Exercise 2: Identity Resolution

Web Data Integration
Agenda

1. Exercise Overview

2. Preparing the Inputs
   1. Check your data
   2. Create gold standard

3. Java Project Template
   1. Load your data
   2. Experiment with matching rules
   3. Use blocking
   4. Evaluate results
   5. (Extra task) Learn matching rules
1. Exercise Overview

Project Phase 2: Identity Resolution

Duration: October 18\textsuperscript{th} – November 7\textsuperscript{th}

Tasks: Extend Java project template to

1. Identify records in different data sets that describe the same real-world entity
2. Experiment with different combinations of similarity measures (matching rules)
3. Use blocking to speed up the comparisons
4. Evaluate quality of your approach (F1 / Reduction Ratio)
5. Extra task: Learn matching rule using RapidMiner

Result: Correspondences between records in different data sets that describe the same entity
2. Prepare the Inputs: Check Your Data

• Your input is the output of Exercise 1
  • Vocabularies are aligned
  • Unique IDs are in place

• Are there duplicates in your data?
  • At least 1000 entities should be contained in at least two datasets.

• Is there enough attribute overlap?
  • At least 5 attributes should be contained in at least two datasets.

• Which combination of attributes can you use to detect duplicates?
  • name/title, creation/founding date, location/address, height, colour, …
Prepare the Inputs: Create Gold Standard

• To evaluate identity resolution algorithms, you need a gold standard
  • .csv file containing pairs of (comma-separated) IDs of entities that match and do not match
• You have to create it manually
• Include non-trivial cases
  • ...
• Gold standard should contain more negative than positive examples

gold.csv:
1-9309,2-9309,true
1-9310,2-9310,true
1-9311,2-9311,false
1-9312,2-9312,true
1-9313,2-9313,false
1-9314,2-9314,false
1-9315,2-9315,false
1-9316,2-9316,true
Prepare the Inputs: Create Gold Standard

• Make gold standard big enough
  • At least 1% (or 100 pairs, if your datasets are huge) of entities

• You need a gold standard for all data sets / classes that you want to use in the fusion part
  • Minimum is a gold standard for your main class
  • Only makes sense if you have overlapping attributes that you can use

• Proceed iteratively
  • Create a smaller gold standard, go through the whole exercise, then come back to improve the gold standard by adding corner cases (and fixing errors)
Create Gold Standard: Bad Pair of Data Sets

- Example of a bad decision on a pair of records for creating a gold standard
  - Not much intersection of attributes – just titles
  - Impossible to formulate interesting matching rules

```xml
<movie>
  <title>Madagascar</title>
  <date>2005-05-26</date>
</movie>
<movie>
  <title>Mission: Impossible</title>
  <date>1996-05-21</date>
</movie>
<movie>
  <title>Mission: Impossible II</title>
  <date>2000-05-23</date>
</movie>
```

```xml
<movie>
  <title>Madagascar: Escape 2 Africa</title>
  <studio>Paramount</studio>
  <genre>Animation</genre>
  <budget>150</budget>
  <gross>462.3</gross>
</movie>
<movie>
  <title>Made of Honor</title>
  <studio>Sony</studio>
  <genre>Comedy</genre>
  <budget>40</budget>
  <gross>106</gross>
</movie>
```
Create Gold Standard: Good Pair of Data Sets

- Example of a good decision on a pair of records for creating a gold standard
  - 3 shared attributes: title, director, date
  
  ➔ Matching rules can experiment with different combinations of these attributes.

```xml
<movie>
  <title>Black Swan</title>
  <director>
    <name>Darren Aronofsky</name>
  </director>
  <date>2010-01-01</date>
</movie>

<movie>
  <title>The Fighter</title>
  <director>
    <name>David O. Russell</name>
  </director>
  <date>2010-01-01</date>
</movie>

<movie>
  <title>Black Swan</title>
  <director>
    <name>Aronofsky, Darren</name>
  </director>
  <date>2011-01-01</date>
</movie>

<movie>
  <title>Social Network, The</title>
  <director>
    <name>Fincher, David</name>
  </director>
  <date>2011-01-01</date>
  <globe>yes</globe>
</movie>
```
Winter has come!

