UNIVERSITÄT MANNHEIM

Web Data Integration: Exercise 2, Identity Resolution

Christian Bizer, Heiko Paulheim, Volha Bryl

Exercise 2, Identity Resolution

- **Agenda**
  - Exercise overview
  - Preparing the inputs
    - Check your data
    - Create gold standard
  - Template project structure
    - Load your data
    - Experiment with matching functions
    - Use blocking
    - Evaluate results
    - (Extra task) Learn matching rules

- **Timing: October 24th – November 7th**

Exercise Overview

- In this exercise you will experiment with
  - matching functions and their combinations
  - blocking keys
  - evaluation metrics
  - learning

- Your task is to extend a template Eclipse Java project
  - Using resources introduces in lectures
    - *SecondString* Library for similarity metrics
    - *Xpath/JAXP* for working with XML input
  - For the extra task, rule learning, you will use RapidMiner

- …but first, look at your data.

Do you know you 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 50% of the instances should be in at least two datasets
  - At least 50% of the attribute values should be in at least two datasets
- What to use to detect duplicates in your use case?
  - Name/title, creation/founding date, location/ address, height, color, …

```
<movie>
        <id>1-9311</id>
        <title>Winter's Bone</title>
</movie>
<movie>
        <id>1-9312</id>
        <title>Black Swan</title>
        <date>2011-01-01</date>
        <globe>yes</globe>
</movie>
<movie>
        <id>1-9313</id>
        <title>Blue Valentine</title>
        …
```

Prepare Gold Standard

- To evaluate identity resolution algorithms, you need a **gold standard**
  - .cvs file containing pairs of (comma-separated) IDs of entities that match
- You have to create it **manually**
- Include **non-trivial cases**
  - Movies “Godfather, part 3, The” and “The Godfather, III”
  - Scientists Albert Einstein, A. Einstein and Einstein, Albert
  - …

gold.cvs:

1-9309,2-9309
1-9310,2-9310
1-9311,2-9311
1-9312,2-9312
1-9313,2-9313
1-9314,2-9314
1-9315,2-9315
1-9316,2-9316

Prepare Gold Standard

- Make it **big enough**
  - At least 1% (or 100 pairs, if your datasets are huge) of entities
- You should have a gold standard file for all pairs of datasets
  - …but we understand it is not feasible
  - …so **select the biggest and the most interesting dataset pairs**
- Proceed iteratively
  - Create a smaller gold standard, go through the whole exercise, then come back to improve the gold standard

- Important assumption
  - If “1,2” is the only pair in your gold standard containing “1”, we assume you’ve checked there are no other duplicates of 1 in your data
    - Might look obscure, but will become clear later in the exercise

Prepare Gold Standard

- Example of a **bad** decision on a pair of datasets for creating a gold standard
  - Not much intersection of attributes – just titles

Dataset 1:

```
<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>
```

Dataset 2:

```
<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>
```

Prepare Gold Standard

- Example of a **good** decision on a pair of datasets for creating a gold standard
  - 3 attributes to experiment with: title, director, date

Dataset 1:

```
<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>
```

Dataset 2:

