LSAfun - An R package for computations based on Latent Semantic Analysis

Fritz Günther · Carolin Dudschig · Barbara Kaup

Abstract In this article, the R package *LSAfun* is presented. This package enables a variety of functions and computations based on Vector Semantic Models such as Latent Semantic Analysis (LSA) Landauer, Foltz and Laham (*Discourse Processes* 25:259–284, 1998), which are procedures to obtain a high-dimensional vector representation for words (and documents) from a text corpus. Such representations are thought to capture the semantic meaning of a word (or document) and allow for semantic similarity comparisons between words to be calculated as the cosine of the angle between their associated vectors. *LSAfun* uses precreated LSA spaces and provides functions for (a) *Similarity Computations* between words, word lists, and documents; (b) *Neighborhood Computations*, such as obtaining a word's or document's most similar words, (c) plotting such a neighborhood, as well as similarity structures for any word lists, in a two- or three-dimensional approximation using Multidimensional Scaling, (d) *Applied Functions*, such as computing the coherence of a text, answering multiple choice questions and producing generic text summaries; and (e) *Composition Methods* for obtaining vector representations for two-word phrases. The purpose of this package is to allow convenient access to computations based on LSA.

Keywords Latent semantic analysis · Distributional semantics · Computer software · R

F. Günther (✉) · C. Dudschig · B. Kaup
University of Tübingen, Tübingen, Germany
e-mail: fritz.guenther@uni-tuebingen.de

Introduction

In the last two decades, a class of semantic models emerged in computational linguistics that conceptualize word meanings as vectors in a high-dimensional semantic space (Landauer, McNamara, Dennis, & Kintsch, 2007, for example). In such a semantic space, words that are similar in meaning will tend to be in similar areas of the space. Such models are referred to as *Vector Semantic Models* or *Distributional Semantic Models*. The theoretical foundation for this approach is the *distributional hypothesis* (Sahlgren, 2008; Harris, 1954), which states that words with a similar meaning also tend to occur in a similar context. There exists a wide range of such models (Jurgens & Stevens, 2010, for an overview; Jones et al., in press). However, the focus of this article and the software package presented in it lies on Latent Semantic Analysis, LSA (Derweester et al., 1991; Landauer & Dumais, 1997; Landauer et al., 1998).

In LSA, pre-defined documents are used as the word context. A different approach is taken by moving-window models such as HAL (Lund & Burgess, 1996), where the words that occur in a certain window before or after a specific word define its context. The BEAGLE model (Jones & Mewhort, 2007) even uses both kinds of information to establish a semantic space. It has been shown that the different models succeed in capturing different kinds of similarity, depending on which type of information they use (Jones et al., 2006).

Therefore, the focus on LSA is due to the fact that there exists a large body of research specifically concerned with LSA, and that it is probably the most prominent of those models.

The LSA algorithm

We will shortly describe the LSA algorithm, followed by an overview on how LSA has been used in research. The LSA algorithm is described in detail in Martin and Berry (2007) and can be sketched as follows: The main source of information for the LSA algorithm are word occurrences and co-occurrences in a large collection of text documents, a corpus. In a first step, a term-by-document frequency matrix is constructed from such a text corpus. In this matrix, every row represents a word in the corpus, every column represents a document, and each cell m_{ij} then specifies how often word i occurs in document j. By applying this procedure, words that often occur together will get assigned very similar vectors.

In a second step, a weighting scheme such as the log-entropy weighting (Dumais, 1991; Martin & Berry, 2007) is applied to this matrix to strengthen the impact of low-frequency words while weakening the impact of high-frequency words, since low-frequency words typically have a narrower, more defined and specific meaning.

In a third step, a singular value decomposition (SVD) procedure is applied to this weighted matrix. This is a two-mode generalization of the principal component analysis (Mardia, Kent, & Bibby, 1979) that also constitutes the basis of popular factorial analysis. The SVD procedure decomposes the m words $\times n$ documents weighted matrix M_w into three components as follows:

$$M_w = U \Sigma V^T, \tag{1}$$

where U is a $m \times r$ orthogonal matrix, V is a $n \times r$ orthogonal matrix, and Σ is a $r \times r$ diagonal matrix. The parameter r hereby describes the rank of the original term-by-document matrix M_w, which can be seen as the dimensionality of the LSA space it gives. U contains the word vectors as row vectors, and V contains the document vectors as row vectors (Σ contains the *singular values* as diagonal entries). Word and document vectors are thereby represented in the same semantic space. Applying this SVD enables LSA to capture deeper and more basic semantic dimensions. It is this step that makes indirect co-occurrences count, that is co-occurrences with a third word that two words are both related to. Landauer and Dumais (1997) even state that about 70 % of a word's nearest neighbors never co-occur in the same document.

In a fourth and last step, a dimensionality reduction is applied, so that U and V are reduced from r to k dimensions (whereas $k \leq r$). Typically, k is chosen to be a number of about 300 dimensions (Landauer & Dumais, 1997). Such a dimensionality reduction is applied to remove noise and uninformative dimensions from the LSA space.

LSA also allows the computation of a vector representation of a sentence or text document consisting of multiple words. The standard procedure here is to sum up all the word vectors in the sentence. Although this mechanism does not capture syntax or word order information, it does indeed seem able to capture the semantics and topic of a document (Landauer, 2007). (In fact, Landauer argues in this article that, in most situations, word order information does not even play an important role.)

Finally, representing words and documents by means of vectors in a semantic space allows for the computation of word and document similarities, which probably is the most useful feature of LSA. The most commonly used method for computing the similarity between two words or documents is using the cosine of the angle between two vectors. This cosine value ranges from -1 to +1. A cosine value of 0 indicates orthogonal vectors, i.e. unrelated words or documents, while a value of 1 indicates identical vectors. The higher the cosine between two words (or documents) is, the higher is their semantic similarity. However, it is also possible to obtain negative cosine values between -1 and 0 in LSA. One possible procedure to deal with such negative cosine values is to set them to 0, since negative cosine values cannot be reliably interpreted.

Nearly all of the functions included in the package *LSA-fun* (Günther, 2014) for R (R Core Team, 2013), which will be presented in the remainder of this manuscript, rely on the cosine similarity as a measure of semantic similarity.

