Evaluation of Tensor-based Machine Learning Algorithms for Matching Human Need Descriptions

RSA Technical Report c01/2015

Author: Heiko Friedrich, Dipl. Inf.

Date: January 2015

Version: 2.4

Smart Agent Technologies
Research Studios Austria
Forschungsgesellschaft mbH

About this Document

This report describes the research on *matching (Scalability)* in the context of the *COIN USS WON project (Usability, Scalability, and Security on the Web of Needs)* during June 2014 and January 2015.

Table of Contents

1. Introduction

The term *matching* refers to the problem of finding relevant and suitable *needs* for a given need X in the *Web of Needs* (Kleedorfer, Busch, Pichler, & Hueme, 2014) system. These needs (called *matches*) can then be provided to the owner (user) of need X as suggestions to fulfill X. If the user accepts one of these suggestions he can open a *connection* from his need X to the suggested need(s).

A node in WON that is responsible for computing matches between needs is called a *matcher*. It implements a certain matching algorithm or approach.

The goal of the research described in this report was to create a concept and a prototypical implementation of a matching approach. Furthermore the quality and performance of this approach should be evaluated.

The following chapters describe the realized work packages to achieve this goal. All the outcome artifacts and detailed technical documents of these work packages can be found in the project folder.

2. Work Packages

First a problem description of the topic was done together with requirements for matching approaches in the Web of Needs.

Afterwards the literature was searched for similar approaches and algorithms that could be used to solve the problem of matching.

An algorithm was selected based on the requirements and a concept was created to use it in WON as a matching approach. The concept was implemented as a prototype.

To evaluate the implementation test data was created and an evaluation framework was developed to compare different approaches.

In the end the results of the evaluation were analyzed to assess the matching approach.

The following work packages have been realized pretty much in the order stated below.

2.1. Problem Description

The matches can be chosen by using different criteria.

The first class of criteria are need attributes like type (e.g. *supply* or *demand*), description, heading, tags, location, date, time information.

The second criteria for finding matches can be past connections between needs that can be used to learn about future connections. In this case the problem can be viewed as a link prediction problem in a graph. Needs would be nodes (with attributes) and past connections between them would be edges. New edges or links could be predicted as possible connections by a suitable algorithm.

Machine learning (e.g. link prediction) algorithms can be used to solve the problem of using past connections (in addition to need attributes) to predict new connections.

2.1.1. Requirements

Several requirements were collected to assess and select suitable approaches to solve the matching problem.

We were especially focusing on machine learning algorithms because we wanted to compare these to non-machine learning algorithms which we had already implemented using text-based search with *Apache Lucene Solr*[1].

Also we wanted to model different kinds of relations like it is possible in *RDF*[2].

Here are the main requirements:

- Quality of prediction (as far as assessable)
- Machine learning approach
- Being able to deal with RDF data
- Model different kind of relations
- Include different attributes (like type, description, tags, location, date, etc.)
- Working on sparse matrixes (graphs with low number of edges)
- Scalability and Performance (distributable approach)
- Easy to implement or implementation available

[1] http://lucene.apache.org/solr/
[2] http://www.w3.org/RDF/

2.2. Literature Research

We were looking at about 20 research papers that can roughly divided into the following approaches:

- Matrix or tensor factorization methods
- Link prediction algorithms in graphs (with attributes)
- Recommender systems

We didn't find many approaches in the field that were able to combine the link prediction machine learning aspect with the handling of additional attributes.

2.2.1. Approach Selection

However the most promising approach we found was *RESCAL* (Nickel, Tresp, & Kriegel, 2011). It is a machine learning, tensor-based factorization approach.

The advantages of the approach are the following:

- Can be used for link prediction problems
- tensor-structure fits well to RDF data
- It is possible to model relations between different kind matching information (connections, needs, attributes, etc.)
- Good Scalability since the approach can be distributed (e.g. using Map-Reduce, not implemented in standard RESCAL see hint below) and scales in O(n) with n as number of entities
- Working implementation in python available

The disadvantages are these:

- The whole computation has to be repeated if new needs are added and cannot be done in real time (minutes to hours). So it must be done asynchronously and some kind of folding in for new needs must be developed

- Configuration is not easy since there are many configuration parameters

However the advantages outweigh the disadvantages. The asynchronous computation is not a fundamental problem since the concept of matching works asynchronous anyhow. That means owners of need are informed when new matching needs occur in the system.

Hint: *There is a distributed optimized implementation of RESCAL available called Ext-RESCAL[3]. Here also the attributes can be mapped to a different matrix than the entities which reduces runtime. However we didn't use it so far.*

[3] https://github.com/nzhiltsov/Ext-RESCAL

2.3. Concept

No matter how the input data about needs will be structured in detail after provided to the system by the user we expect it to contain at least some kind of heading and/or description in text form (together with other data like pictures, tags, location, date, etc).

The concept of the matching approach can be divided into two parts then.

First the input data is preprocessed to extract information using for instance natural language processing from the heading and description text of the need.

Second this information (together with the past connection data) must be mapped to the tensor structure of the RESCAL algorithm to use RESCAL as a matching algorithm.

2.3.1. Preprocessing

For this kind of natural language preprocessing we used to different approaches. The first one was *Gate*[4] which can be used out of Java (current implementation of Web of Needs is also Java based).

Gate is a natural language processing framework that lets you create processing components and order them in a processing pipeline. Processing rules can be specified using a simple domain specific language.

We used Gate to do the following preprocessing steps on need description text:

- Extract keywords about the need topic from the need heading
- Extract keywords about the need topic from the need description (optional)
- Extract the need type
- Filter out stop words (optional)
- Filter out location information (optional)
- Apply stemming (optional)

As a second, alternative preprocessing approach we used the python libraries *sklearn*[5] and *nltk*[6] to extract keywords about the need topic from headings and

[4] https://gate.ac.uk/
[5] http://scikit-learn.org/stable/

descriptions. Here we applied *tf/idf*[7]-measures to get additional information about important keywords from word/corpus frequency relations.

Using the two approaches just described the extracted (keyword) information for every need is mapped as attributes to the structure of the RESCAL algorithm.

2.3.2. Matching with RESCAL

As mentioned earlier we use *RESCAL* (Nickel, Tresp, & Kriegel, 2011) as a machine learning link prediction algorithm for matching.

The input and output data structure of RESCAL is a mathematical object called tensor. This tensor can be viewed as a set of several slices of N x N matrices where N represents the number of entities or needs (and attributes) in our case.

Each of these slices represents a certain kind of aspect or relation between these entities (needs and attributes). The most important kind of relation between needs is the slice which describes connections between needs. Using RESCAL we can compute probabilities of future connections (matches) between every combination of two needs based on past connections. After this computation has been executed a threshold value can be used to get the most likely matches.

