Meta-cognition evaluation
project journal

1. Short project description (as given by the project description document)

Starting from a set of student verbalizations, our aim is to measure the similarity between these metacognitions and the initial read texts for better estimating the one's comprehension level. The following criteria will be taken into consideration:

- lexical similarity based on lexical chains
- semantic similarity (LSA, LDA)
- cue phrases

2. Team and repository

Our team is formed of only two members:

- Maruseac Mihai
- Neață Sofia

All resources used for this project including this journal will be stored in a Git repository over GitHub at https://github.com/mihaimaruseac/nlp. Although the repository is public, this cannot be a problem for this project.

3. Motivation for choosing this project

The main reason why we have decided to choose this project is because sometimes each of us had to do a review of a book, of an article, etc and wanted to see how much of the review is similar with the original text and how much is new text. Also, this can be used to test the coverage of the original text given by the summary.

For example, we can use this tool to test how much a blog article reviewing a book uses phrases from the original text and how much coverage of the book is given by the article. We don't want to give spoilers of the book, for example.

Lastly, this can be used to test how much a student understood from the lecture by analyzing his summary of the lecture. If the results are not satisfactory the student knows that he needs to pay more attention next time and reread / rewatch the lecture if possible.

4. Preview of tools used (as of 26.02.2012)

As of now, since the state of the art is not analyzed, we can only give a preview of the tool which will be used in the project.

Basically, we will be using either WEKA or RapidMiner for implementing Machine Learning algorithms which will be needed in the process of developing the project. For kernel machines we can use either libSVM or Shogun, depending on the complexity of the task where we will need them, if any.

As for the Natural Language Processing tools which we will use we analyzed both OpenNLP from Apache or MontyLingua. However, we will use GATE for two reasons: it supports Romanian (in case we will port the project for this language) and it has plugins to interact with WEKA, libSVM and other tools.

5. State of the art (11.03.2012)

Metacognition [6] refers to higher order thinking that involves active control over the thinking process involved in learning. It is a "thinking process" based on one's experience and knowledge about his/her own cognitive activities. This is why, the shortest definition used to "metacognition" is "thinking about thinking".

Basically, a student before actually learn details about a concept, he/her must be able to do so (that means he/she must know how to learn). This is why the teachers must not only transmit some pieces of information, but also must be aware of how to develop students' metacognitive abilities.

Metacognition can be refered to as a system [2] [4]. In the most general sense, a system is a configuration of parts connected and joined together by a web of relationships. Thereby, the most important components of metacognition are knowledge and strategy. In this case knowledge refers to the information one has and use in order to achieve his/her goal of learning another piece of information. In other words, metacognitive knowledge [3] is used in the work monitoring process in order to identify the main task that one must work on, to determine the progress his/her work etc. On the other hand, metacognitive strategies [3] are used to direct one's activities to his/her goal ("draw" the path of the learning process). So, the learning process requires resources allocation, task division and plans to achieve the resulting sub-tasks (the actual activities, but one must be aware also about the time, the intensity and the speed for each of them). Another component of the metacognition system is experience[1]. It refers to past experiences that have something to do with the current developed task. Many times, experience is not recognized as an independent component of the metacognition system, but rather a sub-component of the metacognitive knowledge.

Metacognitive knowledge can be declarative, procedural and conditional [5]. Declarative knowledge answers to the question "what" (e.g. a journal entry). Procedural knowledge answers to the question "how" (e.g. the required steps to write a journal entry); this type of knowledge underlies the metacognitive strategies. Conditional knowledge answers to the questions "when" and "why" (e.g. a journal entry must be written every time the users clicks on an image).

In order to evaluate what the student knows we must be able to compare his written text with the original one and see how many concepts are touched. First of all cue phrases have to be identified and the text should be split in the main ideas components. If the student's work is too poor such that there are no clue phrases the main ideas must be still extracted somehow.

After having the main ideas running a latent semantic analysis on each of them will transform the text into a high-dimensional vector in the space of words. For better results, instead of using LSA it's best to use a probabilistic model[7]. For a refined model we can use lexical chains of words for grouping words together before applying LSA or PLSA.