Applications of LSA

There is a large body of research on what LSA is able to perform. In a variety of studies, LSA cosine similarities have been found to predict lexical priming effects (Landauer & Dumais, 1997; Landauer et al., 1998; Jones et al., 2006). In a typical priming study, participants respond to a word (the *target*) after being shortly presented with another word (the *prime*), for instance in a lexical decision task. It is a common finding that reaction times are faster the more similar the prime is to the target. In an extensive study, Jones, Kintsch, and Mewhort (2006) re-analyzed several experiments that observed priming effects of this type. They found that word pairs in high-similarity groups often had significantly higher LSA cosines than word pairs in low-similarity groups, and that these differences predicted priming effects.[1]

LSA has also been used to automatically answer multiple choice tests such as the synonym test of the TOEFL (Test of English as a foreign language) (Landauer & Dumais, 1997). This has been achieved by computing the cosine similarities between the word in question and each of the answer alternatives. Then, the alternative with the highest cosine to the

---

[1]These authors make a distinction between associative and semantic priming, which is beyond the scope of this article. They find that LSA best predicts associative priming effects, while HAL and BEAGLE, better predict semantic priming effects or even both effects.

word in question was chosen as an answer. With this procedure, LSA reached a score of 51.5 % correct. This was practically identical to the average performance of human learners with English as a foreign language (52.5 % correct), and sufficiently high to pass this test.

In a broad overview, Landauer, Foltz, and Laham (1998) give many further examples on the performance of LSA: For example, LSA is able to automatically assign reviewers to articles to be reviewed, based on samples of their own papers (Dumais & Nielsen, 1992), and can provide learners with new informational texts, based on the texts they have already read (Wolfe et al., 1998). Landauer, Foltz and Laham also demonstrate that LSA similarities correlate with human judgments on the similarity of word pairs. This correlation was stronger the more information LSA had available (i.e. the more documents the LSA algorithm was based on). The correspondence of LSA cosine similarities and human word associations was also demonstrated by Wandmacher et al. (2008). In the domain of university exams, Landauer, Foltz and Laham demonstrate that LSA managed to pass multiple choice test on introductory psychology courses (even though it scored lower than the average human student). On the other hand, Lenhard, Baier, Hoffmann, and Schneider (2007) demonstrated that LSA was able to automatically evaluate open answer student exams by comparing the students' answers to standard solutions. The difference between LSA and a human scorer was not greater than the difference between different human scorers.

As a last example, LSA has been combined with Multidimensional Scaling (MDS) and Principal Component Analysis (PCA) to successfully perform categorization tasks. This has lead to impressive and very interesting results (Louwerse, 2011). For example, Louwerse and Zwaan (2009) applied an MDS to a matrix containing the pairwise LSA similarities for US city names to obtain a two-dimensional solution. These authors were able to show that the MDS coordinates of the cities correlated with the actual geographical position of those cities. It has also been shown (Laham, 1997; Louwerse, 2011; Jones & Mewhort, 2007) that combining MDS or PCA with semantic space similarities allows to categorize words semantically.

Software implementing vector semantic models

A very useful package with a huge variety of algorithms for creating semantic spaces (amongst which LSA, HAL and BEAGLE are only some) from corpora is the S-Space package (Jurgens & Stevens, 2010), which is written in Java. This package even allows the user to implement his or her own algorithms to create semantic spaces. Furthermore, it includes functions to compute word similarities

such as cosine similarities, and to access the neighbors of words. The *Semantic Vectors* package (Widdows & Ferraro, 2008; Widdows & Cohen, 2010), which is also designed for creating semantic spaces from corpora and written in Java, primarily uses Random Projection to build semantic spaces (Bingham & Mannila, 2001; Kanerva, 1988), but also implements the LSA algorithm. This package also allows includes functions for neighborhood computations and basic similarity computations.

The toolkit *DISSECT* (Dinu, Pham, & Baroni, 2013) is written in Python and allows users to build semantic spaces with a special focus on compositional semantics (i.e. the meaning of phrases and sentences as well as words). However, in order to build a semantic space, this toolkit already needs co-occurrence matrices, not just text corpora. It also includes functions for similarity and neighborhood computations, as well as a wide range of composition methods. Also written in Python is the *gensim* software framework (Řehůřek & Sojka, 2010), which has an emphasis on scalability and is specialized in processing very large corpora. This framework also comes with an LSA implementation, amongst other algorithms. It further provides functions for similarity computations, including similarities between documents.

In R, the package *lsa* (Wild, 2011) can be used to create semantic spaces spaces based on the LSA algorithm. This package also includes basic functionalities for similarity and neighborhood computations.

Another software implementing Vector Semantic Models is the package *Word-2-Word* (Kievit-Kylar & Jones, 2012). The focus of this package especially lies on visualizing word similarity structures (as can be obtained from semantic spaces). It also comes with implementations of different algorithms to create semantic spaces (such as LSA, HAL and BEAGLE) that can then be visually explored.

Outlines for this package

The R package *LSAfun* presented here allows for various computations based on LSA. Its functionality will be explained and demonstrated in the remainder of this article. *LSAfun* is not designed for actually creating LSA spaces, because this is already established in R by means of the package *lsa* (Wild, 2011), as well as in the software mentioned above. However, LSA spaces that were created using the *lsa* package (or other software) can be used and investigated by the functions delivered with *LSAfun*. Thus, *LSAfun* allows a user to work with his own as well as pre-created LSA spaces available from a repository, and provides a range of useful higher-level functions. There are at least four advantages *LSAfun* adds to the existing software for semantic spaces:

First, *LSAfun* combines and extends the high-level functions provided by the existing software. For instance, all of the software implementations described above implement neighborhood computations and allow for simple similarity computations. However, as will be shown in this article, *LSAfun* includes a collection of additional functions (such as MDS for word lists, neighborhood plots, automatic summaries, a function for solving multiple choice tests and composition methods) that are only partially included in the existing software. Furthermore, it can serve as an interface between the different existing software applications: For example, composition methods (as implemented in *DISSECT*) can be applied to LSA spaces built with the *S-Space* package, which does not include those functions. In this case, *LSAfun* additionally implements the composition model by Kintsch (2001), which is not part of the *DISSECT* toolkit.