Except the connection slice we use other slices to improve the quality of the matching algorithm by providing additional information about the needs, their attributes and relations (e.g. connections) between them.

The following information has been used in the slices of the RESCAL tensor for computing matches:

- (Past) Connections between needs
- Keyword heading attributes that belong to a need and /or ...
- ... Keyword description attributes that belong to a need
- Type of a need (optional)
- Category (similar to tags) of a need (optional)

So for the default RESCAL mapping there is at least the connections slice plus one attribute slice needed which represents keywords of the heading and/or the description of the need. More slices can be added optionally to improve matching

[6] http://www.nltk.org/
[7] http://en.wikipedia.org/wiki/Tf%E2%80%93idf

quality (e.g. type, category or not implemented aspects like, date or location information).

2.3.3. Prototype Implementation

Since there is already an open source implementation of the RESCAL algorithm available in Python we could create a prototype of a machine learning matching algorithm by implementing the following steps:

- Creating Gate natural language processing rules
- Integrating Gate into Java to execute it from command line and link it to further Java preprocessing
- Mapping of the (extracted) need information in Java to the input tensor structure of RESCAL and writing it to disk to be read by RESCAL
- Mapping the RESCAL output to meaningful matching format

Having the full executable chain of preprocessing, input mapping, RESCAL and output mapping these steps could be automated to implement a full prototype of a matcher.

As described later further implementations were necessary to implement an evaluation framework for the algorithm.

2.4. Evaluation

To evaluate the implemented prototype we needed possible use case scenario, realistic data for testing, an evaluation framework and tools to analyze the results.

2.4.1. Evaluation Data

The scenario we choose is inspired by the sharing economy. That means people can offer things they don't need any more (e.g. furniture, books, clothes) and search for things they want.

We used realistic test data from this field and created ~ 4000 needs with heading and description text. For most of the evaluations only the heading text was used to extract keywords from since it more precisely described the need topic. In this case this resulted of ~ 4500 extracted keywords from all headings and a total of ~ 13500 keyword-attributes for all needs (multi usage). So for most of the evaluations the average number of keyword attributes per need was ~ 3,4.

These needs could roughly be mapped to 180 categories of different size.

Since we didn't have the data about connections between these needs we created them by manually matching these ~ 4000 needs with one another completely. This resulted in ~ 8000 connections between the needs in total (in average 2 connections per need).

HINT: In the default versions of the evaluations we used all this connection information as input to the machine learning algorithm. It highly depends on the concrete use case of Web of Needs if this is a realistic approach.

There may not be all possible past connections available for a need in the system or training base. For some use cases it may be more realistic to have at most only one or two (possible) past connections per need available to learn from. This would be the case if there is usually just one connection opened between two needs, the transaction executed and afterwards both needs closed. There could have been much more connections to other needs possible but we don't get any information about it since the need is already satisfied and closed after establishing one successful connection.

As described later there is an evaluation also for these scenarios (limit number of connections to learn from) but we didn't use it as the default case.

2.4.2. Evaluation Framework

Now we could use this data to run cross-validations, predict (previously masked) connections and evaluate the quality of our matcher prototype. This masking of connections for later prediction in the cross validation can be done in two ways:

1. Cross validation (and masking) based on needs (default case)
2. Cross validation (and masking) based on connections

If the first (default) approach is done the cross validation (randomly) chooses needs and masks all of their connections to other needs. This simulates the case that a new need is put into the system without any connections. Connections have to be predicted for that need now.

In second approach the cross validation (randomly) choose not need but connections and masks them. This simulates the case that the matcher predicts additional connections for a need in the system that has already established connections.

We implemented an evaluation framework (in Python) which does this cross validation. In addition we implemented a simple non-machine learning algorithm based on document-word vector (cosine) similarity measure to have a comparison value for RESCAL prediction quality.

We automated the whole evaluation process of specifying input parameters for different evaluations, executing the matching algorithms (including the two parts preprocessing and mapping) and collecting the output data. For this automation of the process we used the Python library *luigi*[8].

This way we are able to start the whole evaluation with all different executions and configurations with just a single click or simple command line statement.

The evaluation framework produces a log file for every test execution (cross validation) which contains input (configuration) data, execution details and the following computed quality measures about the matching:

- Accuracy
- Precision
- Recall

[8] https://github.com/spotify/luigi

- F-score measure (f-score is a mixture of precision and recall, we use f0.5-score to rate the precision double compared to the recall)
- AUC (area under the curve of precision recall curve)
- Confusion matrix
- Computation of optimal threshold (for maximizing a certain f-score)

Also ROC and precision-recall curves are produced for all iterations of all cross fold validations.

HINT: *Another measure which is often used is for example P@10 (or P@k), given cut-off rank 10 (or any constant k), considering only the topmost 10 (or k) results for the precision computation. We didn't implement this but it could be useful since users usually don't look at all search results.*

2.4.3. Evaluation Configurations

We have created about 20 evaluation types that execute cross fold validations of about 80 different matching configurations. These evaluation types are based on a default configuration that was varied in one aspect to see the effect (of this individual aspect/change) on matching performance.

The default configuration can be summarized like that:

- Using all input connections in the test data set to learn from (see HINT in chapter 2.4.1 about this topic)
- Using cross fold validation based on needs (see Chapter 2.4.2 case 1 of cross fold evaluation, prediction of new need in the system)
- For preprocessing using stop word filtering
- For preprocessing using no stemming
- For RESCAL using fixed configuration parameters (e.g. rank=500, init=nvecs), but some configuration parameters have to be adjusted for individual runs (e.g. lambda values regarding over fitting ,threshold)
- For RESCAL using the default slices: connection slice and one keyword attribute slice (from the heading)

Using the different evaluation types we tested configurations differing from the default configuration like the following:

- Additional input information (e.g. additional need type, header vs. description attributes, category information slices)
- Different preprocessing steps (e.g. stop words filtering off, stemming)
- Different number of input needs and connections
- Influence of highly connected needs (hubs)
- Different configuration parameters for the algorithms (e.g. machine learning parameters like thresholds, regularization parameters, number of internal clusters, convergence values)
- Combinations of machine learning and non-machine learning algorithms (e.g. Intersection, Or-combination of results)
- Using cross validation based on connections, not on needs (see Chapter 2.4.2 case 1 of cross fold evaluation, prediction of new need in the system)

2.4.4. Results Analysis

The computed measures listed above are used to compare the quality of different matching configurations of the algorithms for instance in diagrams.

However to analyze the properties of the algorithms in more detail and find out how exactly to improve them even more specific data is needed.

