CS 784: Advanced Topics in Database Management Systems: Data Science

Project Stage IV

Designing Matcher and Predicting Labels for Unlabeled Pairs

Date: May 13, 2016

Group Member Information
Artsiom Hovarau: hovarau@cs.wisc.edu
Apul Jain: apul@cs.wisc.edu
Zhuowei Cai: zhuoweic@cs.wisc.edu

Website: http://pages.cs.wisc.edu/~zhuoweic/index.html

Introduction:

In this project stage, we designed a machine learning and rule-based matcher which predicts whether two product descriptions correspond to the same (match) product or not (mismatch).

For this project stage, we were given 20K labelled product pairs, from which 10K pairs are randomly sampled (set X) to train our model. The remaining 10K samples (set Y) are used to evaluate our model. Below is the performance of our matcher (trained on set X and applied on set Y):

Blind test results (Stage 3):
Precision: 97.16%
Recall: 91.80%
F1: 94.41%

Previous results on Set Y (Stage 3):
Precision: 96.07%
Recall: 89.29%
F1: 92.55%

Final Results on Set Y (Stage 4):
Precision: 96.13%
Recall: 92.06%
F1: 94.06%

Details about the Matcher:

Design of feature vector:

Our Matcher M is designed using features derived from the product tuple pair. For each tuple we used the following attributes to design our feature vector:

Attributes used to design feature vector:

1. NAME = "Product Name"
2. TYPE = "Product Type"
3. CLASS = "Product Segment"
4. BRAND = "Brand"
5. CATE = "Category"

6. GTIN = "GTIN"
7. COMP = "Country of Origin: Components"
8. SHDC = "Product Short Description"
9. UPC = "UPC"
10. MANF_PART = "Manufacturer Part Number"
11. WARR = "Warranty Information"
12. MANF = "Manufacturer"
13. COLOR = "Color"
14. PROD_LEN = "Assembled Product Length"
15. ACTUAL_COLOR = "Actual Color"

We computed similarity measures for every attribute listed above, namely

1. Jaccard
2. Cosine
3. Levenshtein
4. Jaro
5. Jaro with q-gram = 3
6. Smith-Waterman
7. Jaro-winkler
8. Needleman-wunsch
9. Affine
10. Jaccard
11. Overlap coefficient
12. Cosine
13. Monge-elkan

In addition, for each pair we also used the lengths of the attributes (2 per attribute) as a feature which gave us an additional ~1% improvement in Precision and ~2% in Recall. So for each tuple we have in total #attribute x #similarity measure (15 x (13 + 2) = 15 x 15) dimensional feature vector. We trained several machine learning models using this feature vector. Based on our experiments and observations, Random Forest achieves the highest precision/recall.

Model used: Random Forest
In this model, we learn a set of decision trees where each node in the decision tree represents a condition on some attribute to split the tree based on the cross entropy of the attribute.

<u>**Training and Testing phase:**</u>

First we split dataset into two sets: Set X (training set) and Set Y (test set). Each set consists of 10K tuple pairs.

- Training phase:
    1. For each tuple pair in the Set X, we compute a 15 x 15 dimensional feature vector using the similarity measures and length of the attributes as described above.
    2. Train Random Forest classifier with number of estimators = 1000, which achieves the best performance based on our experiments.

- Test phase:
    1. Apply the constructed Matcher M above on the Set Y.
    2. Get the predicted labels and calculate the **confidence score** for each label.
    3. Apply threshold on confidence score. We used **threshold = 0.235.**
    4. Labels with confidence score < threshold are predicted as “unknown”. Rest are considered as valid prediction by model.
    5. Calculate the precision and recall to estimate the performance of the model.

*Confidence score is computed as the difference between the likelihood of one label and another predicted by the Random Forest classifier.

The above technique gave us precision/recall of 96/89% on set Y. We further applied rule-based techniques to postprocess the labeling and improve the performance in stage 4.

<u>**Rule-based techniques:**</u>

After the model predicts the labels, we applied rules to see if a tuple pair match. Specifically we look at some critical attributes of the product pair. Two products are definitely different if these attributes are different. In particular we used the following rules:

1. For product category: Batteries, we used the rule that if mAH rating does not match, it is a mismatch

    <u>Corresponding Regex:</u> re.search("\s[0-9]+\s*mah\s", prod_name.lower())

2. Color: For products like iPhone, iPad, Mac, we figured that if color doesn’t match, then the entire product pair must be a mismatch. We write a color

extractor to extract color from the product description and compares the color of two products. If they are not the same, output mismatch.

Other rules that we tried which did not give us much improvement:

1. Refurbished: If a prod1[“name”] contains keyword “refurbished” in the product name but prod2[“name”] doesn’t or vice versa, we consider it to be a mismatch. However, there were several examples where this was not really true.

2. Assembled length/width: There are products with different assembled length/width but still they match.

Example (matched):

- `Epson Expression Home XP-400 Small-in-One Printer/Copier/Scanner`
- `Epson Expression Home C11CC07201 XP-400 Wireless All-in-One (`**`Refurbished`**`)`

Final Results on Set Y (Stage 4):
Precision: 96.13%
Recall: 92.06%
F1: 94.06%

Discussion: Why can’t P/R be further improved?
Because the rules are not generic enough to even cover a significant number of mis-labelled pairs of a specific type. For example, there is no rule for specific product TVs which could look at the size and identify if it is a mismatch or match.

What’s more, the data has lots of missing values. Usually one product tuple in the tuple pair misses a value in an attribute while the other does not. This makes a rule-based comparison very difficult.

Another problem as mentioned in the previous section is that some product names contain confusing information. For example, initially we thought that if one product has keyword “Refurbished” in its Product Name or one of its other attributes while the other doesn’t, then they should be labelled Mismatch. However, it turns out that there are such pair that is labelled Match.

Furthermore, the above remark also concerns the attributes such as Assembled Product Weight/Height, where even if the corresponding attributes of both products are present and different, they can still be a match. So indeed, we could not use these attributes when writing rules, for otherwise it might negatively affected the already computed true positives, which is proven in our experiment.

Feedback on the python string matching package:

We did not spot anything wrong regarding the result from the library functions. However, some of the functions in the package are pretty slow. This especially concerns the following: **Needleman-wunsch, Smith-waterman, Affine**, **Monge-elkan**.

Thus, it would be better if the time complexity of those functions are added into the documentation so that user of the package can make some trade-off between time and performance in advance.

Also, for missing values (None in Python), the package can provide some default parameters that automatically fill up the missing value. In this way, users don't need to write their own 'wrapper' function for every similarity function in the library. In the current package, the function will throw an error if any parameter is None.

Another essentially important remark is that some functions do not return values strictly in the range [0, 1] like **Levenshtein, Smith-waterman** which return ***non-negative*** values, and **Needleman-wunsch, Affine** which sometimes return ***negative*** integer values. Therefore if there were an additional flag inside as a parameter in the functions setting which set to true would have them return a normalized value (between 0 and 1), it would free users in the future from not only having to write 'wrapper' normalizers but also from looking into the description of these functions, which could be very much confusing and time-consuming.

There is also an in-built package in Python called difflib providing a class SequenceMatcher which can be used for computing the similarities between two strings. However, it is very rudimentary in a way that it contains only a few functions and therefore it is not appropriate for data scientists.

So if the above-mentioned remarks are to be taken into account, the py-stringmatching package would be a great contribution to the field of the practical data science.