How-To Guide to Entity Matching

January 24, 2017

This short document is a preliminary how-to guide to entity matching. A full-fledged detailed how-to guide will be published in the March-June 2017 time frame.

1 Entity Matching

Entity matching (EM) decides if disparate data pieces refer to the sam real-world entity. Many EM scenarios exist. In this document we will only focus on the scenario of matching tuples between two tables. Specifically, given two tables A and B, we want to find all pairs of tuples (a, b), where tuple a is from A and tuple b is from B and the two tuples refer to the same real-world entity. Figure 1 shows an example of such EM scenarios.

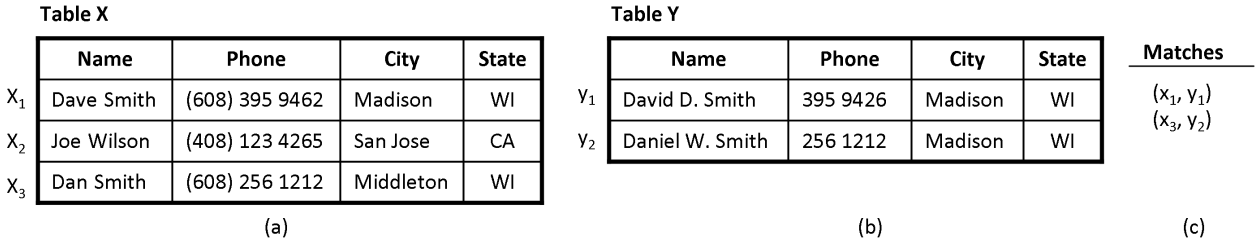

Table X

	Name	Phone	City	State
x_1	Dave Smith	(608) 395 9462	Madison	WI
x_2	Joe Wilson	(408) 123 4265	San Jose	CA
x_3	Dan Smith	(608) 256 1212	Middleton	WI

(a)

Table Y

	Name	Phone	City	State
y_1	David D. Smith	395 9426	Madison	WI
y_2	Daniel W. Smith	256 1212	Madison	WI

(b)

Matches
(x_1, y_1)
(x_3, y_2)

(c)

Figure 1: An example of entity matching: given the two tables in (a) and (b), we want to find all tuple pairs across the two tables that refer to the same real-world person; these pairs are called *matches* and are shown in (c).

Note that if we want to find matches within a single table, we can also view it as finding matches between two (identical) tables.

2 The Process of Performing Entity Matching

In practice, EM is typically carried out in three stages:

- Requirement analysis: In this stage, we discuss with the business owner to understand the requirements for EM. Examples include using rule-based EM for understandability, having precision at least 95% and recall at least 80%, etc.
- Development: Recall that we want to match two tables A and B. In this stage, we will experiment with data samples to develop an accurate EM workflow, one that satisfies the requirements in the previous stage.

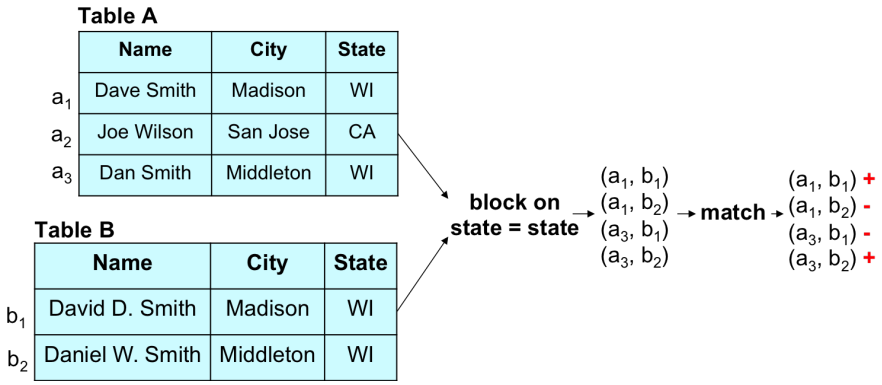

Figure 2: The blocking and matching steps of an EM worklow.

