UNIVERSITY OF MANNHEIM

# Exercise: Identity Resolution

Web Data Integration

# Agenda

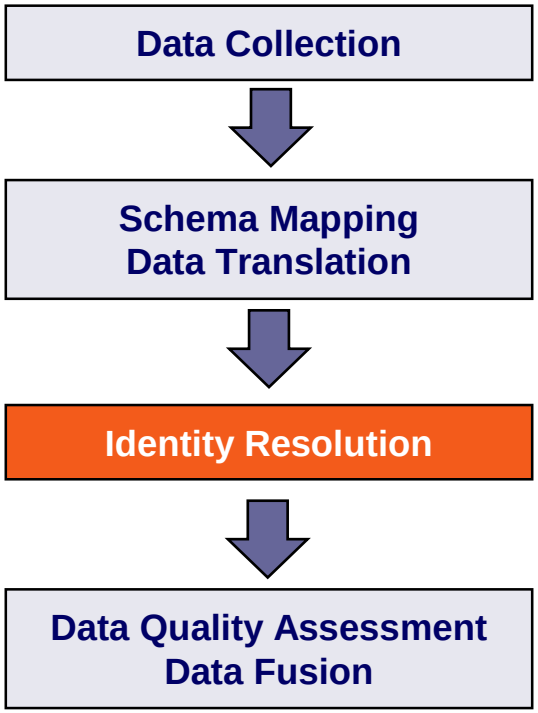
1. Exercise Overview
2. Use Case for this Exercise
3. The WInte.r Framework
    1. Loading Data
    2. Creating a Matching Rule
    3. Running the Identity Resolution
    4. Evaluating the Matching Result
    5. Learning a Matching Rule
4. Hands-on: Tasks of the Exercise

# 1. Exercise Overview

## Learning goal

**Learn how to use the WInte.r Framework in order 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 the quality of your approach (F1 / Reduction Ratio)




## Result

1. Correspondences between records in different data sets that describe the same entity
2. A well-founded idea about the quality of the correspondences

# 2. Use Case for this Exercise

1. Download the .zip of the project from the course page
2. Unzip it and look at the sample files in \data\input\
   - .xml input data sets in input folder
   - .csv gold standard
3. Open the project in a Java IDE (import as maven project)

The project serves as a quick-start for todays tasks and contains:

- Two datasets describing movies
- A gold standard for evaluating correspondences between records
- Pre-implemented classes and a data model for movies
- A fully implemented identity resolution workflow using WInte.r
- Several alternative blocking functions and comparators

# Explore the Data

- We provide you with the following files:
  - actors.xml: dataset with 149 movies
  - academy_awards.xml: dataset with 4580 movies
- Vocabularies are aligned
  - Every input file has the same schema
- Unique IDs are in place
  - The IDs are **globally** unique
- Which combination of attributes might be suitable to detect duplicates?
  - movie title, movie date, list of actors, director name …

# Example

- Example of a movie in both datasets

```
<movies>
        <movie>
          <id>actors_104</id>
          <title>Stalag 17</title>
          <date>1954-01-01</date>
          <actors>
            <actor>
                  <name>William Holden</name>
                  <birthday>1918-01-01</birthday>
                  <birthplace>Illinois</birthplace>
                </actor>
          </actors>
        </movie>
    ...
</movies>
```

(i) actors dataset

```
<movies>
        <movie>
          <id>academy_awards_3059</id>
          <title>Stalag 17</title>
          <date>1953-01-01</date>
          <oscar>yes</oscar>
          <director>
                <name>Billy Wilder</name>
          </director>
          <actors>
            <actor>
                  <name>Robert Strauss</name>
                </actor>
          </actors>
        </movie>
    ...
</movies>
```

(ii) academy_awards dataset

# The Gold Standard

- To evaluate identity resolution algorithms, you need a manually verified **gold standard**
  - .csv file containing pairs of (comma-separated) IDs of entities that match and do not match

```
gs_academy_awards_2_actors_test.csv:

academy_awards_4529,actors_2,TRUE
academy_awards_4500,actors_3,TRUE
academy_awards_4421,actors_82,FALSE
academy_awards_4475,actors_4,TRUE
academy_awards_3305,actors_21,FALSE
```

