1 Wikipedia-based Semantic Analysis

The task of natural language processing has long been both a subject of interest and a source of great challenges in the field of artificial intelligence. The difficulty varies greatly depending with the different language processing tasks; certain problems, such as text categorization, are relatively straightforward to convert to a purely mathematical problem, which in turn can be solved by a computer, whereas other problems, such as computing semantic relatedness, necessitates a deeper understanding of a given text, and thus poses a greater problem. This sections aims firstly to give a brief introduction to some of the most prominent techniques used in language processing in order to explain my chosen method of explicit semantic analysis (ESA), and secondly to explain in detail my practical implementation of an ESA-based text interpretation scheme.

1.1 Methods

This section outlines a few methods used in natural language processing, going into some detail on ESA while touching briefly upon related techniques.

1.1.1 Bag-of-Words

An example of a categorization problem is the 'bag of words' approach, which has seen use in spam filters. Here, text fragments are treated as unordered collections of words drawn from various bags, which in the case of spam filters would be undesired mails (spam) and desired mails (ham). By analysing large amounts of regular mail and spam, the probability of drawing each of the words constituting a given text from each bag can be computed, and the probability of the given text fragment representing draws from each bag can be computed using Bayesian statistics.

More formally, the text T is represented as a collection of words $T = \{w_1, w_2, \ldots, w_n\}$, and the probability of T actually representing draws from bag j is hence

$$P(B_j|T) = \frac{P(T|B_j)P(B_j)}{P(T)}, \tag{1}$$

$$= \frac{\prod_i P(w_i|B_j)P(B_j)}{\sum_j \prod_i P(w_i|B_j)P(B_j)}, \tag{2}$$

for an arbitrary number of bags labelled by j. This method is simple and powerful whenever a text is expected to fall in one of several discrete categories (such as spam filters or language detection). However, for more complex tasks it proves lucrative to attempt instead to assign some kind of meaning to text fragments rather than to consider them analogous to lottery numbers or marbles. The objective of both explicit and latent semantic analysis is to establish a high-dimensional 'concept space' in which words and text fragments are represented as vectors. The difference between explicit and latent semantic analysis is the method used to obtain said concepts, a explained in the following sections.

1.1.2 Semantic Analysis

Salton et al proposed in their 1975 paper *A Vector Space Model for Automatic Indexing*[?] an approach where words and text fragments are mapped with a linear transformation to vectors in a high-dimensional space,

$$T \to |V\rangle = \sum_i v_i |i\rangle, v_i \in \mathbb{R}, \tag{3}$$

where a similarity measure of two texts can be defined as the inner product of two normalized such vectors,

$$S(V,W) = \langle \hat{V} | \hat{W} \rangle = \frac{\sum_i v_i w_i}{(\sum_i v_i^2)\,(\sum_i w_i^2)}, \tag{4}$$

and the cosine quasi-distance can be considered as a measure of semantic distance between texts:

$$D(V,W) = 1 - S(V,W). \tag{5}$$

This approach has later seen use in the methods of Latent Semantic Analysis (LSA) and Explicit Semantic Analysis (ESA). Both methods can be said to mimic human cognition in the sense that the transformation from (3) is viewed as a mapping of a text fragment to a predefined *concept space* and thus, processing of texts rely heavily on external repositories of knowledge. This should remind the reader of the language processing of ordinary humans: The word 'dog', for instance, carries more information than its three letters to a human reader as we immediately associate the word with related concepts such as mammals, common household pets, canines etc.

The difference between LSA and ESA is how the concept space is established. Although I have used solely ESA for this project, I will give an extremely brief overview of LSA for completeness following (Landauer 1998 [?]). Blahblahblah Vigtigt at man medtager de ledende momenter fra korrelationsmatricen og meget hygelig detalje at folk direkte påstår at det efterligner menneskelig indlæring. Slut med præcis hvorfor koncepterne er latente.

Skriv det her efter søvn!

ESA is explicit in the sense that the concepts correspond directly to certain parts of the external text corpora one has employed to construct a semantic analyser. I have used the English Wikipedia, so naturally each concept consists of an article, although the process could easily be tuned to be more or less fine-grained and associate instead each concept with e.g. a subsection or a category, respectively.

I wish to point out two advantages of ESA over LSA.

First, it is more successful, at least using the currently available text corpora and implementation techniques. A standard way of evaluating the performance of a natural language processing program is to measure the correlation between relatedness scores assigned to pairs of words or text fragments by the computer and by human judges. In both disciplines, ESA has outperformed LSA since it was first implemented ([?], p 457).

Second, the concepts employed in ESA are directly understandable by a human reader, whereas the concepts in LSA correspond to the leading moments of the correlation matrix . For example, to test whether the first semantic analyser built by my program behaved reasonably, I fed it a snippet of a news

dobbelttjek at det passer.

article on CNN with the headline "*In Jerusalem, the 'auto intifada' is far from an uprising*". This returned an ordered list of top scoring concepts as follows: "Hamas, Second Intifada, Palestinian National Authority, Shuafat, Gaza War (2008-09), Jerusalem, Gaza Strip, Arab Peace Initiative, Yasser Arafat, Israel, West Bank, Temple Mount, Western Wall, Mahmoud Abbas", which seems a very reasonable output.