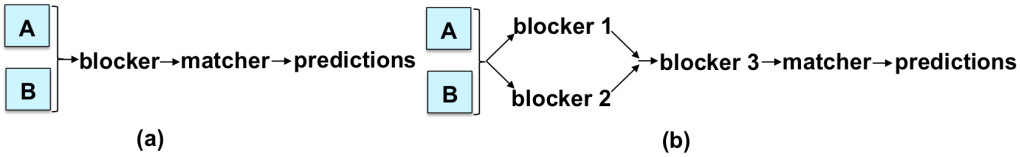

Figure 3: Example workflows considered in this document.

- Production: Once the workflow has been developed, in this stage we will put it in production, by running it on the entirety of data. The focus here is scalability, quality monitoring, crash recovery, and logging, among others.

In the current document, we will focus only on the development stage, deferring the production stage to a later time.

Blocking and Matching: Before discussing the development stage, we discuss two fundamental steps of a typical EM workflow: blocking and matching. Recall that our goal is to match two tables A and B. In practice, these tables can be quite large, such as having 100K tuples each, resulting in 10 billions tuple pairs across A and B. Trying to match all of these pairs is clearly very expensive. Thus, in such cases the user often employs domain heuristics to quickly remove obviously non-matched pairs, in a step called blocking, before matching the remaining pairs, in a step called matching.

Figure 2 illustrates the above two fundamental steps. Suppose that we are matching the two tables A and B in (a), where each tuple describes a person. The blocking step can use a heuristic such as if two tuples do not agree on state, then they cannot refer to the same person to quickly remove all such tuple pairs (this is typically done using indexes, so the blocking step does not have to enumerate all tuple pairs between A and B). In other words, the blocking step retains only the four tuple pairs that agree on state, as shown in (b). The matching step in (c) then considers only these tuple pairs and predicts for each of them a label match or not-match (shown as + and - in the figure).

Supported Workflows: For now we will only focus on EM workflows that consist of a blocking step followed by a matching step. Specifically, we assume the EM package provides a set of blockers and a set of matchers (and the user can easily write his or her own blocker/matcher). Given two tables A and B to be matched, the user applies a blocker to the two tables to obtain a set of tuple pairs, then applies a matcher to these pairs to predict "match" or "no-match". The user can use multiple blockers in the blocking step, and can combine them in flexible ways. Figure 3 illustrates both cases.

Further, we will only consider learning-based matchers for now. Specifically, we require the user to label a set of tuple pairs (as match or no-match), then use the labeled data to train matchers. In the future,

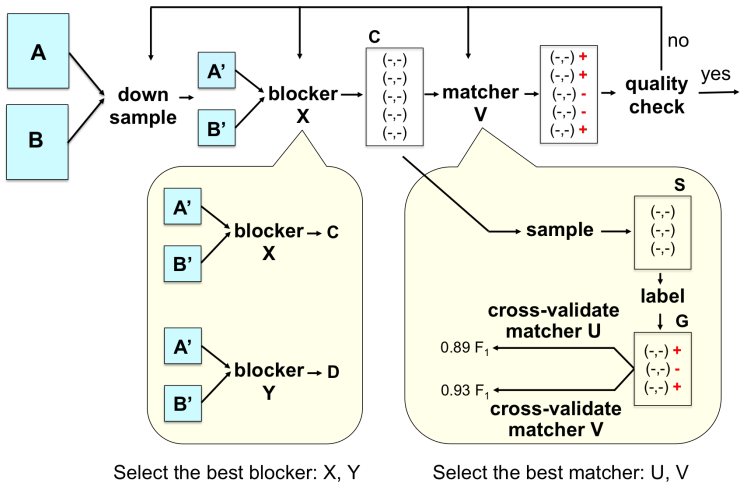

Figure 4: Steps of the development stage supported in this document.

we will consider more powerful EM workflows, such as using multiple matchers, including rule-based ones, or being able to add rules to process the output of the matchers.

The Development Stage: We now can describe the development stage that we support in detail. Figure 4 illustrates the development stage. In the figure, suppose we want to match two tables A and B, each having 1 million tuples. Trying to explore and discover an accurate workflow using these two tables would be too time consuming, because they are too big. Hence, the user will first "down sample" the two tables to obtain two smaller versions, shown as Tables A' and B' in the figure, each having 100K tuples, say (see the figure).