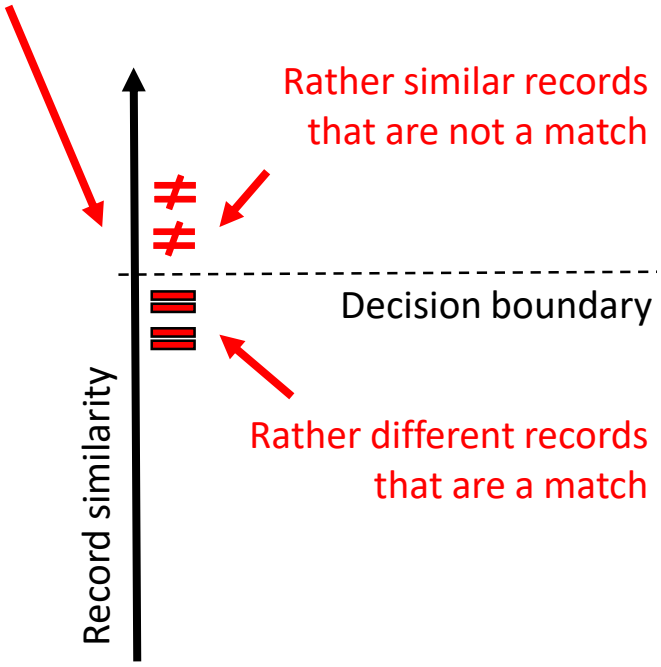
- The file includes non-trivial cases

```
<movie>
    <id>actors_141</id>
    <title>Forest Gump</title>
    <actors>
        <actor>
            <name>Tom Hanks</name>
            <birthday>1956-01-01</birthday>
            <birthplace>California</birthplace>
        </actor>
    </actors>
    <date>1995-01-01</date>
</movie>
```

```
<movie>
    <id>academy_awards_902</id>
    <title>Forrest Gump</title>
    <director>
        <name>Robert Zemeckis</name>
    </director>
    <actors>
        <actor>
            <name>Tom Hanks</name>
        </actor>
        <actor>
            <name>Gary Sinise</name>
        </actor>
    </actors>
    <date>1994-01-01</date>
    <oscar>yes</oscar>
</movie>
```

# The Gold Standard

- You need ground truth (gold standard) for the evaluation
- To create a gold standard, manually label a set of record pairs as matches or non-matches including corner cases
- Rule of thumb for creating a *suitable* gold standard with *acceptable* manual effort:
  1. match records using several simple matching techniques (goal: avoid selection bias) and sort record pairs according to their similarity
  2. if available, use information about likely matches (e.g. ISBN or GTIN numbers that exist in multiple sources)
  3. manually verify a fair amount of the resulting pairs (e.g. >500 pairs) including
     1. matching record pairs (randomly chosen, 20% of GS)
     2. corner case matches and non-matches (30% of GS)
     3. non-matching record pairs (randomly chosen, 50% of GS)