Second, in *LSAfun* there is no need to create a semantic space for working with the provided functions. Instead, users can freely download a semantic space from our repository, as will be explained later in this article. Hence, neither a text corpus nor software expertise are needed to work with this package. This can be considered an advantage of *LSAfun* especially for novices in the field.

Third, *LSAfun* is implemented as a standard package in R, which can be considered a *lingua franca* in statistical computing for psychology and cognitive sciences. This again seems particularly relevant for users who are not familiar with other programming languages, such as Java or Python.

Fourth, by being implemented in R, *LSAfun* allows users to directly integrate results obtained from semantic space analyses into their data analysis, without any copying or converting. This makes *LSAfun* a user-friendly and efficient tool for researchers. Examples for such an integration are given at the end of this article.

Getting started

LSAfun is available at CRAN (http://CRAN.R-project.org/package=LSAfun) and can be installed using the following command in the R console:

```
> install.packages("LSAfun")
```

The package can then be loaded by using

```
> library(LSAfun)
```

To get an overview over the functions provided by this package while using R, use the command

```
> help(package="LSAfun")
```

Getting an LSA space

Almost every function implemented in *LSAfun* relies on an LSA space in which the computations can be performed. In the R workspace, such an LSA space has to be a matrix object in which each row represents a word vector of the space, with row names specifying these words as character strings. This implies that all word vectors have the same dimensionality. In principle, any rectangular table can be loaded into the R workspace as a matrix, as long as it is stored in a format that is readable for R (such as .txt or .csv, for example). Concretely, there are several ways of loading such a semantic space into R, which will be presented here:

The first option is to create your own LSA space directly in R, using the package *lsa*. The basis for this is a collection of text documents one wants to create a LSA space from. This package requires only a few steps to build the LSA space:

1. Setting the working directory to the directory where the documents are stored
2. Creating a term-by-document frequency matrix with `textmatrix()`
3. Optionally applying a weighting scheme with `weightings()`
4. Conducting the SVD and performing a dimensionality reduction with `lsa()`

This procedure gives the LSA space as a named matrix, which is exactly the format required by the *LSAfun* functions. However, since the package *lsa* implements a full SVD instead of a sparse SVD, the size of the corpus that can be processed is limited to a few thousand documents.

The second option is to create your own LSA space using other specialized software, as described in the section on related work and software. The *S-Space* package (Jurgens & Stevens, 2010) exports semantic spaces in the .sspace data format. The *S-Space* homepage gives detailed instructions on how this data format can be converted to a matrix object as used by *LSAfun* (https://github.com/fozziethebeat/S-Space/wiki/FrequentlyAskedQuestions). The *DISSECT* toolkit, as well as the *gensim* framework, already store the semantic spaces they build as matrix objects (`numpy` or `scipy` matrices in Python). These can be converted to R-readable matrix objects in a few steps; one possible option is to save such matrices as .gzip files, which R can open with the `load()` command. The *Semantic Vectors* package includes the option to generate the LSA space as a plain text file, representing each vector in a single line in the format name $|a_1 | \ldots |a_n$ (Widdows & Ferraro, 2008). Such a file can be imported into R as a data frame using `read.table()` (with setting `sep="|"`), which can then be converted into the required matrix format using

`as.matrix()`. If you want to use *LSAfun* with an LSA space created from your own corpus, using *SemanticVectors* is probably the best option to create this LSA space, since it is very easy to use and its output file can be directly imported in R.

The third option is to directly download one of our pre-created LSA spaces provided at http://www.lingexp. uni-tuebingen.de/z2/LSAspaces. This website contains, amongst others, LSA spaces built from collections of blog entries in German, English and Dutch. It also provides an LSA space created from the TASA (Touchstone Applied Science Associates, Inc.) corpus, which consists of 37,651 different documents covering a broad variety of different topics (such as Literature, Arts, Science, Economics and Social Studies). This LSA space contains 92,393 different terms. This TASA LSA space will be used for code examples in the remainder of this article.

All the LSA spaces from http://www.lingexp. uni-tuebingen.de/z2/LSAspaces can be loaded in the R workspace as matrix objects via the following commands:

```
> getwd("Where_you_saved_the_semantic_space")
> load("name_of_space.rda")
```

Computing similarities

A very basic feature of LSA is that it allows for computations of similarities of different words as well as different documents. A comparison of words and documents is also possible, because LSA represents both in a single semantic space. The LSA homepage (http://lsa.colorado. edu/) allows for comparisons between one or multiple words in the *Matrix Comparison*, *One-To-Many Comparison* and *Pairwise Comparison* applications. It also allows for the comparison of documents and/ or sentences in the *Sentence Comparison* application. For information on the LSA homepage, see Dennis (2007).

LSAfun provides four specialized functions for different basic similarity computations:

- `Cosine()` computes the cosine similarity between two single words
- `multicos()` computes all pairwise word similarities for two lists of words
- `costring()` computes the similarity between two documents consisting of multiple words
- `multicostring()` computes the similarities between a document and a list of single words

As stated above, the vector representation for a document is computed by summing up all the vectors for all the words

in the document. The similarity functions for documents then use this vector representation to compute similarities. Table 1 gives examples of how these function are actually used in R. These examples also demonstrate that combining *LSAfun* with the TASA LSA space gives adequate results.

The functionalities presented in this section can also be obtained by working with the `query()` and `cosine()` commands implemented in the *lsa* package by Wild (2011). However, while these require some additional programming, the functions in *LSAfun* are designed to directly provide those functionalities without programming requirements.

Neighborhood computations

The LSA homepage also provides an application called *Near Neighbors*. This application takes single words or documents as input and computes their nearest neighbors, i.e. the words with the largest LSA cosine to this input. In *LSAfun*, this functionality is implemented in the function `neighbors`. A more generalized version of the `neighbors()` function is the `choose.target()` function. This provides the possibility to not just compute the nearest words to a given input, but to randomly sample words within any given range of similarity to the input. Both functions take single words as well as documents as an input. Table 2 shows how the neighborhood functions are used. A similar functionality is implemented in the *lsa* package with the function `associate()`, which gives all words to a given word with a similarity above a certain threshold.