Next, suppose the EM package provides two blockers X and Y. Then the user will experiment with these blockers (for example, executing both on Tables A' and B' and examining their output) to select the blocker judged the best (according to some criterion). Suppose the user selects blocker X. Then next, he or she executes X on Tables A' and B' to obtain a set of candidate tuple pairs C.

Next, the user takes a sample S from C, and labels the pairs in S as "match" or "no-match" (see the figure). Let the labeled set be G, and suppose the package provides two matchers U and V. Suppose further that U and V are learning-based matchers (for example, one uses decision trees and the other uses logistic regression). Then in the next step, the user will use the labeled set G to perform cross validation for U and V. Suppose V produces higher matching accuracy (such as F1 score of 0.93, see the figure). Then the user will select V as the matcher, then apply V to the set C to predict "match" or "no-match", shown as "+" or "-" in the figure. Finally, the user may perform quality check (by examining a sample of the predictions), then go back and debug and modify the previous steps as appropriate. This continues until the user is satisfied with the accuracy of the EM workflow.

Once the user has been satisfied with the EM workflow, the production stage begins. In this stage the user will execute the discovered workflow on the original tables A and B. Since these tables are very large, scaling is a major concern (and is typically solved using Hadoop or Spark). Other concerns include quality monitoring, exception handling, crash recovery, etc.

3 Guide for the Development Stage

We now discuss how to perform each step in the development stage. In what follows we will use "we", "you", and "the user" interchangeably, when there is no ambiguity.

3.1 Down Sampling

If the two tables A and B are too large, then as discussed earlier, it will be difficult to operate directly on them to find the most accurate EM workflow. So we will have to down sample them into two smaller tables A' and B'. For now, a reasonable target is to down sample to tables of size 100K tuples.

Random sampling however does not work, because tables A' and B' may end up sharing very few matches, i.e., matching tuples (especially if the number of matches between A and B is small to begin with). Thus we need a tool that samples more intelligently, to ensure a reasonable number of matches between A' and B'.

We have developed such a tool, implemented as a command in the py_entitymatching package. This command first randomly selects B_size tuples from table B to be table B'. For each tuple $x \in B'$, it finds a set P of $k/2$ tuples from A that may match x (using the heuristic that if a tuple in A shares many tokens with x, then it is more likely to match x), and a set Q of $k/2$ tuples randomly selected from $A \setminus P$. Table A' will consist of all tuples in such Ps and Qs. The idea is for A' and B' to share some matches yet be as representative of A and B as possible.

3.2 Exploring and Understanding the Tables

Next, we may want to explore and understand the tuples in the tables. A few things to look for:

- For each attribute, check to see if it has a lot of missing values. If it does, then it will cause problems later in the blocking and matching stages (we will discuss this a bit later).
- Check also the type of the attribute. This type can be string, textual (that is, long string), numeric, categorical, boolean. Check also to see if the values of this attribute should conform to any format (e.g., dates typically follow a format).
- Check to see how "dirty" the attribute is. For example, an atribute can be categorial, but the same value appears in many different variations (e.g., the values of attribute City are "San Francisco", "SF", "City of San Francisco", "San Franc.", etc.). In such cases, if we do blocking later on the attribute, blocking will not work very well unless we have cleaned the values of this attribute.

Currently there is no good tools to explore and undersand the tables. Commands in the pandas package seem to be the best bet here.

3.3 Exploring and Understanding the Notion of Match

Our goal is to find tuples across two tables that match. Interesting, in many real-world EM scenarios, we often start with very little understanding of what it means to be a match, and trying to understand this is often far more complicated than we originally think.

To illustrate, suppose we want to match restaurants across two tables. What do we mean by matching restaurants? If two restaurants have the same names and addresses, it seems natural that they should match. But what about restaurants that have the same names but different addresses, such as different branche locations of Chipotle, would they still match?

This lack of a thorough understanding of the notion of match has serious implications. For example, we may do blocking based on a wrong notion of match, or label data based on the wrong notion, and then need to redo the work again.

As a result, at the start of the EM process, it is important to take some time to explore the data and try to understand what different notions of match can be defined, and which one we should use (this is tricky

because it requires us to go back to the business owner and have a discussion; often the business owner has not even thought or is aware of this, and so this can take a while).

