NTNU
Det skapende universitet

Department of
Computer and Information Science

Report: Recognizing Textual Entailment in Java

Henning Funke & Thomas Kinnen

IT3105 - Kunstig intelligens programmering

Table of Content

1 Implementations for parts 1, 2 and 3

In the first three parts we implement several systems that determine if a hypothesis is entailed by a text.

1.1 Basic framework

We created a framework that enables us to make, use and evaluate a variety of RTE-systems in an easy way. So we have all systems of all parts of the project embedded in one big test environment. It loads the syntactically parsed data and wraps it in an object oriented structure *Text*. It is able to provide the data either as a linked graph or as a list of words.

To judge the performance of our systems we decided to implement our own evaluation code. The huge amount of file operations, that goes along with using the provided python script has a big impact on the execution time.

To achieve a common basis for evaluation and easy implementation of new systems we make use of object oriented techniques. All of our RTE-systems are derived from the interface *IEntailmentRecognizer*, that requires functionality to estimate the entailment propability of two given *Text* instances.

Further we implemented an abstract class *BasicMatcher*, that is the basis for a category of RTE-systems, that determine the entailment by comparing the sentences word by word. Thus we are able to create new systems of this category by implementing only one procedure, that chooses or calculates the desired property.

1.2 Systems in part 1

With the help of our *BasicMatcher* class it was fairly easy to create most systems of part 1. For each of *Word, Lemma and Part of Speech matching* we could use the *BasicMatcher* and extract and compare the particular part, by implementing the abstract function.

Considering that this kind of analysis ignores most of the semantics the results are quite impressive. These quite simple implementations still give better results than some pretty complicated techniques. Combined with the check for equal polarity, *Lemma matching* gives the third best results of all. We implemented IDF weighting, using the whole dataset to calculate the word quotes, but it did not improve our results.

BLEU algorithm The *BLEU* algorithm, uses n-grams largely, so we implemented the functionality to calculate them in a seperate class to be able to use it for different purposes. The *BLEU* algorithm, determines the number of shared n-grams and weights them according to the length. The descrption suggests to use 4 as maximum for n and to weigh the grams of different length by using the arithmetic mean. In our implementation we leave the maximum n as parameter and add an alternative weighing technique. By this we can use try different setups to figure out which works best.

Surprisingly the test which used a maximum n of 2, which degenerates the algorithm to a bigram matcher worked best with a result of 61.875%.

1.3 Systems in part 2

Our implementations for the Tree Edit Distance algorithm by Zhang & Shasha offer using a replaceable cost function. Thus we can implement new cost functions easily and compare the results. To verify the correctness of our implementation we use a JUnit testcase that contains an example presented in the lecture.

Entailment judgement

To judge the entailment we take every sentence from the text and calculate its distance from the hypothesis. The best score will be used to judge the entailment. The reasoning behind this is that the sentence from the text with the smallest distance d to the hypothesis supports it. With this result we tried to add 1 and invert it: $v = 1/(1 + d)$. Then the result is one, when the distance is zero, which indicates very similar sentences. However the results when we just returned $1 - d * 10$.

IDF insertion cost

We have the functionality to add new cost functions easily and calculated IDF already for the prior task. So we changed the edit cost values in the algorithm to floating point numbers and implemented a cost function that returns the words IDF value for insertion, which improved the performance by about 5%.

1.4 Systems in part 3

For part 3 we implemented a learning algorithm provided by the Mahout[1] library. Mahout is a powerful and scalable machine learning library, providing algorithms for recommendations, clustering, and classification.

[1] Mahout website: `http://mahout.apache.org/`

We implemented the *BasicMahoutMatcher* by using the *OnlineLogisticRegression* algorithm. The results were not very promising at first and it is fairly easy to get really bad results from the machine learner if the parameters are not correctly set or the wrong features are selected. The following features are used for the *BasicMahoutMatcher*:

- Lemma Matching
- IDF Lemma Matching
- Lemma+POS Matching
- BleuScore
- 2-Gram Overlap

The resulting score is not very good, as no semantic information and almost no structural information is used. The *BasicMahoutMatcher* scores 56.4%.

Evaluating the Machine Learner

We implemented a 10-fold cross-validation to verify if our changes to the features improved the matcher or made the results worse. The code for this can be found in the *Main* class under the function name *crossValidate(ArrayList<THPair> pairs, IMachineLearnerRecognizer mlearner)*. We also used the 10-fold cross validation to find the best threshold for the machine-learner results. The Mahout classifier returns probabilites for each category, therefore we needed to find out the best threshold here as well. We used the average of all 10 runs.

2 Building our own RTE system

In this section we describe our best RTE system.

2.1 Considerations from the prior systems

All prior systems turned out to be worse than the normal lemma matching, Which to this point was still our best system with a score of 63.25%. We did not see any big potential in improving the basic lemma matching further, so we decided to try to tune our machine learning matcher *BasicMahoutMatcher* in the form of *MahoutMatcher*. Our basic idea was to use the best non-machine-learner systems as features for the machine learner, hoping this would give the machine learner a good basis to work with.