So in addition to the measures listed above statistical data about individual needs like their true positives, false positives and true negatives is produced for very detailed analysis.

We found that it is also very helpful to represent the result data visually in a graph and be able to analyze and create hypothesis about the matching.

This is why a *GEXF*[9] graph is generated for every cross fold validation execution that represents the input data. Needs are represented as nodes and connections represented as edges in the graph. Need attributes like keywords, type or category information is attached to the corresponding nodes in the graph.

[9] http://gexf.net/format/

Since also detailed result information about the matching is added per node (need) to the graph it can be assessed visually in a graph visualization tool like *Gephi*[10] (see image below).

This result information represents matching measures per need (e.g. precision, recall, accuracy, f-score, number of true positives, true negatives, false positives, false negative). It allows observing graph properties about the matching very fast and form hypothesizes (e.g. if we see that matching is bad in big clusters => is hat due to over fitting) that can be tested then.

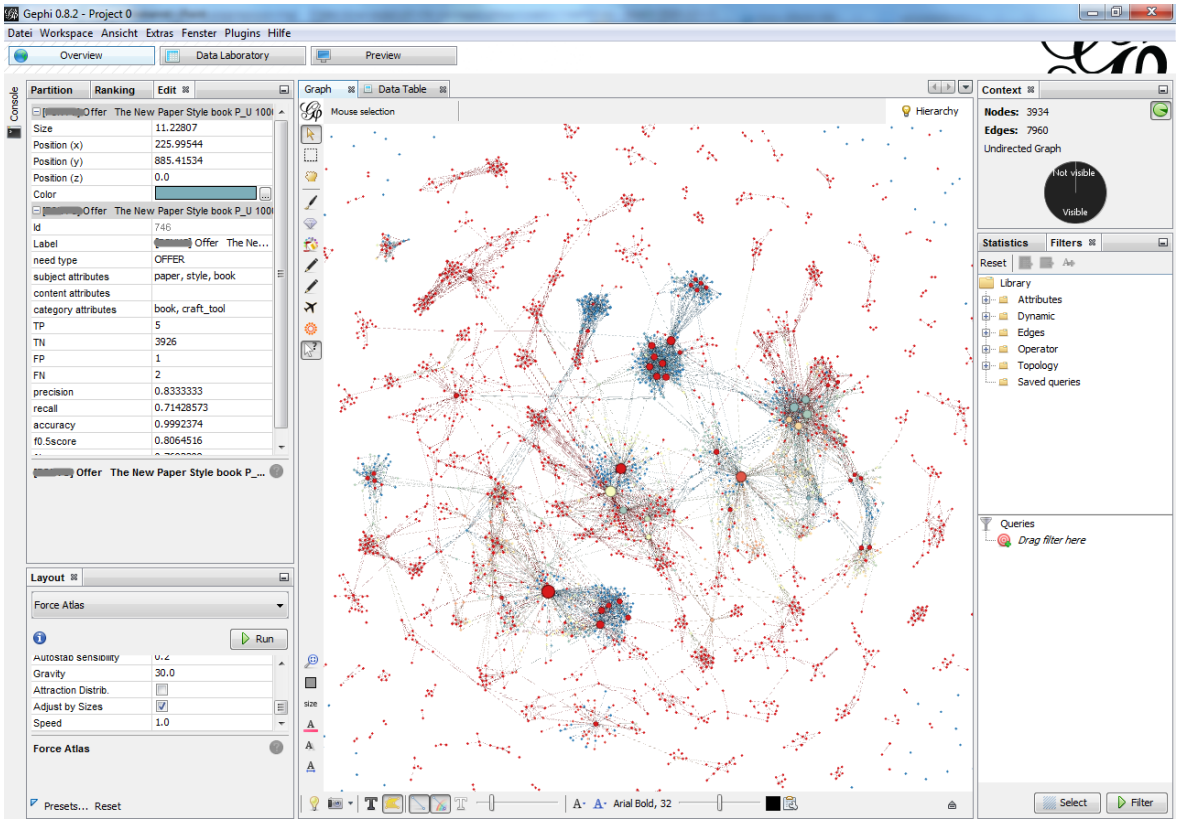


Figure 1 - Graph visualization with Gephi (size of nodes reflects degree, color reflects f-score measure)

As mentioned above we evaluated about 20 different kinds of configurations and strategies for the different algorithms. In the following just the most important findings are represented.