There is currently no good tools to explore different notions of match. You can do one of the three things (or possibly all of the three):

- Have a brain storming with the business owner. Try to think about all possible match scenarios and ask him/her to clarify whether each scenario is considered a match.
- Select a typical-looking tuple in Table A', try to find all similarly looking tuples in Table B'. Now consider if each such pair should be a match. Carefully note down all possible notions of match, for later discussion with the business owner.
- A way to partially automate the above process is to use the command to debug the blocking process in the package py_entitymatching. This command can take two tables A' and B' and returns a set of tuple pairs judged to be a match between the two, based on Jaccard similarities between the tuples. You can then examine these tuple pairs and note possible definitions of match.

It is critical that you perform the above exercise and have a clear definition of match before you proceed further.

3.4 Blocking

Next, you will perform blocking on the tables A' and B' (the two smaller tables obtained from downsampling the original tables). Specifically, suppose the EM package provides two blockers K and L. Then the user will experiment with these blockers (for example, executing both on Tables A' and B' and examining their output) to select the blocker judged the best (according to some criterion). Suppose the user selects blocker L. Then next, he or she executes L on Tables A' and B' to obtain a set of candidate tuple pairs C. We now elaborate on this process.

Selecting the Best Blocker: Many blocking solutions have been developed, e.g., overlap, attribute equivalence (AE), sorted neighborhood (SNB), hash-based, rule-based, etc. From our experience, we recommend that the user try successively more complex blockers, and stop when the number of the tuple pairs surviving blocking is already sufficiently small. Specifically, the user can try overlap blocking first (e.g., "matching tuples must share at least k tokens in an attribute x"), then AE (e.g., "matching tuples must share the same value for an attribute y"). These blockers are very fast, and can significantly cut down on the number of candidate tuple pairs. Next, the user can try other well-known blocking methods (e.g., SNB, hash) if appropriate. This means the user can use multiple blockers and combine them in a flexible fashion (e.g., applying AE to the output of overlap blocking).

Finally, if the user still wants to reduce the number of candidate tuple pairs further, then he or she can try rule-based blocking.

Debugging Blockers: Given a blocker L, how do we know if it does not remove too many matches? We have developed a debugger to answer this question, implemented as a command in py_entitymatching. Suppose applying L to A' and B' produces a set C of tuple pairs $(a \in A', b \in B')$. Then $D = A' \times B' \setminus C$ is the set of all tuple pairs removed by L.

The debugger examines D to return a list of k tuple pairs in D that are most likely to match ($k = 200$ is the default). The user U examines this list. If U finds many matches in the list, then that means blocker L has removed too many matches. U would need to modify L to be less "aggressive", then apply the debugger again. Eventually if U finds no or very few matches in the list, U can assume that L has removed no or very few matches, and thus is good enough.

Knowing When to Stop Modifying the Blockers: How do we know when to stop tuning a blocker L? Suppose applying L to A' and B' produces the set of tuple pairs $block(L, A', B')$. The conventional wisdom is to stop when $block(L, A', B')$ fits into memory or is already small enough so that the matching step can process it efficiently.

In practice, however, this often does not work. For example, since we work with A' and B', *samples* from the original tables, monitoring the size of $block(L, A', B')$ does not make sense. Instead, we want to monitor the size of $block(L, A, B)$. But applying L to the large tables A and B can be very time consuming, making the iterative process of tuning L impractical. Further, in many practical scenarios (e.g., e-commerce), the data to be matched can arrive in batches, over weeks, rendering moot the question of estimating the size of $block(L, A, B)$.

As a result, in many practical settings users want blockers that have (1) high pruning power, i.e., maximizing $1 - |block(L, A', B')|/|A' \times B'|$, and (2) high recall, i.e., maximizing the ratio of the number of matches in $block(L, A', B')$ divided by the number of matches in $A' \times B'$.

Users can measure the pruning power, but so far they have had no way to estimate recall. This is where our debugger comes in. The user can use our debugger to find matches that the blocker L had removed, and when he/she finds no or only a few matches, he/she can conclude that L had achieved high recall and stopped tuning the blocker.

3.5 Sampling and Labeling Tuple Pairs

Let L be the blocker we have created. Suppose applying L to tables A' and B' produces a set of tuple pairs C. In the next step, user U should take a sample S from C, then label the pairs in S as matched / no-matched, to be used later for training matchers, among others.