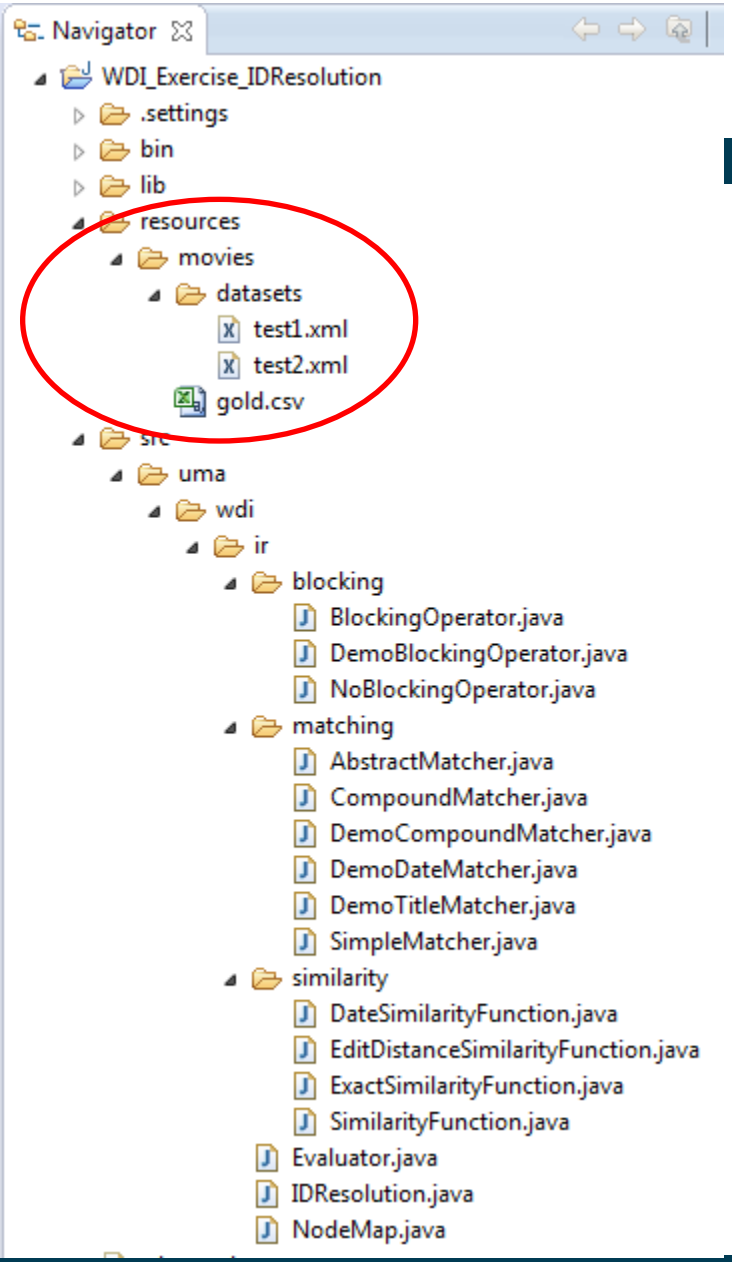
```
<movie>
        <title>Black Swan</title>
        <director>
                <name>Aronofsky, Darren</name>
        </director>
        <date>2011-01-01</date>
</movie>
<movie>
        <title>Social Network, The</title>
        <director>
                <name>Fincher, David</name>
        </director>
        <date>2011-01-01</date>
        <globe>yes</globe>
</movie>
```

Start with the Template Project

- Download the .zip of the project from the course page
  - http://dws.informatik.uni-mannheim.de/en/teaching/courses-for-master-candidates/ie-670-web-data-integration/
- Unzip it and look at the sample input files in **\resources\movies**
  - .xml input datasets in **datasets** folder
  - .cvs gold standard
- Open the project in **Eclipse**