Plots and multidimensional scaling

An additional feature of *LSAfun* is the possibility to generate plots that allow one to actually have a look on one's semantic space. Note that the *Word-2-Word* package (Kievit-Kylar & Jones, 2012) is a specialized software for exactly this purpose and implements, amongst others, the features presented here.

Since semantic spaces are typically high-dimensional, they usually cannot be plotted as a whole. Instead, the functions `plot_neighbors()` and `plot_wordlist()` plot subsets of the semantic space: Either the neighborhood of a given word (or document), or the similarity structure for a given list of words. This is achieved by means of four steps:

1. The nearest *n* words to the given input are computed (for `plot_neighbors()` only)

Table 1 Code examples for similarity functions

```
> Cosine("lion","tiger",tvectors=TASA)

[1] 0.6285499


> multicos("tiger lion cow",

          "cat zoo",

          tvectors=TASA)



          cat          zoo

tiger 0.3696037 0.33791316

lion  0.4732016 0.48697617

cow   0.1460171 0.02467506


> costring("The lions go on a hunt.",

          "The antelopes start to panic.",

          tvectors=TASA)

[1] 0.252278


> multicostring("The lions go on a hunt",

                "elephant antelope jump airplane",

                tvectors=TASA)



                elephant   antelope      jump    airplane

expression in x 0.08919608 0.2215349 0.1855564 0.01647845
```

2. All cosine values between all of these n neighbors plus the original input are computed and stored in a $(n+1) \times (n+1)$ matrix (for `plot_neighbors()`); if the function `plot_wordlist()` is used, all pairwise similarities between the given words in the list are computed and stored in a cosine matrix)

3. Either a principal component analysis (PCA) (Mardia, Kent, & Bibby, 1979), or a multidimensional scaling (MDS) is applied to this cosine matrix

4. The resulting matrix is truncated to two or three dimensions, so a two- or three-dimensional vector is assigned to each of the n neighbors and the input

The two- or three-dimensional vectors are then plotted in either a two-dimensional plot, or in a dynamic three-dimensional plot using the R-package *rgl* (Adler et al., 2014). However, when using these functions a user should always be aware of the fact that what is plotted is just a subset of the actual semantic space. Furthermore,

Table 2 Code examples for the neighborhood, plot, and MDS functions

```
> neighbors("lions",n=10,tvectors=TASA)
      lions         lion    elephants        beasts     leopards
  1.0000000    0.7620240    0.6662553     0.6370110    0.6322121
rhinoceros          zoo      antelope       animals    antelopes
  0.5940019    0.5939769    0.5936863     0.5916549    0.5783232


> choose.target("lions",lower=.2,upper=.3, n=10,tvectors=TASA)
   bleating         puma         simba      roarings      maximus
  0.2547728    0.2007193    0.2835835     0.2655955    0.2334613
trumpeting      sidenote       lookout      victoria     unwisely
  0.2628147    0.2282063    0.2591131     0.2152288    0.2327938


> plot_neighbors("lions",n=10,tvectors=TASA,method="PCA",
                 dims=3,connect.lines="all",alpha="shade")
> plot_neighbors("music",n=12,tvectors=TASA,method="MDS",dims=2)


> words <- c("lion","tiger","leopard","cat",
            "potato","carrot","leek","onion",
            "student","university","professor","exam",
            "car","ship","airplane","bike")


> plot_wordlist(words,method="MDS",dims=2,tvectors=TASA)
```

it is only a low-dimensional approximation to this high-dimensional subset. Table 2 shows how the plot and MDS functions are used in R, and Fig. 1 gives an impression of the results of `plot_neighbors()`. Figure 2 gives the results of a two-dimensional MDS that was applied to a list containing words from four different semantic categories (cats, food, university, and vehicles), using `plot_wordlist()`.

In a three-dimensional plot, the argument `connect.lines` controls the number c of connecting lines that are to be drawn from every single word.

Such lines connect each word to its c nearest neighbors in the plot (`connect.lines="all"` draws all pairwise connecting lines). Setting the arguments `alpha="shade"` or `col="rainbow"` will adjust the luminance or the color of these lines, so that they indicate the cosine similarities between the connected words. Using these options generates plots that deliver a good insight into the similarity structure of the plotted words (an example for `alpha="shade"` can be seen in Fig. 1).
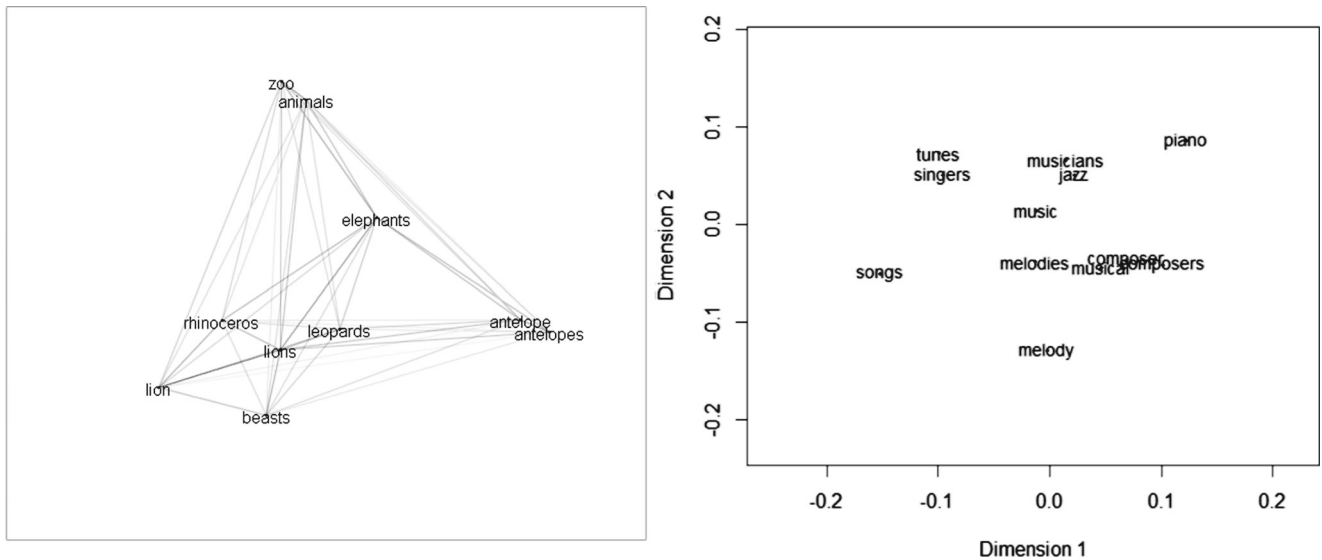
Fig. 1 (*left panel*) A plot of the ten nearest neighbors to the word *lions*, using `plot_neighbors()` with an PCA (screenshot of the three-dimensional plot). (*right panel*) A plot of the twelve nearest neighbors to the word *music*, using `plot_neighbors` with a two-dimensional MDS