2.2 Implementation

We already had the basic *BasicMahoutMatcher* from part three, which we decided to tune. The base of this matcher was the *OnlineLogisticRegression* learning algorithm distributed with the Mahout machine-learning library. We played around with all our

basic matchers as features, but ended up only using a quite small subset in the final version. Only 6 features were used:

- Lemma Matching
- IDF Lemma Matching
- Lemma+POS Matching
- BleuScore
- Polarity
- WordNet Synonym Matching

For all features for which we had matchers (all except Polarity) we used the matchers estimate whether a text/hypothesis pair was entailing or not as value, as this is already a convinient value between 0 and 1. All matchers except for BleuScore already contain the Polarity measurement already as a sort of "death" criteria, where we set the estimate to 0 if the polarity doesn't match. We still use it as seperate feature, by setting the value to 1 if it matches and 0 if it doesn't, this proved helpful (without polarity feature, the result was 62.6%).

As we were not happy with the results we were able to achieve with our basic set of matchers, we decided to implement the WordNet library using JWI[2] as access to WordNet and the JavaSimLibrary[3] to calculate distances on the WordNet graph. JavaSimLibrary provides implementations of the Lin and Jiang & Conrath measures.

We implemented three different matchers based on WordNet:

- WordNetDistanceMatching
- LinSimilarityMatching
- SynonymMatching

The *WordNetDistanceMatching* recognizer uses the Jiang and Conrath distance and calculates the average distance between all the words in a sentence, normalized with the hypothesis size.

The *LinSimilarityMatching* uses the Lin similarity measurement. In contrast to the *WordNetDistanceMatching* the *LinSimilarityMatching* does not calculate averages, instead it is an implementation of the *BasicMatcher* class, and returns whether two nodes in the graph match by checking if there similarity is bigger than 0.8. This value was found by trying different values.

The *SynonymMatching* is also a *BasicMatcher* that checks whether two graph nodes have a overlap in synonyms with 1 graph level depth.

[2]JWI website:`http://projects.csail.mit.edu/jwi`

[3]JavaSimLibrary website: `http://pertomed.spim.jussieu.fr/~lma/jsl/`

2.3 Where our system failed

One example where our system failed was:

> <t>In Nigeria, by far the most populous country in sub-Saharan Africa, over 2.7 million people are infected with HIV.</t>
>
> <h>2.7 percent of the people infected with HIV live in Africa.</h>

We assume this happenend because there are many words in the hypothesis that also occur in the text, this giving a good lemma match for example. Polarity also matches, thus this is of no help. We do not see how our measurements could have detected this as not being an entailment.

Another example:

> <t>Rockweed has been harvested commercially in Nova Scotia since the late 1950's and is currently the most important commercial seaweed in Atlantic Canada.</t>
>
> <h>Marine vegetation is harvested.</h>

Here the first two words of the hypothesis care not found in the text, thus leading to e.g. a very bad lemma matching. A deeper synonyms analysis might help in this case. Other WordNet methodes like word stem could be useful as well.

3 Results

In this table, the results from all of our RTE-systems are presented. The parameters describe the setup used for the test. For the tests with the *TreeDistMatcher* "costs" denotes the used cost function, for operations on the tree. "depth" for *BleuScoreMatching* indicates the maximal used n-gram length.

It is important to notice that the results from the *MahoutMatcher*, can not be compared directly with the other percentages of correct guesses. That is because 10 fold-cross validation was used. It allows to determine a fitting threshold for each fold. The final performance result will be shown when applied to the test data, where we can apply it on the full range.

technique	parameters	threshold	correct
MahoutMatcher	10 fold cross-validation	avg. 0.5225 [0.475-0.55]	65.625% [57.5%-75%]
BasicMahoutMatcher	10 fold cross-validation	avg. 0.615 [0.55-0.775]	56.375% [51.25%-65%]
SynonymMatching		0.675	63.375%
LemmaMatching		0.675	63.25%
LemmaAndPosMatching		0.625	62.875%
LexicalMatching		0.575	62%
BleuScoreMatching	depth=2	0.425	61.875%
IDFLexicalMatching		0.325	61.625%
IDFLemmaMatching		0.5	61.375%
BleuScoreMatching	depth=4	0.5	61.25%
BleuScoreMatching	depth=3	0.425	61.125%
BleuScoreMatching	arithm. mean, depth=2	0.075	60.875%
LinSimilarityMatching		0.35	60.5%
WordNetDistanceMatching		0.775	60.25%
TreeDistMatcher	costs: WeightedLemma	0.725	57.125%
TreeDistMatcher	costs: WeightedIDF	0.4	55.75%
BleuScoreMatching	arithm. mean, depth=3	0.05	55.375%
BleuScoreMatching	arithm. mean, depth=4	0.05	51.875%
TreeDistMatcher	costs: FreeDeletion	0.05	51.7%

4 Future plans

Polarity measures could be implemented as a sort of filter around normal matchers, which can be added on demand. Currently the polarity is used in all *BasicMatchers*.

There are probably a lot of interesting possibilities with the WordNet API which have not been explored yet. Combining these with a more tuned version of the Mahout learning algorithms might lead to better results in the future.

The code could be better documented using javadoc. We think that if the code is properly documented it could be released as a generic RTE java library that can be used by others for their work. All code can be found at: `https://github.com/nihathrael/java-rte`