UNIVERSITEIT TWENTE.

Data Science [201400174]

Course year 2022/2023, Quarter 1B

DATE
December 20, 2022

EXCERPT

Data Integration [DINT]

TEACHERS
Faizan Ahmed
Ellen-Wien Augustijn
Nacir Bouali
Faiza Bukhsh
Rolf de By
Karin Groothuis-Oudshoorn
Maurice van Keulen
Mahdi Khodadadzadeh
Elena Mocanu
Estefania Talavera
Brenda Voorthuis
Shenghui Wang

COURSE COORDINATOR
Karin Groothuis-Oudshoorn (quartile 1A)
Maurice van Keulen (quartile 1B)
Faizan Ahmed (quartile 2A)

PROJECT OWNERS
Faiza Bukhsh
Karin Groothuis-Oudshoorn
Maurice van Keulen
Elena Mocanu
Mannes Poel
Michel van Putten
Mohsen Jafari Songhori
Luc Wismans

Topic 8

Data Integration [DINT]

8.1 Introduction

Topic teachers: Shenghui Wang and Maurice van Keulen

An often-needed activity in Data Science is combining the data from two independent data sources. Its purpose is usually data enrichment: Given a data set with information about a certain entity (e.g., patients, users, products, locations, etc.), we would like to add additional information *about these same entities* from a different data source. This activity is often called **data integration**.

Complications that may arise are

- There is *no or no reliable ID* that links every entity of source A with a single unique entity in source B. Therefore, the entities have to be *matched* using the attributes both sources have in common. This typically involves string matching functions and the aggregation of multiple matching scores on several common attributes into one matching score for a pair of entities.
- There are usually *thresholds* involved: if the entity matching score is above a certain threshold, we consider them to be the same. Obviously, this approach will make errors: false positives and false negatives. Therefore, *evaluation* of the matching quality in terms of precision and recall is important. Furthermore, this also allows for a (sample-based) search for a best threshold setting.
- *Multiplicity*: Often an entity of source A should have one single unique matching entity in source B. The above approach may produce clusters of entities considered possibly the same. It is not trivial how to select the eventual 1-to-1 pairs.
- The common attributes in both sources may have different names. Therefore, a manual or automatic *mapping* needs to take place. Furthermore, the format of the values may differ (unit of measurement, date format, conventions in names or other strings, etc.), so some transformation may be needed to obtain a uniform format (*standardization*). This also falls under the activity of mapping.
- The values of the common attributes of both sources may contradict each other, i.e., we may encounter *inconsistency*. There some approach is needed for selecting the best one or combining both to create a best value.
- Files may be larger than the main memory of a single computer, i.e., *scalability* may be an issue. Therefore, a streaming-based implementation of the above activities is needed.

A similar activity is solving the semantic duplicate problem: a data set which contains for some real-world objects, multiple instances (e.g., customer create more than one account). Solving this problem requires very similar methods and techniques when viewed as matching the data set with itself and merging the instances of found matches.

8.1.1 Global description of the practicum and project

The global concept for this topic is the *match-map-merge-evaluate* data processing pipeline. The learning goals are tied to these steps in the pipeline and the abovementioned complications that may arise.

The main idea behind the topic assignments is to develop a data processing pipeline enriching the information of movies from a TV guide with additional information available from IMDB.

This challenge contains the following complications: no identifier for matching, not every movie has a match, there are movies with the same name (e.g., "King Kong" and "Die Hard"), common attributes with different names, other naming conventions in the titles of movies and names of actors, scalability (the IMDB data set is about 300GB), and even after standardization there are inconsistencies in the attribute values (e.g., actors in the one source but not in the other, same for genres).

The leading principle of the assignments is to step-by-step develop this data processing pipeline and learning and practicing with the individual steps in the process. The order is not match-map-merge-evaluate, but match-evaluate-map-merge: after the initial step of matching, we'd like to immediately learn how to evaluate this step. The steps of mapping and merging will include extension of the evaluation regarding the quality of the results of those steps.