[10] http://gephi.github.io/

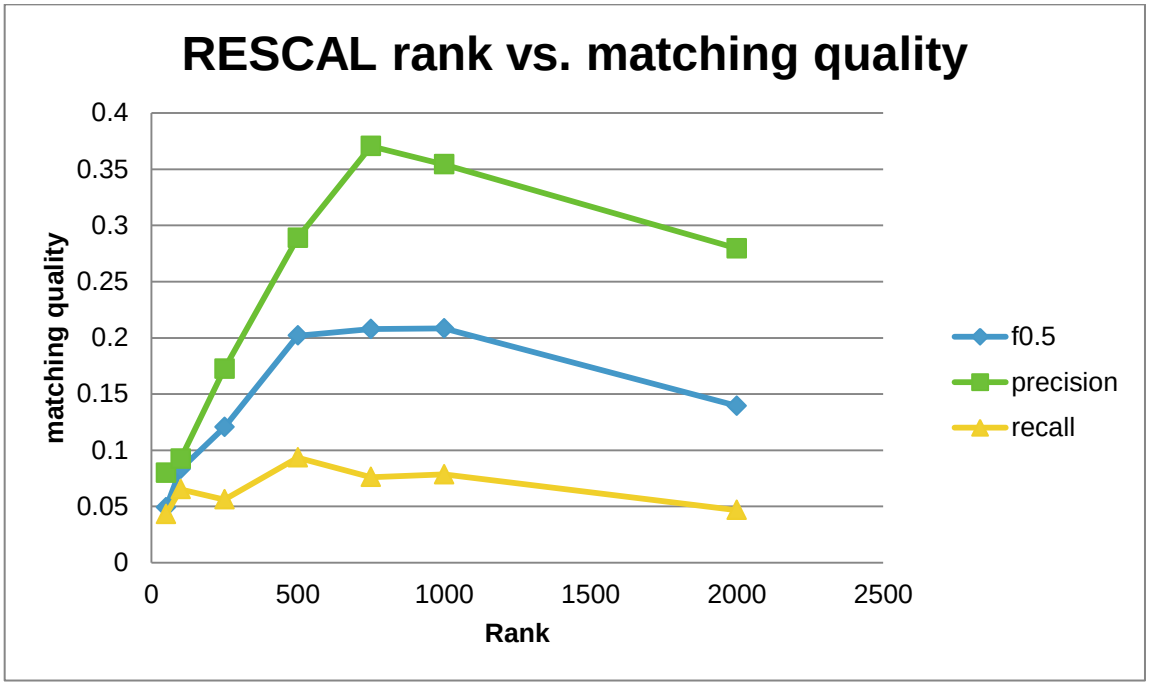
2.4.5. Configuration Parameters

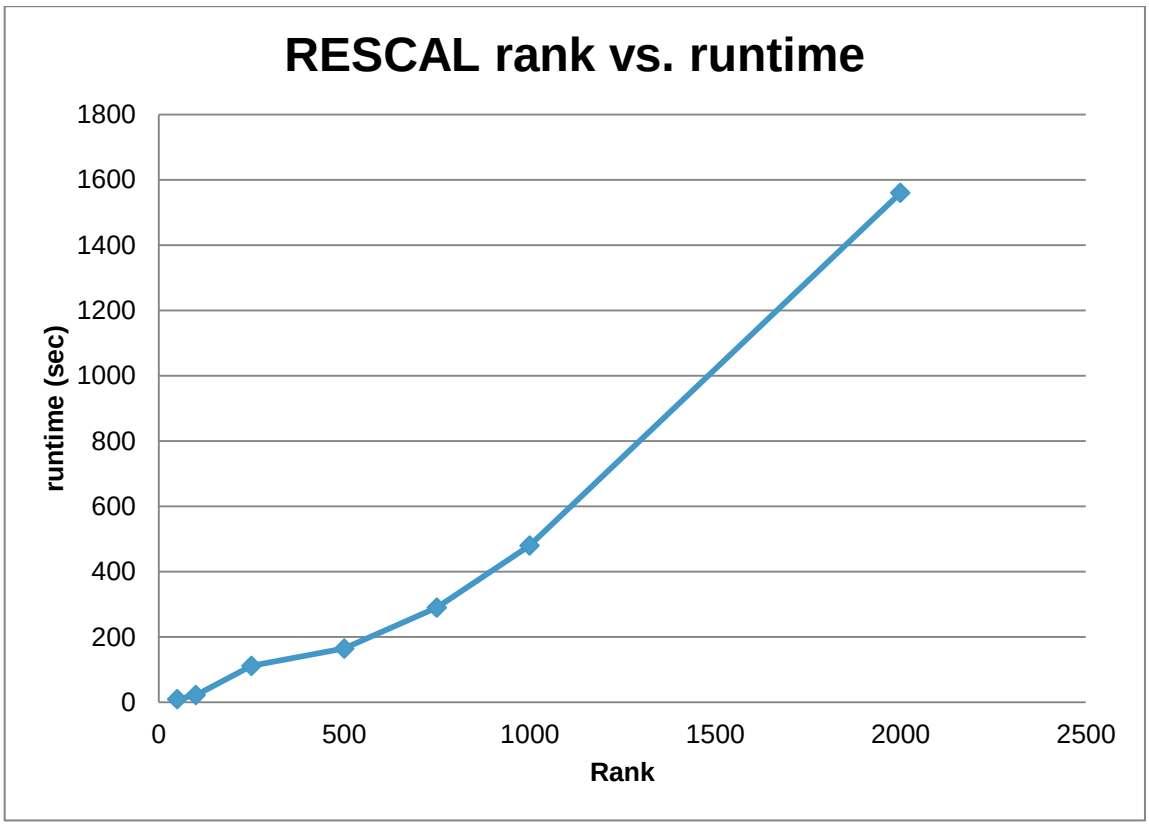
The main two configuration parameters for RESCAL are the *threshold* and the *rank*. Increasing the threshold basically increases the precision and lowers the recall. So we adapt the threshold in a way that results in an optimal f0.5-score measure for every evaluation we execute.

The rank specifies roughly how many lateral clusters RESCAL uses for its computation. Increasing the rank basically increases matching quality up to some maximum but also increases the runtime linearly.

We computed an optimal rank for our (default) evaluations based the following diagrams:

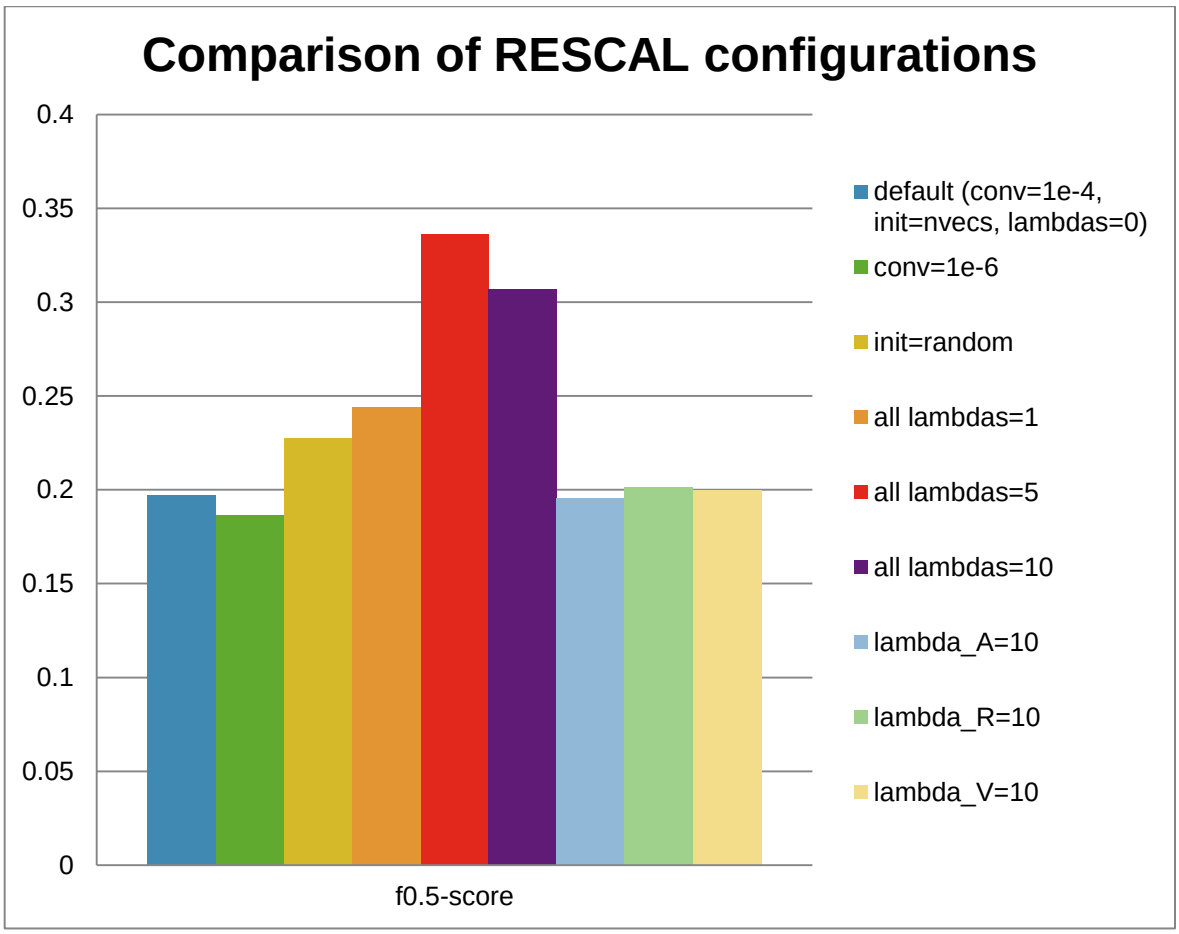
HINT: *the following evaluations have been done with RESCAL configuration parameter lambda set to 0 so they might suffer from over fitting!*