Start with the Template Project

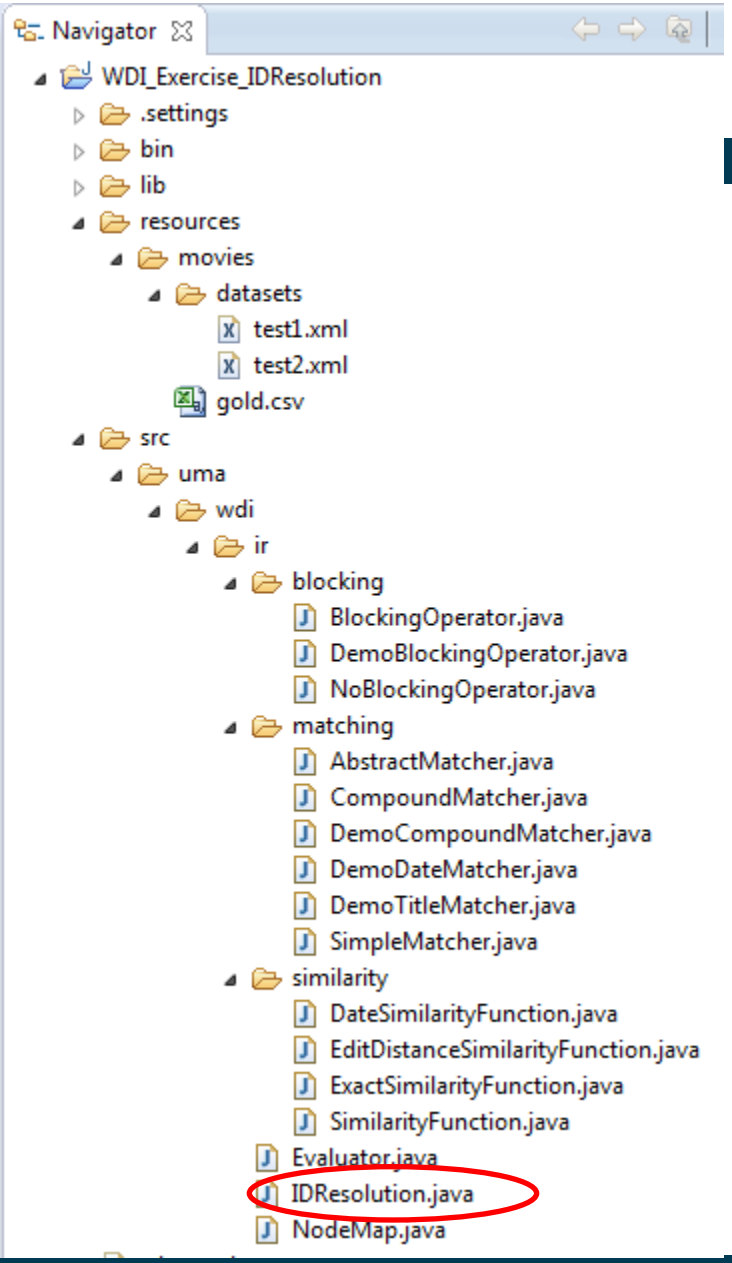
- Download the .zip of the project from the course page
  - http://dws.informatik.uni-mannheim.de/en/teaching/courses-for-master-candidates/ie-670-web-data-integration/
- Unzip it and look at the sample input files in **\resources\movies**
  - .xml input datasets in **datasets** folder
  - .cvs gold standard
- Open the project in **Eclipse**
- **We have implemented for you**
  - Loading/storing input datasets and gold standard
  - Infrastructure for matching 2 datasets and calculating evaluation metrics
  - Examples of matchers, similarity metrics and blocking keys
  - Output of the results, preparing data for RapidMiner



Template Project Structure

- **Input**
  - **Datasets (xml) and gold standard (cvs)**
- Blocking keys
- Matchers:
  - Match 2 things (2 xml nodes)
  - Simple and compound
- Similarity measures
  - Compare 2 strings, dates, numbers
- Match all and calculate evaluation metrics
- Main class : start from here
- Store and print data

Template Project Structure



- Input
  - Datasets (xml) and gold standard (cvs)
- Blocking keys
- Matchers:
  - Match 2 things (2 xml nodes)
  - Simple and compound
- Similarity measures
  - Compare 2 strings, dates, numbers
- Match all and calculate evaluation metrics
- **Main class : start from here**
- Store and print data

Define your inputs

- ***Class IDResolution***
  - In the ***main()*** function you specify
    1. xpath to unique object IDs
    2. paths to input .xml files and .csv gold standard file
    3. which blocking key to use
    4. which matcher to use

```
String fnGold = "resources/movies/gold.csv";
String fnDataset1 = "resources/movies/datasets/test1.xml";
String fnDataset2 = "resources/movies/datasets/test2.xml";
String idPath = "/movies/movie/id";
runEvaluation(fnDataset1, fnDataset2, idPath, fnGold,
     new DemoBlockingOperator(), new DemoTitleMatcher(), true);
```

