Application of machine learning in analysis of answers to open-ended questions in survey data

Philip Pries Henningsen

Kongens Lyngby 2013
IMM-B.Sc-2013-25

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.Sc-2013-25

Summary (English)

The goal of the thesis is to implement a framework for analyzing answers to open-ended questions in a semi-automated way, thereby lessening the cost of including open-ended questions in a survey. To do this, techniques from the machine learning branch of computer science will be explored. More specifically, a methods known as *latent semantic analysis* and *non-negative matrix factorization* will be the focus of the thesis. This techniques will be used to extract topics from the answers, which enables me to cluster the answers according to these topics. The clustering will be done using k-means clustering. To implement all of this, the Python programming language is used.

Summary (Danish)

Målet for denne afhandling er at implementere et framework til at analysere svar til åbne spørgsmål semi-automatisk, derved mindske udgiften der følger med ved at inkludere åbne spøgsmål i et spørgeskema. For at gøre dette, vil jeg udforske metoder fra machine learning grenen af computer science. Mere specifikt, så vil fokus være på metoderne der kendes som *latent semantic analysis* og *non-negative matrix factorization*. Metoderne vil blive brugt til at finde emner i svarene, hvilket gør mig i stand til at klassificere svarene i kraft af disse emner. Til at klassificere svarene bruger jeg k-means clustering. Programmeringssproget Python vil blive brugt til at implementere teknikkerne.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring a B.Sc. in Softwaretechnology.

Lyngby, 01-July-2013

Philip Henningsen

Philip Pries Henningsen

Contents

Summary (English) **i**

Summary (Danish) **iii**

Preface **v**

1 Introduction **1**
1.1 How LSA is traditionally used 2

2 Theory **5**
2.1 Transformations 5
2.1.1 The *bag-of-words* model 5
2.1.2 Tokenizing and stemming 6
2.1.3 Weighting by TF-IDF 7
2.2 Latent analysis 7
2.2.1 Latent semantic analysis 7
2.2.2 Non-negative matrix factorization 9
2.3 Clustering 10

3 Implementation **13**
3.1 Data representation 13
3.1.1 TF-IDF 14
3.2 Latent semantic analysis 15
3.3 Spellchecker 16
3.4 The framework 18

4 Results **19**
4.1 Topic extraction 19
4.1.1 Without TF-IDF 19

4.1.2 With TF-IDF 27
4.2 Clustering 32
4.2.1 LSA 32
4.2.2 NMF 40

5 Conclusion **49**

A Code **51**

Bibliography **61**

CHAPTER 1

Introduction

Optimizing the procedure of collecting and analyzing data from surveys has led to closed-ended questions being the prevalent form of question used surveys and questionnaires. Closed-ended questions provide the means to easily and automatically analyze on data from surveys, while open-ended questions are almost exclusively done by human coding. This is a very costly approach, leading to open-ended questions being rarely used and even more rarely analyzed. This project is an attempt at lessening that cost while retaining the advantages open-ended questions provide.

To do this, I will explore methods from the machine learning branch of analyzing textual data to group the answers according to topics. Hopefully, this will provide a semi-automated way of analyzing the answers, vastly reducing the overhead inherent when including open-ended questions in surveys. The analysis will be performed on a dataset consisting of roughly 4000 answers from a survey made for Hi3G by the media agency Mindshare.

The main focus will be on the method known as *Latent Semantic Analysis*, which, as the name implies, is a method used to finding latent semantic structures in a collection of documents. In other words, reduce the semantic space, thereby 'combining' several terms into one. LSA builds on a very powerful mathematical method called *singular value decomposition* which is also used in methods such as *Principal Component Analysis*. Indeed, what I would like to do is very similar to PCA, only in PCA the aim is to create a model that explains as much variance as possible. Instead, the thought behind this project, and LSA, is that the original data is in truth 'flawed', since multiple words can describe the same concept and one word can be used about several concepts. This means I want to vastly reduce the dimensions of the data in order to find the topics inherent in the data.

In addition to LSA, a method known as *non-negative matrix factorization* will also be used. This is a different way of factorizing a matrix (as opposed to SVD) that introduces the additional constraint that all values in the resulting matrices must be positive. This turns out to be a very difficult problem to solve, but I will explain more about this in chapter 2.

The aim of the analysis, whether by using LSA or NMF, is to reduce the semantic space to a dimension that facilitate a quick overview of what the answers in the survey contain. In other words, the aim is to extract a number of topics the respondents are 'talking' about.

The end product should be a framework that can be molded to work in conjunction with the system normally used to analyze survey data at Mindshare. Since this system is still in the process of being built, the aim is to provide a flexible implementation instead of a plug-and-play implementation.

The thesis will start out with examining previous work done using machine learning to analyze open-ended questions as well as looking at what LSA has been used for in the past. In chapter 2 I will explain the theory behind both LSA and NMF, as well as the bag-of-words model, using TF-IDF to weight the data and the k-means clustering algorithm. Once that is all explained, chapter 3 goes through how the methods has been implemented in the Python programming language. Chapter 4 examines the results I get when combining all of this, including comparisons of weighted vs. unweighted data. Lastly, the results are summarized and interpreted in chapter 5.

1.1 How LSA is traditionally used

Originally, LSA was developed to find a better way of indexing and retrieving documents from a corpus. The idea was to reduce the semantic space, thereby reducing the effect of synonymy and polysemy. Once you have done this to your corpus, it would then be possible to treat a search query as a new document, reduce the dimensionality and then find similar documents in the corpus based on the length between the two documents in the new semantic space [DD90].

While information retrieval was the main objective when developing LSA, it has been used in numerous ways. This ranges from being used to solve the synonymy part of the *Test Of English as a Foreign Language* (*TOEFL*) to analyzing answers to open-ended questions in a study in the *Millennium Cohort* [KKH][LJLS11].

Chapter 2

Theory

The data used in this paper was provided by the media agency Mindshare. The data consists of 3590 answers to the open-ended question *'What do you think the message in the commercial is?'* from a questionnaire done for Hi3G by Mindshare.