The maximum of matching quality can be achieved at around rank=500 (f0.5-score is about 0,2) and the runtime of one execution of RESCAL is still acceptable (about 2 minutes).

Other configuration parameters are shown in the next diagram. They are compared to the RESCAL default configuration with rank=500:



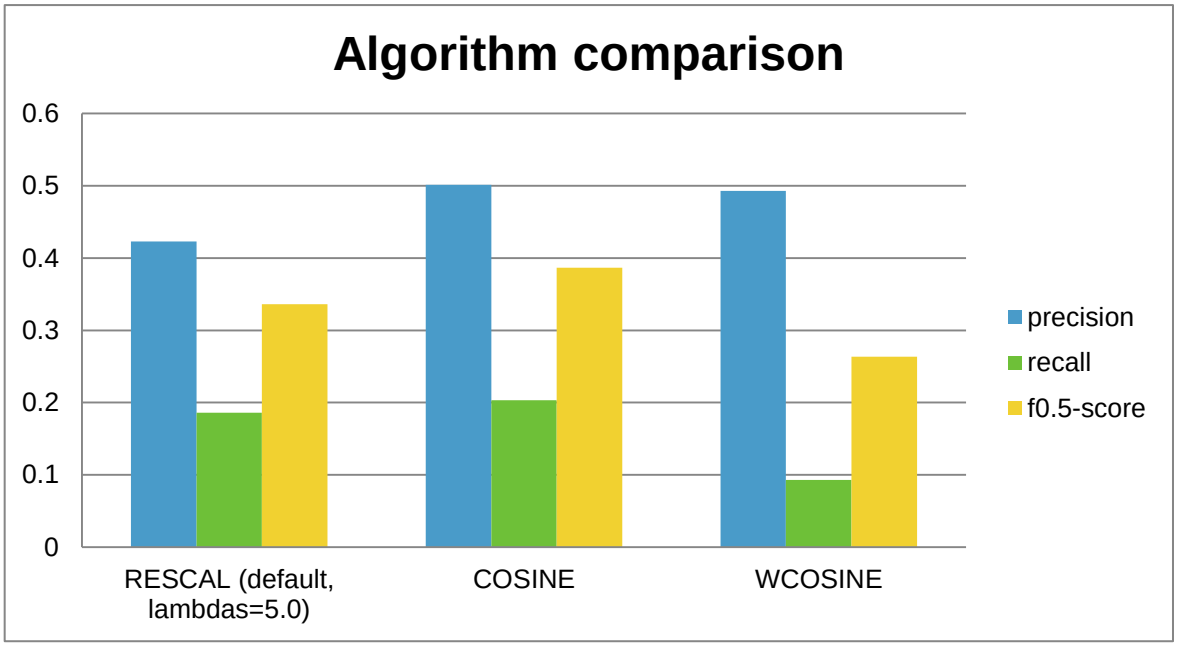
We found out that especially the lambda parameters could improve the matching of RESCAL by actually avoiding over fitting. The f0.5-score of RESCAL (default) can be increased by using lambdas=5.0 (compared to lambda=0.0) from about 0,2 to around 0,33.

HINT: However the influence of the lambda parameters was relatively lately discovered so some of the following evaluations don't have this parameter included in the default RESCAL configuration.

2.4.6. Algorithm Comparison

The first question after the default configuration is determined was how RESCAL as a matcher performs in comparison to a non-machine learning algorithm. For the non-machine learning algorithm we used a simple document-word-vector similarity approach (in the following called COSINE since it uses cosine similarity to compare the vectors).

The following diagram shows the comparison of the RESCAL, COSINE and WCOSINE (*idf-weighted* version of COSINE because the term frequencies are currently missing in the input, that means *tf* is either 0 or 1) matching qualities (precision, recall and f0.5 score measure):



As it can be seen the COSINE algorithm provides better matching quality (compare f-score measures) as WCOSINE and RESCAL in the default configuration (but with the lambda configuration values set to 5.0) we used.

We expect three main reasons responsible for that:

1. The need attributes alone (without past connection information) already provide a good hint which needs should be matched. Depending to which extend this statement is true this could be a strong reason not to use a machine learning algorithm as a matcher!

2. There may be too less data available for the machine learning algorithm. The expectation is that using more data (e.g. number of needs, number of attributes per need, different attributes) could increase the matching quality of RESCAL.

3. The way RESCAL works or how we use it may not be perfectly suited to solve the matching problem. As we will also see later RESCAL performs much better if additional connections for needs should be predicted that have already connections to other needs and thus are not new in the system (this