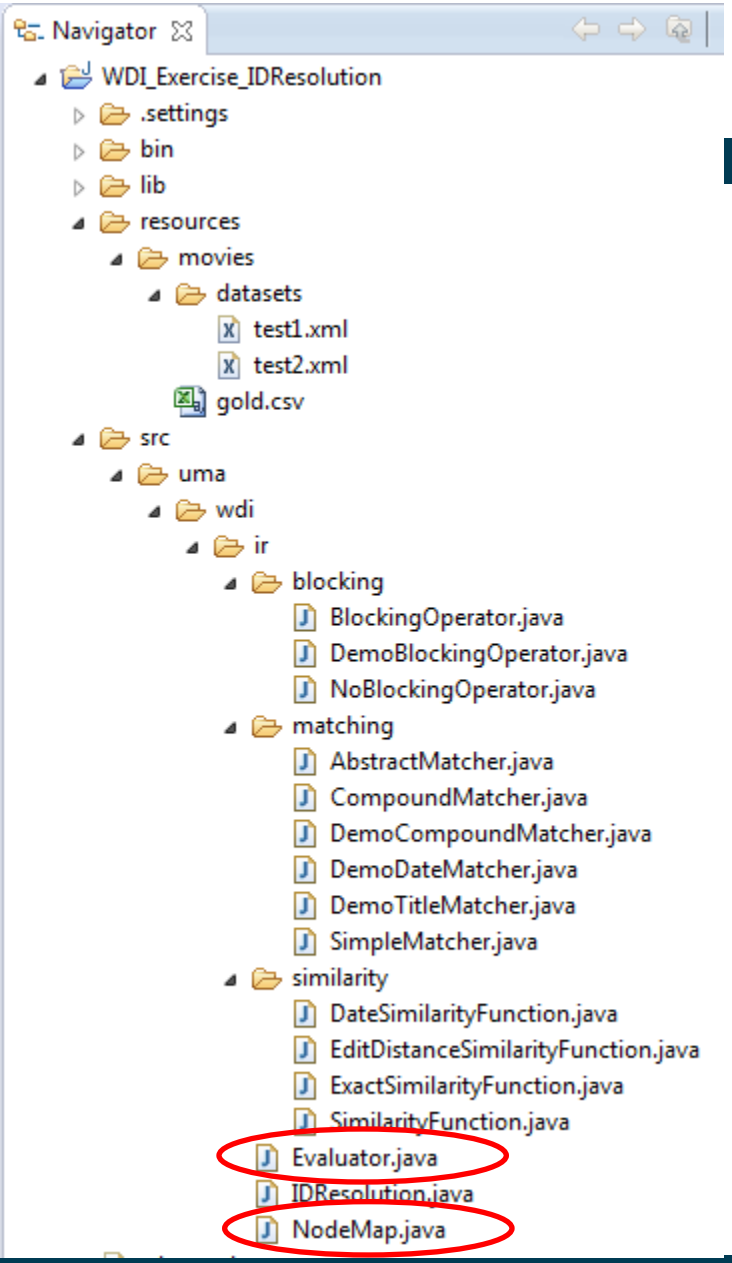
2

1

3

4

Template Project Structure



- Input
  - Datasets (xml) and gold standard (cvs)
- Blocking keys
- Matchers:
  - Match 2 things (2 xml nodes)
  - Simple and compound
- Similarity measures
  - Compare 2 strings, dates, numbers
- **Match all and calculate evaluation metrics**
- Main class : start from here
- **Store and print data**

What you don't need to implement

- *class NodeMap*
  - Loads your xml data in a hash map of xml nodes, node ←→ ID
- *class Evaluator*
  - Loads .cvs gold standard
  - Calculates matching scores for all pairs of entities
  - Computes and outputs P/R/F1/runtime/num-of-matching-operations

