248



The 8 different text files of “deid” code were compared successfully using both string and semantic methods. When the string comparisons were performed, very few txt file comparisons were found to be “similar” (e.g. codes 1 & 2, and codes 2 & 6 using the Jaro and Levenshtein distance methods). By contrast, when certain common syntax was extracted from the text files by my program, and like “messages” were grouped together, certain patterns emerged that were more common across multiple text files. Five clusters arise from the semantic textual similarity (STS) diagram (the darker areas corresponding to higher text similarity). Some clusters were expected and simply represent those text strings that were defined as similar in my code: (1) top left, the identifier that was chosen for deidentification (date\_pattern, phone pattern, and age\_pattern); (2) center, between those that recreate a string of the text to deidentify (e.g. “result=str(str…”)); (3) and bottom right, which opens the output file and exports/prints the string values to the output file. However, other unexpected clusters appeared: (4) Top right, the STS noted that the variable names used as identifiers (e.g. date\_pattern) were often used in the compilers (age\_reg, date\_reg, and ph\_reg), thus showing some similarity in function; and (5) bottom left, the STS noted similarity between the variable names used as identifiers and the output handles and the text used to export.

In summary, while the string similarity metrics note similarities in syntax, the STS notes contextual similarities which offer a more robust comparison.