- The **Web Data Integration** Framework (WInte.r) provides methods for end-to-end data integration

- Implements methods for
  - Pre-Processing
  - Schema Matching
  - Identity Resolution
  - Data Fusion
  - Evaluation

- Open Source under Apache 2.0 License
- [https://github.com/olehmberg/winter](https://github.com/olehmberg/winter)
1. Download the .zip of the project from the course page

2. Unzip it and look at the sample files in \data\input\n   • .xml input data sets in input folder
   • .csv gold standard

3. Open the project in Eclipse (import as maven project)
   • The project serves as a quick-start for todays tasks
     • It contains a data model for movies
     • It contains a fully implemented identity resolution workflow using WInte.r
     • It contains several blocking functions and comparators
Identity Resolution Walkthrough: Movie Use Case

1. Loading Data
2. Creating a Matching Rule
3. Running the Identity Resolution
4. Evaluating the Matching Result
5. Learning a Matching Rule
• First Step: Define your data model!
  • Create Java classes for your entities
  • Implement the *Matchable* Interface

```java
public class Movie implements Matchable {

  public Movie(String identifier, String provenance) {
    id = identifier;
    this.provenance = provenance;
    actors = new LinkedList<>();
  }

  private String title;
  private String director;
  private LocalDateTime date;
  private List<Actor> actors;

  public String getTitle() {
    return title;
  }

  public String setTitle(String title) {
    this.title = title;
  }

  ...
}
```
Second Step: Define how to load your model from XML files

- Extend the `XMLMatchableReader` class
- It evaluates an XPath expression
- Calls `createModelFromElement` for each result

```java
public class MovieXMLReader extends XMLMatchableReader<Movie, Attribute> {
    @Override
    public Movie createModelFromElement(Node node, String provenanceInfo) {
        // get the ID value
        String id = getValueFromChildElement(node, "id");

        // create a new object with id and provenance information
        Movie movie = new Movie(id, provenanceInfo);

        // fill the attributes
        movie.setTitle(getValueFromChildElement(node, "title"));
        ...

        // return the new object
        return movie;
    }
}
```
Loading Data: XMLMatchableReader

- Methods provided by XMLMatchableReader

```java
getValueFromChildElement(node, "title");
getListFromChildElement(node, "director");
getObjectListFromChildElement(
    node,
    "actors",
    "actor",
    new ActorXMLReader(),
    provenanceInfo);
```
Loading Data: Load an XML file

- Third Step: Load your data set
  - Create a new data set
  - Specify
    - The file that contains your data
    - The XPath to the XML elements that represent your records

```java
// create a new data set
HashedDataSet<Movie, Attribute> ds = new HashedDataSet<>();

// load an XML file
new MovieXMLLoader().loadFromXML(
    new File("data/input/academy_awards.xml"), // the file to load
    "/movies/movie", // XPath to elements
ds); // data set to fill
```
• Alternative: Use the Default Model for a simple schema
  • de.uni_mannheim.informatik.dws.winter.model.defaultmodel
  • Basically a key/value map supporting atomic values and lists
  • Data is modelled using the Record and Attribute classes

```java
// Map the XML Element names to attribute names in the data set
Map<String, Attribute> nodeMapping = new HashMap<>();
nodeMapping.put("title", Movie.TITLE);
nodeMapping.put("date", Movie.DATE);

new XMLRecordReader("id", nodeMapping).loadFromXML(sourceFile, "/movies/movie", ds);
```
Creating a Matching Rule

- A matching rule specifies which attributes to compare and how to create an overall similarity value

\[
sim(x,y) = 0.3s_{\text{name}}(x,y) + 0.3s_{\text{phone}}(x,y) + 0.1s_{\text{city}}(x,y) + 0.3s_{\text{state}}(x,y)
\]

- \(s_{\text{name}}(x,y)\): using the Jaro-Winkler similarity measure
- \(s_{\text{phone}}(x,y)\): based on edit distance between x’s phone (after removing area code) and y’s phone
- \(s_{\text{city}}(x,y)\): based on edit distance
- \(s_{\text{state}}(x,y)\): based on exact match; yes → 1, no → 0

- **Similarity Measures** specify the similarity of two values
- **Comparator**s specify how to compare the values of attributes
- **Matching Rules** specify how to combine the different similarity values
Creating a Matching Rule: Similarity Measures