8.1.2 Study material and tools

Recommended reading (also our sources for the topic lecture):

1 Euzenat Jérôme, Shvaiko P. (2013). Ontology matching (2nd ed.). Springer Berlin Heidelberg. https://link-springer-com.ezproxy2.utwente.nl/book/10.1007/978-3-642-38721-0

2 Castano S., Ferrara A., Montanelli S., Varese G. (2011) Ontology and Instance Matching. In: Paliouras G., Spyropoulos C.D., Tsatsaronis G. (eds) Knowledge-Driven Multimedia Information Extraction and Ontology Evolution. Lecture Notes in Computer Science, vol 6050. Springer, Berlin, Heidelberg. https://doi-org.ezproxy2.utwente.nl/10.1007/978-3-642-20795-2_7

The assignments are meant to be implemented in Python. There are hints with the assignments for certain packages that can be used and such. It is not obligatory to do the assignments in Python: you can choose another programming language, but be aware that we (teachers and teaching assistants) will not be able to help you (much) with programming issues. You'd be on your own with that.

If you are in need of a DBMS to store and manipulate data, you can use the DAB system to obtain a database for this course on our faculty's server. See the DS course guide for the DPV topic for instructions on how to obtain a database using DAB.

8.1.3 Deliverables and obligatory items

The solution to the assignments is (a) a Python programme and (b) output of this Python programme. This needs to be submitted to the appropriate Canvas assignment. A submission should include a ZIP-file with all source and output files as well as a PDF file with the source code and results as well. The latter is for the purpose of accelerating the checking of the assignments.

8.2 Data set

The topic assignments are based on two data sources:

A. `Tvguide-ds.xml`
A data set with information about 100 top movies from a TV guide. The data set contains limited general information about these movies (name, year, rating, running-time, country, genres, actors) as well as TV-guide specific information (rank, format, user-rating, reviews).

B. `Imdbfull-ds.xml`
An excerpt of about 250.000 movies from the well-known IMDB knowledge base. The data set has much more general information about movies.

Both sources have their own movie identifiers: 'tvgid' for the TV guide source and 'imdbid' for the IMDB source.

There is also a file `ground_truth.txt` which holds the correct match for each of the movies in the TVguide source. It is used to evaluate the performance of the matching approaches you will develop in the assignments.

8.3 Description of the practical assignments

8.3.1 Assignment 1: Matching

This assignment focuses on matching the movies from the TV guide data set with the movies in the IMDB data set based only on string similarity of the titles. The sources are in XML, so we use the standard Python XML processing modules[1] `xml.dom` and `sxml.sax` for reading and parsing them. Since we are interested in enriching the TV guide data set with information from IMDB, the former is our main source. It is not very large while the IMDB source is, so in the assignments we use the following approach to processing it (do not immediately code this; it is just an overview of steps. in the assignments below you will step by step develop it):

- Read the whole TV guide source into main memory using the XML DOM method.
- Read the IMDB source one movie at-a-time using the XML SAX method.
- After each IMDB movie, we match it with all movies in the TV guide and we keep the matching scores in main memory as tvgid-imdbid-score triples if the score is above a certain threshold. We keep these triples clustered per TV guide movie, so that we can later determine quickly which is the best match for each TV guide movie.
- Write the tvgid-imdbid-score triples of the resulting best match for each TV guide movie to a file.

We implement this approach in a few steps in the subassignments below and then further improve it in the subsequent assignments.

☞ **8.1**

(a) We start with setting up the reading and processing of both files. Write a program that reads the TV guide source using the DOM method and show that it works by counting how many movies there are in the TV guide source as well as printing the full information of one movie (first or last one; in this way, you can explore the structure of the data in this source). There should be 98 TV guide movies.
Tip: There are three different methods offered in Python: from easiest to most complex: `xml.etree.ElementTree`, `xml.dom.minidom`, and `xml.dom`. All suffice here, so you choose. There are many examples on the web; just Google "python xml example" to quickly find out how to use these methods. Our advice is: if you are not so experienced with Python, `xml.etree.ElementTree` is a good choice. If you are more experienced, `xml.dom.minidom` is better: it can accomplish the same with less code (but more complex to understand).

(b) Add to your program code that reads and processes the IMDB source using the SAX approach. Show that it works by counting how many movies there are in the IMDB source as well as printing the full information of one movie (first or last one; in this way, you can explore the structure of the data in this source). There should be 243856 IMDB movies.

