Entity Matching: Matching Entities Between Multiple Data Sources

Ivan Bilan

Learning Objectives
- Illustrate the steps required to build an entity matching pipeline
- Explain how entity matching can be applied in the tourism industry
- Demonstrate how to engineer an end-to-end entity matching pipeline using Python

1 Introduction and Theoretical Foundations

1.1 Entity Matching Problem Statement

Entity matching describes the approach of finding records that refer to the same real-world entity across different databases or any other data storage types. These entities are identified by cross-checking their identifiers, such as name, address, phone number, and the like. Entity matching, also known as record linkage, has applications in various scientific fields and industrial solutions, ranging from matching people in census data (Christen, 2012), bibliographic databases (Christen, 2012), forensic data and DNA matching (Tai, 2018), physical objects like businesses, and many more. It is mainly used to consolidate records of the same type and especially used with textual data. For example, when matching company entities from two databases, they can be matched by name and address. Additionally, entity matching is often used in the process of finding duplicates within the same database.

The challenge of entity matching arises mainly from there being no cross-industry standard on how to store such entities and their identifiable markers in a consistent

I. Bilan (✉)
TrustYou GmbH, Munich, Germany

R. Egger (ed.), *Applied Data Science in Tourism*, Tourism on the Verge,
https://doi.org/10.1007/978-3-030-88389-8_19

way. The simple example of matching the same company entity from two databases becomes challenging if these entities are stored in different formats. For example, in one case, the company may be stored in the database using just two columns, one for the full name and legal entity type and another one for the full address, whereas, in the second database, the company entity might have separate columns for the name, legal entity, city, street, building number, and so on. Working on an entity matching solution requires a significant amount of time to invest in generating quality training data that, for a subset of records, includes the match status of whether the items in a record pair belong to a single real-world entity or not (Christen, 2012).

1.2 Entity Matching Examples in the Travel Industry

Entity matching is also widely used in the tourism industry, and there are various scenarios in which a company needs to rely on entity matching approaches. One of the most used applications involves the deduplication of hotel reviews that are crawled from various sources. Another widespread example is matching hotels or accommodations between various sources or deduplicating them within one source (Bayrak et al., 2019; Kozhevnikov & Gorovoy, 2016). One scenario relating to this might be to extend a database when onboarding a new hotel chain. Even if the company already has hotels in their database, they might not have all of the hotels the clients want to use for their solution. As such, the company would need to align its database to the database of their clients in order to find the correct data of a specific hotel and deliver their analytic insights to the correct hotel.

Another typical scenario is the need of consolidating all reviews for specific hotels from various websites that post traveler reviews. Often, differences can be observed in how the hotels are displayed on such websites. The names of the hotel can differ, especially if the websites are based in different countries, and addresses can also vary or display varying levels of detail. Some sources may display the phone number and the official website of the hotel, or, if the hotel belongs to a specific chain of hotels, these may sometimes simply point to the chain's headquarters instead.

Table 1 shows a general example of how two hotel records involving the same entity can be stored in various data sources. This particular example illustrates why entity matching goes beyond merely matching two databases via a direct comparison of database columns. One of the specific issues with this example is that the

Table 1 Example of different records from the same hotel entity

	Hotel name	Address	City	Street	Zip	Country
Data Source 1	Hotel Kaltbräu City	Tal 11, Lehel, München	–	–	–	Deutschland
Data Source 2	Hotel Kaltbraeu City	–	Munich	Tal 11	80331	Germany

Table 2 Example of different records from the same hotel entity in different languages

	Hotel name	Address	Phone number	Website
Data Source 1	サンシャイン	東京都豊島区東池袋2-3-4	0356411167	www.japanhotels.co.jp/hotelsunshine
Data Source 2	Sunshine	2-3-4 Higashi-Ikebukuro, Toshima-ku, Tokyo	+081 035-641-1167	japanhotels.co.jp

addresses are saved in very different formats. In the first example, the address is stored in a single address column, while in the second example, each address identifier, such as city, street, or zip code, is given separately. There is also a small difference in how the name is normalized in each data source, with the second data source normalizing German umlauts by using the English alphabet. Such a record cannot be matched directly by its name, and the address information is scattered between various database columns.