At a first glance, this step seems very simple: why not just take a random sample and label it? Unfortunately in practice this is far more complicated.

For example, suppose C contains relatively few matches (either because there are few matches between A' and B', or because blocking was too liberal, resulting in a large C). Then a random sample S from C may contain no or few matches. But the user U often does not recognize this until U has labeled most of the pairs in S. This is a waste of U's time and can be quite serious in cases where labeling is time consuming or requires expensive domain experts (e.g., labeling drug pairs). Taking another random sample does not solve the problem because it is likely to also contain no or few matches.

To address this problem, we propose that user U sample and label in iterations. Specifically, suppose U wants a sample S of size n. In the first iteration, U takes and labels a random sample S_1 of size k from C, where k is a small number. If there are enough matches in S_1, then U can conclude that the "density" of matches in C is high, and just randomly sample $n - k$ more pairs from C.

Otherwise, the "density" of matches in C is low. So U must re-do the blocking step, perhaps by creating new blocking rules that remove more non-matching tuple pairs in C, thereby increasing the density of matches in C. After blocking, U can take another random sample S_2 also of size k from C, then label S_2. If there are enough matches in S_2, then U can conclude that the density of matches in C has become high, and just randomly sample $n - 2k$ more pairs from C, and so on.

What Should Be the Sample Size? This is a tricky question. But there is a procedure to estimate the sample size, which we will describe in the next writeup of the how-to guide. For now, as a rule of thumb, it is reasonable to take a sample of size 300-500. Recall that we want to take this sample incrementally, in multiple iterations. In each iteration we recommend to take a smaller sample of size 50.

Distributed Labeling: The sample (of 300-500 pairs) is often distributed among a few people to label. A potential problem here is that if they do not all agree a priori on the definition of match, then collaborative labeling will create many problems, due to inconsistent labeling. Even if they have agreed a priori, they may

not have anticipated all possible match definitions, and if some of them improvise with the match definition along the way, then inconsistent labeling may still arise.

To avoid such problems, we recommend that initially, each person label just a small set of pairs, say 20. Then they should get together and compare notes on their match definitions and resolve any inconsistency. This will also give them a chance to discover if they have a skew problem (that is, too few positive examples), in which case they probably have to redo the blocking process to increase the density of positive examples.

3.6 Selecting a Matcher

Once user U has labeled a sample S, U uses this labeled set G to select a good initial learning-based matcher. Today most EM systems supply the user with a set of such matchers, e.g., decision tree, Naive Bayes, SVM, etc., but do not tell the user how to select a good one.

The py_entitymatching package addresses this problem. Specifically, user U first calls a command in py_entitymatching to create a set of features $F = \{f_1, \ldots, f_m\}$, where each feature f_i is a function that maps a tuple pair (a, b) into a value. This command creates all possible features between the attributes of tables A' and B', using a set of heuristics. For example, if attribute $name$ is textual, then the command creates feature $name\_3gram\_jac$ that returns the Jaccard score between the 3-gram sets of the two names (of tuples a and b).

Next, U splits the labeled set G into a development set I and evaluation set J.

Next, U converts each tuple pair in the development set I into a feature vector (using features in F), thus converting I into a set H of feature vectors.

Next, the user needs to fill in missing values if any in H, otherwise the learning algorithms of scikit-learn won't work (they won't work on missing values).

Let M be the set of all learning-based matchers supplied by the EM system. Next, U uses another command to perform cross validation on H for all matchers in M, then examines the results to select a good matcher. The above command highlights the matcher with the highest accuracy. However, if a matcher achieves just slightly lower accuracy (than the best one) but produces results that are easier to explain and debug (e.g., a decision tree), then the user may want to select that matcher instead.

3.7 Debugging a Matcher

Let the selected matcher be X. In the next step user U debugs X to improve its accuracy.

We recommend that user U debug in three steps: (1) identify and understand the matching mistakes made by X, (2) categorize these mistakes, and (3) take actions to fix common categories of mistakes.

Identifying and Understanding Matching Mistakes: U should split the feature vectors H of the development set I into two sets P and Q, train X on P then apply it to Q. Since U knows the labels of the pairs in Q, he or she knows the matching mistakes made by X in Q. These are *false positives* (non-matching pairs predicted matching) and *false negatives* (matching pairs predicted not). Addressing them helps improve precision and recall, respectively.

