CS838 Project Stage 3 Report

Fangzhou Mu, Nafisah Islam, Meera George

I. Datasets

We used the datasets *tracks.csv* and *songs.csv* to perform entity matching on songs. We did not work on our own dataset due to the following reasons:

1) there are surprisingly few *exact* matches between the laptops sold on Amazon and the laptops reviewed on Pcmag. In many cases, two laptops belong to the same model but differ by their serial numbers, which results in subtle differences in their specifications.
2) Although we could match laptops approximately by ignoring the differences in their serial number, we fear that this would introduce potential complications in future project stages.

Two smaller datasets, *track_sample.csv* and *songs_sample.csv*, are producted by downsampling the full datasets. They contain ***8592*** and ***10000*** tuples, respectively. ***1119*** tuple pairs survived after blocking and were stored in *pairs_passed.csv*. ***400*** tuple pairs were sampled and labelled (*labelled_data.csv*) for use in the matching step. This dataset was further split into a training set, (*train.csv*) which contains ***200*** tuple pairs, and a test set (*test.csv*), which contains ***200*** tuple pairs. All datasets generated in this stage can be found at Github link

II. Code

Code for data preprocessing, blocking and labelling can be found in *blocking.ipynb*. Code for matcher training, debugging and testing can be found in *matching.ipynb*.

III. Data Preprocessing (1 hr) and Blocking (4 hrs)

We first changed the entire tables to lowercase. This avoided case-sensitive string matching, which is not desired in our setting. Downsampling was performed as suggested. After trying out different rules, we finally settled down with the following three rules which were applied in a sequence to remove tuple pairs that are unlikely to match.

1) ***Pairs whose song names have a jaccard score (3-grams) lower than 0.1 were removed.*** These pairs have very different song names and therefore are unlikely to match. This step dramatically reduce the number of candidate pairs. The feature used in this rule was automatically generated.

2) ***Pairs whose song names have less than 25% common words were removed.*** This borrowed the same idea from 1) but examined the song names at word level. This rule, *title_function(x, y)*, was hand-constructed.

3) ***Pairs whose artists have less than 25% common words were removed.*** This is rooted in the observation that tuples in one table usually contain more than one artists and two tuples usually match when they share one artist. This rule, *artists_function(x, y)*, was hand-constructed.

By using the debugging tool supplied by Magellan, we examined the tuple pairs that had been blocked and confirmed that very few actual matches had been wrongly removed. In addtion, we confirmed by examining the tuple pairs that survived blocking that we ended up with a reasonale number of positive examples and negative examples. Therefore, we concluded that our blocker is good enough.tv

IV. Data Labelling (1 hr)

We labelled the data based on the following criteria.

if two tuples has the same song name (excluding version information)
    if they share common artists ***MATCH***
    else ***MISMATCH***
else ***MISMATCH***

This simple rule is well justified. In many applications, we would simply want to know how popular a song is regardless of its different versions. In this case, variants of the same song should have the same name and share common artists who created the song initially. When a tuple's song name or artists is missing, we assume that it does match with any tuple. This decision is reasonable because there are not many tuples that contain missing values, and often times it is the artist information that is missing. Given the fact that many different songs have the same name but different artists, we are not confident enough to conclude that two tuples match without seeing both of their artist information.

V. Tuple Matching (10 hrs)

The labelled data was split into a development set *I* and a training set *J*, each containing 200 tuple pairs. Using automatic feature generation in *py_entitymatching*, set of features *F* is generated. We removed features that take '*id'* as parameter from *F* as it does not contribute effectively to decide the matching between the tuple pairs in *A* and *B*. Since there were many features in *F* that match the tables based on *year* parameter, we retained the feature that calculates the *level similarity* score between the *release year of the movie in A (Tracks table) and release year of the song in B (Songs table)* and dropped others. Next, *I* was converted into a set *H* of feature vectors using features in *F*. Missing values in *H* was changed to 0. We performed cross-validation on *H* for matchers Decision Tree, Random Forest, SVM, Logistic Regression, Linear Regression and Naïve Bayes. Below is the reported Precision, Recall and F1 of the matchers:

Matcher	**Precision**	**Recall**	**F1**
Decision Tree	0.926108	0.888781	0.905739
Random Forest	***0.953444***	***0.932410***	***0.941354***
SVM	0.762289	0.992308	0.857984
Logistic Regression	0.929450	1.000000	0.962912
Linear Regression	0.939540	0.983612	0.959953
Naïve Bayes	0.938986	0.954917	0.946326

Random Forest (*X*) was selected after the cross-validation as it gave the best score of precision.