Table 2 shows an extreme case of the above in which two records from the same hotel entity are stored between two data sources with very different identifiers. The first data source shows the hotel's name and address in Japanese, while the other data source is written using the transliterated English form. Moreover, the phone numbers do not fully match since, in one of the sources, the number has the country-specific identifier attached at the beginning. Additionally, the websites also do not fully match.

More difficulties arise when dealing with a large number of records; for example, there might be numerous hotels at the same address or the names of the hotels might be very similar. Moreover, some data records may have incomplete data points when compared to others.

1.3 Overview of the Stages of an Entity Matching Approach

Entity matching approaches usually follow similar steps. The process usually starts with the (1) data pre-processing step followed by (2) pre-filtering potential candidate pairs, (3) record pair comparison, and, finally, (4) classification of each record pair as a true or negative match. This process is illustrated in Fig. 1.

Pre-processing data is necessary because the structure of the data and its general representation usually varies among data sources. Many steps can be taken to pre-process data. In the particular hotel examples provided above, the address data points can be stored uniformly between data sources by either splitting them into the smallest available identifiers or by joining them together into one address identifier. The data can also be normalized to a single language or a language-agnostic representation. For example, instead of using the name of the hotel in various languages, it can be transliterated into the English alphabet. For most languages,

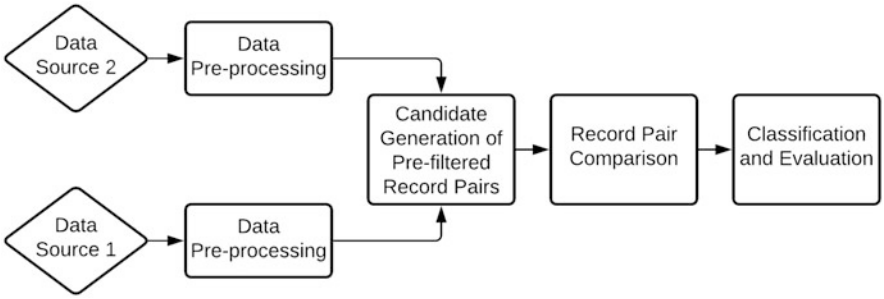

Fig. 1 General steps of an entity-matching pipeline

textual identifiers can be normalized to English with the help of various Unicode normalization techniques.[1] Other specialized tools dedicated to such transliterations, for instance, jaconv[2] for Japanese, also exist. Additionally, the transliterated form can be transformed into a phonetic representation that would only reflect how the text is pronounced and not necessarily how it is written. This is possible with such phonetic algorithms as Soundex[3] (a detailed overview of it and many other similar algorithms is given by Christen (2012)). Moreover, as the type of hotel or another part of the address often end up in the hotel name field of various record representations, names can be cleaned up by removing general words in the name field that do not necessarily belong to the actual name.

Before comparing records, the next step is to pre-filter potential matching record pairs. This step can technically be omitted if the amount of records in need of comparison is very small. However, when comparing large databases, computational limitations may be reached fairly quickly. If the record pairs were not pre-filtered, the entity matching pipeline will have to compare every single record from the first source to each record in the second source and then repeat the process for each individual record in source one. This approach has quadratic computational complexity (Christen, 2012); hence, the pre-filtering step is recommended for production-ready solutions. The pre-filtering of potential matching record pairs is also known in the research literature as indexing, blocking (Kirsten et al., 2010), or candidate generation (Kong et al., 2016). In the end, the ultimate goal of this step is to find a way to limit the number of comparisons needed for each record. Continuing with the hotel example from above, a fast way to do blocking is to only compare records from the first source to records in the second source that are within the same city or on the same street. This will largely alleviate the computational complexity of the task.

After the blocking step, each record pair needs to be compared. Even after extensive pre-processing, many differences between record identifiers may still

[1] *Unicode Normalization Forms* https://unicode.org/reports/tr15/

[2] *Japanese character interconverter* https://github.com/ikegami-yukino/jaconv

[3] *The Soundex Indexing System* https://www.archives.gov/research/census/soundex

remain; in other words, relying on one of the identifiers between the sources to be an exact match or not is often not enough. As a result, a measure of how similar the identifiers are to each other is required. There are various approaches to conduct such a comparison, and it usually depends on the type of identifier. When comparing names or addresses, various string similarity algorithms can be used on the text itself, or such a comparison can be done using word embedding vector representations of the compared identifiers. Both approaches can be combined as well.

