Matching and Mapping for the TSDB (KairosDB)

SEED project

November 29, 2015

Contents

1 Introduction

The document records the process of implementing the mapping and matching operations for TSDB (KairosDB.

2 SEED Mapping and Matching

2.1 Mapping

Mapping rename the column/field names of the imported data set to terms in Building Energy Data Exchange Specification (BEDES) [5]. In the process, the program search through the terms in BEDES and returns a suggested field name for each of the imported field in the dataset. user can 1) choose which field they want to retain or ignore 2) modify the suggested mapping and input the BEDES term. During the input process, there is a list of 20 strings in the drop-down menu under the input bar each string contains the current input as a sub string (Figure 1).

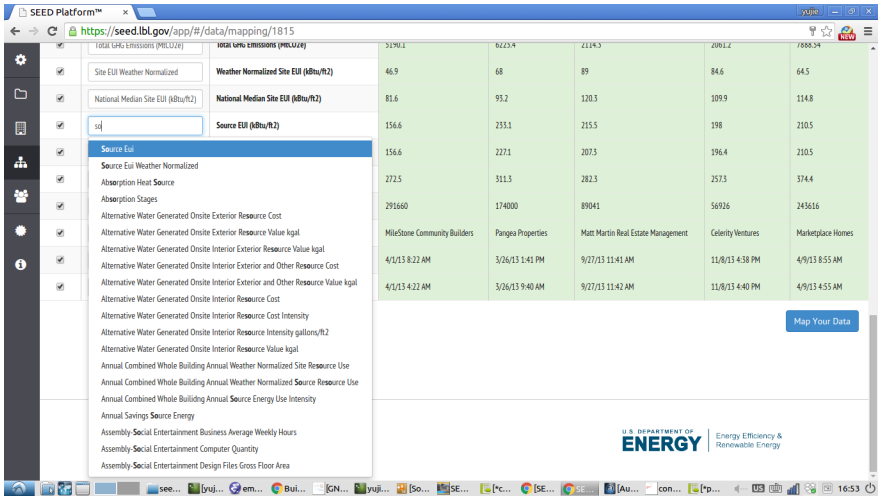

Figure 1: Drop-down list for input BEDES term

[3]

The guideline for the process is "to improve results in matching buildings across different data files, map as many of the following four (4) fields as possible: Tax Lot ID, PM Property ID, Custom ID, Address Line 1" [3].

2.2 Matching

In this process the fields in the two input tables, the building list and the PM data, are combined. The combining process utilizes some common fields from the two tables. The process uses fuzzy string searching [2,6] to auto map the records in the two tables and returns a confidence score for the matching. In the table below we can see the leading zeros does not affect the matching result.

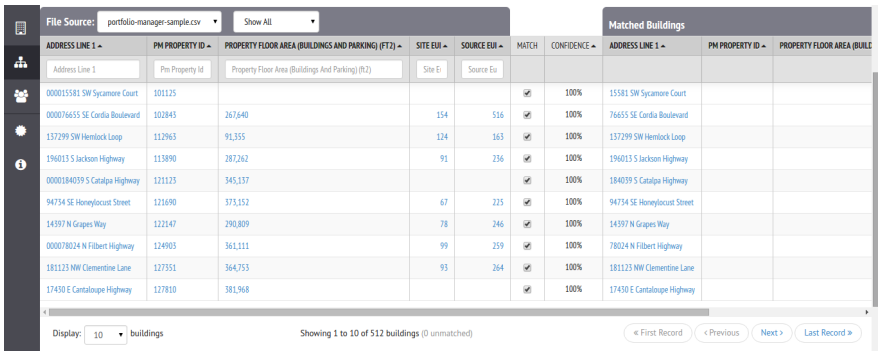

Figure 2: Matching result with confidence score

[3]

The matching can be manually corrected by clicking on the value of the shared field in the source table and one can choose **one or more** records from the drop-down list that matches the record in the source table (Figure 3).