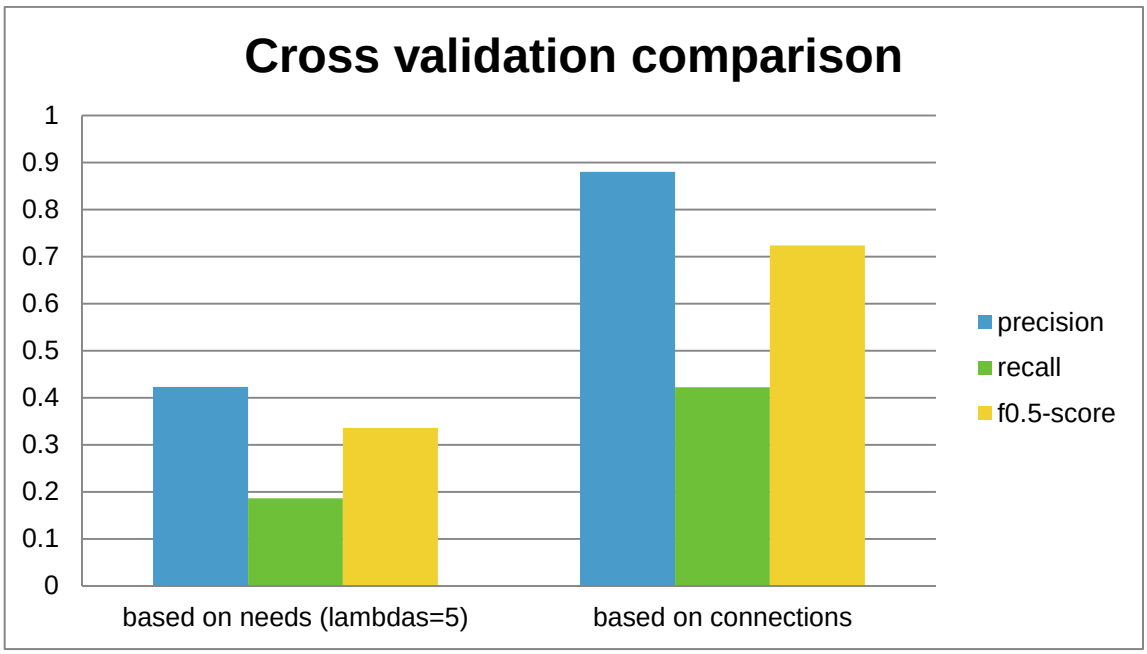
is the second case of cross validation described in chapter 2.4.2 and according evaluation results are described in the next chapter).

2.4.7. Cross validation comparison

As already described in chapter 2.4.2 there are two different ways the cross validation can be done.

The first one is the default one and represents the case of predicting connections of a new need (called *based on needs* since in the cross validation all connections of a randomly chosen need are masked).

The second one represents the case of predicting connections of an existing need which may already have connections (called *based on connections* since in the cross validation randomly chosen connections are masked).



Here we clearly see where the strengths of RESCAL are. In a highly connected graph RESCAL is very good (f0.5 score around 0,7) at “reconstructing” edges that have been deleted or masked (cross validation based on connections).

However if there are no other edges at the node (new need) then it is way harder (f0.5 score around 0,33) for RESCAL to find the missing connections (cross validation based on needs).

We expect that the case of predicting connections for a new need is much more likely than predicting connections for a need that is already connected to other needs. But this also depends on the concrete use case scenario of Web of Needs.

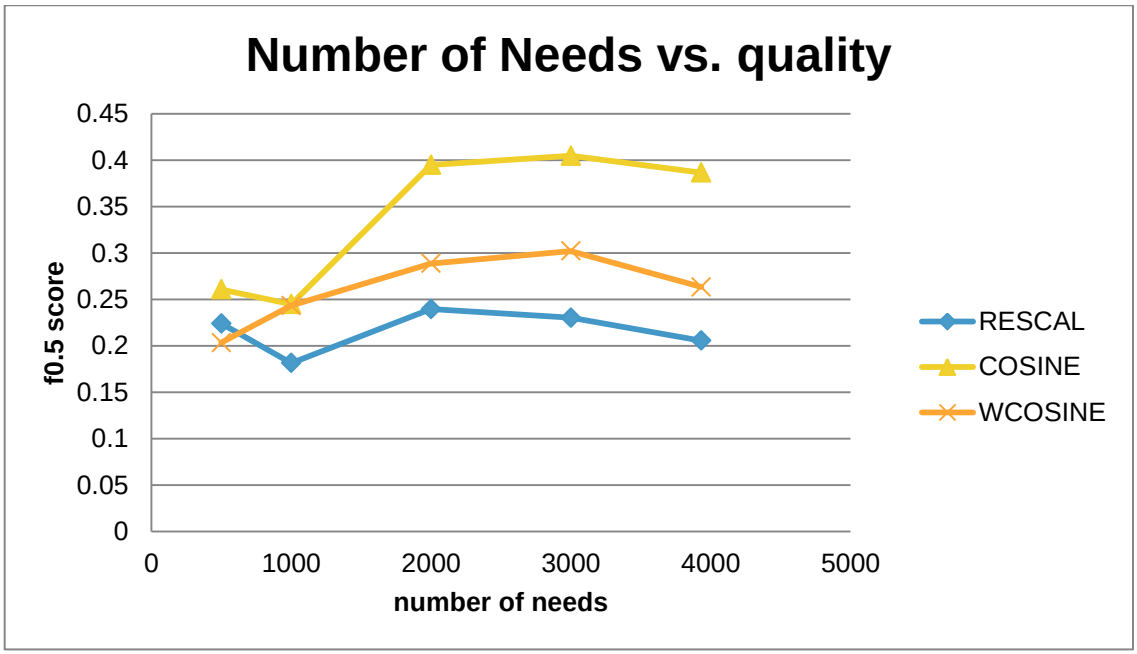
The decision for using RESCAL as matching algorithm thus depends highly on the concrete use case scenario.

2.4.8. Number of needs and connections

As stated above we are expecting that the matching quality of RESCAL could be increased if more data is available.

But we just had 4000 needs in the test set that could not be increased easily. Thus we just evaluated the effect of decreasing the number of needs to see if the predictions got worse:

HINT: *the following evaluations have been done with RESCAL configuration parameter lambda set to 0 so they might suffer from over fitting!*