There are many approaches available to compute string similarity between two textual data points, such as Levenshtein Edit Distance (Hyyrö, 2003), Jaro Winkler (Keil, 2019), Hamming, Monge-Elkan string comparison algorithms (Cohen et al., 2003), and many more. Christen (2012) gives a detailed overview of various string comparison algorithms and how they can be applied to record pair comparison. The ultimate use-case of these algorithms for the entity matching task is to provide a numeric similarity score between two texts. For example, when comparing different written variants of hotel names in two data sources, such as "Batu Faly Shamrock Beach 22" and "Batu Faly Shamrock Villa," such hotel names would not be matched to the same entity when using a one-to-one comparison. Instead, for the latter example, one of the string comparison algorithms could be applied, for instance, Hamming Similarity, which would output a similarity score of 0.70 (on a scale of 0 to 1).

After the pipeline compares each record to its potential match candidate and also each of the record's identifiers (name, address, phone, etc.) to its mirror identifiers from the compared record in the second source, a classification decision needs to be made on whether a record pair is indeed a match or not. In this regard, mainly two approaches to record pair classification can be exemplified: a threshold-based approach and an approach using an end-to-end neural network classifier.

The threshold-based approach is a simple way of applying entity matching and, if necessary, can be used without a training set. It entails computing a similarity score for each record identifier, summing it up, and deciding whether a record pair is a match if the sum exceeds a set threshold value (Christen, 2012). For example, given two record identifiers in both databases, name and address, each of them can be compared, providing a maximum similarity score of 1 for each identifier, which adds up to 2 if both identifiers are exactly the same. If the threshold of the sum of all similarity scores is set to 1.5, only the record pairs that have a joint similarity score at least that high will be matched. Another example: if the address is the same and yields a similarity score of 1, but the name only yields a similarity score of 0.5, the record pairs will be matched as referring to the same entity because of the threshold. This grows in complexity with more identifier fields, and setting a correct threshold is a matter of experimenting with the quality evaluation of the output of such classifiers. This method can also be applied without having a pre-annotated gold standard. Furthermore, it can be used to help build a new training set much faster than having to annotate data from scratch, without any additional indications of the probability of a record pair being a match.

There are various alternatives to threshold-based classification, one of which is the neural network-based approach. Such approaches can work end-to-end without

defining any specific similarity scores or thresholds. However, they often require a considerable amount of annotated training data. A neural network-based entity matching system typically uses the contextual representation of record identifiers with learned vector embeddings, such as word2vec (Mikolov et al., 2013), fastText (Joulin et al., 2016), or BERT (Devlin et al., 2019) (see Chapter "Text Representations and Word Embeddings" for more details). In addition to the word embeddings, a neural classifier based on recurrent neural networks, a Transformer encoder (Vaswani et al., 2017), or similar neural approaches are used for classification. Zhao and He (2019) provide a more detailed overview of the variants and complex structures of such entity matching systems. In the next section, DeepMatcher, one of the most popular, open-source end-to-end neural entity matching frameworks will be presented (Mudgal et al., 2018).

2 Practical Demonstration

2.1 Data Formatting and Pre-processing

Let us take a look at a hands-on example of matching hotel entities between data sources. For this demo, we will use Python and various data analysis libraries like "pandas".[4] The sample dataset contains two different sources with 84 hotel records each. Around two-thirds of the dataset consists of record pairs each referring to a single hotel entity. One-third of the record pairs have very similar record identifiers, like name and address, however, they are not referring to the same hotel entity. All of the hotels provided in this chapter and the dataset are automatically generated and do not refer to real-world hotels.

Let us load the data from each source into a data frame:

```
import pandas as pd
df_source1 = pd.read_csv("./data/source_one.csv")
df_source2 = pd.read_csv("./data/source_two.csv")
```

Table 3 shows a sample of records from our first source, and Table 4 shows a data sample from our second source.