First the transformations that needs to be done on the data set before I have something I can use with LSA and NMF will be explained. This includes the bag-of-words model, tokenizing and stemming and TF-IDF transforming.

2.1 Transformations

A range of transformation was performed on the data to first of all enable us to analyze the data, and second of all improve the results obtained.

2.1.1 The *bag-of-words* model

Since the data is in natural language, it can be difficult doing any sort of analysis with the raw data. A way of transforming textual data to a numerical repre-

sentation is the *bag-of-words* model, also known as a *document-term matrix.* In this matrix, each row, or *document vector*, corresponds to the term frequency for each term in a document. Each column, or *term vector*, corresponds to the frequency of that term in each document. In other words, each unique term is given a index (column) and the frequency of each term is computed for each document.
For the two simple documents *'3 has the best net. Best 4G too.'* and *'the fastest net is provided by 3'*, the bag-of-words model would be:

#	3	has	the	best	net	4G	too	fastest	is	provided	by
Doc1	1	1	1	2	1	1	1	0	0	0	0
Doc2	1	0	1	0	1	0	0	1	1	1	1

The bag-of-words model is a simplified representation of the original text, since all grammar and word order is ignored. While there is, without question, important information in the grammar and word order, the emphasis in this project is to extract topics from the data. If the dimensionality is not reduced, each (stemmed) term can be seen as a topic with LSA or NMF then combining terms to form true topics. Because of this, the grammar and word order is not really important to us - only the terms are. In other words, it is the content, and not the context, that matter to us. In this case, the bag-of-words model fits my needs perfectly, without introducing unnecessary complexity.
Having the answers represented as a matrix enables the use of powerful methods that depend on the data being numerical and in a matrix-like format.

2.1.2 Tokenizing and stemming

To transform the data from several strings of varying lengths to a bag-of-words, each document is tokenized. That is, transformed to a list of tokens. A token, to begin with, can be any word or punctuation. To filter out a lot of useless tokens, all stop words - excessively common words such as *'the'*, *'and'* and *'is'* - and punctuation is removed. The remaining tokens should, hopefully, be a list of relevant terms. This list will still contain duplicates in meaning, if not literally. E.g. the terms *'fast'* and *'fastest'* describe the same concept, but would be two separate terms in the list of tokens. To account for this, each token is reduced to their stem (e.g. *'fastest'* to *'fast'*). This process is called stemming and is essential in order for us to compare documents.

2.1.3 Weighting by TF-IDF

A popular way of weighting textual data is *term frequency inverse document frequency* (*tf-idf*). As the name of the method suggests, tf-idf is a way of emphasizing rare words while understating common words in the corpus. The tf-idf is computed by the product of two statistics: The term frequency (*tf*) and the inverse document frequency (*idf*). Various way of calculating the term frequency exists, but in my case I will be using the raw frequency of the term t in the document d divided by the total number of words n_d in d. That is:

$$tf(t,d) = \frac{f(t,d)}{n_d} \; .$$

The inverse document frequency of a term t, which is a measure of how rare t is across all documents, is the logarithm of the total number of documents $|D|$ divided by the number of documents containing t, denoted by N_t:

$$idf(t,D) = log\frac{|D|}{N_t} \; .$$

As mentioned, the tf-idf can then be calculated as the product of the tf and idf:

$$tfidf(t,d,D) = tf(t,d) \times idf(t,D) \; .$$

2.2 Latent analysis

The main focus will be on *Latent Semantic Analysis* (*LSA*), which is a powerful method of finding hidden meaning in textual data. A method called *non-negative matrix factorization* (*NMF*) will be used to compare the results obtained from using LSA.

The section will begin with the theory behind LSA including the matrix factorization method known as *singular value decomposition* and then move on to explaining NMF along with its advantages and shortcomings compared to LSA.

2.2.1 Latent semantic analysis

Finding latent semantic structure in a collection of documents starts with a powerful mathematical technique called *singular value decomposition* (*SVD*). SVD is a factorization of a matrix X

$$X = USV^T$$

such that U and V have orthonormal columns and S is diagonal. U and V are said to contain the *left* and *right singular vectors* while S is said to contain the *singular values*. If X has the shape $n \times m$, $m < n$ the SVD matrices has the following shapes: U has the shape $n \times m$, S the shape $m \times m$ and V^T the shape $m \times m$ [DD90]. The crux of LSA, then, is that the singular values in S are constructed to be ordered in decreasing magnitude. I can then keep only the first k singular values and set the rest to zero. The product of the resulting matrices $\hat{X}$ can then be shown to be the closest approximation in a least-squares sense to X of rank k [DD90]:

$$X \approx \hat{X} = U_k S_k V_k^T$$

which is the matrix $\hat{X}$ of rank k with the best least-squares-fit to X.
While the matrix is normally reconstructed when doing LSA, I will not be doing that. Recall that I are dealing with a document-term matrix, or bag-of-words, with documents as rows and terms as columns. Applying SVD to this matrix with d documents and t columns will result in a U matrix containing a row for each document and a column for each component. The V matrix will contain a row for each term and a column for each component. Conversely, the V^T matrix contains a row for each component and a column for each term.

V^T can be interpreted as a set of topics, with the most significant topic residing in the first row and the least significant row residing in the m'th row. Each column of V^T corresponds to a term and the value for each row is the importance of that term in the given topic.

U can be interpreted as a set of documents, with each column being a topic. The value for each element, then, is the importance of the topic of the elements column in the document of the elements row.

The trick is to pick the correct number of topics to include. I want to reduce the semantic space to a dimension that would make it easy for a human being to interpret what the general sentiments of the answers to a open-ended question are, without losing too much information. One method could be to calculate the variance explained for each component, but that is not really a good way to measure the quality of the number of topics picked. For now, I will be content with a performance based criteria. In other words, I will experiment with different values and see which one performs the best for this set of data.

Having picked a suitable number of topics, I can then cluster the documents according to the weights for each topic. This will be explained in a later section.