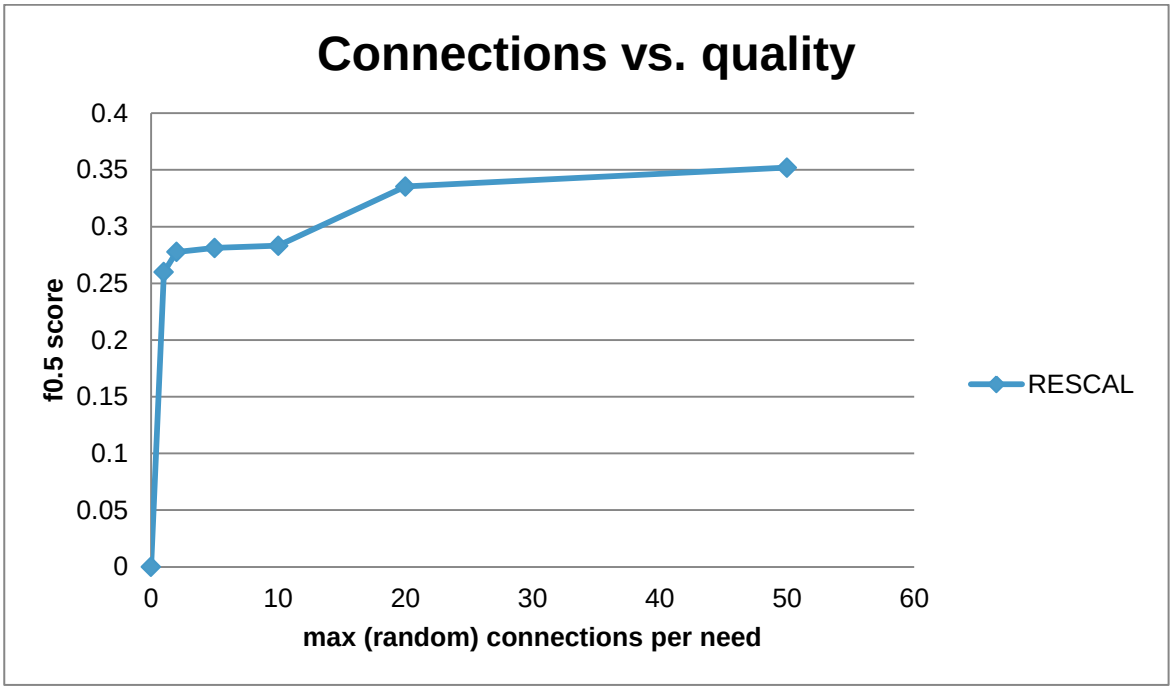


However a decrease of matching quality was not found. The less needs used in the evaluation the more the variance of the result increases what makes clear statements on evaluations with less than 2000 needs hard to do.

So it is still unclear if the matching quality of RESCAL could be improved with more data (needs or attributes).

Changing the number of input connections (per need) the machine learning algorithm can use to learn from is especially interesting since this depends heavily on the use case scenario of Web of Needs as already described in chapter 2.4.1.

From all connections of each need only X have been chosen randomly and used as input for the machine learning algorithm. The next diagram shows the influence of the number of (random) input connections on the matching quality. The lambda parameters were adjusted for every run to achieve an optimal result:



In this diagram we can see that the matching quality basically increases for an increasing number of input connections to learn from. If there is only a maximum of 1 connection per need to learn from the matching quality is ~ f=0,26, but if there are maximum 50 connections per need available to learn from the matching quality is ~ f=0,35. Since the number of input connections is dependent on the use case this result should be taken into account when deciding about using RESCAL for a specific use case in WON.

2.4.9. Improvement Strategies

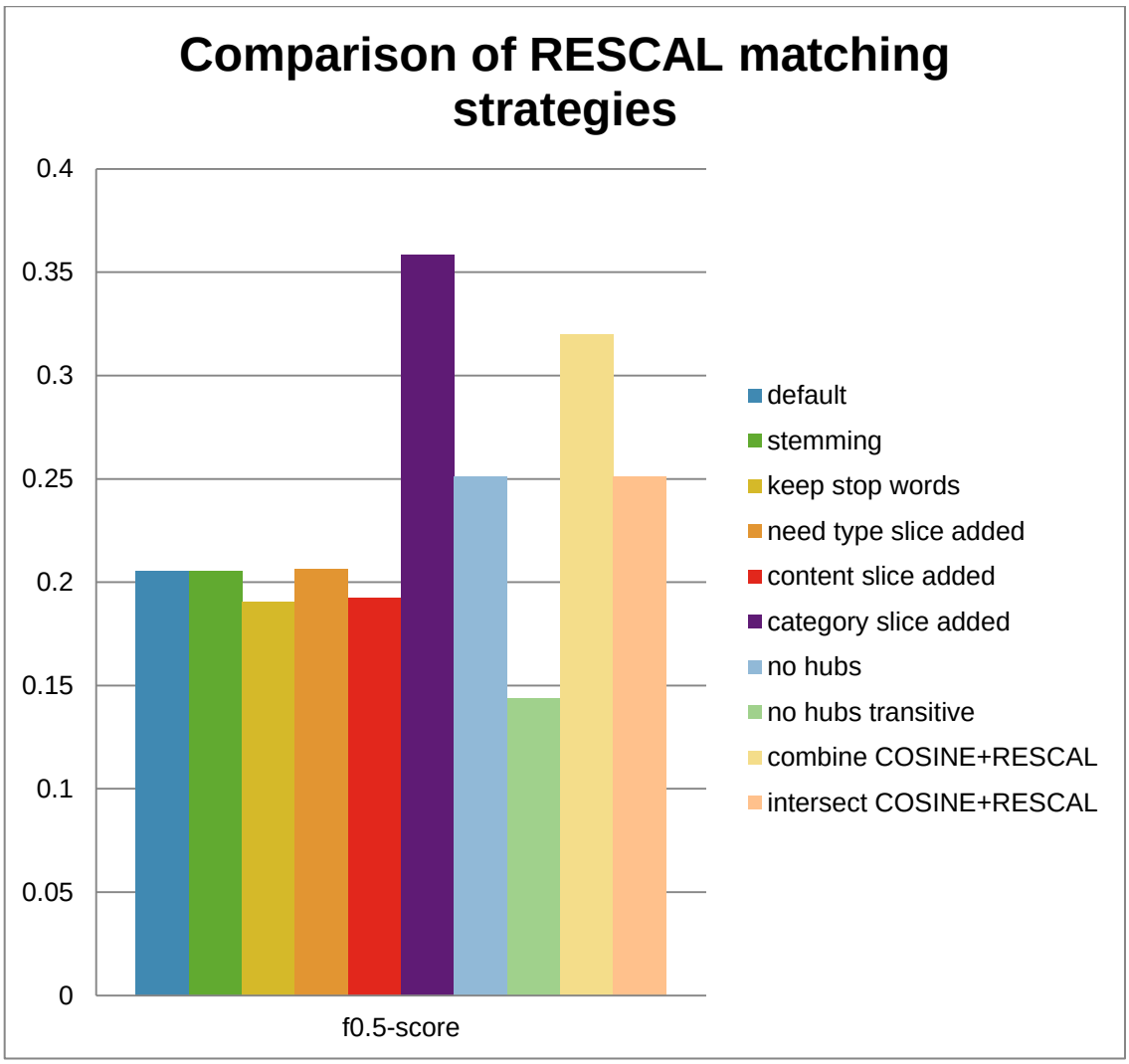
To improve the matching quality of the default configuration of RESCAL we tested several strategies.