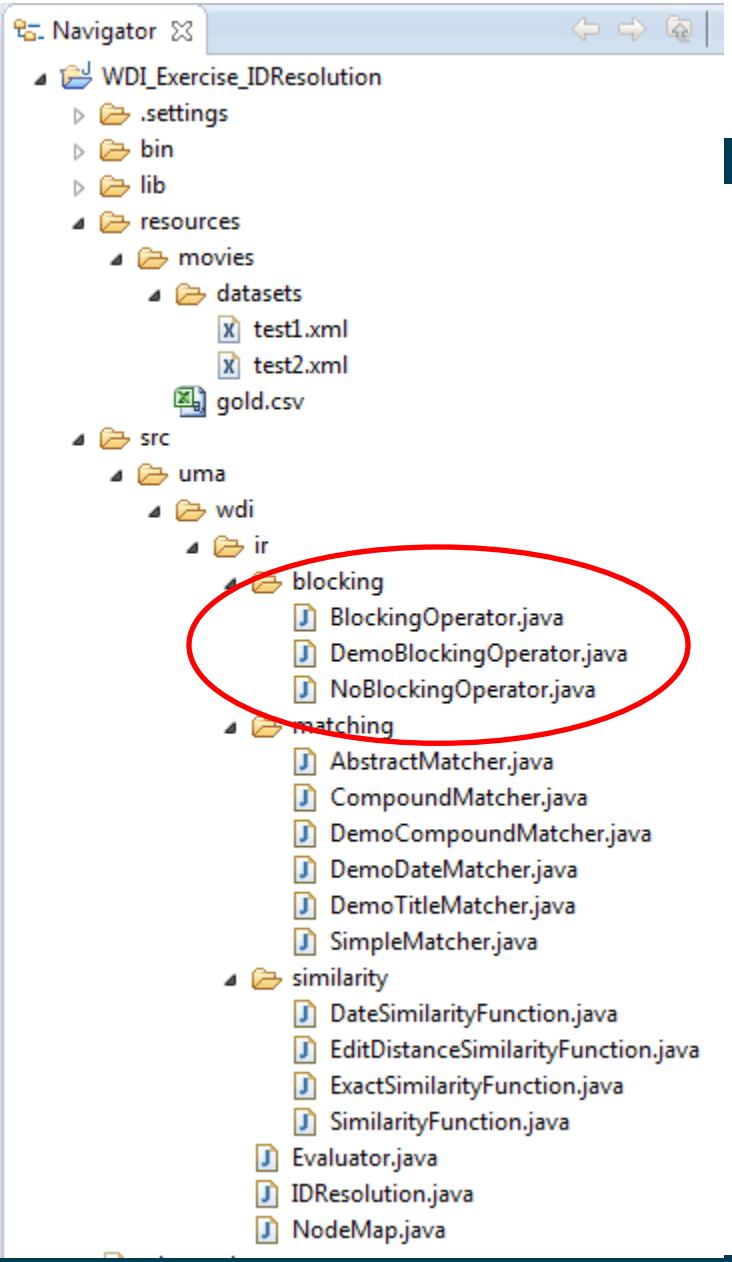
```
String fnGold = "resources/movies/gold.csv";
String fnDataset1 = "resources/movies/datasets/test1.xml";
String fnDataset2 = "resources/movies/datasets/test2.xml";
String idPath = "/movies/movie/id";
runEvaluation(fnDataset1, fnDataset2, idPath, fnGold,
     new DemoBlockingOperator(), new DemoTitleMatcher(), true);
```

What you need to implement

- **Blocking operator**
  - Defines how a blocking key is constructed
- **Matchers**
  - Simple: matches values defined by xpath and 2 nodes
  - Compound: e.g. linear combinations of simple matchers

```
String fnGold = "resources/movies/gold.csv";
String fnDataset1 = "resources/movies/datasets/test1.xml";
String fnDataset2 = "resources/movies/datasets/test2.xml";
String idPath = "/movies/movie/id";
runEvaluation(fnDataset1, fnDataset2, idPath, fnGold,
     new DemoBlockingOperator(), new DemoTitleMatcher(), true);
```

Template Project Structure



- Input
  - Datasets (xml) and gold standard (cvs)
- **Blocking keys**
- Matchers:
  - Match 2 things (2 xml nodes)
  - Simple and compound
- Similarity measures
  - Compare 2 strings, dates, numbers
- Match all and calculate evaluation metrics
- Main class : start from here
- Store and print data

Define blocking key operator

- If you don't need blocking, use *NoBlockingOperator*
- See *class DemoBlockingOperator* for an example
  - Uses movie creation year with last digit → 0 (e.g 1910, 1980, 2010)

```
// no blocking
runEvaluation(fnDataset1, fnDataset2, idPath, fnGold,
      new NoBlockingOperator(), new DemoTitleMatcher(), true);
// use blocking
runEvaluation(fnDataset1, fnDataset2, idPath, fnGold,
      new DemoBlockingOperator(), new DemoTitleMatcher(), true);
```

- What blocking key to use?
  - **Look at your data**
  - Examples: country, type, first digits of a zip-code, first letters of a title, …
  - Experiment with different operators, compare recall/precision versus running time/number of matching operations