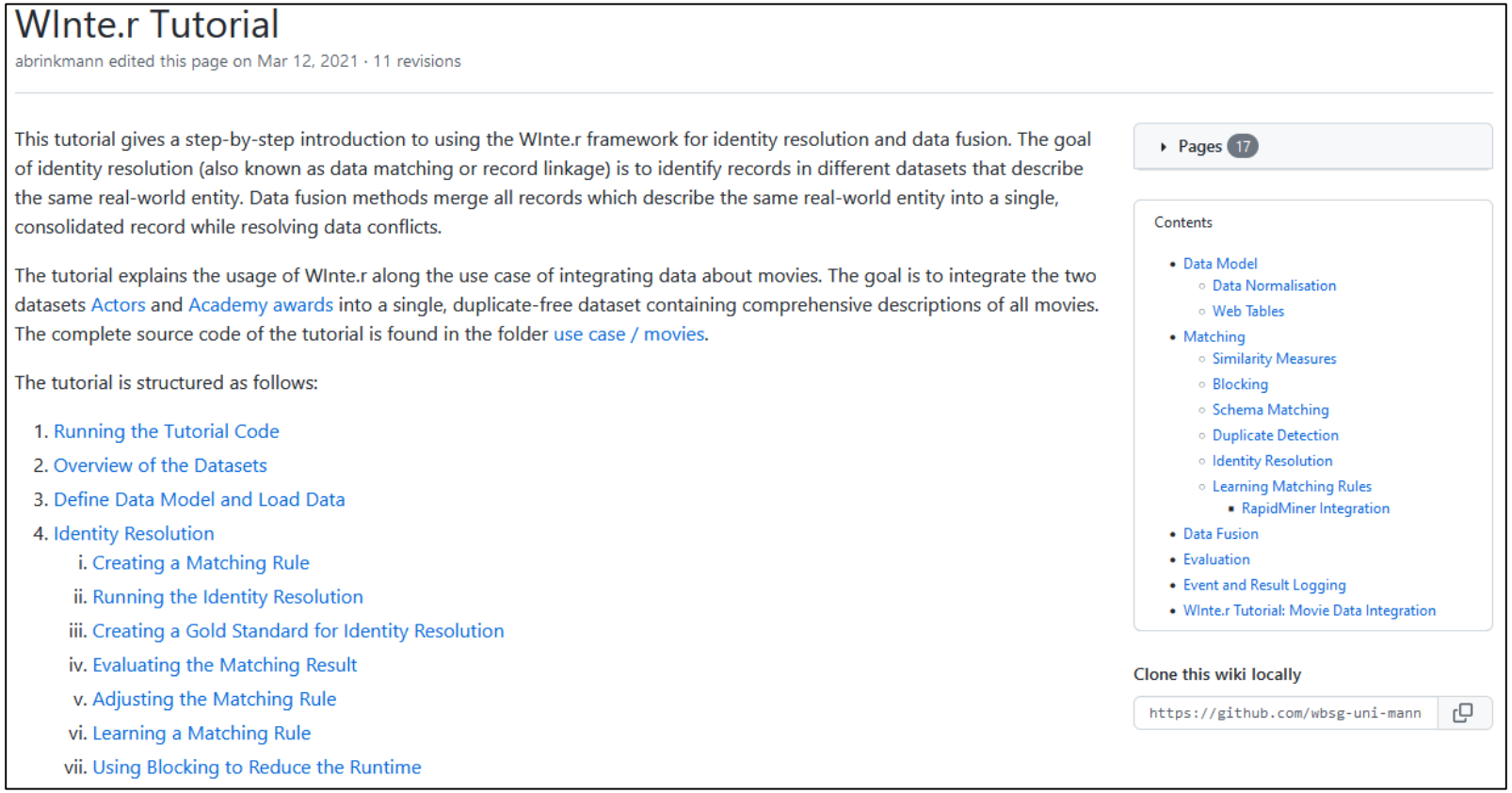
# 3. The WInte.r Framework

- The **W**eb Data **Integr**ation Framework (WInte.r) provides methods for end-to-end data integration
- Implements methods for
  - Data Pre-Processing
  - Schema Matching
  - Identity Resolution
  - Data Fusion
  - Evaluation

- Open Source under Apache 2.0 License
- https://github.com/wbsg-uni-mannheim/winter

# WInte.r Tutorial

https://github.com/wbsg-uni-mannheim/winter/wiki/WInte.r-Tutorial



## WInte.r Tutorial

abrinkmann edited this page on Mar 12, 2021 · 11 revisions

This tutorial gives a step-by-step introduction to using the WInte.r framework for identity resolution and data fusion. The goal of identity resolution (also known as data matching or record linkage) is to identify records in different datasets that describe the same real-world entity. Data fusion methods merge all records which describe the same real-world entity into a single, consolidated record while resolving data conflicts.

The tutorial explains the usage of WInte.r along the use case of integrating data about movies. The goal is to integrate the two datasets Actors and Academy awards into a single, duplicate-free dataset containing comprehensive descriptions of all movies. The complete source code of the tutorial is found in the folder use case / movies.