In this particular example, the first source has columns referring to name, address, and phone number, while the second source has name, city, country, zip, street, and phone number. To compare each record identifier separately, we need to make sure that the two datasets have the same columns. In this case, we have two options:

1. In the second source, combine each single address identifier into one address column for each record.

[4] https://pandas.pydata.org

Table 3 Data sample of hotel records from source 1

Hotel name	Address	Phone number
Valley Country Inn	433 East Route 77, Johnscity, AZ 87030, United States of America	922-456-1178
Comfort Inn Westcity	1821 Harrison Drive, Westcity, WY 81340, United States of America	

Table 4 Data sample of hotel records from source 2

Hotel name	Street	City	Zip code	Country	Hotel phone number
Valley Country Inn Bed & Breakfast	East Route 77433	Johnscity	87030	United States	+1 922-456-1178
Comfort Inn Westcity	Harrison Drive 1821	Westcity	81340	United States	+1 342-562-8865

2. In the first source, programmatically extract city, country, zip, and street name from the single address column.

The second option will allow us to be more precise in our comparisons and help us create better record candidate pairs. Fortunately, there are tools in the Python ecosystem that will allow us to do such data pre-processing. Using a Python package called "postal,"[5] we start by extracting specific address information from the first source. This particular Python library allows us to automatically parse a single string of address data into separate address units like city, street, country, zip code, etc. It has a simple interface, and with just one function, we can get the expected results. Let us look at a quick demonstration below.

```
from postal.parser import parse_address
print(parse_address("433 East Route 77, Johnscity, AZ 87030, United
States of America"))

Output: [(433, 'house_number'), (east route 77, 'road'), (Johnscity,
'city'), ('az', 'state'),
(87030, 'postcode'), ('united states of america', 'country')]
```

With this function, we can take the data from the first source and transform it to have the same columns as the data in the second source. Table 5 shows one record from the first source after having applied the address transformation step.

We still have multiple inconsistencies between data sources in terms of how each column is represented. When we investigate each column in more detail, we can see that the country names are inconsistent. For example, in one source, we can see "United States," while in the other "United States of America." Furthermore, the

[5] https://github.com/openvenues/pypostal

Table 5 Data sample of hotel records from source 1 after address transformation

Hotel name	Street	City	Zip code	Country	Hotel phone number
Valley Country Inn	433 east route 77 az	johnscity	87030	united states of america	922-456-1178

phone numbers in the first source have no country code, while the records in the second source have the country code attached to the phone number. The first issue can be easily resolved by using the "pycountry" Python library,[6] which allows one to look up a country name in almost any format and to normalize it to a predefined one. For example, we can look up the country names we have in our data and transform all of them into a uniform two-character country code using the following code snippet:

```
import pycountry
pycountry.countries.search_fuzzy("United States")[0].alpha_2
pycountry.countries.search_fuzzy("United States of America")[0].
alpha_2

Output: "US" for both
```

The issue with the phone number formatting can also be clarified by using another specialized Python library. With "python-phonenumbers,"[7] we can get a direct local national number:

```
import phonenumbers
phonenumbers.parse("+1 922-456-1178", None).national_number

Output: "9224561178"
```

Various other pre-processing steps can be applied to the data to make it more uniform throughout, and the approaches that need to be taken usually depend on the type of record identifier. After the pre-processing step is finalized, we need to pre-filter the potential candidate record pairs.

2.2 Candidate Generation

As discussed in the theoretical part, we do not want to compare each record from the first source to each record in the second source. To avoid this, we can use various approaches to candidate generation of potential match pairs. In this demo, we will

[6] https://pypi.org/project/pycountry/

[7] https://github.com/daviddrysdale/python-phonenumbers

make sure to only compare hotels within the same country. This should decrease the number of comparisons we need to compute considerably. In real-world applications, candidate generation is one of the most important steps for the entity matching algorithm. For the purpose of this demo, we are only generating candidates based on the country. However, there are many ways this step can be refined. Explore the various options and approaches to candidate generation by following the tutorials provided on the documentation page of the "py_entitymatching" Python library.[8]

We will first generate the candidate pairs based on the country code:

```
import py_entitymatching as em
# Instantiate the overlap blocker object
ob = em.OverlapBlocker()

# apply the candidate generation step based on a predefined column
match_candidate_pairs_df = ob.block_tables(df_source1, df_source2,
'country', 'country', word_level=True, overlap_size=1,
l_output_attrs=['name', 'phone', 'city', 'zip', 'street'],
r_output_attrs=['name', 'phone', 'city', 'zip', 'street'])
```