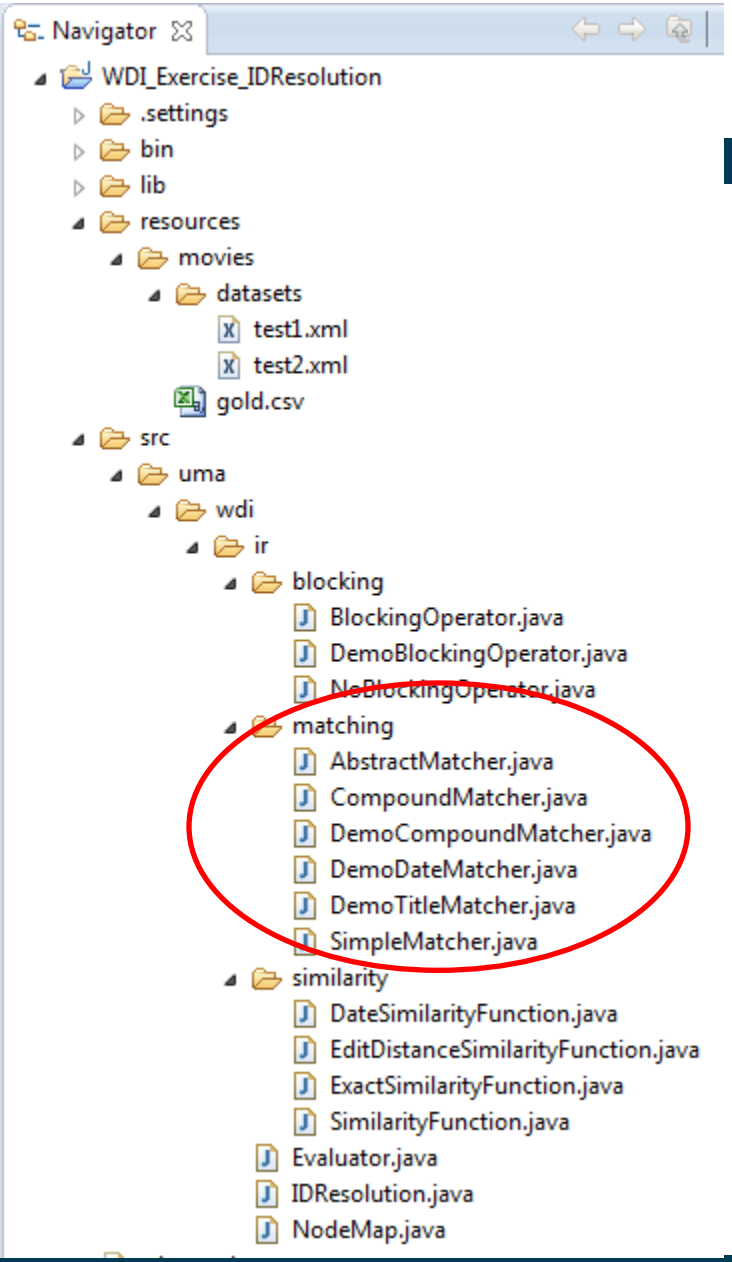
Define blocking key operator

- Your operator should implement *BlockingOperator interface*
- Create a new class (e.g. copy-pasting *DemoBlockingOperator*)
- Put your code for creating a blocking key in g*etBlockingKey()*

```
public class YourBlockingOperator implements BlockingOperator
{
     private XPathExpression blockingKeyXPath;
     private Map<Node,String> cache = new HashMap<Node,String>();
     …
     public String getBlockingKey(Node node)
     {
          if(cache.containsKey(node)) return cache.get(node);
          ...
          return blk;
     }
}
```

your code here

Template Project Structure



- Input
  - Datasets (xml) and gold standard (cvs)
- Blocking keys
- **Matchers:**
  - **Match 2 things (2 xml nodes)**
  - **Simple and compound**
- Similarity measures
  - Compare 2 strings, dates, numbers
- Match all and calculate evaluation metrics
- Main class : start from here
- Store and print data

Define your matching strategy

- Decide which attributes to compare and how
  - **Look at your data**
- Add new similarity functions
- Look at the examples we provided for you
  - *DemoDateMatcher* and *DemoTitleMatcher*
- Your matcher should extend *class SimpleMatcher*

your simple matching strategy here

```
public class DemoTitleMatcher extends SimpleMatcher{
     public DemoTitleMatcher() {
          super("title",
               new LevenshteinSimilarityFunction(),0.5);
     }
}
```