The tutorial is structured as follows:

1. Running the Tutorial Code
2. Overview of the Datasets
3. Define Data Model and Load Data
4. Identity Resolution
   i. Creating a Matching Rule
   ii. Running the Identity Resolution
   iii. Creating a Gold Standard for Identity Resolution
   iv. Evaluating the Matching Result
   v. Adjusting the Matching Rule
   vi. Learning a Matching Rule
   vii. Using Blocking to Reduce the Runtime

Pages 17

Contents

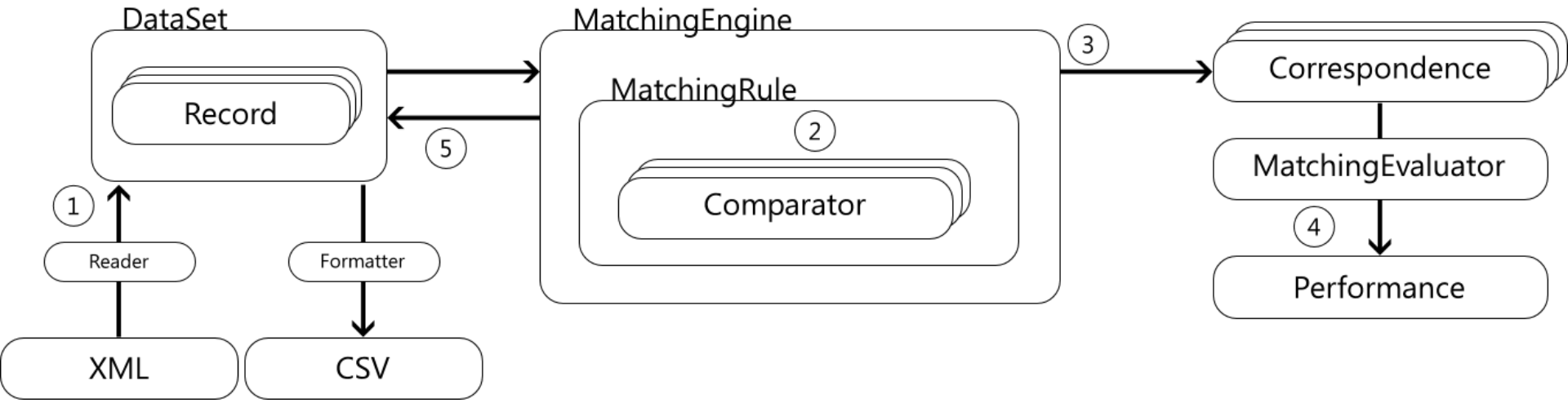
- Data Model
  - Data Normalisation
  - Web Tables
- Matching
  - Similarity Measures
  - Blocking
  - Schema Matching
  - Duplicate Detection
  - Identity Resolution
  - Learning Matching Rules
    - RapidMiner Integration
- Data Fusion
- Evaluation
- Event and Result Logging
- WInte.r Tutorial: Movie Data Integration

Clone this wiki locally

https://github.com/wbsg-uni-mann

# 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



# 3.1 Loading Data: Define the Data Model

- First Step: Define your data model!
  - Create Java classes for your entities
  - Implement the *Matchable* interface

```
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 void setTitle(String title) {
  this.title = title;
 }
...
}
```

- The data models for this exercise are already implemented in Movie.java and Actor.java in the package:

  de.uni_mannheim.informatik.dws.wdi.ExerciseIdentityResolution.model

# Loading Data: Create an XML File Reader

- Second Step: Define how to load your model from XML files
  - Extend the *XMLMatchableReader* class
  - Override the createModelFromElement method
    - Creates a new movie instance

```
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*

```
getValueFromChildElement(node, "title");
```

```
getListFromChildElement(node,"director");
```

```
getObjectListFromChildElement(
 node,
 "actors",
 "actor",
 new ActorXMLReader(),
 provenanceInfo);
```

```
<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>
```

# Loading Data: Load an XML file

- Third Step: Load the provided data sets
  - Create a new *HashedDataSet* object
  - Specify
    - The file that contains your data
    - The XPath to the XML elements that represent your records

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

// load an XML file
new MovieXMLReader().loadFromXML(
 new File("data/input/academy_awards.xml"),  // the file to load
 "/movies/movie",                            // XPath to elements
 ds);                                        // data set to fill
```

