Web Data Integration

Exercise 2

Identity Resolution
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 24\textsuperscript{th} – November 13\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

<table>
<thead>
<tr>
<th>gold.csv:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-9309,2-9309, true</td>
</tr>
<tr>
<td>1-9310,2-9310, true</td>
</tr>
<tr>
<td>1-9311,2-9311, false</td>
</tr>
<tr>
<td>1-9312,2-9312, true</td>
</tr>
<tr>
<td>1-9313,2-9313, false</td>
</tr>
<tr>
<td>1-9314,2-9314, false</td>
</tr>
<tr>
<td>1-9315,2-9315, false</td>
</tr>
<tr>
<td>1-9316,2-9316, true</td>
</tr>
</tbody>
</table>
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>

<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>
```
3. Java Project Template

1. Download the .zip of the project from the course page

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

3. Open the project in Eclipse
   - We have implemented for you
     - Loading/storing input datasets and gold standard
     - Infrastructure for matching two data sets and calculating evaluation metrics
     - Examples of matching rules, similarity measures, and blocking functions
     - Output of the results, preparing data for RapidMiner learning task
Project Template Walkthrough: Movie Use Case

1. Loading Data
2. Creating a Matching Rule
3. Running the Matching Engine
4. Evaluating the Matching Result
5. Learning a Matching Rule
- First Step: Define your data model!
  - You should define a proper object model that implements the `Matchable` Interface

```java
public class Movie extends Record {

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

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

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

public interface Matchable {
    String getIdentifier();
    String getProvenance();
}

public abstract class Record implements Matchable {
    ...
}
```
Project Template: Loading Data

- Second Step: Define how to load your model from XML files
  - By implementing a class that extends MatchableFactory

```java
public class MovieFactory extends MatchableFactory<Movie> {
    @Override
    public Movie createModelFromElement(Node node, String provenanceInfo) {
        String id = getValueFromChildElement(node, "id");

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

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

        // load the list of actors
        List<Actor> actors =
            getObjectListFromChildElement(
                node, "actors", "actor",
                new ActorFactory(), provenanceInfo);
        movie.setActors(actors);

        return movie;
    }
}
```
Methods provided by MatchableFactory

```java
public abstract class MatchableFactory
    <RecordType extends Matchable> {

    public abstract RecordType createModelFromElement(
        Node node,
        String provenanceInfo);

    protected String getValueFromChildElement(
        Node node,
        String childName) { ... }

    protected List<String> getListFromChildElement(
        Node node,
        String childName) { ... }

    protected <ItemType extends Matchable>
        List<ItemType> getObjectListFromChildElement(
        Node node,
        String childName,
        String objectNodeName,
        MatchableFactory<ItemType> factory,
        String provenanceInfo) { ... }
}
```

Project Template: Loading Data

```xml
<movie>
    <id>academy_awards_2</id>
    <title>True Grit</title>

    <director>
        <name>Joel Coen</name>
        <name>Ethan Coen</name>
    </director>

    <actors>
        <actor>
            <name>Jeff Bridges</name>
        </actor>
        <actor>
            <name>Hailee Steinfeld</name>
        </actor>
    </actors>
</movie>
```
– Alternative: Use the DefaultRecord for simple schema
  • Basically a key/value map supporting atomic values and lists
  • You might have to adjust the factory to correctly interpret your schema
    • All XML elements below the XML element representing your record will be added with their name as key
    • If an XML element contains multiple child elements, it is interpreted as list
      • Lists contain all text of the child XML elements

```java
public class DefaultRecord extends AbstractModel {
    public String getValue(String attributeName) { ... }
    public List<String> getList(String attributeName) { ... }
}
```
Third Step: Load your data set

- Use the DataSet class and provide the following parameters:
  - The file reference for your data set
  - The factory that creates your record
  - The XPath to the XML elements that represent your records

```java
public class DataSet<RecordType> extends Matchable {

    public void loadFromXML(
        File dataSource,
        MatchableFactory<RecordType> modelFactory,
        String recordPath) throws ...
    }

public class Movie {
    // properties
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<movies xsi:noNamespaceSchemaLocation="..."
    xmlns:xsi="...">
    <movie>
        ...
    </movie>
</movies>
```

```java
DataSet<Movie> ds = new DataSet<>();

ds.loadFromXML(  
    new File("usecase/movie/input/academy_awards.xml"),  
    new MovieFactory(),  
    "/movies/movie");
```
Project Template: Creating a Matching Rule

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.wdi.identityresolution.matching.Comparator;

public class MovieDirectorComparatorJaccard extends Comparator<Movie> {
    TokenizingJaccardSimilarity sim = new TokenizingJaccardSimilarity();

    @Override
    public double compare(Movie entity1, Movie entity2) {
        // 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;
    }
}
```
Project Template: Creating a Matching Rule

- Note: You may have to implement additional similarity measures!
  - Extend the SimilarityMeasure class
  - Contains a function that accepts two values and returns their similarity

```java
public abstract class SimilarityMeasure<DataType> {
    public abstract double calculate(DataType first, DataType second);
}
```

```java
public class LevenshteinSimilarity extends SimilarityMeasure<String> {
    @Override
    public double calculate(String first, String second) {
        if (first == null || second == null) {
            return 0.0;
        } else {
            Levenstein l = new Levenstein();
            double score = Math.abs(l.score(first, second));
            score = score / Math.max(first.length(), second.length());
            return 1 - score;
        }
    }
}
```
Project Template: Creating a Matching Rule