It should be noted that traditionally a term-document matrix is used instead of a document-term matrix. This would entail that U would be the term-feature matrix, V the document-feature matrix and V^T the feature-document matrix. For my purposes, a document-term matrix was more suited, since I would have to transpose both U and V^T to get the matrices I truly want. The reason for this, is that U would then contain a row for each term and a column for each topic and V^T would contain a row for each topic and a column for each document. This is the opposite, so to say, of what I wanted. Since there are no consequences except the matrices being 'mirrored', I decided to use a document-term matrix.

2.2.2 Non-negative matrix factorization

Another way of extracting topics from a document-term matrix, is *non-negative matrix factorization* (*NMF*). As with SVD, it involves factorizing the original matrix into several matrices. Where it differs from SVD, and LSA, is that it introduces an additional constraint and that is has no singular values. The rank of the new matrices can also vary, which means the factor of dimensionality reduction has to be decided when computing the factorization. The additional constraint enforced by NMF, is that the resulting matrices of the factorization has to be positive. This can make the result easier to interpret, since negative correlation has been eliminated.

The factorization consists of two matrices in NMF: A term-feature matrix W and a feature-document matrix H [LS99]:

$$V \approx WH \ .$$

The rank of W and H can be chosen arbitrarily, which decides the number of topics, or features, that the data is reduced to. This means that granularity can be controlled in two ways, instead of only one. By reducing or increasing the rank of the matrices, topics will be either expanded or collapsed, resulting in finer or coarser topics. In addition to this, the minimum weight required for a term to appear in a topic can be adjusted, just as with LSA.

Note that the factorization mentioned above depends on the data being in a term-document matrix. As mentioned, this results in a term-feature matrix (when I want a feature-term matrix) and a feature-document matrix (when I want a document-feature matrix). I could easily transpose the resulting matrices to get the result I want, but since I am already using a document-term matrix previously, it is far easier to continue using this. This does mean that the matrices are swapped, so to say, with W being a document-feature matrix and H being a feature-term matrix.

The NMF algorithm used in this project is very similar to the algorithm used for k-means clustering (or rather, they are both similar to the *Expectation-Maximization algorithm*). Actually, the NMF problem has been found to be a generalization of the k-means problem [DHS05]. As such, the current implementations suffer from some of the same problems that the k-means clustering algorithm suffer from: Non-deterministic, dependent on initialization and so on.

There is also the case of deciding on the update function. I will be using euclidean distance, but I am sure a case could also be made for divergence instead.

2.3 Clustering

A well suited method for clustering the documents after performing LSA and NMF on the data is *k-means clustering*. Solving the k-means problem means placing k centroids and assigning each data point to a centroid, such that the mean of the distances to each centroid is as small as possible. This is actually a NP-hard problem [ADHP09][MN09], but algorithms that provide a reasonable solution fast exist. One such algorithm is *Lloyd's algorithm*, which is commonly known as simply the k-means algorithm. In general, the algorithm can be described by the following steps:

1. Place k centroid in the virtual space of the data.
2. Assign each data point to the nearest centroid.
3. Recalculate the position of each centroid to be the mean of the assigned points.
4. Repeat steps two through three until there is no change.

Lloyd's algorithm is not deterministic, however, and given a bad initialization of the centroids can provide an unsatisfactory solution.
To solve the problem of how to initialize the centroids, there exists several methods. Two very common ones are *Forgy* and *Random Partition*. Forgy picks k random data points from the data set as the initial centroid, while Random Partition randomly assigns each data point to a cluster and then sets the initial centroids to be the means of each cluster. The one used in this project is called *k-means++* [AV07].

The k-means++ algorithm in its entirety is as follows:

1. The first centroid is chosen randomly from all the data points.

2. Calculate the distance $D(x)$ for each point x and the nearest centroid.
3. Choose a new data point as the next centroid, with the probability of choosing a data point x being proportional to $D(x)^2$.
4. Repeat steps two through three until k centroids has been placed.

Using this algorithm would seem to increase the computation time, and indeed the initialization is slower than other methods, but the k-means algorithm actually converges much faster when initializing in this way. This means the computation time is actually lowered, even though the initial selection of centroids is slower [AV07].

Aside from having to deal with the problem of selecting the initial centroids, there is also the matter of how many clusters to use. This is a problem up for debate, with several different approaches on how to solve it. The simplest one is the rule of thumb $k \approx \sqrt{n/2}$, where n is the number of data points [Mar79]. In the future other metrics such as the *Akaike information criteria* or the *Bayesian information criterion* could be explored.

Chapter 3

Implementation

In this chapter, a general description of the implementation of the methods examined in the theory chapter will be provided. Beginning with the bag-of-words representation, I will go through each of the methods described in chapter 2 and describe how I have implemented them.

3.1 Data representation

As mentioned in the previous chapter, the data is going to be represented as a bag-of-words. This representation is implemented in the `BagOfWords` class. The important attributes in this class is the document-term matrix, the list of terms and a dictionary for looking up which index belongs to a given term.

The class can either be instantiated with a list of answers or with the path to a file containing the answers. If instantiated with a path, the answers will be read from the file to a list. After this, each answer will be tokenized and stemmed, as explained in the theory chapter. This results in a list, with each entry corresponding to the stemmed tokens of an answer. From this list, I create the full set of unique terms, which is used to look up which term a given index is. I also need to be able to do a look up the other way - i.e. look up the index given a term. This is done by creating a dictionary from two zipped lists: The term list and a list containing every integer from 0 to the length of the term list.

With the foundation in place, I can now begin creating the document-term matrix. I simply iterate through each token of each answers, look up the index of the token and increment the corresponding element of the matrix:

```
matrix = np.zeros([len(tokens), len(terms)])
for doc, row in enumerate(tokens):
    for token in row:
        term = index_lookup[token]
        matrix[doc, term] += 1
```

This creates the document-term matrix. The full code listing can be found in the appendix.