# ALTERNATIVE Loading Data: The Default Model

- Alternative to creating customized data models:

  Use the Default Model for a simple schema
  - `de.uni_mannheim.informatik.dws.winter.model.defaultmodel.*`
  - A key/value map supporting atomic values and lists
  - Data is modelled using the *Record* and *Attribute* classes

```
HashedDataSet<Record, Attribute> ds = new HashedDataSet<>();

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

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

- More info about data management in the WInte.r tutorial at:
  https://github.com/wbsg-uni-mannheim/winter/wiki/Data-Model

# 3.2 Creating a Matching Rule

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

$$sim(x,y) = 0.3s_{name}(x,y) + 0.3s_{phone}(x,y) + 0.1s_{city}(x,y) + 0.3s_{state}(x,y)$$

$s_{name}(x,y)$: using the Jaro-Winkler similarity measure

$s_{phone}(x,y)$: based on edit distance between x's phone (after removing area code) and y's phone

$s_{city}(x,y)$: based on edit distance

$s_{state}(x,y)$: based on exact match; yes ➔ 1, no ➔ 0

- ***SimilarityMeasure***s specify the similarity of two values
- ***Comparator***s specify how to compare the values of attributes
- ***MatchingRule***s specify how to combine the different similarity values

# Creating a Matching Rule: Similarity Measures

- A similarity measure calculates a similarity between two values
  - WInte.r provides different similarity measures for string, numeric, date and list attribute
    - More Info at: https://github.com/wbsg-uni-mannheim/winter/wiki/SimilarityMeasures
  - Extends the SimilarityMeasure class
  - Accepts two values and returns their similarity

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

 @Override
 public double calculate(String first, String second) {
  if(StringUtils.isEmpty(first) || StringUtils.isEmpty(second))
    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
- Creating attribute comparators:
  1. apply specific preprocessing
     - lower-case the values
  2. calculate similarity value
     - Use Jaccard Similarity
  3. re-scale the similarity value
     - square similarity

```
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

- Defining a Matching Rule:
  - Use the LinearCombinationMatchingRule class
  - Specify final threshold
  - Add comparators and their 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
```

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

$$match_{Movie}(m_1, m_2) = \begin{cases} 1 & \text{if } sim_{Movie}(m_1, m_2) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

# 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 be evaluated by the matching rule

- Example: use the decade of a movie‘s release as blocking key
  - Extend RecordBlockingKeyGenerator
  - Override generateBlockingKeys(…)

```
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)
          );
 }
}
```

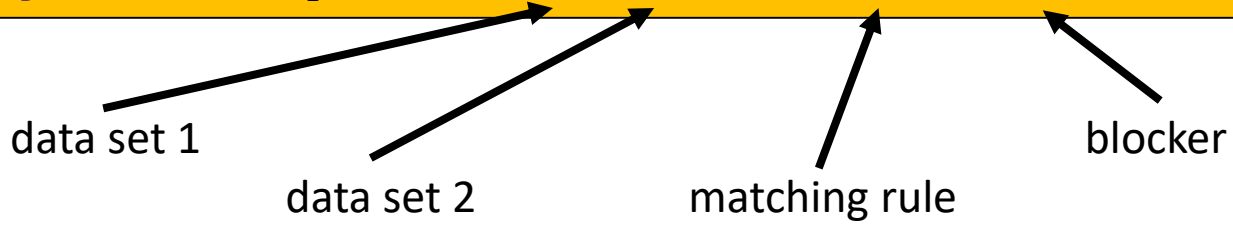
# 3.3 Running the Identity Resolution

- Create a MatchingEngine instance and run the identity resolution

```
// 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);
```




- Write the resulting correspondences into a file

```
new CSVCorrespondenceFormatter().writeCSV(new
File("usecase/movie/output/academy_awards_2_actors_correspondences.csv"), correspondences);
```

# 3.4 Evaluating the Result

- First Step: Load the gold standard
  - Use the *MatchingGoldStandard* class
- Second step: evaluate the result
  - Use the *MatchingEvaluator* class

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

// evaluate the result
MatchingEvaluator<Movie, Attribute> evaluator = new MatchingEvaluator<>();
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()));
```

