CS839 Project Stage 3: Entity Matching

Date: 4/18/2018 Team name: Big HIT

Team members: **H**oai Nguyen, **I**saac Sung, **T**rang Vu

Summary: This report describes how we have performed Entity Matching (EM). This is a continuation of project stage 2 in which we created two tables of movie data, one from IMDb [1] and one from TMDb [2]. In this project stage, we developed a matcher that matches tuple pairs between these two tables through various blocking and matching strategies in the py_entitymatching module [3]. The goal is to create a matcher with more than 90% precision and as high as recall as possible. Table 1 below summarizes the minimal information necessary for this report, and the sections proceeding that explain our project in further detail.

Table 1: Project summary

Entity type	Movies from internet movie databases		
Entity Sources	www.imdb.com	www.themoviedb.org	
Data Tables	A.csv: The IMDb movies table, contains 3500 tuples B.csv: The TMDb movies table, contains 5490 tuples C.csv: Candidate set of all tuple pairs that survive the blocking step, contains 2547 tuples G.csv: Labeled sample set of candidate tuples, contains 400 tuples with 171 positive labels I.csv: Development set used for classifier training, contains 200 tuples J.csv: Evaluation set used to evaluate predictions made by selected classifier trained on I.		
Blocker	An attribute-equivalence blocker combined with a black-box blocker. (see details below).		
Cross Validation on Set I			
Classifiers	**Precision**	**Recall**	**F1**
Decision Tree	0.965	0.980	0.970
Random Forest	0.975	0.990	0.981
SVM	0.900	0.176	0.287
Naive Bayes	0.965	1.00	0.981
Logistic Regression	0.979	0.980	0.959
Linear Regression	0.964	0.960	0.959
Final best matcher (Y = Logistic Regression)	Logistic Regression (highest precision score)		
	0.979	0.980	0.959
Evaluation on Set J			
	Precision	**Recall**	**F1**
Decision Tree	0.967	1.00	0.983

Random Forest	0.989	1.00	0.994
SVM	1.00	0.341	0.509
Naive Bayes	0.967	1.00	0.983
Logistic Regression	1.00	1.00	1.00
Linear Regression	0.967	1.00	0.983
Time Estimates	Blocking: 3 days	Labeling: 1 hour	Finding best matcher: 3 hours
Conclusion	We have developed a matcher that met the classification requirements, which are having at least 90% precision and as high recall as possible.		

1. Entity Type and Sources

The entity type for this project stage was movies gathered from internet databases. The two sources we used were www.imdb.com and www.themoviedb.org. Many features were extracted from the web pages, including title, director, cast, release year, genre, user rating, etc. The IMDb movie table has 3500 tuples, and TMDb movie table has 5490 tuples. The two tables have the same schema, which has a total of 18 attributes, and multiple values in an attribute are separated by semicolons.

2. Blocker Description

Each table had 3,000+ items, so a full cross-matched comparison without any blocking would have resulted in 9,000,000+ tuple pairs, which would take a long time to compute. Thus, we implemented a blocking stage before we attempted matching. Our selected blocker is an attribute-equivalence blocker combined with a black-box blocker provided by the py_entitymatching package. Specifically, we first applied an attribute-equivalence blockers on tuples that do not have the same 'release_year'. Then we applied a black-box blocker using our own blocking rule on the candidate set produced from the first blocker. We defined a blocking rule such that if a tuple pair that does not have at least one match in either 'directors', 'writers', or 'cast', it will be blocked. For a pair of movies, we extracted directors, writers and cast to 3 separate lists. We iterate through the directors' list for each tuple and compute the similarity score between each pair of directors using string Jaccard measure provided by the py_stringmatching package [4]. If the score is greater than 0.8 (almost exact match), then we set a flag indicating there is a match in director name to True. We did the same for the other lists. If one of these flags is True, the tuple pair survives the blocking rules, otherwise, it would be blocked. This blocker however, takes longer time to run compared to other existing blockers (approximately 10 minutes). Applying this blocker on the IMDb and TMDb data tables, we got a set of candidates C of 2547 (pairs of) tuples.

3. Sample Set

From our candidate set C achieved through blocking, we sampled 400 tuples and manually labeled this set iteratively so that we ensure that the density of the positive labels in this set is reasonable. Our labeled data set G has 171 positive labels.

4. Matcher Selection and Evaluation

4.1 Matcher Selection

We divided the labeled data (set G) equally to set I which is used to select the best matcher and set J, which is used to evaluate the selected matcher's prediction. We performed 10-fold cross validation using 6 different learning methods provided in Magellan: Decision Tree, Random Forest, SVM, Naive Bayes, Logistic Regression, Linear Regression on set I, and selected the matcher with highest precision to be the best matcher. The precision, recall and F1-scores of these matchers are shown in Table 2 below (extracted from Table 1). Our goal is to get a precision score of at least 90% and as high as recall as possible.

Table 2: Precision, recall and F1 scores for 6 different matchers from cross validation

Matchers	**Precision**	**Recall**	**F1**
Decision Tree	0.965	0.980	0.970
Random Forest	0.975	0.990	0.981
SVM	0.900	0.176	0.287
Naives Bayes	0.965	1.00	0.981
Logistic Regression	0.979	0.980	0.978
Linear Regression	0.964	0.969	0.959

Based on the results above, all classifier except SVM meet the classification requirement of at least 90% precision and high recall and therefore we don't need to do any debugging. Logistic regression classifier has the highest precision so we chose it to be our final best matcher (matcher Y) with the precision, recall, and F1-scores highlighted in blue in Table 2 above.