1.2 Constructing a Semantic Analyser

The process of applying ESA to a certain problem may be considered as the two separate subtasks of first a very computationally intensive construction af the machinery required to perform ESA, followed by the application of said machinery to some collection of texts. For clarity, I'll limit the present section to the details of the former subtask while description its application and results in section 1.3.

The construction itself is divided into three steps which are run in succession to create the desired machinery:

1. First, a full Wikipedia XML-dump[1] is parsed to a collection of files each of which contains the relevant information on a number of articles. This includes the article contents in plaintext along with some metadata such as designated categories, inter-article link data etc.

2. Then, the information on each concept (article) is evaluated according to some predefined criteria, and concepts thus deemed inferior are purged from the files. Furthermore, two list are generated and saved, which map unique concepts and words, respectively, to an integer, the combination of which is to designate the relevant row and column in the final matrix. For example, the concept 'Horse' corresponded to column 699221 in my matrix, while the word 'horse' corresponded to row 11533476.

3. Finally, a large sparse matrix containing relevance scores for each word-concept pair is built and, optionally, pruned (a process used to remove 'background noise' from common words as explained in section 1.2.3)

These steps are elaborated upon in the following.

1.2.1 XML Parsing

The Wikipedia dump file comes in a rather large (50GB unpacked for the version I used) which must be parsed to extract each article's contents and relevant information. To this end, I wrote a SAX parser, which processes the dump sequentially to accommodate its large size. When running, the parser walks through the file and sends the various elements it encounters to a suitable processing function depending on the currently open XML tag. When treating article contents, I mainly used code from the pre-existing Wikiextractor project to remove Wiki markup language from the content. This code is included in section **??**. The remainder of the parser is my work, and is included in section **??**. Throughout this process, the parser keeps a list of unique words encountered outgoing link information for each article. These two lists, along with the article contents, are saved to files at the end of the step.

[1] These are periodically released at `http://dumps.wikimedia.org/enwiki/`

1.2.2 Index Generation

The next step reads in the link information previously saved and adds it as ingoing link information in the respective article content files. The point of this approach is that link information is initially saved as a hashmap so the link going to a given article can be found quickly, rather that having to search for outgoing links in every other article to determine the ingoing links to each article, which would require n^2 computations.

Following that, articles with sufficiently few words and/or ingoing/outgoing links are discarded an index lists for the remaining articles and words are generated to associate a unique row/column number with each word/concept pair. The code performing the step described here is included in section **??**.

1.2.3 Matrix Construction

This final step converts the information compiled in the previous steps to a very large sparse matrix. The program allows for this to be done in 'chunks' in order to avoid insane RAM usage. Similarly, the matrix is stored in segments with each file containing a set number of rows in order to avoid loading the entire matrix to interpret a short text.

The full matrix is initially constructed using a DOK (dictionary of keys) sparse format in which the i, jth element simply counts the number of occurrences of word i in the article corresponding to concept j. This is denoted $\text{count}(w_i, c_j)$. The DOK format works as a hash mapping tuples (i, j) to the corresponding matrix element and is the fastest format available for element-wise construction. The matrix is subsequently converted to CSR (compressed sparse row) format, which allows faster operations on rows which performs much quicker when computing TF-IDF scores and extracting concept vectors from words, i.e. when accessing separate rows corresponding to certain words.

Each non-zero entry is then converted to a TF-IDF score according to

$$T_{ij} = (1 + \ln(\text{count}(w_i, c_j))) \ln\left(\frac{n_c}{df_i}\right), \tag{6}$$

where n_c is the total number of concepts and

$$df_i = |\{c_k, w_i \in c_k\}| \tag{7}$$

is the number of concepts whose corresponding article contains the ith word - hence the name *inverse document frequency*. Each row is then L^2-normalized (divided by their Euclidean norm):

$$T_{ij} \to \frac{T_{ij}}{\sqrt{\sum_i T_{ij}^2}}. \tag{8}$$

Finally, each row is pruned to reduce spurious associations between concepts and articles with a somewhat uniform occurrence rate. This was done in practice by following the pragmatic approach of Gabrilovich [**?**] of sorting the entries of each row, move a sliding window across the entries, truncating when the falloff drops below a set threshold and finally reversing the sorting.

The result of this step is the matrix which computes the interpretation vectors as described in 1.1.2. The code is included in section **??**.

1.3 Applications & Results

Having constructed a semantic interpreter, I wrote a small Python module to gather data from twitter and analyse it. The module consists mainly of a SemanticAnalyser class, which loads in the previously mentioned index lists and provides methods for various computations such as estimating the most relevant concepts for a text, determining semantic distance etc, and a TweetHarvester class which can access the Twitter API and return tweets matching a given search term either by 'mining', i.e. actively querying a set number of matching tweets, or by 'listening', i.e. passively waiting for tweets relevant to the query to be posted. This code is included in section **??**.

Indsæt fine grafer OSV!!!

DO IT NAW!

1.4 Code

Skal kode være her eller i appendix?