- A similarity measure calculates a similarity between two values
  - Extend the SimilarityMeasure class
  - Accepts two values and returns their similarity

```java
public class TokenizingJaccardSimilarity extends SimilarityMeasure<String> {

    @Override
    public double calculate(String first, String second) {
        if (first == null || second == null) {
            return 0.0;
        } else {
            // use the SecondString library to calculate the similarity value
            Jaccard j = new Jaccard(new SimpleTokenizer(true, true));
            return j.score(first, second);
        }
    }
}
```
Creating a Matching Rule: Comparators

- Example: Calculate Jaccard similarity between movie’s directors
  - First lower-case the values
  - Then calculate similarity
  - Finally, square similarity

- First Step: Create attribute comparators
  1. (optional) apply specific preprocessing
  2. calculate similarity value
  3. (optional) re-scale the similarity value

```java
import de.uni_mannheim.informatik.dws.winter.matching.rules.Comparator;

public class MovieDirectorComparatorJaccard implements Comparator<Movie, Attribute> {

    TokenizingJaccardSimilarity sim = new TokenizingJaccardSimilarity();

    @Override
    public double compare(Movie entity1, Movie entity2, Correspondence<Attribute, Matchable> schemaCor) {

        // preprocessing
        String s1 = entity1.getDirector().toLowerCase();
        String s2 = entity2.getDirector().toLowerCase();

        // calculate similarity value
        double similarity = sim.calculate(s1, s2);

        // postprocessing
        similarity *= similarity;

        return similarity;
    }
}
```
Creating a Matching Rule: Combine Comparators

• Now we define a matching rule
  • Use the LinearCombinationMatchingRule class
  • Specify final threshold
  • Add comparators and corresponding weights

```
LinearCombinationMatchingRule<Movie, Attribute> rule =
    new LinearCombinationMatchingRule<>((0.5)); // final threshold
rule.addComparator(new MovieTitleComparator(), 0.6); // comparator & weight
rule.addComparator(new MovieDateComparator(), 0.4); // comparator & weight
```

\[
sim_{Movie}(m_1, m_2) = 0.6 \cdot sim_{title}(m_1, m_2) + 0.4 \cdot sim_{date}(m_1, m_2)
\]

\[
match_{Movie}(m_1, m_2) = \begin{cases} 
1 & \text{if } sim_{Movie}(m_1, m_2) \geq 0.5 \\
0 & \text{otherwise}
\end{cases}
\]
Define a Blocker

• Listing all pairs of records in two datasets D and E is in $O(|D||E|)$
• Blockers create fewer pairs which speeds up the matching runtime

• You can choose between
  • `de.uni_mannheim.informatik.dws.winter.matching.blockers`
  • `NoBlocker`
    • Calculates all pairs, i.e. no blocking
  • `StandardRecordBlocker`
    • Uses a blocking function to create pairs
  • `SortedNeighbourhoodBlocker`
    • Uses the sorted neighbourhood method
Define a Blocking Function

• A blocker creates pairs based on a blocking function
  • Records for which the blocking function returns the same value will become a pair

• Example: use the decade of a movie’s release as blocking key

```java
public class MovieBlockingFunction extends RecordBlockingKeyGenerator<Movie, Attribute> {

  @Override
  public void generateBlockingKeys(Movie instance, Processable<Correspondence<Attribute, Matchable>> correspondences, DataIterator<Pair<String, Movie>> resultCollector) {

    resultCollector.next(new Pair<>(
        Integer.toString(instance.getDate().getYear() / 10), instance));
  }
}
```
Running the Identity Resolution

• Create a MatchingEngine instance and run the identity resolution

```java
// create & configure the blocker
Blocker<Movie, Attribute> blocker = new StandardRecordBlocker<>(new MovieBlockingFunction());

// create a matching engine
MatchingEngine<Movie, Attribute> engine = new MatchingEngine<>();

// run the matching
Processable<Correspondence<Movie, Attribute>> correspondences = engine.runIdentityResolution(ds1, ds2, null, rule, blocker);
```
Evaluating the Result

• First Step: Load your gold standard
  • Use the MatchingGoldStandard class
  • It will inform you about some possible errors in your gold standard
    • Pay attention to the error output!