3.1.1 TF-IDF

The `BagOfWords` class also contain the possibility to TF-IDF transform the data. This is done in the method **`tfidf_transform`**:

```
def tfidf_transform(self):

    num_docs = self.num_docs
    tdo = [None] * len(self.terms)

    for doc, row in enumerate(self.matrix):
        num_words = reduce(lambda x, y: x + y, row)
        for term, freq in enumerate(row):
            if freq != 0:
                if tdo[term] is None:
                    tdo[term] = sum(
                        [1 for d in self.matrix if d[term] > 0])
                tf = freq / num_words
                idf = log(abs(num_docs / tdo[term]))
                self.matrix[doc][term] = tf * idf
```

First, I get the number of documents in the bag-of-words and initialize the term-document-occurrence list to be a an empty list for each unique term. Then I iterate through each row of the matrix. I calculate the number of words by using the Python function `reduce`, which takes a function (in my case a *lambda* *expression*) and a list. In short, I simple sum the first two elements of the list, then sum the result of that with the third element and so on, until I have the total number of words in the document. This is used later, when calculating the term frequency.

Next I iterate through each term that actually occurs in the document and calculate the tf-idf. This is done in four stages: Calculate how many documents the term occurs in (tdo), calculate the term frequency (tf), calculate the inverse document frequency (idf) and last, calculate the product of tf and idf (tf-idf). This is all done in accordance with the procedure explained in the theory chapter.

3.2 Latent semantic analysis

The latent semantic analysis is implemented in the class `LSA`. After being instantiated with a `BagOfWords` object, the SVD will be calculated and the U, S and V^T matrices are extracted. To calculate the SVD, I use the Scientific Python, `scipy`, package. `scipy` in turn uses the Fortran package *Linear Algebra PACKage* (*LAPACK*). LAPACK includes a highly efficient, and widely used, implementation of SVD, which made it undesirable to attempt to implement it ourselves.

After having extracted the matrices from the SVD, it is possible to find the desired number of topics and extract what terms, that are above the threshold, are contained in the topic. This is done in the `_find_topics` function:

```python
def _find_topics(self, num_topics, min_weight):
    topics = [{} for _ in range(num_topics)]
    for topic, terms in itertools.islice(enumerate(self._vt),
        0, num_topics):

        for term, weight in enumerate(terms):
            if abs(weight) > min_weight:
                topics[topic][self.bag.terms[term]] = weight

    return topics
```

First I initialize the topics list to be an empty dictionary for each topic. Then I iterate through each component, or topic, in V^T, using the `itertools.islice` function to only iterate through the number of topics I need. After that, all that remains is to iterate through each term in the current topic, check if the weight exceeds the minimum weight and then look up the term if it does.

Something similar is done to only include the number of topics I want in each document vector of U:

```
def _find_doc_weights(self, num_topics):

    doc_weights = []
    for doc, topics in enumerate(self._u):
        doc_weights.append(list(topics[:num_topics]))

    return doc_weights
```

I simply iterate through each document and slice the list to only include the topics I want. Now that I have a list of documents defined by the importance of each topic, I can use k-means to cluster the documents. To do this, I use the `scikit-learn` implementation of k-means.

3.3 Spellchecker

Early on a problem with the data became apparent: Respondents do not take the time to spell check their answers. This poses a problem, since besides polysemy and synonymy a term can now also be spelled in a million ways. To account for this, I set out to implement a spelling corrector in Python. My implementation builds heavily on Peter Norvigs short spelling corrector, which can be found here: http://norvig.com/spell-correct.html.

To sum up what Peter Norvig writes on his webpage, what we want is the correction c out of all possible corrections such that

$$\underset{c}{\operatorname{argmax}}(P(c|w))$$

where w is the original word. Using *Bayes' Theorem*, we get:

$$\underset{c}{\operatorname{argmax}}(\frac{P(w|c)P(c)}{P(w)})$$

Since $P(w)$ is the same for all c, we can ignore that part:

$$\underset{c}{\operatorname{argmax}}(P(w|c)P(c))$$

There is three parts to the resulting expression. $argmax_c$ simply makes sure we get the best probability score by going through all c. $P(w|c$ is called the *error* *model.* This is how likely an respondent has entered w when the correct word was c. $P(c)$ is called the *language model.* This is how likely the word c is to appear in a Danish text.

The reason for expanding the original, seemingly simpler, expression is that $P(c|w)$ implicitly contains both the probability for c to be in the text, and the probability of the change from c to w. It is simply cleaner to explicitly separate the factors.

The next step is to find a large Danish corpus, that can be relied upon for word frequency. Then the corpus can be read and the frequency for each word saved. This is later used to calculate $P(c)$.

Once that is done, the different permutation of w has to be generated. All permutations of *edit distance* two are created, just to be safe. Each of the permuations are looked up in our word frequency list, since there's no use in including words we don't even know.

Last, the error model needs to be defined. The one Peter Norvig uses, and which I used, is somewhat naive: All known words of edit distance one is considered infinitely more probable than known words of edit distance two and known words of edit distance 0 are infinitely more possible than known words of edit distance one. This is done by first looking at possible candidates with no edit distance, then at possible candidates with one edit distance and finally at possible candidates with two edit candidates, short-circuiting as soon as a set of candidates are found.

While I had a hunch that erroneous spelling could have an influence on the analysis, I had trouble getting the spelling corrector to work in a reliable fashion. Peter Norvig was able to achieve 67% correct with his English corpus. I had no real way of testing my precision, but it was evident by looking at samples of corrections that it was not working very well. I assume my corpus was at fault, even though I tried several. If I had more time I would try to collect a custom corpus that fit the words in my data or use the data itself as a corpus. This might affect the stemming process, but could work regardless. In any case, I decided to drop the spelling corrector before it consumed too much of my time and the results has not seemed to suffer too much because of it.

3.4 The framework

The original aim of this project was to end up with a product that could be used by Mindshare to analyze answers to open-ended questions. While this is still the goal, the system used by Mindshare to analyze survey data (and the system which this project is going to be integrated in) is currently being reworked. This makes it difficult to create a plug-and-play executable for Mindshare. Instead, I have implemented what amounts to a framework of sorts. It consists of routines to perform LSA, NMF and to read data into a bag-of-words model. Once the Mindshare system is complete, it should be easy to integrate my framework in their system.