Applied functions

LSAfun includes higher-level functions that implement several computations based on LSA. Landauer and Dumais (1997) proposed a method of determining the coherence of a text consisting of multiple sentences. Coherence is thereby defined as the conceptual overlap between the individual sentences of a text: In a coherent text, the understanding of a new sentence is facilitated because most concepts used in this sentence are already part of the preceedings discourse. The method for coherence

computation proposed by Landauer and Dumais (1997) works as follows:

1. The local coherence between two adjacent sentences is defined as the cosine between those sentences.
2. The global coherence of a text is the mean local coherence of the sentences in this text.

In *LSAfun*, this mechanism is implemented in the function `coherence()`, which basically applies the function `costring()` to a whole text. The `coherence()` function can be useful in examining the effects of coherence for text comprehension, for example by selecting coherent and incoherent text material for an experiment. On the other hand, it allows determining the coherence of a text produced by learners or students.

The function `MultipleChoice()` is a modification of `multicostring()` that selects the closest element of a set of possible options to a given text (or single word) input. This function first computes the vector for the question, and then computes the cosine similarities between the question and the possible answers. It then chooses the answer that has the highest similarity with the question.

We tested the accuracy of the `MultipleChoice()` function on the TOEFL synonym test (Landauer & Dumais, 1997). This test contains 80 target words (the questions) with four different answer alternatives per question. The task in this test is to find thesynonym to the target word among the answer alternatives. Of the 79 questions where the correct target word was present in the TASA LSA space, the function `MultipleChoice()` solved 45 correctly (56.25 %).
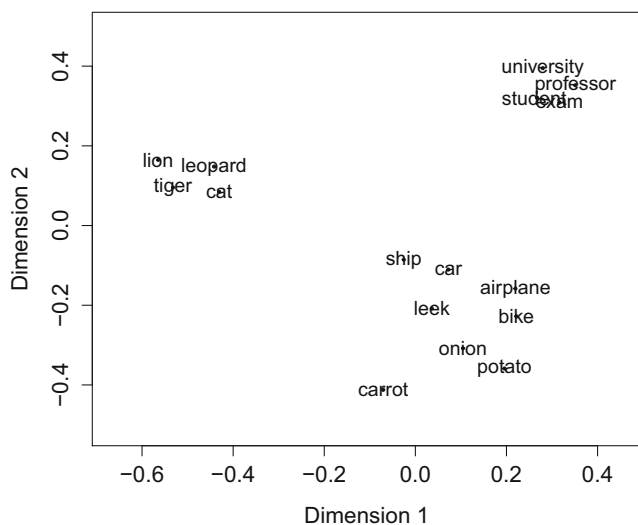

Fig. 2 The results of a two-dimensional MDS on words from four different semantic categories (*cats, food, university, vehicles*). This plot was generated using the function `plot_wordlist()`

In comparison, Landauer and Dumais report that the model answered 51.5 % of the questions correctly.

Table 3 contains code examples for the coherence computations and the automatic multiple choice answers.

The function `genericSummary()` applies a method for a generic and automatic text summary proposed by Gong and Liu (2001). This method is designed for summarizing a text by its k most important sentences:

1. Decompose the text into its sentences
2. Construct a term-by-sentence frequency matrix
3. Perform a SVD to obtain a singular value matrix Σ and a right singular vector matrix V^t. Each column vector of V^t then represents a sentence of the original text.
4. Select the k'th right singular vector from this matrix.
5. Select the sentence which has the largest index value with this k'th right singular vector, which is then included in the summary. Given the assumption that each row vector represents a topic of the text, this is exactly the sentence that scores the highest value on this topic, compared to the other sentences.
6. Repeat until k reaches a predefined number

This mechanism relies on the assumption that the SVD identifies the most important topics of the text and includes these as dimensions of the semantic space. In the end, the sentences best corresponding to these topics are included in the summary. Note that there is no need to specify a semantic space in which the computations can be performed for this function; this is because the semantic space is directly created from the text input. Table 4 contains code examples for this generic summary function.

Composition methods

As mentioned above, the standard LSA approach for computing the vector representation of a complex expression (an expression consisting of more than one word) is to compute the vector sum for all words that are part of this sentence/ document. However, this is not the only possibility for obtaining vector representations for complex expressions.

Mitchell and Lapata (2008) proposed various methods for computing vector representations for two-word-phrases. In four of these models, an explicit formula is given for computing the i-th element of the expression vector p from the two word vectors u and v:

1. Additive Model: $p_i = u_i + v_i$
2. Weighted Additive Model: $p_i = a * u_i + b * v_i$
3. Multiplicative Model: $p_i = u_i * v_i$
4. Combined Model: $p_i = a * u_i + b * v_i + c * u_i * v_i$,

with a, b and c being constant numeric values.

The latter three composition models pose an interesting alternative to the standard approach of summing up word vectors: A Weighted Additive Model allows one word to have more impact on the meaning of a complex expression than another word. In an adjective-noun phrase such as *brown cow*, one could assume that the noun plays a more important role for the meaning of the phrase than the adjective that just modifies the noun. Furthermore, such a model allows including word order information, for example by assigning a higher weight to the first or the last word in a complex expression.