# Inspect and Improve the Results – Event Logging

- Winte.r supports detailed event logging which can help you find what went wrong and improve your results
- Default logging level: info on input data, gold standard, # blocks, results
- Trace (tracefile) logging level: default **+** blocking keys and frequency, missing, wrong and correct correspondences

```
public class Logger logger = WinterLogManager.activateLogger("trace");
```

```
[correct] academy_awards_1272,actors_135,0.85
[wrong] academy_awards_1115,actors_101,0.7
[missing] academy_awards_4270,actors_9
...
```

```
Blocking key values:
BlockingKeyValueFrequency
MA,MO   24
CA      21
...
```

# Inspect and Improve your Results – Result Logging

Get more detailed logging by activating the debug reports for:

- Blocking: overview of key values and their frequencies
- Matching: input values, pre-processed values, similarity scores, post-processed similarity scores

Activate the *Blocking* report before running blocking:

```
// create & configure the blocker
Blocker<Movie, Attribute> blocker = new StandardRecordBlocker<>(new MovieBlockingFunction());
blocker.collectBlockSizeData("data/output/debugResultsBlocking.csv", 100);
```

Activate the Matching report after loading your gold standard and before adding comparators to your matching rule:

```
// create & configure the blocker
LinearCombinationMatchingRule<Movie, Attribute> rule = new LinearCombinationMatchingRule<>(0.5);
rule.activateDebugReport("data/output/debugResultsMatchingRule.csv", 1000, gs);
```

# Inspect and Improve your Results - Example

```
LinearCombinationMatchingRule<Movie, Attribute> rule =
  new LinearCombinationMatchingRule<>(0.7);
rule.addComparator(new MovieTitleComparator(), 0.2);
rule.addComparator(new MovieDateComparator(), 0.8);
```

```
*       Evaluating result
*
Academy Awards <-> Actors
Precision: 1.0000
Recall: 0.0213
F1: 0.0417
```

## Matching Rules Debug Report

| Total Similarity | Is Match | MovieDateComparator 2Yearsrecord1 PreprocessedValue | MovieDateComparator 2Yearsrecord2 PreprocessedValue | MovieDateComparator 2Years postproccesed Similarity | MovieTitleComparator Jaccardrecord1 PreprocessedValue | MovieTitleComparator Jaccardrecord2 PreprocessedValue | MovieTitleComparator Jaccard postproccesed Similarity |
|---|---|---|---|---|---|---|---|
| 0.6 | **1** | 1978-01-01T00:00 | 1979-01-01T00:00 | **0.5** | Coming Home | Coming Home | **1** |
| 0.6 | **1** | 1952-01-01T00:00 | 1953-01-01T00:00 | **0.5** | High Noon | High Noon | **1** |
| 0.6 | **1** | 1955-01-01T00:00 | 1956-01-01T00:00 | **0.5** | Marty | Marty | **1** |
| 0.6 | **1** | 1948-01-01T00:00 | 1949-01-01T00:00 | **0.5** | Johnny Belinda | Johnny Belinda | **1** |

We inspect the debug report. The MovieDate attribute is noisy while MovieName is cleaner and should stronger affect the matching decision.

```
LinearCombinationMatchingRule<Movie, Attribute> rule =
  new LinearCombinationMatchingRule<>(0.7);
rule.addComparator(new MovieTitleComparator(), 0.8);
rule.addComparator(new MovieDateComparator(), 0.2);
```

```
*       Evaluating result
*
Academy Awards <-> Actors
Precision: 0.9333
Recall: 0.8936
F1: 0.9130
```

# 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

```
// create the matching rule
String options[] = new String[] { "-S" };
String modelType = "SimpleLogistic"; // use a logistic regression
WekaMatchingRule<Movie, Attribute> matchingRule
 = new WekaMatchingRule<>(0.5, 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

- The *WekaMatchingRule* class can be parametrized with **linear** (logistic regression) and **non-linear** models (decision tree).