- Second Step: Define your matching rule
  - Use the LinearCombinationMatchingRule class
  - Specify final threshold
  - Add comparators and corresponding weights

```java
public class LinearCombinationMatchingRule
  <RecordType extends Matchable>
  extends MatchingRule<RecordType> {

  public LinearCombinationMatchingRule(double finalThreshold) { ... }
  public void addComparator(Comparator<RecordType> comparator, double weight) { ... }
  ...
}

LinearCombinationMatchingRule<Movie> rule =
  new LinearCombinationMatchingRule<>((0.5);

rule.addComparator(new MovieTitleComparator(), 0.6);
rule.addComparator(new MovieDateComparator(), 0.4);
```
Project Template: Creating a Matching Rule

- Feel free to implement your own matching rules!
  - Extend the MatchingRule class
  - Override the compare function
  - Implement the generateFeatures function if you want to optimise the rule

```java
public class LinearCombinationMatchingRule
    <RecordType extends Matchable>
    extends MatchingRule<RecordType> {
...
    @Override
    public double compare(RecordType record1, RecordType record2) {
        double sum = offset;
        for(int i = 0; i < comparators.size(); i++) {
            Pair<Comparator<RecordType>, Double> pair = comparators.get(i);
            Comparator<RecordType> comp = pair.getFirst();
            double similarity = comp.compare(record1, record2);
            double weight = pair.getSecond();
            sum += (similarity * weight);
        }
        return sum;
    }
    ...
}
```
Third Step: Choose a blocker

- CrossProductBlocker
  - Calculates all pairs, i.e. no blocking
- PartitioningBlocker
  - Uses a blocking function to create pairs
- SortedNeighbourhoodBlocker
  - Uses the sorted neighbourhood method

```java
public abstract class Blocker
    <RecordType extends Matchable> {

    public double getReductionRatio() { ... }

    public abstract List<Pair<RecordType, RecordType>> generatePairs(
        DataSet<RecordType> dataset1,
        DataSet<RecordType> dataset2);
}
```
Fourth Step: Define your blocking function

- Generate a blocking key for a given record

```java
public abstract class BlockingFunction
    <RecordType extends Matchable> {

    public abstract String getBlockingKey(RecordType instance);
}

public class MovieBlockingFunction
    extends BlockingFunction<Movie> {

    @Override
    public String getBlockingKey(Movie instance) {
        return Integer.toString(instance.getDate().getYear() / 10);
    }
}
```
Project Template: Running the Matching Engine

Create and configure a MatchingEngine instance

```java
public class MatchingEngine<RecordType extends Matchable> {

    public MatchingEngine(
        MatchingRule<RecordType> rule,
        Blocker<RecordType> blocker) { ... }

    public List<Correspondence<RecordType>> runMatching(
        DataSet<RecordType> dataset1,
        DataSet<RecordType> dataset2) { ... }

    public DataSet<DefaultRecordType> generateTrainingDataForLearning(
        DataSet<RecordType> dataset1,
        Blocker<Movie> blocker = new PartitioningBlocker<>(new MovieBlockingFunction());
        MatchingEngine<Movie> engine = new MatchingEngine<>(rule, blocker);
    }

    // run the matching
    public List<Correspondence<Movie>> correspondences = engine.runMatching(ds1, ds2);

    public RecordType getFirstRecord() { ... }
    public RecordType getSecondRecord() { ... }
    public double getSimilarityScore() { ... }
}
```
Project Template: Evaluating the Result

- First Step: Load your gold standard
  - Use the GoldStandard 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
public class MatchingEvaluator<RecordType>
    extends Matchable {

    public Performance evaluateMatching(List<Correspondence<RecordType>> correspondences,
        GoldStandard goldStandard) {
    }

    public class Performance {
        public double getPrecision() { … }
        public double getRecall() { … }
        public double getF1() { … }
    }

    public class GoldStandard {
        public void loadFromCSVFile(File file) throws … {
        …
    }

    GoldStandard gs = new GoldStandard();
    gs.loadFromCSVFile(new File("usecase/movie/goldstandard/gs_academy_awards_2_actors.csv"));

    MatchingEvaluator<Movie> evaluator = new MatchingEvaluator<>(true);
    Performance perf = evaluator.evaluateMatching(correspondences, gs);

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

First Step: Generate a data set for RapidMiner

- Your data set is generated for all records in the given gold standard
- Every Comparator in your matching rule becomes a feature in this data set
  - Marked by [number]
- The label from the gold standard is included to be used for learning
  - Field „label“
- 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

```
"[0] MovieTitleComparator","[1] MovieDateComparator","finalValue","isMatch","label"
"0.41666666666666663","1.0","0.09541666666666648","false","0"
"1.0","0.81","1.01782","true","1"
"0.7272727272727273","0.0","-0.15227272727272734","false","0"
"0.9523809523809523","0.81","0.9297723809523807","true","1"
"1.0","0.6400000000000001","0.87808","true","1"
```

```
DataSet<DefaultRecord> features =
engine.generateTrainingDataForLearning(ds1, ds2, gsTraining);
features.writeCSV(
    new File("usecase/movie/output/optimisation/academy_awards_2_actors_features.csv"),
    new DefaultRecordCSVFormatter());
```
Second Step: Learn a Linear Regression in RapidMiner

- You will learn weights for all records in the gold standard that you specify.
- So, you must split your gold standard into a training set TR and test set TE!
- Use X-Validation on TR for the estimation of the performance in RapidMiner.
Third Step: Adjust your matching rule

- Enter the offset (a.k.a. intercept) and coefficients in the definition of your matching rule
- Evaluate the new matching rule on the test set TE of your gold standard!
- Compare the performance of your handwritten rule and the learned rule on TE

```java
LinearCombinationMatchingRule<Movie> rule =
    new LinearCombinationMatchingRule<>(-1.497, 0.5);
rule.addComparator(new MovieTitleComparator(), 1.849);
rule.addComparator(new MovieDateComparator(), 0.822);
```
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
...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