A similar treatment is done to the original text. In fact, the original text can be split into ideas from the beginning. Both alternatives have to be tested.

Then, for each idea of the original document we will compute the cosine similarity between the vector obtained from the student's work and the original document. This will provide an overview over how well this idea was understood. If this particular idea is missing in the student's work then the overview value will be 0.

The last step involves averaging all the overview values weighted by some predefined importances given to the ideas of the original text.

To evaluate the model a human expert will label several examples and precision and recall will be computed.

6. Architecture

The application for this project will be written in Python. It allows faster prototyping and shorter development times at the cost of a longer run time. However, this disadvantage is acceptable.

The architecture of the executable will be a big pipeline transforming text to a hyper-dimensional vector in the space of word meanings.

The project will create as an artifact a simple executable which will receive as command line arguments two files: one containing the original text and one containing the text written by the student. As output, the application will produce a number of statistics related to the similarity between those two texts. All those statistics can then be used to estimate and evaluate the meta-cognition level achieved by the student.

The first step of the application (see figure 1: TODO) is to split the original text and the user text into a list of main ideas. We do this in order to increase the capabilities of meta-cognition evaluation that our project does. Moreover, having this separation in main ideas we can give weights to some of them in order to better evaluate what the student knows. For example, if the student had to learn three concepts: A, B and C but concept A was far more important than the other two we can give to each idea related to A a greater weight than to the other ideas. Thus, a student which will have learned A but failed to learn B and C will be considered better than a student who luckily understood B and C without recognizing that they are instances of the more important A concept. This idea will also help should we choose to use this for movie reviews, for book reviews, summaries, etc.

The above paragraph doesn't mean that a whole text analysis would not be done. Besides comparing each idea, we will also analyze both texts using the same pipeline. First, we considered using only the main ideas of the text or only the entire text, not both options. But it was proven that more information can be gained at a small cost – the total running time increases by a constant factor. The advantages far outweigh this: we can concentrate on the really important ideas, giving bigger weights to them.

Splitting the text is done by considering paragraphs: one idea per paragraph. An alternative would be to identify cue words: connectors, adverbs, etc. and split according to them. This will work if there are more ideas in the same paragraph but will fail if an idea has multiple sub-ideas: it will generate too many small phrases, increasing the time complexity of the algorithm. This increase is too costly compared to the benefits gained: a fine-grained structure of the text.

In conclusion, a result of the first stage in the pipeline is a vector of texts, both for the original document and for the student's work. The second step of the pipeline will work on each text from this vector, be it a single idea or the entire text. This stage, presented in figure 2 (TODO), pairs a text from the original document with a text obtained from the student's document such that both represent the same thing. It also keeps a count of pairings made and inputs discarded: if an idea from the original text wasn't covered by the student then that idea is discarded as it cannot be paired with anything the student produced.

The third step of the pipeline will receive this list of pairings and will compute the similarity between the first and the second element in each pair. There are many ways to compute this similarity. For example, we could use Latent Semantic Analysis or Probabilistic Latent Semantic Analysis for this. Both of them work pretty well but cannot cover polysemy. Also, they use the Bag Of Words assumption, which makes the algorithm ignore an important aspect of both texts: similar runs of words or similar runs of syntax tags. However, both LSA and PLSA offer good metrics, so we will keep them.

To solve the polysemy problem we will use WordNet. In fact, we can compute several metrics by pairing each word from text1 with each word from text2 and keeping those with similarity above a given threshold. The first way to compute the similarity will be to analyse the overlap between the dictionary entries of the two words (Lesk, 1986). This metric is very easy to compute and we will keep it.

Another metric using WordNet is to find the least common sense of the two words (LCS) and compute the similarity between two words w1 and w2 as given by the following formula.

$$Similarity(w1, w2) = \frac{2 * depth(LCS)}{depth(w1) + depth(w2)}$$

It is a simple computation, it can be done very quickly. We will also keep this metric, originally used by Wu and Palmer in 1994.