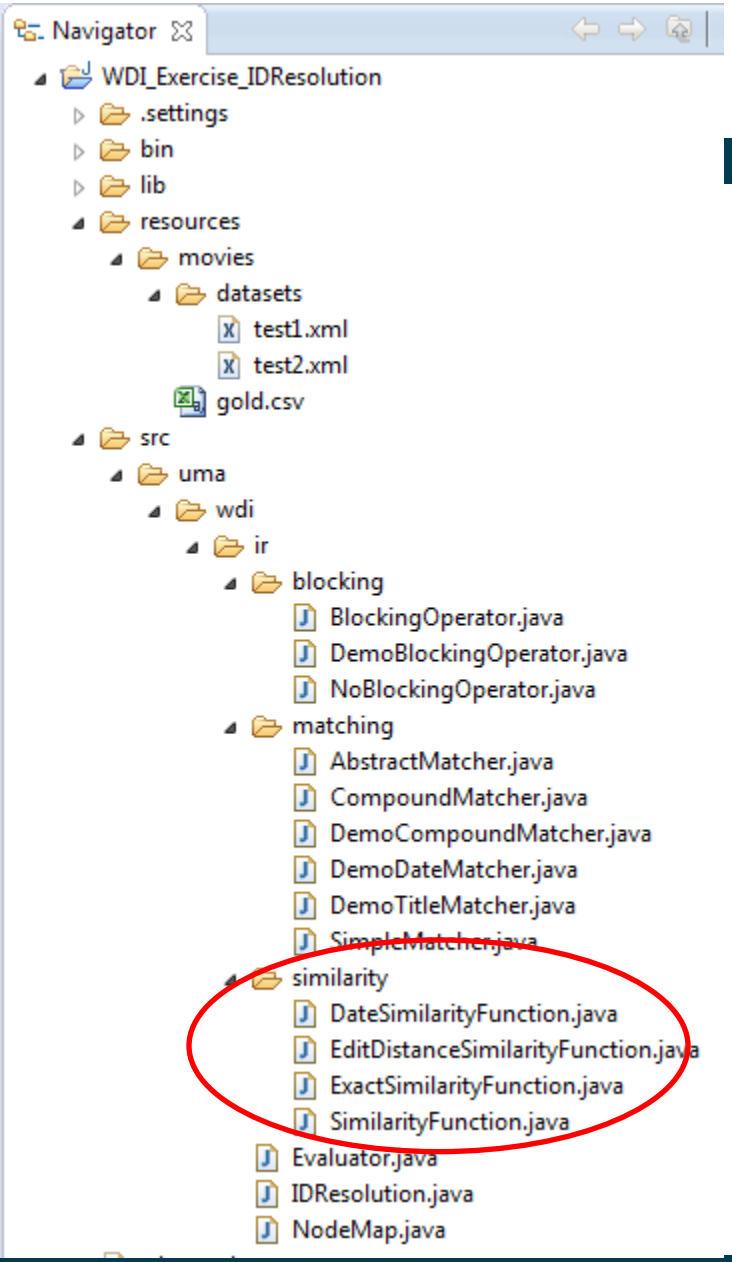
Define your matching strategy

- Matching strategy can involve
  - comparing several attributes
  - combining results (as a weighted sum)
  - or applying rules (“if titles are similar, compare dates”)
- Extend *class CompoundMatcher* to experiment with linear combinations of simple matchers
  - See *DemoCompoundMatcher* for an example

define matchers and weights here

```
public class DemoCompoundMatcher extends CompoundMatcher {
     public DemoCompoundMatcher() {
     AbstractMatcher m1 = new DemoTitleMatcher();
     double w1 = 9;
     AbstractMatcher m2 = new DemoDateMatcher();
      double w2 = 1;
     …
}
```

Template Project Structure



- Input
  - Datasets (xml) and gold standard (cvs)
- Blocking keys
- Matchers:
  - Match 2 things (2 xml nodes)
  - Simple and compound
- **Similarity measures**
  - **Compare 2 strings, dates, numbers**
- Match all and calculate evaluation metrics
- Main class : start from here
- Store and print data

Define your similarity functions

- Your similarity function should
  - **Compute similarity score between two strings**
  - Return 0 for total dissimilarity, 1 for total similarity, 0<x<1 otherwise
  - Implement *SimilarityFunction* interface
- You can use metrics from *SecondString* library
- See *uma.wdi.ir.similarity* for examples
  - *DateSimilarityFunction, ExactSimilarityFunction, LevensteinSimilarityFunction*

your similarity measure here


```
public class ExactSimilarityFunction implements SimilarityFunction
{
      public double compare(String s1, String s2) {
            return s1.equalsIgnoreCase(s2) ? 1.0 : 0.0;
      }
}
```

Run the evaluation

- *IDResolution.runEvaluation*() outputs a number of metrics
  - **Precision, recall, F1**
    - Precision is calculated on a partial gold standard
  - **Number of matching operations**
    - Gets less with the use of blocking
  - **Runtime**
    - Gets less with the use of blocking
    - Note: to measure the runtime, do some “warming up” (run several times, take the average)