[1] `https://docs.python.org/3/library/xml.html`

Tip: Use Python package `xml.sax` for XML processing. Below a piece of Python code that processes the IMDB file with SAX until a maximum number of movies and collects all titles in `string_list`.

```
from xml.sax import parse
from xml.sax.handler import ContentHandler

class imdb_Handler(ContentHandler):
  def __init__(self, total_movies):
    self.tag = None #Used to store the tag name thats currently being parsed
    self.string_builder = [] #Used to store the complete title/actor names
    self.string_list = [] #Stores all the movie titles in a list
    self.movie_key = 0 # counter for number of movies seen so far
    self.total_movies = 5 # upper limit for number of movies to process

  def startElement(self, name, attrs):
    self.tag = name #Initialize tag name
    self.movie_key += 1

  def endElement(self, name):
    self.tag = None #Remove tag name when the tag ends
    if name == "movie" and self.movie_key>=self.total_movies: #Check if upper limit is reached
      raise Exception("")
    if name == "title": #If tag is title
      self.string_list.append(" ".join(self.string_builder)) #append title to list
    self.string_builder = [] #Clear string builder list

  def characters(self, content):
    if content.strip() != "" and self.tag is not None:
      self.string_builder.append(content) #Add to string builder list

handler = imdb_Handler(15) #Pass the number of movies you want to retrieve as a parameter
try:
  parse("imdb-ds.xml", handler)
except Exception as e:
  print(e)
  ass
print(handler.string_list)
```

NB: If you have enough main memory in your computer, you could read the whole document using XML DOM. You are not supposed to do that in these assignments; we like you to learn how to process data in a streaming fashion.

(c) Add to your program code that selects a small sample of titles of both sources (say, 5 each) and determines the string similarity between the 5x5 pairs. Do this using the string similarity functions *Levenshtein Distance, Jaro-Winkler Distance, Hamming Distance*, and *American Soundex* seperately. Show that it works by printing the pairs of titles and their string similarity scores for each string similarity function. For example, for "Hamming Distance" it could output that TV guide movie "Ever After" and IMDB movie "Prisoner of Zenda" have a distance of 17.

Tip 1: Use Python package `jellyfish`[2] for string similarity functions.

Tip 2: The `soundex` function does not return a score, but a code. One can determine matches by selecting those strings that have the same soundex code, but that would only produce a set of matches for which it is unclear which one is the best match. Therefore, combine soundex with one of the other similarity functions *sim* to determine a "soundex+sim(a,b)" score as follows:

$$soundex{+}sim(a,b) = \begin{cases} sim(a,b) & soundex(a) = soundex(b) \\ 0 & \text{otherwise} \end{cases}$$

[2]Can be installed using `pip install jellyfish`. Package source: `https://github.com/jamesturk/jellyfish`. Documentation: `https://jamesturk.github.io/jellyfish/`

(d) Modify your program such that it does the matching for all $98 \times 243856 \approx 24M$ (t,i) pairs, where t is a TV guide movie and i is an IMDB movie. Choose a string similarity function $sim(t,i)$ that you find most suitable and establish a self-set lower threshold τ, so that the similarity score is only retained if $(sim)(t,i) \geq \tau$, i.e., only a limited number of candidate matching pairs is kept in main-memory and afterwards written to a file.
NB: It is possible that your τ is so high that for some t there is no i with $(sim)(t,i) \geq \tau$. In such a case, the conclusion is that there is no candidate match. Note that it is indeed possible in general, that there need not exist a match for each entity in the other source. In our case, this is also the case, i.e., there are some TV guide match movies that have no match in the IMDB source.

(e) Add to your program a step that determines a single most probable match in IMDB for each movie in the TV guide source, or decides that no such match is present. Note that an IMDB movie cannot be a match for two or more TV guide movies. Write the resulting matches with their scores as tvgid-imdbid-score triples to a different file.

□

8.3.2 Assignment 2: Evaluation

☞ **8.2** Matching is an imperfect process: the results need not be entirely correct. Therefore, an important next step is to evaluate the quality of the result. We provide you with a *ground truth*, i.e., a data set with for each TV guide movie, the movie in IMDB which in the real world is indeed the same one as the TV guide movie. In other words, the ground truth provides you with the correct answers. This you can use to evaluate the correctness of the result of your matching algorithm as a means to assess the performance of this matching algorithm.

(a) Extend your program so that it not only computes the similarity scores for one similarity function, but for all similarity functions of the previous assignment. And add a step that determines the precision, recall, and F1 of the result of your matching algorithm based on the given ground truth. Compare the performance on these metrics for the different similarity score functions.
NB: If your algorithm finds a match for each TV guide movie, then precision and recall are the same.
Tip: The "sklearn" package has nice functions for evaluating such well-known metrics, for example `precision_recall_fscore_support`. You can use the following example code to load the ground truth and compute the metrics given that your own matching result is in the variable "result":

```
from sklearn.metrics import precision_recall_fscore_support

relevant = []
with open ("ground_truth.txt","r") as f:
  for line in f.readlines():
    relevant.append(line.strip().split(",")[1])

pre,rec,f1,av = precision_recall_fscore_support(relevant, result, average='micro')
print("Precision: ",pre,"\nRecall:",rec,"\nF1 score:",f1)
```

NB: You can also design your own functions to calculate precision, recall and F1. This allows you to decide whether one TV guide movie has a match or not based on different thresholds.