The last metric analyzed in regard with WordNet consists of using the formula of Leacock and Chodorow (1998)

$$Similarity = -\log\left(\frac{length}{2 * D}\right)$$

where length is the shortest path between the two concepts, using node-counting and D is the maximum depth of the taxonomy which contains both concepts

Another approach would be to use Explicit Semantic Analysis via Wikipedia articles. While this technique maps words to meaning in the high-dimensional space of natural concepts it requires doing inverse searches on a Wikipedia's database dump and will take too much to compute. We will not use this approach unless we can afford to do this: development time will allow implementing it and deployment method will allow running it.

As can be seen, we have several similarity metrics for the same pair which reaches this step. Which one will we choose?

In fact, this third stage of the pipeline will compute several similarity metrics for the same pair, returning a list of results. If debugging is enabled, the user of the application will see all of these results. Either way, these numbers are combined via a mixing function and a single similarity measure will be returned for each pair that entered this third stage. See figure 3 (TODO) for details.

Lastly, the fourth stage of the pipeline – presented in figure 4 (TODO) – will take each input similarity and the weight given to that input source (idea from text or entire text). If debugging is enabled the user will receive all these products. Otherwise, only those above a given threshold will be kept and will be returned to the user in the following format: the score, the original text and the student's text corresponding to the analyzed fragment.

After all these outputs are presented to the user, the last step – figure 5 (TODO) – will simply average all these statistics using a new mixing function (we won't use the arithmetic mean, this won't work quite well).

Climbing on the ladder of abstractions, all of those steps can be seen together in figure 6 (TODO). As it is easily seen, this looks like a pipeline and we can use this as a clever way to speed up the computations: parallelize as many operations as possible. We can do this because the entire processing can we written as a single functional equation (written in a Haskell-like language):

$$MC\ O\ S = avg\ \$\ filter\ TF\ \$\ zipWith * w_{ideea}\ \$\ map\ mix\ \$\ map\ SC\ \$\ MI\,(getIdeas\ O)(getIdeas\ S)$$

Where O is the original text, S is the student's work and MC O S is the coefficient we receive as a metacognition evaluation. TF is a threshold filtering function, SC represents all the similarity computations between ideas. MI is the idea matcher.

Figure 6 introduced above functions both as a quick overview of the entire transforming pipeline and as a scheme of the communication between the main modules of the application: each block in the figure will be a separate module and each stage will be another bigger module incorporating the smaller ones. In the end, the application will have these modules, each of them turned into a class. An UML description of the classes used by the application is given in figure 7 TODO.

We have tried to limit the work done by each class and decided to use a pipelined approach because this is in fact a MapReduce implementation and it should be easy to change it to other languages, to port it on a cluster, etc.

As software tools, we have decided to use RapidMiner instead of WEKA because it has more analysis features suitable for text mining and because it can be called from command line, thus it can be invoked by our program only when it is needed. Moreover, it's multilayered data-view concept maps very close to the pipeline we described above.

Between OpenNLP, MontyLingua and GATE we will use MontyLingua since it is implemented in Python just like our project is. Moreover, it is organized into a list of libraries and we can import only the libraries we need. Even more, it doesn't require training and it is also enriched with common sense knowledge, making it less vulnerable to common NLP errors.

7. to be continued

8. Bibliography

[1] http://studenttavern.com/2008/04/metacognitive-letters-evaluating-yourself/
[2] http://onlinelibrary.wiley.com/doi/10.1111/1467-8527.t01-1-00156/abstract
[3] http://www.hent.org/world/rss/files/metacognition.htm
[4] http://www.hent.org/world/rss/files/systems_think.htm
[5] http://www.education.com/reference/article/metacognitive-process-text-comprehension/
[6] http://catdir.loc.gov/catdir/samples/cam033/2002024499.pdf
[7] "*Probabilistic Latent Semantic Analysis*", Thomas Hofmann , UAI'99, Stockholm
[8] "*Measuring Metacognition and Self-Regulated Learning in Educational Technologies*", The 3rd Workshop on Metacognition and Self-Regulated Learning in Educational Technologies, http://www.andrew.cmu.edu/~iroll/workshops/its08/
[9] http://dickgrune.com/Programs/similarity_tester/
[10] http://www.slideshare.net/sharaf/text-similarity
[11] ftp://ftp.cs.vu.nl/pub/dick/similarity_tester/TechnReport