Here is a short description to each strategy (and the aspect changed in comparison to the default) shown in the diagram below:

- *Stemming* is part of the Gate pre-processing and generates stemmed keyword attributes instead of the original items
- *Keep stop words* is part of the Gate pre-processing and doesn't filter out stop words from a list of ~650 English stop words (filter stop words is the default case) before generating keyword attributes
- *Need type slice added* means adding a slice to RESCAL that describes every need as either a *WANT (demand)* or *OFFER (supply)*
- *Content slice added* means adding a keyword slice from the content (description) of needs to RESCAL in addition to the already existing keyword slice from the heading of a need
- *Category slice added* means adding an additional slice with category information (similar to tags) about each need to RESCAL
- *No hubs* removes the needs with more than 10 connections from the evaluation
- *No hubs transitive* is like *No hubs* but in addition adds transitive (artificial) new connections for needs one hub away for learning. Here the idea was to connect not only WANT need types with OFFER need types but also the same need types. Since the algorithm in this configuration does anyway not distinguish between WANT and OFFER there might be a change for better learning this way
- *Combine COSINE+RESCAL* logically OR-combines the results of the two algorithms
- *Intersect COSINE+RESCAL* logically AND-combines (intersects) the results of the two algorithms

The results of the different RESCAL strategies can be seen in the next diagram:

HINT: *the following evaluations have been done with RESCAL configuration parameter lambda set to 0 so they might suffer from over fitting!*



As can be seen in the diagram above the pre-processing steps (stemming, filter stop) words we tested didn't improve the results. The same is true for adding information about the need type (need type slice added) and additional but less accurate keyword data from the need description (content slice added).

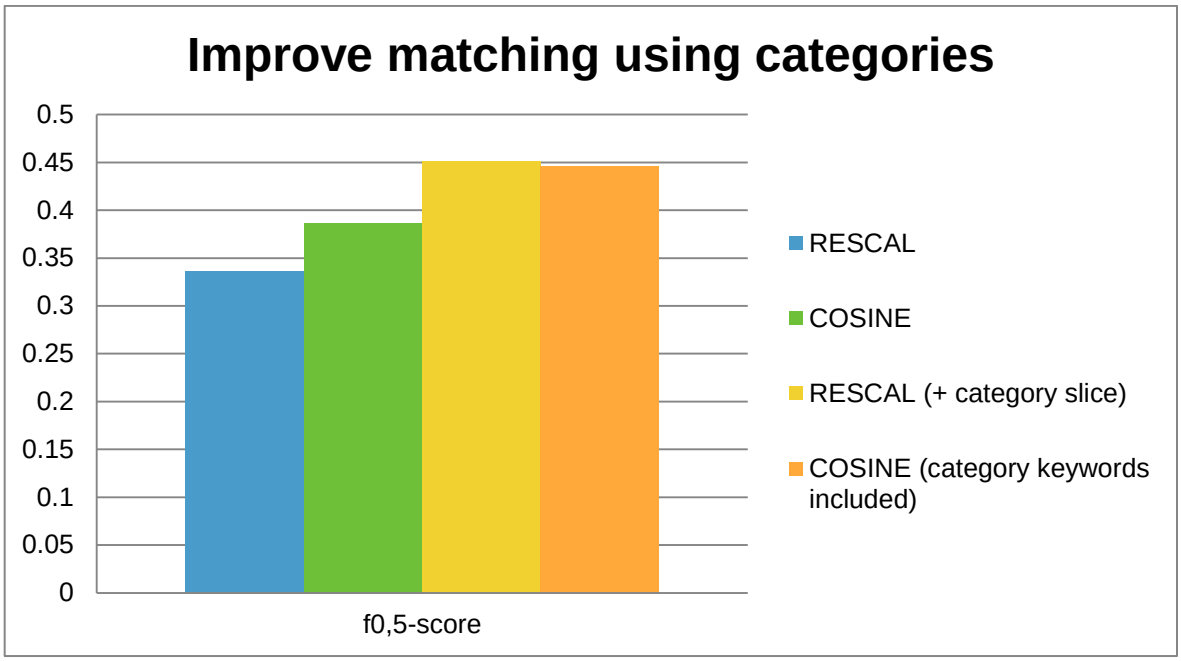
The *no hubs* evaluation improved the matching but we think this is due to decreased over fitting and has to be tested again.

The combinations of RESCAL and CONSINE (combine, intersect) also perform better than the default RESCAL but this is just because of the better performance of COSINE compared to RESCAL.

The only real improvement of matching quality could basically be achieved by adding a category slice with new information. This is reasonable since the category data adds very useful hints for matching.

To be able to compare this improvement we have added the same category information as keywords for the COSINE algorithm.

In the diagram below the results of this comparison can be found (this time over fitting for RESCAL is prevented, this is why results are ~ 0,1 better than above):



Using this keyword information it can be seen that RESCAL and COSINE achieve almost the same matching performance. However RESCAL might be more flexible in adding more and more attribute data to improve the matching.

2.5. Conclusion

Creating a concept and implementing a good (machine-learning) matching algorithm for Web of Needs is not an easy task. However we found RESCAL, a quite promising machine learning link prediction algorithm and implemented a prototype matcher.

This matching approach was compared to a very simple (non-machine learning approach with cosine similarity) using an automated evaluation framework in Python. For detailed analysis we used Gephi which appeared to be a very valuable tool for graph visualization.

Deciding about using a certain algorithm for matching depends on the concrete use case scenario. Particularly on the graph structure that is created by the system and the connection data that is available for learning.

In the cross validation evaluations (see chapter 2.4.7) we saw that RESCAL performs much better if single missing connections are predicted in a strongly connected graph than to predict connections of new needs that do not have any connections yet. Thus we expect for RESCAL as a matching algorithm to perform better for long living needs that establish many connections over time than for use cases were needs are closed after the first successfully established connection.

Many use cases are probably closer to the later use case scenario (loosely connected graph) than to the former one (highly connected graph). Here we saw that COSINE, a simple non-machine learning algorithm outperforms RESCAL in our default configuration. However the matching quality of RESCAL could be improved by using additional information like for instance category information (see 2.4.9).

In parts an open question is still if more data would also improve the RESCAL matching. As we saw by adding a category slice more attribute data indeed improves matching quality. Also having more connection data available improves matching quality (see chapter 2.4.7 and 2.4.8). In the number of need we were however limited to 4000 test needs and evaluation with less test needs didn’t provide clear insights about the influence of number of test needs on the matching.

The overall conclusion is therefore that the use of RESCAL depends strongly on the concrete use case scenario of Web of Needs. Also it should be considered that the effort of implementing and configuring RESCAL as a matching algorithm is quite high. Furthermore it cannot be executed in real time when a new need is created and needs to run asynchronously. However in a highly connected graph were link data is more important than attribute data of the nodes it clearly outperforms simple document comparison (cosine similarity) algorithms.

2.6. References

Kleedorfer, F., Busch, C. M., Pichler, C., & Hueme, C. (2014). The Case for the Web of Needs. *16th IEEE Conference on Business Informatics (CBI 2014).* Geneva, Switzerland.

Nickel, M., Tresp, V., & Kriegel, H.-P. (2011). A Three-Way Model for Collective Learning on Multi-Relational Data. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, (pp. 809-816). Bellevue, Washington, USA.