Based on a small dataset of 84 records in each source, by simply overlapping the two sources using the country code, we will need to do computations on more than 746 record pairs. This number grows quadratically with the dataset's size if no blocking is performed, making the selection of the right approach during candidate selection very important. We can filter out a few more record pairs by, for example, only allowing record pairs in which at least one word in the hotel name exists in both record pairs. This is controlled by defining a column on which blocking can occur and the size of the word overlap. In our example, this is set to 1:

```
strict_match_candidate_pairs_df = ob.block_candset
(match_candidate_pairs_df, 'name', 'name', word_level=True,
overlap_size=1, show_progress=False)
```

Now we have reduced the number of record pairs we need to compare to only around 262. There are many ways this number can be reduced even more; however, it all depends on which record identifiers are available. For example, if you have the longitude and latitude coordinates of the hotels, you can base your candidate generation on a radial distance around a specific hotel from one of the sources and only match it with other hotels within that given distance.

At this stage, both data sources are combined into one data frame including each potential candidate match pair (record identifiers from the first source and record identifiers from the second source) produced by the candidate generation step available for direct comparison. Table 6 illustrates what the dataset should look

[8] http://anhaidgroup.github.io/py_entitymatching/v0.3.x/user_manual/guides.html#stepwise-guides

Table 6 Data sample of hotel records following the candidate generation step

record_pair_id	ltable_name	ltable_phone	ltable_country	rtable_name	rtable_city	rtable_country
1	Valley Country Inn	9224561178	US	Valley Country Inn Bed & Breakfast	9224561178	US
2	Valley Country Inn	9224561178	US	Beach Country Inn	9223453468	US

like at this stage (the columns that start with "ltable_" come from the first source, while the "rtable_" identifiers are from the second source). Only a subset of record identifiers is shown in this example.

A data frame with the candidate pair from both sources provided in one row allows for within-row record identifier comparison.

2.3 Record Pair Comparison (Threshold-based)

Our next step is to compare each identifier in each potential record pair from the previous step and compute a similarity score for each column comparison between the first and second sources. We can use a binary comparison for simple columns, such as phone number and zip code, to see if they are a full match or not. For more complex columns, such as name, street, and city, we can use one of the available string comparison algorithms implemented in the "textdistance" Python library[9] or the "jellyfish" Python library.[10] In this particular demo, we will use the Hamming distance. However, you should experiment with various string similarity measures and see which one yields the best result for your particular data type. The following is an example of how to calculate the Hamming Distance for two strings:

```
import textdistance
textdistance.hamming.normalized_similarity('Batu Faly Shamrock
Beach 22', 'Batu Faly Shamrock Villa')

Output: 0.7037037037037037
```

The similarity score generates a value between 0 and 1 and indicates how similar the two strings are, with 1 denoting complete similarity. We have previously mentioned that we can compare zip codes and phone numbers in a binary fashion: assign 1 if they match entirely and 0 otherwise. However, matching the phone number or the zip code is less significant than matching a name or full street address. For this reason, we should boost the scores of each of these columns accordingly. For the demo, we will boost the name of the hotel by multiplying the final score by 3 and the address by 2. If the phone matches, we will assign a score of 1 to that record identifier comparison, and if the name matches only partially, we will multiply the score by 3, which in our example from above, will boost the similarity score to

[9] https://pypi.org/project/textdistance/
[10] https://pypi.org/project/jellyfish/

approximately 2.1. For the purpose of a production-ready system, the boosting weights should be computed using a more elaborate approach, for example, a machine learning approach that defines the best boosting values based on the accuracy of the final prediction on the whole gold standard.