Once everything is integrated, a heatmap of documents vs. topics as well as the topics should be an addition to the automatically generated reports generated by Mindshares system. Ideally, a way of changing the parameters of the latent analysis should be a possibility. This would enable human intervention in order for the best possible result, since the number of topics, clusters and the magnitude of the minimum weight for a term to be included can change from data set to data set.

Chapter 4

Results

Both LSA and NMF was used on the same data set (with and without being TF-IDF transformed). This chapter will examine the results of both methods as well as the effects they have on the clustering.

I will begin with the task of extracting topics, first without using TF-IDF and then with TF-IDF. LSA and NMF will be compared in both cases. Then I will move on examining the resulting clustering for LSA and NMF.

4.1 Topic extraction

One of the main part of this project is to extract what topics are in the answers. The data is read into a bag-of-words model and then analyzed with LSA and NMF.

4.1.1 Without TF-IDF

First I run LSA and NMF on the data without doing TF-IDF. The result of this can be seen below.

4.1.1.1 LSA

Doing LSA on the data without any TF-IDF transformations yielded some fairly interesting results. Let's examine the first three topics using a minimum term weight of $|0.1|$

Topic #1

Variance explained: 0.0932

telefon: -0.102

få: -0.105

mobil: -0.119

abonnement: -0.130

kan: -0.150

køb: -0.174

3: -0.553

bil: -0.724

While the fact that all the terms are negative might be confusing, this is actually a useful topic. The topic is generally dominated by the terms *'3'* and *'bil'* (stemmed form of *cheap*) with the rest being about the subscription. The topic would seem to be about 3 having cheap subscriptions, explaining roughly 9% of the variance.

Topic #2

Variance explained: 0.0535

3: 0.704

køb: 0.218

bil: -0.644

The second topic is heavily dominated by the term *'3'*, which is the company the survey is done for. The second term in the topic is *'køb'*, with a fairly significant weight but no where near the first term. The third term is *'bil'*, the stemmed form of *'billig'*. This term is negatively correlated to the two other terms, however. This does not necessarily mean that the respondents are saying *'3'* is expensive, only that they do not mention *'billig'* often when mentioning

'3' and *'køb'*. It would be safe to assume that the second topic is something akin to *'køb 3'*, that is that the message of the commercial is *'buy 3'*. Without having been involved in the process of designing the commercial, I assume this is pretty correct. The second topics accounts for ≈ 5% of the variance.

Topic #3

Variance explained: 0.0343

køb:	0.544
kan:	0.440
få:	0.254
abonnement:	0.251
mobil:	0.206
famili:	0.186
tdc:	0.160
hel:	0.153
telefon:	0.141
saml:	0.123
pris:	0.115
bil:	−0.135
3:	−0.379

The third topic is much more scattered, even containing what might be considered stop-words in this context. Again, the term *'køb'* is one of the dominating factors of the topic, but the rest is more varied. However, combining the terms would give a topic of something in the area of *'buy family mobile subscriptions from TDC'* with the term *'3'* actually being negatively correlated, which is interesting. This would indicate that there is a significant amount of respondents that think the commercial is about TDC and not 3. I think increasing the minimum weight for a term to be included would improve the results for this topic. That is, it would not change the results as such, but it would make it easier to interpret. If the minimum weight is increased to 0.2, the result is:

Topic #3

Variance explained: 0.0343

køb:	0.544

kan: 0.440

få: 0.254

abonnement: 0.251

mobil: 0.206

3: −0.379

Sadly, the correlation with the term *'TDC'* is lost in this case, but the negative correlation with *'3'* is still there. While not as explicit as having *'TDC'* in the topic, the result still says the topic is clearly *not* used in conjunction with *'3'*. At the very least, this would give a clue that the respondents has failed to identify the brand of the commercial. If this minimum weight is applied to all three topics, the result is:

Topic #1

Variance explained: 0.0932

3: −0.553

bil: −0.724

Topic #2

Variance explained: 0.0535

3: 0.704

køb: 0.218

bil: −0.644

Topic #3

Variance explained: 0.0343

køb: 0.544

kan: 0.440

få: 0.254

abonnement: 0.251

mobil: 0.206

3: −0.379

The first topic is still not very helpful and the second topic is unchanged. As explained, the third topic is more easily interpreted but some of the information is lost. The choice would seem to be granularity versus accessibility. In any case, it is a parameter that would have to be experimented with for each data set before a judgment of which value is best can be made.

Normally much more than three topics would be picked. To get an image of how many topics that should be included, it can be useful to calculate the variance explained by number of principal components. This is done by squaring each singular value and dividing by the sum of all the singular values squared. The variance plotted can be seen in figure 4.1.

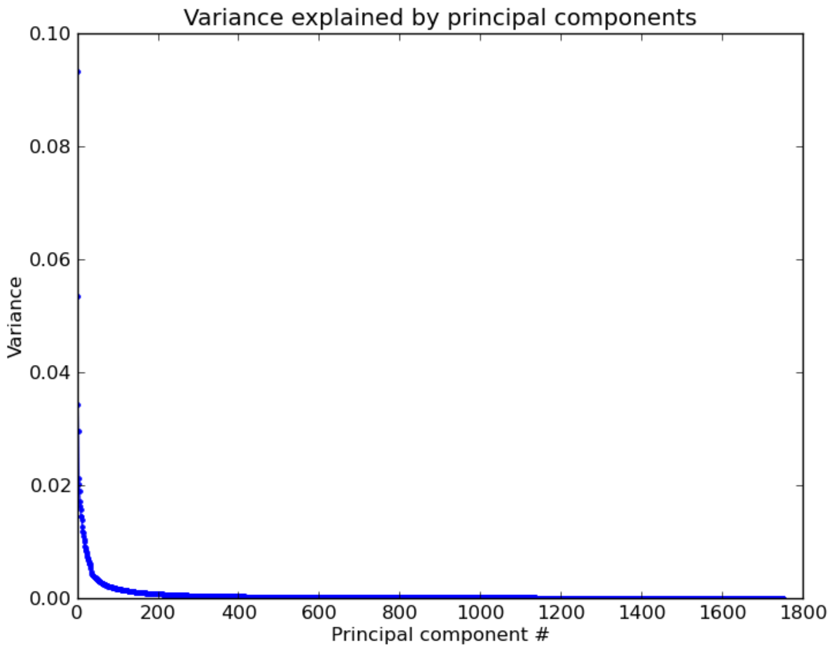

Figure 4.1: The variance explained by each principal component.