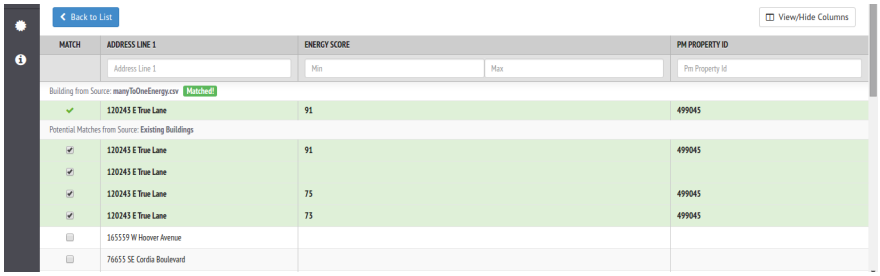

Figure 3: Manually correct matching result by selecting one or more potential record

[3]

In the matching process, one table is the source and the other is the target. For each record/row in the target table, there is a unique record in the source table that matches this record, a match will be successful, but the score of confidence will not be 100%. if there are multiple records in the target that matches the source.

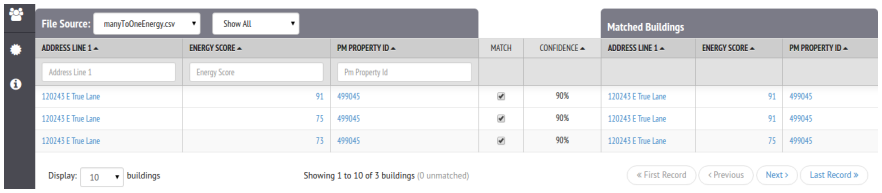

Figure 4: Three records in the target table (PM table) matches one record in the source table (Building table)

[3]

3 Implementation strategy of mapping and matching: approximate string matching

The approximate string matching aims at finding strings that *approximately* matches some pattern. The matching is normally evaluated by some edit distance, which is the minimum number of primitive operations (e.g. insertion, deletion and substitution) needed to convert the approximate match to an exact match [6]. There are several versions of the set of primitive operations. One common definition is the Levenshtein distance, which include single character operations as insertion, deletion and substitution.

There is a package in Python called FuzzyWuzzy [4] that evaluates the difference between strings with Levenshtein distance. The package is built upon the Python package difflib (which has a class called "SequenceMatcher" that compares two sequences (str, unicode, list, tuple, bytearray, buffer, xrange) as long as they are hashable (those that can become a dictionary key). Immutable types (number, string, tuples) are all hashable in Python). There are some explanation of the fuzzywuzzy package here. The key functions include [1]:

```
from fuzzywuzzy import fuzz
# simple ratio: pure edit distance
# similar to difflib.SequenceMatcher
fuzz.ratio("this is a test", "this is a test!")

# partial ratio: when s1 and s2 have very different lengths (WOLG, s1
< s2),
partial_ratio(s1, s2) returns fuzz.ratio(s1, s2') where s2' is a
sub string of s2 and len(s1) == len(s2')

fuzz.partial_ratio("this is a test", "this is a test!")

# token sort ratio:
# to deal with the word re-order of strings
break strings to tokens, sort tokens and then re-assemble them to strings before calculating t
fuzz.token_sort_ratio("fuzzy wuzzy was a bear", "wuzzy fuzzy was a bear")
```

```
# extracting a list of tuples of (str, score) where str is in choices
# and score is the matching score between query and choice)
extract(query, choices, processor=None, scorer=None, limit=5)
```

However, if the address line is selected as the field for matching calculation, a **substitution of common abbreviations** should be performed before the string searching process (Figure 2 in this paper).

References

[1] JeffPaine et al. seatgeek/fuzzywuzzy. web, November 2015. https://github.com/seatgeek/fuzzywuzzy.

[2] Lawrence Berkeley National Laboratory. Seed 1.1 tutorial. web, November 2015. https://windows.lbl.gov/projects/SEED/IntroTutorial/SEED%201.2%20Overview.htm.

[3] Lawrence Berkeley National Laboratory. Seed platform. web, November 2015. https://seed.lbl.gov/app/#/data/mapping/1815.

[4] Python Software Foundation. fuzzywuzzy 0.8.0. web, November 2015. https://pypi.python.org/pypi/fuzzywuzzy.

[5] U.S. Department of Energy. Bedes. web, November 2015. https://bedes.lbl.gov/.

[6] Wikipedia. Approximate string matching. web, November 2015. https://en.wikipedia.org/wiki/Approximate_string_matching.