Next, we need to sum up all the similarity scores and set a threshold at which we could regard a record pair as an actual match. If we compare name (maximum boosted similarity score of 3), street (maximum boosted similarity score of 2), city (maximum binary similarity score of 1), phone (maximum binary similarity score of 1), and zip code (maximum binary similarity score of 1), all of them add up to a maximum score of 8. For the demo, we will set the threshold for a record pair match at 5.5. The threshold has to be low enough to account for some records missing data in various columns but also high enough to reach a reasonable level of prediction accuracy. For the production-level solutions, make sure to compute a suitable threshold for your use case by doing additional experimentation on your data. Table 7 depicts a simplified illustration of what the data should look like at this stage of the entity matching process, namely, with columns storing the computed and boosted similarity scores for each record identifier in a separate column and the summed up score of all similarity scores for all identifiers.

After filtering out all the record pairs with a total similarity score lower than our set threshold, we get 84 record pairs predicted as matches. Now we also need to make sure that we only allow for a one-to-one comparison; in other words, each record from the first source can only be matched to one record from the second source. Such a limitation is not always applicable, however, as there might be multiple records of the same entity within the same source. As such, the final application depends on the type of data being matched.

Let us extract one-to-one predicted pairs and evaluate the quality based on an annotated gold standard. Our threshold-based classifier reaches 82% precision and 78% recall on our dataset. Using a simple threshold-based approach, we have already achieved significant precision and recall levels without having to rely on an extensive training corpus of thousands of annotated record pairs. This is one of the main advantages of such simple approaches. Next, we will explore a neural-based entity matching system called DeepMatcher.

Table 7 Data sample of hotel records with similarity scores

record_pair_id	ltable_name	ltable_phone	rtable_name	rtable_phone	name_similarity_score	phone_similarity_score	score_sum
1	Valley Country Inn	9224561178	Valley Country Inn Bed & Breakfast	9224561178	2.52	1	3.52
2	Valley Country Inn	9224561178	Beach Country Inn	9223453468	2.24	0	2.24

2.4 Record Pair Comparison (Neural-based)

A neural-based approach to entity matching has its pros and cons. It still requires the pre-processing and candidate generation steps. However, the computation of similarity scores between each record identifier is now a part of a fully automated process within the neural-based approach. The drawback of this approach is its dependency on annotated data with manually matched record pairs. This annotated data is required in order to train a model that will produce acceptable match classification results.

Compared to the threshold-based approach, the amount of code required is significantly lower as the system takes over most of the tasks related to similarity computation and classification. The focus shifts to selecting the right hyperparameters and classifier types for the data type at hand, which is achieved through extensive experimentation. Considering that pre-processing and candidate generation has already been done, it only takes a few lines of code to train and apply a DeepMatcher model:

```
import deepmatcher as dm

# load train, validation and test sets
train, validation, test = dm.data.process(
   path='./data/deep_matcher',
   train='train.csv',
   validation='validation.csv',
   test='test.csv')

# build an Entity Matching model
model = dm.MatchingModel(
       attr_summarizer=dm.attr_summarizers.Hybrid(
               word_contextualizer=dm.word_contextualizers.
               SelfAttention(heads=2)))

# run training with appropriate hyperparameters
model.run_train(train, validation, epochs=5, batch_size=8,
best_save_path='hybrid_model.pth',
pos_neg_ratio=2
)

# predict matches on the test set
model.run_eval(test)
```

You can experiment with different types of models, explanations of which are given in the tutorial page of DeepMatcher[11] as well as in a paper by Mudgal et al. (2018). For our demo, we are using the Self-Attention Transformer encoder with two attention heads, and we will train the model for five epochs. Additionally,

[11] https://github.com/anhaidgroup/deepmatcher/tree/master/examples

DeepMatcher allows for the fine-tuning of other various internal components, for example, how to tokenize the text input, what types of word embeddings to use, and more.

After evaluating the trained model, we reach 60% precision on a small test set of 25 samples. This is a good achievement considering that setting up the whole classification process for DeepMatcher is based on a few lines of code; however, higher precision can only be achieved through a larger training set.

3 Summary

The ease of use is what truly differentiates the neural-based entity matching approaches from a more rule-based and manual approach shown above. However, this approach needs a considerable amount of training data to perform well on a larger scale dataset. The threshold-based approach also gives one the opportunity to fine-tune the smallest details of how the similarity between record identifiers is computed, which is not easily available with DeepMatcher. The choice of which approach to use depends on the end goal, and the model's inference time, type of data, and availability of training data need to be taken into consideration. Do not discard any approach for its apparent simplicity or complexity and always rely on experimentation to define which one works best for your particular application.