A Multiplicative Model also has interesting features: In Vector Semantic Models, one assumes that each entry of a word vector represents one semantic dimension of the word. By applying an element-wise multiplication for a two-word phrase, only the semantic dimensions both words have high values on (i.e. the dimensions both share) will have high values in the resulting vector for the expression. The remaining dimensions will end up with relatively low values (in the most interesting case where one vector has an entry of 0, the vector for the expression will also have an entry 0 on this dimension, regardless of the other vector).

As indicated by its name, the Combined Model combines the features of the Weighted Additive Model and the Multiplicative Model. Mitchell and Lapata (2008) also mention the Predication Model by Kintsch (2001) in their article. This is a model for computing the vector representation of expressions consisting of an *argument* (describing an object, or more generally an entity) and a *predicate* (describing a property of this entity), and works as follows:

1. Select the m nearest words to the predicate
2. Of this set of m words, select the k nearest words to the argument
3. Compute the expression vector as the vector sum of predicate, argument and these k words.

This Predication Model considers syntax information, in that it emphasizes the different roles of arguments and predicates. Furthermore, it considers contextual information: Different neighbors of *runs* will be part of the expression vector for *horse runs* than for *time runs*. The Predication Model is also implemented in a computer program described by Lemaire et al. (2006). This program simulates text comprehension by combining LSA with the Construction-Integration Model (Kintsch, 1998), which is the theoretical foundation for the Predication Model.

In their article, Mitchell and Lapata (2008) examined these methods by setting up an experiment in which participants were presented with short sentences consisting of a noun and an intransitive verb (e.g. *The fire glowed*). These were followed by a second sentence, containing the same

Table 3 Code examples for `coherence()` and `MultipleChoice()`

```
> coherence("The lions go on a hunt.

         They approach a herd of antelopes.

         One lioness sneaks very near towards them.

         She gets ready to jump.",

         tvectors=TASA)


$local
[1] 0.24113457 0.50101178 0.06591721


$global
[1] 0.2693545


> MultipleChoice("Which of the following is

              a typical prey for lions?",

              c("Dogs","Whales","Elephants","Antelopes"),

              tvectors=TASA)


[1] "Antelopes"
```

noun and either a highly related word (e.g. *The fire burned*) or a lowly related word (e.g. *The fire beamed*). The participants were then asked to judge how similar the two phrases were. The phrase vectors were computed according to the five models presented here. Mitchell and Lapata (2008) showed that the similarity between the phrase vectors predicted the human similarity judgements best when they were calculated with the Multiplicative or the Combined Model, as compared to the other models.

In a more recent study, Mitchell and Lapata (2010) further examine these models using human similarity judgments for different types of phrases (adjective-noun, noun-noun, and verb-object phrases), in which they also found that the Multiplicative Model predicted the human judgments best.

In *LSAfun*, these five methods presented by Mitchell and Lapata (2008) are implemented in the function `compose()`. The output of this function is a vector with the same dimensionality as the semantic space used in the computations and can be used as an argument in other functions such as `neighbors()`. The function `Predication()` gives a more detailed output if the predication method is applied, it for example specifies which *k* words are included in the expression vector. Code examples for `compose()` are also included in Table 4.

Data analysis examples

As mentioned before, a main advantage of the implementation of *LSAfun* in R is that it can be used directly for data analyses. To demonstrate this, we included three data sets in the package: The data set `priming` contains simulated reaction time data for a semantic priming experiment with related and unrelated pairs. The item material is taken from Hutchison et al. (2008). The data set `syntest`

Table 4 Code examples for `genericSummary()` and `compose()`

```
> D <- "Lions  are  big  cats  that  mainly  live  in  Africa.

        However,  there  are  also  small  populations  in  Asia.

        Male  lions  have  a  mane,  while  female  ones  do  not.

        Lions  usually  live  in  small  packs  called  prides.

        There  is  one  dominant  male  lion  in  a  pride,

        with  several  related  lionesses  around  him.

        The  lionesses  do  most  of  the  hunting  for  the  pride."


> genericSummary(D,k=1)


[1] "the  lionesses  do  most  of  the  hunting  for  the  pride"


> comp <- compose(x="hunt",y="lions",method="Predication",m=50,k=2,
                  tvectors=TASA)


> neighbors(comp,n=5,tvectors=TASA)


     wild       hunt     hunting     hunted    antelope
0.9110461  0.7854708  0.7491371  0.7256238  0.6986696
```

contains a short synonym/ antonym multiple choice test, and `oldbooks` contains a list of five complete classical books.

Tables 5 and 6 contain the R code for these analyses. In the first analysis of `priming`, a correlation between LSA cosine similarities and reaction times is computed, as well as differences in the LSA cosine similarities for the related and unrelated prime-target pairs (whether a pair is classified as related or unrelated was determined by Hutchison et al. (2008)). In the second analysis of `syntest`, the multiple choice questions are automatically answered, and the percentage of correct answers is computed. In the third analysis of `oldbooks`, we want to recommend another classical book to a reader who just read Bram Stoker's *Dracula* on the basis of its similarity to this book. All example computations will be performed with the

TASA LSA space, which can be loaded into R as explained above.

Validation

To validate the results *LSAfun* gives, we compared them to those results given by other software implementing LSA.

We used the TASA corpus described above to create an LSA space with the *SemanticVectors* package (Widdows & Ferraro, 2008). With this space, we computed the pairwise similarities for 2,000 randomly selected word pairs (4,000 words were sampled randomly with replacement from all the words present in the LSA space, and randomly assigned

Table 5 Data analysis for the priming data

```
load("TASA.rda")


### Priming Analysis

> data(priming)

> sim           <- vector(length=nrow(priming))

> for(i in 1:nrow(priming)){sim[i] <-

    Cosine(priming[i,"Prime"],priming[i,"Target"],tvectors=TASA)}

> priming$sim <- sim

> priming      <- na.omit(priming)

> cor(priming$sim,priming$RT)

[1] 0.6027278

>t.test(sim ~ Relation,priming)

Welch Two Sample t-test


data:  sim by Relation

t = 24.1775, df = 315.345, p-value < 2.2e-16

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

 0.3493281 0.4112201

sample estimates:

  mean in group related mean in group unrelated

            0.382104115              0.001830033
```

to pairs). For those pairs, we computed the pairwise similarities with the `CompareTerms` class provided there. We also exported the LSA space to *LSAfun* and computed the pairwise similarities using the `Cosine()` function. We obtained a perfect correspondence between the two packages, all the cosine values were exactly the same.

