Product Matching

Project Stage 4

Ainur Ainabekova, Nick Schober, Sunny Shah

UW-Madison CS 784

INTRODUCTION

For this stage of the project we needed to improve the precision and recall of our matcher that we developed in stage 3. In this report we describe improvements we made to the matcher, some of the techniques that we tried but didn't get reasonable results and analysis of what can be still done for the future work. Overall in this stage of the project, we did data cleaning compared to the stage 3, used more features and also used hand-crafted rules for matching entities and making predictions.

ACCURACY - PRECISION AND RECALL

Here are our results from applying M to the set Y:

Precision: 95%

Recall: 83%

F-1: 88.59%

MATCHER METHODOLOGY

Description of the matcher for stage 3:

Our matcher uses values of five different product attributes: product name, product type, product segment, product long description and brand. We chose these because they are the most frequently appearing attributes in all product pairs (thus preventing us throwing out too many pairs as "unknown") and are also relevant to classifying whether or not two products are the same. The following are the features we use for training our

model:

- Jaccard, Needleman-Wunsch, Jaro-Winkler, Levenshtein and Cosine similarity functions on all attributes except “product long description”
- Jaccard and Cosine similarity functions on “product long description”

This totals to 22 features. We go over set X (the dataset is divided at the beginning into sets X and Y, chosen randomly), convert each product pair into a feature vector and use 5 fold cross validation with the chosen machine learning model. Following this, we calculate the precision and recall. The machine learning algorithm we use is Random Forests with 100 classifiers. This gave us the best results among all algorithms we tried.

If any attribute except “brand” is missing, we treat the corresponding product pair as unknown. If the brand is missing in any product, we use the brand name extractor we developed in stage 2 to extract the product’s brand from its “product name” attribute.

Following is the precision and recall we obtained by applying the above procedure to set Y:

Precision: 91.5%

Recall: 84.5%

F-1: 87.86%

Following is the precision and recall we obtained by applying the above procedure to the unlabeled data set:

Precision: 92.23%

Recall: 85.46%

F-1: 88.71%

Changes we made to improve the matcher for stage 4:

- Data cleaning
    - Removing all HTML tags from “product long description”.
    - Removing stop words from all attributes such as “or”, “and”, “,”, “of”, “too”, etc.

- Additional features:
    - The similarity functions used for all attributes except “product long description” are - Jaccard, Needleman-Wunsch, Jaro-Winkler, Levenshtein, Smith-Waterman, Affine, Cosine, Tfidf, Soft-tfidf, absolute value of difference in lengths of product name, absolute value of difference in lengths of product long description and number of attributes. Similarity functions used for “product long description” are - Jaccard, Cosine, Tfidf, Soft-tfidf, absolute value of difference in lengths of product name, absolute value of difference in lengths of product long description and number of attributes.
    - In a pair, if one product has the brand attribute but is missing for the other in the pair, we lookup the former’s brand value in the latter’s “product long description”, “product short description” and “product name” attributes. If a match is found, we treat them as having the same brand. If not, we use our brand extractor as in stage 3.
    - We also use the “color” attribute. If only one of the products has color, we use the same logic for finding color of the other product as we do for “brand”, as described above. If the same color cannot be found in the other product’s attributes, we declare the color to be unmatched. In addition, if both products have their color missing, we look up both product’s “product name”, “product long description” and “product short description” to find a color. The color values we check for in these attributes are - "Red", "Green", "Black", "Blue", "Yellow", "White", "Maroon", "Purple", "Orange", "Lavender", "Multicolor", "Gray", "Ivory", "Silver", "Tan". Multiple colors can be found from all attributes for one product.

      We then compute an intersection of the colors retrieved for both products. If the intersection size is greater than 0, we declare the colors to match. Otherwise, they are treated as not being a match. If no colors are found from either product’s attributes, we treat the pair as “unknown”.
    - Total number of features is thus 51 now.

- Hand-crafted rules

  Following rules are applied after applying the machine learning model:

    - If a pair is classified as a match but both of them have the “color” attribute and the colors differ, we set the pair to mismatch.
    - If a pair is classified as a match, both products have the “product long description” attribute such that the length of the smaller description is less than 20% of the length of the larger description and the jaccard value of the product names is less than 0.35, we treat the pair as a mismatch.