```
// create the matching rule
String options[] = new String[1];
options[0] = "-U";
String modelType = "J48"; // use a tree classifier
WekaMatchingRule<Movie, Attribute> matchingRule
 = new WekaMatchingRule<>(0.5, tree, options);
```

- Documentation on Weka classifiers:

http://weka.sourceforge.net/doc.dev/weka/classifiers/Classifier.html

# Project Phase 2

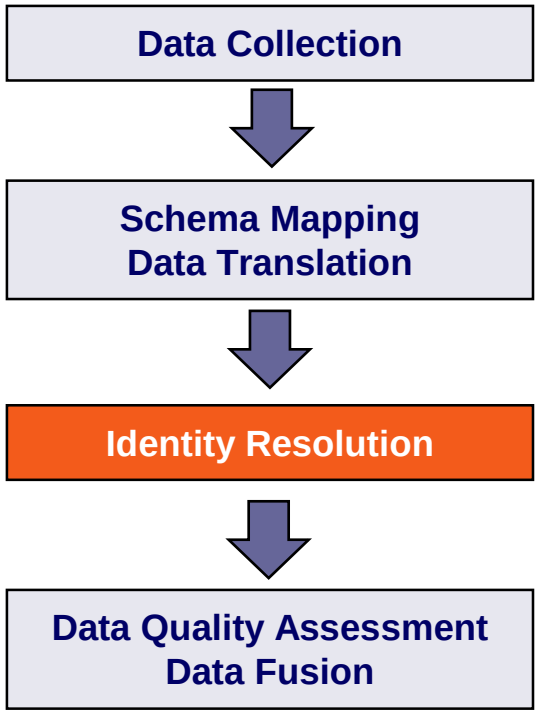
## Project Phase 2: Identity Resolution

Duration: October 12th – November 9th

Tasks: Use WInte.r 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 the quality of your approach (F1 / Reduction Ratio)

Result: Correspondences between records in different data sets that describe the same entity



# 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

- Make gold standard big enough
  - At least 1% (or 500 pairs, if your datasets are huge) of entities
- You need a gold standard for all data sets that you want to use in the fusion part
  - 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)

# Identity Resolution in the Final Project Report

- Results of Phase 2 will be part of your final report
- Make sure you 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 Phase 2 output is Exercise 3 (Data Fusion) input

# Final Report: Tables describing your Experiments

1. Please add the following table to your final report and presentation slides:

| # | Matching Rule | Blocker | P | R | F1 | # Corr | Time |
|---|---|---|---|---|---|---|---|
| 1 | Rule1:Title&Year | No Blocking | 0.71 | 0.95 | 0.82 | 10.230 | 90 min |
| 2 | Rule1:Title&Year | StandardYear | 0.71 | 0.73 | 0.72 | 9.609 | 18 sec |
| 3 | Rule1:Title&Year | SNBYear | 0.71 | 0.89 | 0.79 | 10.215 | 50 sec |
| 4 | Rule2:Title&Actors | SNBYear | 0.81 | 0.89 | 0.83 | 9.919 | 19 sec |

2. Please also report the group size distribution:

```
CorrespondenceSet<Movie, Attribute> correspondences = …
correspondences.printGroupSizeDistribution();
```

```
Group Size      | Frequency
-----------------------------
        2       |       43
        3       |       103
        4       |       2
```

More details in the lecture on data fusion!

# Task for this Exercise

1. Open and run the provided *IR_using_linear_combination.java* in *de.uni_mannheim.informatik.dws.wdi.ExerciseIdentityResolution*
   1. Which performance does the linear combination rule achieve?
2. Understand your results:
   1. Inspect the log files in *data\output* to see which errors were made
3. Try different combinations of comparators or weights in your matching rule
   1. Can you improve the performance?
   2. Can you improve the performance using global matching?
4. 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*
5. Use machine learning (*IR_using_machine_learning.java*)
   1. Which performance does the machine learning rule achieve?
   2. Create a comparator that uses the actors

# ...and now

1. Prepare the gold standard
2. Use WInte.r and
   - Define your inputs
   - Define blocking functions
   - Define your matching rules
   - Run the evaluation
   - (extra) Learn matching rules