Service Section

Main Application Fields: Entity matching is often used to consolidate records from various databases or data sources into one by joining records from various sources that refer to the same real-world entity. This approach is often applied when joining databases of people, for example, when matching census data from various sources. It is also used when joining data sources containing company entities of various types, for instance, hotels, amongst others. A subset of entity matching approaches is also used for data deduplication, for example, deduplication of hotel reviews within one database or deduplication of guest data between subsidiaries of a single hotel chain.

Limitations and Pitfalls: One of the major limitations of entity matching approaches is that there are almost no open-source datasets to work with. Since entity matching can be applied to various entity types and many of these include private information, finding any openly available datasets is extremely hard. The best approach is to usually annotate data internally, which requires a significant amount of time. Another issue is the computational complexity of many entity matching approaches. Comparing two databases, each containing millions of rows, and doing it record-by-record is usually computationally intangible. Much effort is needed to select the most suitable approach for

(continued)

pre-filtering in order to limit the number of record pairs that need to be compared.

Similar Methods and Methods to Combine with: Entity matching relies heavily on advances from other research fields such as string similarity matching, document classification, word embeddings, and many more.

Code: The Python code is available at: https://github.com/DataScience-in-Tourism/Chapter-19-Entity-Matching

Further Readings and Other Sources

Book: Data matching: Concepts and techniques for record linkage, entity resolution, and duplicate detection by Christen (2012).
Video: Deep learning for entity matching: A design space exploration https://www.youtube.com/watch?v=plaONS-Lr8U

References

Bayrak, A.T., Özbek, E.E., Kestepe, S., & Yildiz, O.T. (2019). Intelligent mapping for hotel records representing the same entity (pp. 560–563). In *2019 4th International conference on computer science and engineering (UBMK)*.
Christen, P. (2012). *Data matching: Concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer Publishing Company. Incorporated.
Cohen, W. W., Ravikumar, P., & Fienberg, S. E. (2003, August). A comparison of string distance metrics for name-matching tasks. *IIWeb, 3*, 73–78.
Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
Hyyrö, H. (2003). A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nordic Journal of Botany, 10*(1), 29–39.
Joulin, A., Grave, E., Bojanowski, P., Douze, M., Jégou, H., & Mikolov, T. (2016). FastText.zip: Compressing text classification models. arXiv preprint arXiv:1612.03651.
Keil, J. M. (2019). Efficient bounded Jaro-Winkler similarity based search. In T. Grust, F. Naumann, A. Böhm, W. Lehner, T. Härder, E. Rahm, A. Heuer, M. Klettke, & H. Meyer (Eds.), *BTW 2019*. Gesellschaft für Informatik.
Kirsten, T., Kolb, L., Hartung, M., Groß, A., Köpcke, H., & Rahm, E. (2010). Data partitioning for parallel entity matching. arXiv preprint arXiv:1006.5309.
Kong, C., Gao, M., Xu, C., Qian, W., & Zhou, A. (2016, April). Entity matching across multiple heterogeneous data sources. In *International conference on database systems for advanced applications* (pp. 133–146). Springer.
Kozhevnikov, I., & Gorovoy, V. (2016). Comparison of different approaches for hotels deduplication. In A.-C. N. Ngomo & P. Křemen (Eds.), *Knowledge engineering and semantic web*. Springer Nature.
Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Mudgal, S., Li, H., Rekatsinas, T., Doan, A., Park, Y., Krishnan, G., . . . Raghavendra, V. (2018). *Deep learning for entity matching: A design space exploration. SIGMOD'18* (pp. 19–34). Association for Computing Machinery. https://doi.org/10.1145/3183713.3196926

Tai, X. (2018). Record linkage and matching problems in forensics (pp. 510–517). In *2018 IEEE International conference on data mining workshops (ICDMW)*. IEEE. https://doi.org/10.1109/ICDMW.2018.00081.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., . . . Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (Vol. 30). Curran Associates.

Zhao, C., & He, Y. (2019). *Auto-EM: End-to-end fuzzy entity-matching using pre-trained deep models and transfer learning* (pp. 2413–2424). Association for Computing Machinery. https://doi.org/10.1145/3308558.3313578