• Second step: evaluate your result
  • Use the MatchingEvaluator class

```java
// load the gold standard
MatchingGoldStandard gs = new MatchingGoldStandard();
gs.loadFromCSVFile(new File("gold.csv"));

// evaluate the result
MatchingEvaluator<Movie, Attribute> evaluator = new MatchingEvaluator<> (true);
Performance perf = evaluator.evaluateMatching(correspondences, gs);

// print the performance
System.out.println(String.format("Precision: %.4f\nRecall: %.4f\nF1: %.4f", perf.getPrecision(),
perf.getRecall(), perf.getF1()));
```
Learning a Matching Rule

• Use the **WekaMatchingRule** class
  • Configure it with the model & parameters you want to use
  • Train it on a labelled training set
  • Then you can run it on your data
  • And evaluate it on a *separate* test set

```java
// create the matching rule
String options[] = new String[] { "" };  
String modelType = "SimpleLogistic";  // use a logistic regression
WekaMatchingRule<Movie, Attribute> matchingRule = new WekaMatchingRule<> (0.6, modelType, options);

// add comparators
matchingRule.addComparator(new MovieDirectorComparatorLevenshtein());
matchingRule.addComparator(new MovieTitleComparatorLevenshtein());

// load the training set
MatchingGoldStandard gsTraining = new MatchingGoldStandard();
gsTraining.loadFromCSVFile(new File("training.csv"));

// train the matching rule's model
RuleLearner<Movie, Attribute> learner = new RuleLearner<>();
learner.learnMatchingRule(dataAcademyAwards, dataActors, null, matchingRule, gsTraining);
```
Learning a Matching Rule in RapidMiner

- Alternative: Generate a data set for RapidMiner
  - Your data set is generated for all records in the training set
  - Every Comparator in your matching rule becomes a feature in this data set
  - The output of your current matching rule is included (not required for learning)
    - "isMatch": does the current rule think this is a match?
    - "finalValue": similarity score calculated by the current rule

```java
// generate the feature data set
RuleLearner<Movie, Attribute> learner = new RuleLearner<>();
FeatureVectorDataSet features
    = learner.generateTrainingDataForLearning( dataAcademyAwards, dataActors, gsTraining, matchingRule, null);

// write the data set to a CSV file
new RecordCSVFormatter().writeCSV(new File("output/features.csv"), features);
```
Learning a Matching Rule in RapidMiner

- Second Step: Learn a Linear Regression in RapidMiner
  - Learn a model using the feature data set that you generated from the code
  - Use X-Validation for the estimation of the performance in RapidMiner
Learning a Matching Rule in RapidMiner

• Third Step: Adjust your matching rule
  • Enter the offset (a.k.a. intercept) and coefficients in the definition of your `LinearCombinationMatchingRule`
  • Evaluate the new matching rule on the test set of your gold standard!

```
LinearCombinationMatchingRule<Movie, Attribute> rule =
    new LinearCombinationMatchingRule<>(-1.497, 0.5);  // offset & final threshold
rule.addComparator(new MovieTitleComparator(), 1.849);  // comparator & weight
rule.addComparator(new MovieDateComparator(), 0.822);  // comparator & weight
```
Identity Resolution in the Final Project Report

• Results of Exercise 2 will be part of your final report

• Make sure your know/make notes on
  1. Content and size of your gold standard?
     • Which classes/data sets are included?
     • What „corner cases“ did you include?
  2. Which matching rules did you try?
     • What happens with P/R/F1?
     • Which attribute comparators / similarity measures did you use?
  3. What blockers have you tried?
     • What happened with runtime and number of matches?
     • What blockers / blocking functions have you used?
     • How do P/R/F1 change, and why?

• Note also that Exercise 2 output is Exercise 3 (Data Fusion) input
Task

1. Open and run the provided Java project
   1. Which performance does the linear combination rule achieve?
   2. Which performance does the machine learning rule achieve?

2. Understand your results:
   1. Write code that shows which errors are made

3. Experiment with different Blockers
   1. First, use the NoBlocker to see the maximum runtime
   2. Then, try different blocking keys with the StandardRecordBlocker
   3. Finally, try the SortedNeighbourhoodBlocker

4. Try different combinations of Comparators in your matching rule (don’t use the title to make it interesting)
   1. Can you improve the performance?
   2. Create a comparator that uses the actors
...and now

1. Prepare the gold standard
2. Get the project template and
   • Define your inputs
   • Define blocking functions
   • Define your matching rules
   • Run the evaluation
   • (extra) Learn matching rules