We also repeated this analysis with pairs consisting of two-word phrases to test for the validity of the `costring()` function (i.e. we sampled 8,000 words and assigned them to two-word pairs). Note that almost all of these phrases did not make any sense, but this is not relevant for this validation. Comparing the results from `CompareTerms` in *SemanticVectors* and `costring()` in *LSAfun* also resulted in a perfect correspondence.

For further validation, we tried to replicate a 5000 words × 5000 words pairwise similarity matrix that was computed from an LSA space build from the CHILDES corpus (MacWhinney, 2000). This corpus consists of transcriptions of child-directed speech. We obtained the LSA space as well as the similarity matrix from the Cognitive Computing Laboratory, Indiana University, Bloomington. Both the LSA space and the similarity matrix were created using Python.

Table 6 Data analysis for the multiple choice test and the book comparisons

```
load("TASA.rda")


### Synonym/ Antonym Test

> data(syntest)

> attach(syntest)

> Solution <- vector(length=nrow(syntest))

> for(i in 1:nrow(syntest)){Solution[i] <- MultipleChoice(Question[i
    ],c(Answer1[i],Answer1.1[i],Answer3[i],Answer4[i]),tvectors=TASA)
    }

> syntest$Solution <- Solution

> table(syntest$Solution == syntest$Correct)/nrow(syntest)

FALSE   TRUE

  0.4    0.6


### Book similarities

> data(oldbooks)

> costring(oldbooks$dracula,oldbooks$frankenstein,tvectors=TASA)

[1] 0.922062

> costring(oldbooks$dracula,oldbooks$aroundtheworld,tvectors=TASA)

[1] 0.8511814

> costring(oldbooks$dracula,oldbooks$threemusketeers,tvectors=TASA)

[1] 0.9525076

> costring(oldbooks$dracula,oldbooks$jekyllhyde,tvectors=TASA)

[1] 0.9627005

## On this basis, recommend "Dr. Jekyll and Mr. Hyde"

## We agree with the results on this one
```

When using this CHILDES LSA space, the `multicos()` function of *LSAfun* did perfectly replicate the similarity matrix.

Scalability and run times

For scalability and run time analyses, we used a test laptop

with a 2.70 GHz processor, a 32-bit Windows OS, and 4 GB RAM. Loading an LSA space into the R workspace takes a short amount of time, depending on how big this LSA space is (and, of course, on the computer that is used). For most of the LSA spaces provided at http://www.lingexp.uni-tuebingen.de/z2/LSAspaces, which mostly contain about 100,000 different 300-dimensional term vectors, this takes about 2.5 seconds on our test laptop. A LSA space with 200,000 of such term vectors would take about 4 seconds to load. The size of LSA spaces that can be used with *LSAfun* is restricted only by the working memory of the computer one works with. Our test laptop failed to load LSA spaces with more than 85,500,500 entries from .rda files (this equals a matrix with 285,000 rows and 300 columns). Note, however, that most LSA spaces will not contain such a large amount of different terms.

The run times of simple similarity computations (such as `Cosine()`, `costring()`, `multicos()`, `multicostring()`, `coherence()` and `MultipleChoice()`) is very fast (less than 100 ms) and do not noticeably scale with the size of the LSA space used or any other input parameters. The run times for functions that depend on neighborhood computations (such as `neighbors()`, `plot_neighbors()`, `choose.target()` and `Predication()`) on the other hand scale with the size of the LSA space (but not with any other input parameters). These take about 2.5 seconds for an LSA space containing about 100,000 300-dimensional term vectors, and scale linearly with the size of the LSA space (a LSA space containing twice as many vectors will take about 5 seconds, for example). The run times for those neighborhood-based function also increase approximately linearly with the dimensionality of the LSA space used for computations. While computing the nearest neighbors to a word in a 300-dimensional LSA spaces containing 100,000 terms takes about 2.5 seconds, it takes about 3.2 seconds for a 600-dimensional space, and 4.3 seconds for a 900-dimensional space.

Conclusions

The package *LSAfun* presented here makes a range of LSA-based computations available on an open source basis in R. It allows the user to include her or his own LSA spaces as well as the pre-created LSA spaces available at http://www.lingexp.uni-tuebingen.de/z2/LSAspaces.

The input format for the functions was designed to be as convenient and intuitive as possible. Hopefully, this package makes research based on Vector Semantic Models better available for a larger audience of researchers and teaching staff. Its main purpose is to break down the barriers that the quite extensive computational processes involved in Vector Semantic Models pose especially for people that are new to this research field.

References

Adler, D., Murdoch, D., et al. (2014). Rgl: 3d visualization device system (opengl) [Computer software manual]. Retrieved from http://CRAN.R-project.org/package=rgl (R package version 0.93.996).

Bingham, E., & Mannila, H. (2001). Random projection in dimensionality reduction: Applications to image and text data. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 245–250). New York, NY: ACM.

Dennis, S. (2007). How to use the LSA website. In T.K. Landauer, D.S. McNamara, S. Dennis, & W. Kintsch (Eds.), *Handbook of latent semantic analysis* (pp. 57–70). Mahwah, NJ: Erlbaum.

Derweester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R. (1991). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, *41*, 391–407.

Dinu, G., Pham, N., Baroni, M. (2013). DISSECT: Distributional semantics composition toolkit. *Proceedings of the system demonstrations of acl 2013 (51st annual meeting of the association for computational linguistics)* (pp. 31–36). East Stroudsburg, PA: ACL.

Dumais, S.T. (1991). Improving the retrieval of information from external sources. *Behavior Research Methods, Instrumentation, and Computers*, *23*, 229–236.