It might not be entirely clear from this plot but I calculated that five components explains $\sim$ 25% of the variance, 27 components explains $\sim$ 50% of the variance, 124 components explains $\sim$ 75% of the variance and 370 components explains $\sim$ 90% of the variance. From this I would judge that including anywhere from 10 - 30 topics should be sufficient to cover a significant part of the data.

4.1.1.2 NMF

The same experiment is done using non-negative matrix factorization. To begin with, the minimum weight is set to 0.2. The first topic:

Topic #1

bil:	5.07
telefoni:	0.464
bredbånd:	0.393
oist:	0.335
telefon:	0.328
mobil:	0.292
abonnement:	0.277
mobiltelefoni:	0.209

The topic is heavily dominated by the term *'bil'*, which is the stemmed version of *'billig'* (cheap). The rest range from *'telefon'* to *'bredbånd'* to *'oist'* (telephone, broadband and OiSTER - a cheaper subbrand of 3). All in all a fairly clear topic, the message being something like *'OiSTER has cheap telephone/broadband subscriptions'*.

Topic #2

3:	5.26
netværk:	0.549
køb:	0.355
godt:	0.324
andr:	0.317
kund:	0.307
selskab:	0.286
skift:	0.260
tilbud:	0.223
ved:	0.214
hurt:	0.210

Again, the topic is mainly about one term: *'3'.* The rest could be interpreted to be a mix of two themes. One describing the network (*'netværk'*, *'hurt'*, *'godt'*) and one describing the action of changing carrier (*'køb'*, *'selskab'*, *'skift'*, *'kund'*). Of course this is subject to the error inherent when interpreting something, but it seems like a fair conclusion. An advantage of NMF is that this probably would be broken down into several topics if the number of topics were to be increased.

Topic #3

køb:	2.80
kan:	2.29
få:	1.32
abonnement:	1.21
mobil:	0.983
famili:	0.795
telefon:	0.732
tdc:	0.685
hel:	0.669
pris:	0.576
saml:	0.537
så:	0.371
ved:	0.321
produk:	0.301
oist:	0.265
bredbånd:	0.256
reklam:	0.244
brug:	0.232
ny:	0.204
sælg:	0.200

The last topic is more of a mess than the last two. It seems to be a combination of several topics and it is difficult get something meaningful out of the topic. From the last two topics it would seem the result would really benefit from including more topics. The first three topics if the number of topics is increased to 20:

Topic #1

bil: 5.03

telefoni: 0.445

mobiltelefoni: 0.200

Topic #2

3: 5.33

kund: 0.218

Topic #3

kan: 3.58

brug: 0.257

få: 0.249

The first two topics are much more clearly defined this time around, though the third topic is still not very useful. If the next three topics are included, some of the terms from the third topic last time around might appear again:

Topic #4

køb: 5.03

produk: 0.445

Topic #5

abonnement: 5.33

famili: 0.218

tegn: 5.33

saml: 5.33

Topic #6

bredbånd: 3.58

hurt: 0.257

mobilt: 0.249

Indeed, some of the terms reappear. The terms *'køb'* and *'produk'* (*'buy'* and the stemmed form of *'product'*) is now one topic, *'abonnement'*, *'famili'*, *'tegn'* and *'saml'* (*'subscription'*, the stemmed form of *'family'*, *'sign up'* and *'collect'*) is another and *'bredbånd'*, *'hurt'* and *'mobilt'* (*'broadband'*, the stemmed form of *'fast'* and *'mobile'*) is the last. All three are fairly clear topics.

4.1.2 With TF-IDF

I repeat the experiment with TF-IDF transformed data. The process will not be as rigorous as the last section, now that the basis has been established.

4.1.2.1 LSA

A quick look at the first three topics with a minimum weight of $|0.1|$:

Topic #1

Variance explained: 0.0171

ok: 0.997

Topic #2

Variance explained: 0.0163557197664

oist: -0.101

mobil: -0.101

bredbånd: -0.129

telefoni: -0.140

3: -0.172

køb: -0.173

hurt: -0.254

bil: -0.485

priskr: -0.717

Topic #3

Variance explained: 0.0158312332337

bil: 0.441

hurt: 0.395

køb: 0.179

bredbånd: 0.153

3: 0.133

telefoni: 0.129

oist: 0.100

priskr: −0.687

The variance explained now seems to be much smaller across the board now, which means a larger amount of topics would be needed to explain the same amount of variance as without TF-IDF. Moving past that, it is obvious there is something wrong with the first topic. It is comprised entirely out of a single word with no substance (in fact, since the length of the entire feature vector is 1, the vector is almost entirely made up of the term *'ok'*). This term should probably be considered a stop word. The second and third topics make decent enough sense, but since the variance explained is much smaller this time around, their overall effect is also smaller. In addition to this, they seem to be made up of mostly the same terms, only with reverse sign. Increasing the minimum weight increases the readability, but makes the fact that the second and third topic is almost the same even more clear.

Topic #1

ok: 0.997

Topic #2

hurt: −0.254

bil: −0.485

priskr: −0.717

Topic #3

bil: 0.441

hurt: 0.395

priskr: −0.687

It does not seem like TF-IDF transforming has had any benefit in our case, it has in fact had the opposite effect.

Let's look at the variance again. See figure 4.2 for the plot.