Next U should try to understand why X makes each mistake. For example, let $(a, b) \in Q$ be a pair labeled "matched" for which X has predicted "not matched". To understand why, U can start by using a debugger that explains how X comes to that prediction. For example, if X is a decision tree then py_entitymatching has a debugger which can show the path from the root of the tree to the leaf that (a, b) has traversed. Examining this path, as well as the pair (a, b) and its label, can reveal where things go wrong.

In general things can go wrong in four ways:

- The data can be dirty, e.g., the price value is incorrect.
- The label can be wrong, e.g., (a, b) should have been labeled "not matched".

- The feature set is problematic. A feature is misleading, or a new feature is desired, e.g., we need a new feature that extracts and compares the publishers.
- The learning algorithm employed by X is problematic, e.g., a parameter such as "maximal depth to be searched" is set to be too small.

Categorizing Matching Mistakes: After U has examined all or a large number of matching mistakes, he or she can categorize them, based on problems with data, label, feature, and the learning algorithm.

Handling Common Categories of Mistakes: Next U should try to fix common categories of mistakes by modifying the data, labels, set of features, and the learning algorithm. This part often involves data cleaning and extraction (IE), e.g., normalizing all values of attribute "affiliation", or extracting publishers from attribute "desc" then creating a new feature comparing the publishers.

Proxy Debugging: Suppose we need to debug a matcher X but there is no debugger for X, or the debugger exists but is not very informative. In this case X is effectively a "blackbox". To address this problem, we propose to train another matcher X' for which there is a debugger, then use that debugger to debug X', instead of X. This "proxy debugging" process cannot fix problems with the learning algorithm of X, but it can reveal problems with the data, labels, features, and fixing them can potentially improve the accuracy of X itself.

Selecting a Matcher Again: So far we have discussed selecting a good initial learning-based matcher X, then debugging X using the feature vectors H of the development set I. To debug, user U splits H into training set P and testing set Q, then identifies and fixes mistakes in Q. Note that this splitting of H into P and Q can be done multiple times. Subsequently, since the data, labels, and features may have changed, U would want to compute the feature vectors and do cross validation again to select a new "best matcher", and so on.

3.8 Selecting and Debugging a Matcher: A General Procedure

Based on the discussion above, we now describe a general procedure to select and debug matchers. For concreteness, we will assume that we are using the py_entitymatching package for this process. Further, we will assume that the goal is to create a matcher that achieves precision of at least 90%, and achieve recall as high as possible. In what follows we will often use the term "accuracy" as a shorthand to refer to precision, recall, and F-1.

This is an iterative process, and will take a while. We begin by describing the high-level overview of this process:

1. Split the labeled set G into a development set I and an evaluation set J.
2. Convert I into a set H of feature vectors (using the features in F).
3. Fill in missing values if any in H.
4. Suppose you have four learning-based classifiers that you can use to build learning-based matchers (e.g., Decision Tree, Random Forest, SVM, Naive Bayes). How do you know which one is the best? To answer this question, perform cross validation (CV) for all matchers on H and select the one judged to be the best. Let this one be X.
5. If X has reached desired accuracy (as measured on CV), then exit, using X as the matcher. Otherwise debug X. To do this, split the feature vectors H of the development set I into a training set U and a testing set V. Train X on the t raining set U, then apply X to the testing set V. Examine the false

positives and false negatives that X makes on the testing set V, and debug X accordingly. You can debug X as follows:

- The package py_entitymatching has a set of debuggers for classifiers, so these debuggers can be used to debug X.
- If you know how X works, then you can use your own knowledge to debug X.
- You can perform "proxy debugging", as described above. (Next writeup of the how-to guide will also describe "feature based debugging".)

Once you have exhausted things to do with the false positives and negatives on the testing set V, you may want to do another split of H into a new training set U and a new testing set V, then repeat the above debugging process. Do this as many times as you judge necessary.

During this step, occasionally you may want to measure X's accuracy, by performing CV for X on H. This will tell you if you have been able to improve X's accuracy on the set H.

6. Once X has been debugged, the data/label/features may have changed. So you will repeat Steps 2-5. Specifically, you will perform CV again for all matchers, then select the best matcher, and so on. Eventually you stop when you have obtained the desired accuracy, run out of things to try, or out of time. Let the best learning-based matcher obtained at this point be Y.