4.2 Matcher Evaluation
We trained all 6 matchers on set I and applied the trained matchers on set J to evaluate the predictive power of each of these matchers. The precision, recall and F1 scores of the predictions are shown in Table 3 below (extracted from Table 1)

Table 3: Precision, recall and F1 scores for 6 different matchers on set J

Matchers	**Precision**	**Recall**	**F1**
Decision Tree	0.967	1.00	0.983
Random Forest	0.989	1.00	0.994
SVM	1.00	0.341	0.509
Naives Bayes	0.967	1.00	0.983
Logistic Regression	1.00	1.00	1.00
Linear Regression	0.967	1.00	0.983

Our final best matcher is logistic regression so the precision, recall and F1 scores of the final best matcher (matcher Y) on set J are the same as values highlighted in blue in Table 3 above.

5. Discussion
5.1 Time estimates
It took us about 3 days to find our best blockers, about 1 hours to label our sample set, and another 2-3 hours to find the best matchers. For blocking, we basically tried different existing blockers (overlap, attribute-equivalence, rule-based blockers) first on various attributes and examined the outputs of the

candidate sets as well as of the blocked sets before created our own blocker. The sampling and labeling step can be done easily and efficiently in Excel. Since we got good precision and recall in the first time running cross validation, we did not have to do any debugging in matching step so it did not take us too long.

5.2 Feedback on Magellan

[Hoai] The instruction was long and hard to follow. It would be useful to have a short instruction "Getting Started With Magellan" that shows users how to get started with some toy examples end to end. Then the users can refer to the long page instruction for details. There is an issue with installing the package with Anaconda. The installation process didn't show an error but when you try to run and use the library, you will likely see the following error:

```
Traceback (most recent call last):
File "matcher.py", line 1, in <module>
import py_entitymatching as em
File "/anaconda3/lib/python3.6/site-packages/py_entitymatching/__init__.py", line 42, in <module>
from py_entitymatching.debugblocker.debugblocker import debug_blocker
File "/anaconda3/lib/python3.6/site-packages/py_entitymatching/debugblocker/debugblocker.py", line 14, in <module> from py_entitymatching.debugblocker.debugblocker_cython import
ImportError: dlopen(/anaconda3/lib/python3.6/site-packages/py_entitymatching/debugblocker/debugblocker_cython.cpython-36m-darwin.so, 2): Symbol not found: __ZNSt11logic_errorC2EPKc
Referenced from: /anaconda3/lib/python3.6/site-packages/py_entitymatching/debugblocker/debugblocker_cython.cpython-36m-darwin.so
Expected in: /usr/lib/libstdc++.6.0.9.dylib
in /anaconda3/lib/python3.6/site-packages/py_entitymatching/debugblocker/debugblocker_cython.cpython-36m-darwin.so
```

The solution was to remove Anaconda and then install Magellan with Pip.

[Isaac]
The good: The provided blockers and matchers were very powerful and easy to use once Magellan was installed. The functions for blockers and matchers had nice parameters and flexibility to customize to your own problem set.
The bad: I had to do a bit of Googling to figure out how to install Magellan into my Anaconda work environment. I found this page in the Conda documentation that helped me install Magellan using pip into my conda environment:
https://conda.io/docs/user-guide/tasks/manage-pkgs.html#installing-non-conda-packages.
This might be a helpful link to include to other students in the future, if they are using Anaconda for their Python environment.

[Trang]
The good: I am impressed with the matching tool, especially with feature creation and classifier selection. It works seamlessly, and is easy to use. I also like that the blockers come with many options and flavors and that we can define our own blockers.
The bad: While using the py_entitymatching module is pretty straight forward, it was the installation of the module package and its dependencies that took me longer time and lots of frustration and Googling. I am not sure if this is an issue generalized to everyone but for me, my laptop lacks many dependencies required for proper installation of Magellan. For example, to install xgboost, I have to install a C++ compiler mingw64. For py_entitymatching, I need to get Visual Studio Windows SDK which takes up more than 1GBs of memory, however this requirement (Windows SDK) is not specified in the module installation guideline.

For sampling and labeling step, while it is nice to be able to view the data, it was misleading that I can actually do the labeling in place because I could change the column entries in the dataframe viewer. However, it was unclear how the labeled data can be saved if labeling is done in place. I wasted 3 hours labeling the data directly in the dataframe viewer thinking it would be saved somehow but it wasn't. I ended up labeling the data in Excel which was a lot faster.
The ugly: Thankfully there is none.

5.3 Recommendation

[Trang]
Overall I think the package guidelines can be improved by adding more details in installation, potential errors and how to fix them.
For the rule-based blockers, it would be nice to include a description of the features explaining what they mean. I could make a guess but there is no guarantee my guess is correct without looking at the source codes.

References:

[1] IMDb: http://www.imdb.com/
[2] TMDb: https://www.themoviedb.org/?language=en
[3] py_entitymatching module: http://anhaidgroup.github.io/py_entitymatching/v0.3.x/index.html
[4] py_stringmatching module: http://anhaidgroup.github.io/py_stringmatching/v0.4.x/index.html