Two things can be seen from this plot right away: The curve is less smooth and it flattens out much later than when TF-IDF was not used. The number of components needed to explain $\sim 25\%$ of the variance is 27 (the same number that explained $\sim 50\%$ without TF-IDF), 108 components to explain $\sim 50\%$, 265 components to explain $\sim 75\%$ and 539 components to explain $\sim 90\%$ of

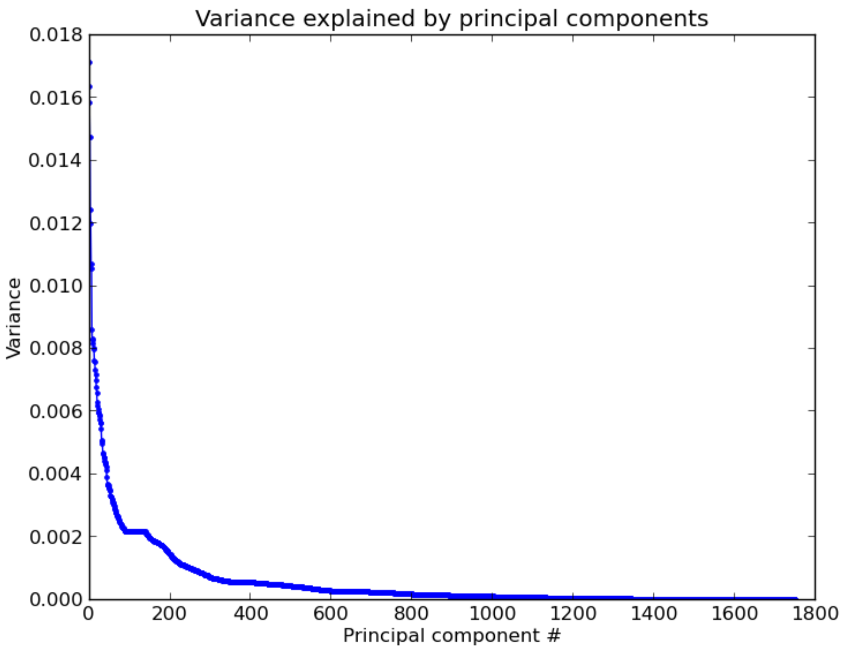

Figure 4.2: The variance explained by each principal component.

the variance. This indicates that using TF-IDF has actually scattered the data more, which is not the intention. The reason for this is probably that the answers are all fairly short sentences using a lot of the same words.

4.1.2.2 NMF

NMF with three topics and TF-IDF:

Topic #1

ok: 4.783

Topic #2

priskr: 4.26

Topic #3

hurt:	2.96
bil:	2.46
køb:	0.986
bredbånd:	0.968
3:	0.855
telefoni:	0.706
netværk:	0.549
mobil:	0.548
oist:	0.542
forbind:	0.481
telefon:	0.437
sælg:	0.427
abonnement:	0.418
mobilt:	0.414
pris:	0.355
godt:	0.333
mobiltelefoni:	0.274
intern:	0.268
rabat:	0.255
tilbud:	0.251
net:	0.227
produk:	0.217
tdc:	0.215
4g:	0.212

Again, the first topic is fairly useless. The second topic indicates that there is some talk about the term *'priskr'* (stemmed form of *'price war'*), which is fairly useful. The last topic is a complete mess, however. This could be rectified by increasing the number of topics. The number of topics is increased to 20 again, and the first six are picked out:

Topic #1

ok:	4.783

Topic #2

priskr: 3.94

Topic #3

bil: 3.80

mobiltelefoni: 0.466

telefon: 0.360

mobil: 0.313

abonnement: 0.313

Topic #4

hurt: 4.26

forbind: 0.705

4g: 0.270

intern: 0.254

går: 0.238

netværk: 0.210

net: 0.209

Topic #5

køb: 3.89

produk: 0.592

telefon: 0.363

mobil: 0.299

mobiltelefon: 0.286

samsung: 0.214

vor: 0.213

Topic #6

rabat: 4.24

As the last time, the third topic has been split into several topics. While they are clearer than with LSA, they are still not as clear as when TF-IDF was not used.

4.2 Clustering

Merely having the topics of the answers are not much use without some way of quantifying the use of those topics. To do this, I intend to cluster the data based on the document-feature matrix, treating each topic as a dimension. This means I end up with a clustering in a significantly smaller dimensional space than the original data. Having fewer dimensions should make the result much easier to interpret than simply clustering the original data. The reason for this, is that, hopefully, many of the terms has been combined, meaning the documents are now defined in terms of a few topics instead of *many* words.

Since the best results came from not TF-IDF transforming the data, the clustering will be done on the original data after having used both LSA and NMF. While the rule of thumb says the number of clusters should be $k \approx \sqrt{n/2} \approx 40$ (where n is number of data points) [Mar79] in our data set, I will be using a size of $k = 10$. 40 is simply too many clusters to be useful in a production environment, and 10 clusters provided very reasonable results.

I begin with the LSA clustering and follow up with clustering using the NMF data.

4.2.1 LSA

By plotting a heat-map for each cluster, I can, in rough terms, see if the answers grouped together are indeed alike. The clusters can be seen in figures 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11 and 4.12.

Light colors indicates a small absolute weight, while dark colors indicate large absolute weight.In this section I will only deal with the result of the clustering, but if used in a production environment, the content of the topics should be taking into account while interpreting the result of the clustering.

In general, the answers in each cluster seems fairly similar, though a good part of the clusters seem to be talking about most of the topics. A more useful result would be like the third or eighth cluster (figure 4.5 and figure 4.10): A few columns stand out compared to the rest, which means there is only positive - or negative - correlation with a small number of topics. This opens for the possibility for some useful interpretations, where as the more mixed clusters does not. Especially the fourth cluster (which can be seen in figure 4.6) is problematic, since it contains almost 50% of the answers.

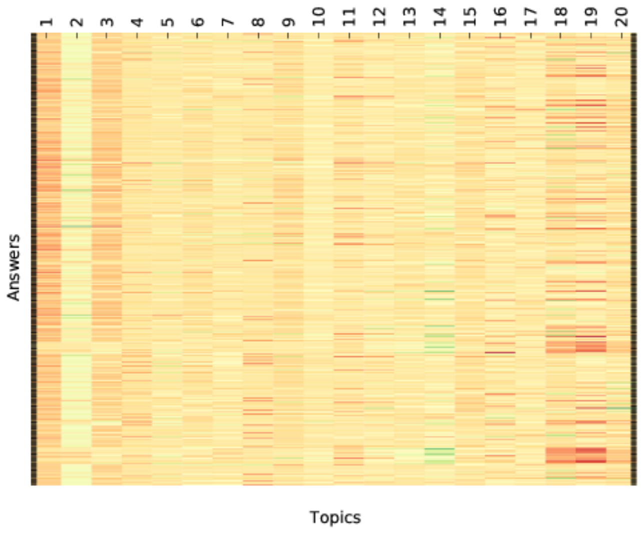

Figure 4.3: Heat-map of the first cluster using LSA. The cluster contains 192 answers.