  Following rules are applied before applying the machine learning model:

    - If one of the products contains word “Refurbished” in “product name”, we treat the corresponding pair as a mismatch.
    - If “color” was not found in either product but their brand names have an edit distance of 1 or 0, we treat the pair as a match. Otherwise, we treat it is an unknown.

The rest of the logic remains the same as for stage 3.

Techniques we tried but that didn’t significantly improve the precision or recall:

- For the pair of attributes where one of them is missing or both are missing, automatically assigning similarity function to 0.5.
- We used the unlabeled dataset from stage 3 to add to the brand names we use in our brand name extractor.
- Assigned different weights to all features, with a higher weight to features based on the “product name”.

Here are the results of applying the matcher from stage 4 to set Y:

Precision: 95%

Recall: 83.1%

F-1: 88.6%

RESULTS ANALYSIS - HOW TO IMPROVE?

Here are some suggestions on how the matcher can be improved to achieve higher precision/recall:

- It is generally difficult to generate hand-crafted rules for the kind of problems where we need domain specific knowledge. One of the reasons why we are not

having higher precision and recall might be the fact that our rules are not complicated enough for this problem and are not covering all the small peculiarities of the training dataset. A closer inspection of the false negatives and false positives to add rules that cover a large subset of them would improve the precision and/or recall.

- It might be necessary to handle missing values for some of our attributes in a different manner than simply declaring the pair containing the corresponding product as “unknown”. If the attribute is missing, some other existing attribute could be used to account and compensate for the missing attribute, or feature based on the missing attribute could be hard-coded to have some value unique to missing values.
- There are also many product pairs where almost all of the attribute values (of the attributes we considered) are similar/exactly the same, so that even a human looking at these two products would declare the pair as a match. However, according to the training data the label is a mismatch. Our matcher identifies them as a match due to the fact that all of the features are so similar. We would need to hand-craft rules for such cases or identify some other attribute that would distinguish them.

DISCUSSION OF THE PY_STRINGMATCHING PACKAGE

The py_stringmatching package has been very useful in the development of the matcher for both stages, especially since an all-encompassing package that has all required similarity functions is very hard to find.

The similarity functions we used have been mentioned earlier. Following are some of the pros we observed of using the package:

- Good documentation, and the examples provided with each similarity function make it a lot easier to understand and use. The explanation of the arguments and return values was simple and easy to understand.
- The API is simple and straightforward. Virtually no learning curve was required, and the hints provided by the IDE for each function’s arguments made it obvious what should be given as arguments to each function.
- Most major similarity functions are included, as well as the required tokenizers.

Following are some places where improvements can be made:

- Needleman-Wunsch and Affine take a somewhat longer time to execute on strings

longer than about 30 in length. This is after execution on a quad-core Intel i7 machine.
- Before releasing to the public, the website for the package can be modified to improve its look-and-feel and the font size could be increased for screens with a greater resolution (this input is based on viewing the site on a 1920x1080 screen).

Other Notes:

- We did not rewrite any of the functionality in the py_stringmatching package, it fully met our needs for this project from a functional perspective.
- We did not identify any bugs with the similarity functions.
- Regarding scaling up, as noted above a couple of the similarity functions seem to take a bit longer time to execute on longer strings. Expanding on this a bit, our overall experience was that we could generate the ~50 features for our set of 10k products (set X) in 20-30 minutes, and when we also needed to do so for the unlabeled dataset(20k additional pairs), the total runtime of our matcher would approach 2 hours.
- This leads into a suggestion for potential additional functionality. One of our group members had a good insight that once we got to the stage of adding hand-coded rules (and thus not changing the features passed to our ML learner between runs), we could save off the feature vectors generated from py_stringmatching once, then load from file to speed the process up as we tweaked our hand-coded rules. What might be nice is to somehow encapsulate this in the process such that a small tweak could be made to one feature (or add/remove a feature) without requiring recalculating the entire set. However, this might be slightly outside the scope of the py_stringmatching package.