7. Eventually you stop when you have reached the desired accuracy, or run out of things to try, or out of time. Now, train Y on the set H, then convert J into a set L of feature vectors, next fill in missing values if any in L, then apply Y to the set L and report the accuracy of Y on the set L.

We now describe these steps in detail.

1. Creating Development and Evaluation Sets: Randomly split the set G into a development set I and an evaluation set J. Convert I into a set H of feature vectors and then fill in the missing values if any in H. The idea is that we will do the development to find the best matcher on I, and we will report its precision and recall on J. J will be a data set that we don't ever look at, we just use it to compute precision and recall.

2. Selecting the Best Learning-Based Matcher: The current py_entitymatching package provides the following learning-based algorithms:

- Decision Tree
- Random Forest
- Support Vector Machine
- Naive Bayes
- Logistic Regression
- Linear Regression

For each algorithm, create a corresponding matcher.

Once you have created these matchers, perform CV for all of them on the set H. Then use the CV result to select a best matcher. Let this matcher be X. (Note that the CV command will select a matcher that it judges to be the best. This is not necessarily the matcher that you want. You may have your own criteria for selecting a matcher judged to be the best. So ignore the best matcher reported by the CV command.)

3. Debugging the Selected Learning-Based Matcher: If X has reached the desired accuracy (as computed by doing CV on H) then exit, reporting X as the best matcher found. Otherwise you need to debug X. To do so, first split the feature vectors H of the development set I into a training set U and a testing set V. Many splits are possible. For example, you can split 50-50.

Next, use U and V to debug X. You can debug in three steps as follows:

1. py_entitymatching has debugger built for Decision Tree and Random Forest. So if X uses one of these learning methods (i.e., X is a Decision Tree based matcher or a Random Forest based matcher), you can use the debugger on U and V to debug X.

2. If there is no debugger in py_entitymatching for X, or if you have used the debugger in py_entitymatching, and you are still unsatisfied with it, then you move to this step. In this step if you know how X works, then you can use that knowledge to try to debug X. For example, if X is SVM and you know how SVM works, then you can try to use that knowledge to debug X.

3. If you have done Steps 1-2 and you are still not satisfied (or if you can't do Steps 1-2, for example, because X is SVM, so there is no debugger for it and you also don't know how SVM works), then you can do "proxy debugging" (or "feature based debugging" which will be described in the next version of the write-up).

Note that by "debugging", it pretty much means you identify a problem and then try to debug the problem. In this context, a problem will mean a false positive or a false negative. So you should examine the false positives and false negatives.

The basic idea is that you train X on U then apply it to V. Since you know the labels on V, you can easily identify the false positives and negatives on V. You should think about what causes these false positives and negatives and what you can do to fix the problems. Typical actions include: checking if the label is incorrect, is there any problem with the tuples (maybe some of their values are incorrect, e.g., the length is mistakenly put into the place of the width)? Maybe the thresholds being set are too high, maybe a new feature has to be added, and so on.

4. Repeating the Steps of Selecting the Best Learning-based Matcher: Eventually you either (a) run out of things to debug, or (b) get tired, or (c) found the accuracy to be acceptable. At this point you might have changed the data (e.g., cleaning it up), the labels, the features, and other stuff. So it may be that X is no longer the best learning-based matcher.

So you should repeat the above two steps. That is, first redo cross validation on H to select the best learning-based matcher. Let this matcher be Y. If this matcher is still X (i.e., this cross validation step selects X again as the best matcher), then you can stop. Otherwise, you may want to debug Y. And so on.

5. Computing the Accuracy of the Best Matcher: At this point you will have a matcher Y. Now train Y on H, then convert J into set L of feature vectors, next fill in missing values if any in L, then apply the trained matcher Y to J and report the precision and recall on L. Remember that L is the feature vectors of the evaluation set J and that you have NOT touched J at all until now.

Note: As described, your goal is to use the set I to develop a good matcher (which consists of a learning-based matcher and a set of rules on top). During the development process, you may wonder what is the accuracy of a particular matcher M. To measure this accuracy, perform CV for M on the set feature vectors H got from the development set I.