(b) As you probably will conclude from the result to the previous subquestion, matching on only the title is not the best algorithm. Let us improve it by involving more attributes in the matching.
To be able to evaluate by how much such an extension improves the matching performance, leave the existing matching algorithm by title only intact. Add to your programme a new algorithm that matches by title as well as year and actors. Develop for each attribute a separate similarity score function and compute a final score by means of a weighted average; choose the weights based on your own common sense.
Then re-evaluate the result of both algorithms on precision, recall, and F1 and conclude whether and by how much your extended algorithm improved the matching performance.

□

8.3.3 Assignment 3: Mapping

The term *mapping* refers to both dealing with different attribute names for the same information between sources as well as dealing with different formats or conventions between sources. Perhaps you didn't think of it explicitly, but the fact that the element names for `actor` and `year` are the same in both sources is more of a coincidence than something one can rely on.

Moreover, you may have spotted by now that TV guide and IMDB use certain conventions for the movie titles and actor names. As a consequence, the matching will not be 100% for obviously the same movie titles and actor names. It make sense to attempt to standardize such conventions, i.e., map the strings in one source to the convention of the other ... and then re-evaluate to see to what degree it improves the matching performance. For example, a title in IMDB may be formatted as "Perfect Nanny, The" while the same title in TVguide would be "The Perfect Nanny". This ", The" at the end of an IMDB title can also be mapped explicitly and specifically to "The " at the beginning of the title. In this way, making the IMDB title uniform to the conventions of the titles in TVguide may improve the matching.

☞ **8.3** Add one more algorithm to your programme that performs the above-described mapping of the movie titles and actor names, i.e., transforms the values of one source to the convention of the other source before the similarity matching step. Evaluate the performance gain of this step by comparing precision and recall for all three algorithms. □

8.3.4 Assignment 4: Merging

☞ **8.4** The whole purpose of doing all this matching and mapping is to enrich the TV guide source with additional attributes from the IMDB data set. This means that we have to *merge* matching movies. We do this in a few steps. Note, that we do not ask you to evaluate the result of this step.

(a) We first focus on the easy cases: attributes in IMDB that are not in the TV guide (`directors`, `locations`, `keywords`, `plots`, and `imdbid`) and single-valued ones for which we assume that IMDB has the most reliable value (`title`, `year`). Add a step to your programme that, for each TV guide movie, merges these attributes of the chosen matched IMDB movie into the TV guide movie. For `title` and `year`, overwrite the value in the TV guide movie whenever IMDB happens to disagree and provides a different value (we are not sure whether are such inconsistencies in the data set)
Inspect a sample of generated merged movies to ascertain that your merge is working correctly.

(b) Now the genre attribute: note that the logical intuition here could be that any genre mentioned in one or both of the sources, would be a useful genre to have in the end. Therefore, improve you merging step by taking the *union* of the genres of the TV guide movie and its matched IMDB movie.
Inspect a sample of generated merged movies to ascertain that your merge is working correctly.

(c) Finally, provide a merge of the actors lists. Intuition here is also to take the union, but the complication is that the actor names need not exactly match, so you do need to take special measures to prevent generating duplicates. Furthermore, also implement the rule that if a role is unique for a movie in both sources, then assume that it must be the same actor; if that case, we will take the name from IMDB as the most reliable one.
Inspect a sample of generated merged movies to ascertain that your merge is working correctly.

□

8.3.5 What we didn't do in the assignments

These assignments have been designed to teach you the basics. What we like to emphasize as well as didn't do, but highly recommend in practice are:

- Always develop alternative approaches and an evalution based on metrics to properly and systematically evaluate alternative approaches and make a choice based on real evidence. That is the academic way of working.
- Also for determining the values of parameters (such as the weights and thresholds in the above), it is recommended to use a systematic approach that evaluates the performance on metrics for a range of values for these parameters. This is called 'hyperparameter tuning'.

- You will need a 'ground truth' for computing the metrics on the results of each alternative approach. A ground truth produced by real judgments of humans is preferred. Be aware of possible biases in the ground truth.