Dumais, S.T., & Nielsen, J. (1992). Automating the assignment of submitted manuscripts to reviewers. In N. Belkin, P. Ingwersen, & A.M. Pejtersen (Eds.), *Proceedings of the Fifteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 233–244). New York, NY: ACM.

Gong, Y., & Liu, X. (2001). Generic Text Summarization Using Relevance Measure and Latent Semantic Analysis. *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 19–25). New York, NY: ACM.

Günther, F. (2014). LSAfun: Applied latent semantic analysis (lsa) functions [Computer software manual]. Retrieved from http://CRAN.R-project.org/package=LSAfun (R package version 0.3.1).

Harris, Z. (1954). Distributional structure. *Word*, *10*, 146–162.

Hutchison, K.A., Balota, D.A., Cortese, M., Watson, J.M. (2008). Predicting semantic priming at the item level. *Quarterly Journal of Experimental Psychology*, *61*, 1036–1066.

Jones, M.N., Kintsch, W., Mewhort, D.J.K., High-dimensional semantic space accounts of priming (2006). *Journal of Memory and Language*, *55*, 534–552.

Jones, M.N., & Mewhort, D.J.K. (2007). Representing word meaning and order information in a composite holographic lexicon. *Psychological Review*, *114*, 1–37.

Jones, M.N., Willits, J.A., Dennis, S. (in press). Models of semantic memory. In J.R. Busemeyer, & J.T. Townsend (Eds.), *Oxford handbook of mathematical and computational psychology*. New York, NY: Oxford University Press.

Jurgens, D., & Stevens, K. (2010). The S-Space package: An open source package for word space models. *Proceedings of the ACL 2010 System Demonstrations* (pp. 30–35). Uppsala, Sweden: ACL.

Kanerva, P. (1988). *Sparse distributed memory*. Cambridge, MA: MIT Press.

Kievit-Kylar, B., & Jones, M.N. (2012). Visualizing multiple word similarity measures. *Behavior Research Methods*, *44*, 656–674.

Kintsch, W. (1998). *Comprehension: A paradigm for cognition*. New York, NY: Cambridge University Press.

Kintsch, W. (2001). Predication. *Cognitive Science*, *25*, 173–202.

Laham, D. (1997). Latent semantic analysis approaches to categorization. In M.G. Shafto, & P. Langley (Eds.), *Proceedings of the 19th Annual Conference of the Cognitive Science Society* (p. 979). Mahwah, NJ: Erlbaum.

Landauer, T.K. (2007). LSA as a Theory of Meaning. In T.K. Landauer, D.S. McNamara, S. Dennis, & W. Kintsch (Eds.), *Handbook of Latent Semantic Analysis* (pp. 3–34). Mahwah, NJ: Erlbaum.

Landauer, T.K., & Dumais, S.T. (1997). A solution to Platos problem: The Latent Semantic Analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, *104*, 211–240.

Landauer, T.K., Foltz, P.W., Laham, D. (1998). Introduction to Latent Semantic Analysis. *Discourse Processes*, *25*, 259–284.

Landauer, T.K., McNamara, D.S., Dennis, S., Kintsch, W. (2007). *Handbook of Latent Semantic Analysis*. Mahwah, NJ: Erlbaum.

Lemaire, B., Denhière, G., Bellissens, C., Jhean-Larose, S. (2006). A computational model for simulating text comprehension. *Behavior Research Methods*, *38*, 628–637.

Lenhard, W., Baier, H., Hoffmann, J., Schneider, W. (2007). Automatische Bewertung offener Antworten mittels Latenter Semantischer Analyse. *Diagnostica*, *53*, 155–165.

Louwerse, M.M. (2011). Symbol interdependency in symbolic and embodied cognition. *Topics in Cognitive Science*, *3*, 273–302.

Louwerse, M.M., & Zwaan, R.A. (2009). Language encodes geographical information. *Cognitive Science*, *33*, 51–73.

Lund, K., & Burgess, C. (1996). Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instrumentation, and Computers*, *28*, 201–208.

MacWhinney, B. (2000). *The CHILDES Project: Tools for Analyzing Talk*, 3rd ed. Mahwah, NJ: Erlbaum.

Mardia, K.V., Kent, J.T., Bibby, J.M. (1979). *Multivariate Analysis*. London, England: Academic Press.

Martin, D.I., & Berry, M.W. (2007). Mathematical foundations behind Latent Semantic Analysis. In T.K. Landauer, D.S. McNamara, S. Dennis, & W. Kintsch (Eds.), *Handbook of Latent Semantic Analysis* (pp. 35–56). Mahwah, NJ: Erlbaum.

Mitchell, J., & Lapata, M. (2008). Vector-based Models of Semantic Composition. *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (pp. 236–244). Columbus, OH: ACL.

Mitchell, J., & Lapata, M. (2010). Composition in distributional models of semantics. *Cognitive Science*, *34*, 1388–1439.

R Core Team (2013). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from http://www.R-project.org/

Řehůřek, R., & Sojka, P. (2010). Software framework for topic modelling with large corpora. *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (pp. 45–50). Valletta, Malta: ELRA.

Sahlgren, M. (2008). The distributional hypothesis. *Rivista di Linguista*, *20*, 33–53.

Wandmacher, T., Ovchinnikova, E., Alexandrov, T. (2008). Does Latent Semantic Analysis reflect human associations? *European Summer School in Logic, Language and Information* (ESSLLI08).

Widdows, D., & Cohen, T. (2010). The Semantic Vectors package: New algorithms and public tools for distributional semantics. *Fourth International Conference on Semantic Computing (ieee-icsc)* (pp. 9–15). Pittsburgh, PA: IEEE.

Widdows, D., & Ferraro, K. (2008). Semantic Vectors: A scalable open source package and online technology management application. *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC 2008)*. Morocco: Marracesh.

Wild, F. (2011). lsa: Latent Semantic Analysis [Computer software manual]. Retrieved from http://CRAN.R-project.org/package=lsa (R package version 0.73).

Wolfe, M.B., Schreiner, M.E., Rehder, R., Laham, D., Foltz, P.W., Landauer, T.K., Kintsch, W. (1998). Learning from text: Matching reader and text by Latent Semantic Analysis. *Discourse Processes*, *25*, 309–336.