Matching by titles only
WITHOUT BLOCKING, run 1:
P = 1.0
R = 1.0
F1 = 1.0
Matching operations = 500
runtime = 764.0

WITHOUT BLOCKING, run 2:
P = 1.0
R = 1.0
F1 = 1.0
Matching operations = 500
runtime = 537.0

WITH BLOCKING:
P = 1.0
R = 0.9090909090909091
F1 = 0.9523809523809523
Matching operations = 100
runtime = 188.0

Learning an Optimal Matcher Combination

- Extra (Bonus) part of the exercise

- What you need:
  - several matchers implemented
  - RapidMiner and basic knowledge on how to use it

- What you get:
  - an optimal linear combination of your matchers

Learning an Optimal Matcher Combination

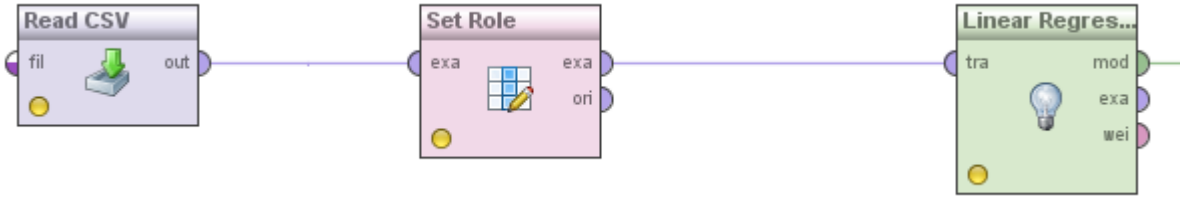
- Creating a training set
  - use the method *runWriteRegressionFile* (see last two lines in *IDResolution.main()*)

- What it does
  - writes a CSV with positive and negative examples
  - as well as the results for all the matchers

```
DemoTitleMatcher,DemoDateMatcher,score
1.0,1.0,1.0
1.0,1.0,1.0
1.0,1.0,1.0
1.0,1.0,1.0
1.0,1.0,1.0
1.0,0.6341225381761154,1.0
1.0,1.0,1.0
1.0,1.0,1.0
1.0,1.0,1.0
1.0,1.0,1.0
1.0,1.0,1.0
0.1578947368421053,0.6341225381761154,0.0
0.08695652173913049,1.0,0.0
0.3157894736842105,0.6341225381761154,0.0
0.1578947368421053,0.6341225381761154,0.0
0.23529411764705888,1.0,0.0
0.26315789473684215,0.6341225381761154,0.0
0.13043478260869568,0.6341225381761154,0.0
0.1428571428571429,1.0,0.0
0.26315789473684215,0.6341225381761154,0.0
0.26315789473684215,0.6341225381761154,0.0
0.26315789473684215,0.6341225381761154,0.0
```

Learning an Optimal Matcher Combination

- Load the data in **RapidMiner**
  - You can reuse the process **linear_regression.rmp** you find in **resources** folder
    - …but change the file path to *your* input .cvs
- Perform linear regression to create an optimal function



Learning an Optimal Matcher Combination

- Look at the results in RapidMiner

```
LinearRegression
   1.187 * DemoTitleMatcher
+ 0.202 * DemoDateMatcher
- 0.388
```

- Create an *OptimalCompoundMatcher* using those values

```
public class OptimalCompoundMatcher extends CompoundMatcher {

  public OptimalCompoundMatcher() {
    super();
    List<AbstractMatcher> matchers = Arrays.asList(new AbstractMatcher[]{new DemoTitleMatcher(),new DemoDateMatcher()});
    List<Double> weights = Arrays.asList(1.187,0.202);
    double offset = -0.388;
    setParameters(matchers, weights, 0.5, offset);
  }

}
```

Identity Resolution in the Final Report

- Results of Exercise 2 will be part of your **final report**
- Make sure your know/make notes on
  - How your created your gold standard
  - What metrics you added and tried
    - What happens with P/R/F1? And with runtime?
  - What blocking functions you tried
    - What happened with runtime and number of matches?
    - How do P/R/F1 change, and why?
  - What functions you combined and how
  - Have you learned new matching rules?

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

...and now

- Prepare the gold standard
- Get the template project and
  - Define your inputs
  - Define blocking keys
  - Define your matching strategy
  - Run the evaluation
  - (extra) Learn matching rules