To improve the accuracy of *X*, we debugged it. *H* was split into sets *P* and *Q* with *train_proportion* = 0.5. On examining the false positives, **2 of the 3 false positives generated were due to an exact match in either *song_title* or *artists* between the tuples**. Since there were multiple features in *F* that match the tables based on the parameters *song_title* and *artists*, we tried with **different subsets of the feature set involving these parameters to remove redundancy and improve its accuracy**. The features *song_title_song_title_jac_qgm_3_qgm_3, song_title_song_title_jac_dlm_dc0_dlm_dc0 and song_title_song_title_lev_sim* seem to give a better representation of the matching between the tuples, hence only these features based on *song_title* parameter was retained (**Iteration 1**). Following this, only feature *song_title_song_title_jac_qgm_3_qgm_3,* the best representor was retained (**Iteration 2**). Next, on exploring the features based on parameter *artists,* since *artists_artists_jac_qgm_3_qgm_3* and *artists_artists_lev_sim* best represent the matching score, only these were retained (**Iteration 3**). Further, only *artists_artists_jac_qgm_3_qgm_3,* the best representor was retained (**Iteration 4**). Since we noted that many false positives were generated due to an exact match in either *song_title* or *artists* between the tuples, a feature that calculates the product of the jaccard score between the *song_title* and *artists* was added to the reduced feature set

obtained in the previous step (**Iteration 5**) In each debugging iteration, the accuracy of the matcher *X* with new feature set was examined. The above steps didn't contribute to an increase in accuracy of *X*. Below are the scores of the discussed debugging iterations:

Debugging Iteration	Precision	Recall	F1
1	83.05%	94.23%	88.29%
2	87.72%	96.15%	91.74%
3	84.75%	96.15%	90.09%
4	75.36%	100.0%	85.95%
5	82.26%	98.08%	89.74%

Since the score of precision dropped in all cases, we proceed with matcher X as the best matcher, i.e. **the final match *Y* is the same as *X***. Next, we converted the test set *J* into a set *L* of feature vectors, filled in the missing values and applied *Y* to *L*. The scores of *all methods (including Y)* are reported below:

Learning Method	Precision	Recall	F1
Decision Tree	92.98%	92.17%	92.58%
Random Forest	***94.55%***	***90.43%***	***92.44%***
SVM	73.86%	98.26%	84.33%
Logistic Regression	95.54%	93.04%	94.27%
Linear Regression	94.96%	98.26%	96.58%
Naïve Bayes	95.61%	94.78%	95.20%

The match Y produces reasonably high scores of precision, recall and F1 as shown in the table above, though Logistic Regression, Linear Regression an Naïve Bayes slightly outperform *Y* on the test set.

VI. Discussion

Due to the structure of the datasets, a good matcher was obtained without too much difficulty. However, there are a few things that can be taken into account if this matcher needs to be further improved. For example, ***some tuples, although rarely, contain information misplaced in some attributes***, i.e., song name appear in the "artists" attribute, and vice versa. To deal with this, we can also compute similarity score between song names and artists and examine those that achieve a unexpectedly high score. If the information is swapped, we correct it and claim that it is a match. Additionally, ***extra knowledge may be exploited to impute the missing values to aviod mistakenly throwing away actual matches***. For example, when the artist information is missing, we can take advantage of the song name and the year to find the most probable artists.

VII. Comments on Magellan

Positives:

The documentation of Magellan is thorough and very clear. The step by step guide to using Magellan and the Jupyter Notebook examples made it easy for us to work with it.

Improvement Points:

(1) A detailed documentation on the explanation of the features automatically generated would have helped us understand and debug the matchers better.

(2) Availability of debuggers for all learning-based matchers (other than Decision Tree and Random Forest) would be very helpful.
(3) In the matching step, combining multiple features (such as multiplication of multiple feature values) as a new feature could be supported.