To ensure the clusters are actually similar, some random samples from some of the clusters are compared. First 10 answers from the second cluster:

- at man kan få rabat ved køb af ny mobiltelefon som kunde ved 3
- pris
- at det skulle være billigt
- telefon på et hurtigt og billigt netværk
- at gribe chancen og købe en ny telefon
- Billigt tilbud
- at alle har en mulighed for at få en smartfone
- Køb en billig mobiltelefon
- at jeg kan købe en mobiltelefon til en god pris
- Billig telefon og abonnement hos 3

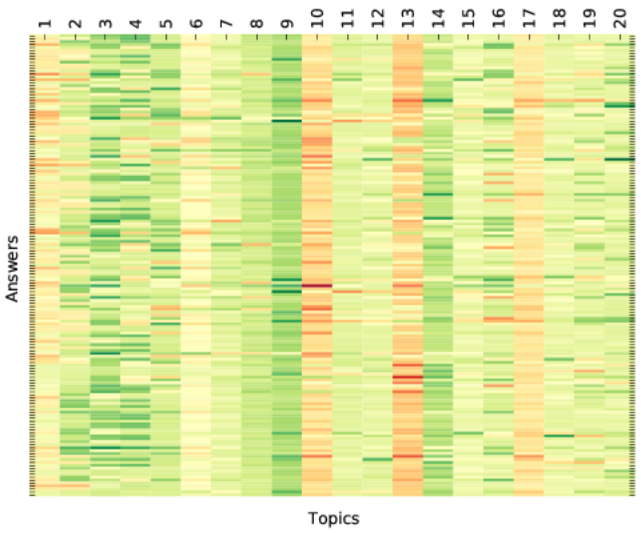

Figure 4.4: Heat-map of the second cluster using LSA. The cluster contains 157 answers.

All of them are about the price, or discount or being cheap. Judging by these 10 answers, the cluster is fairly accurate. Next is 10 from the fifth topic:

- At man kan købe god og billig mobiltelefoni hos 3
- 3 er billigere end andre selskaber 3 har et bedre netværk end andre selskaber
- skift til 3
- kapre kunder
- gode abonnementsvilkår
- at man skal tegne abonnement hos dem
- 3
- friste til salg skifte udbyder
- At 3 er et godt og sjovt mobilselskab
- at 3 kan kuncurere på prisen

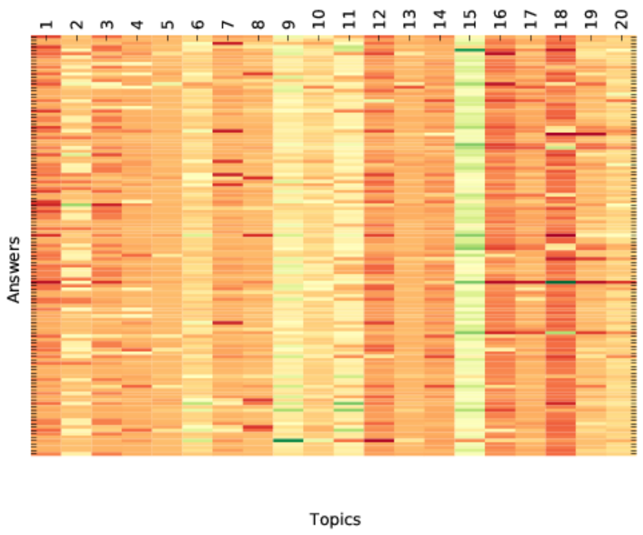

Figure 4.5: Heat-map of the third cluster using LSA. The cluster contains 125 answers.

The central term here is *'3'* and changing subscription to the company. In general the answers are comprised of *'3'* and adjectives describing the company. Finally, 10 answers from the ninth cluster:

- Køb 3 mobilt bredbånd
- At selv om man ikke er hurtig kan man alligevel få rabat
- Man skal gribe tilbuddet
- rabat til alle
- at alle kan få rabat på mobil telefoner hos 3
- Det er billigt
- Billig bredbånd i ferien
- lav hastighed til lav pris dobbelt op for at lokke flere til
- at sælge deres skrammel
- billig mobilnet

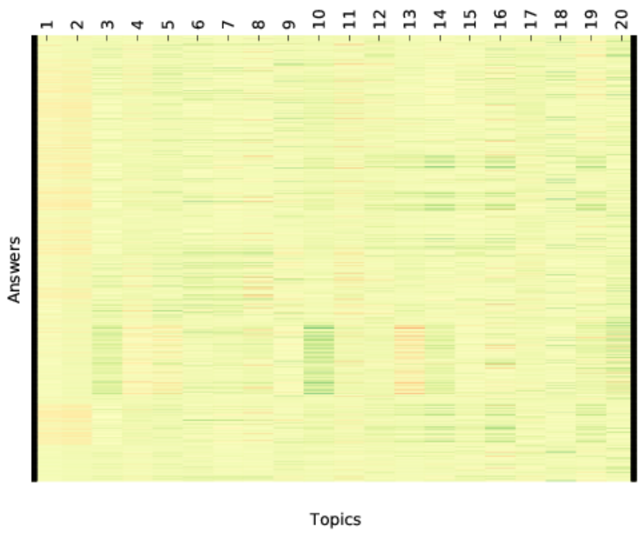

Figure 4.6: Heat-map of the fourth cluster using LSA. The cluster contains 1705 answers.

Again a cluster concerning how cheap 3 is. While the answers do fit fairly well together, an argument could be made for them being put in the second cluster as well.

Of course this is highly anecdotal, but I do think they serve as an indicator for how well the clustering has performed. In this case, it seems like it has performed quite well, though two of the clusters are quite similar.

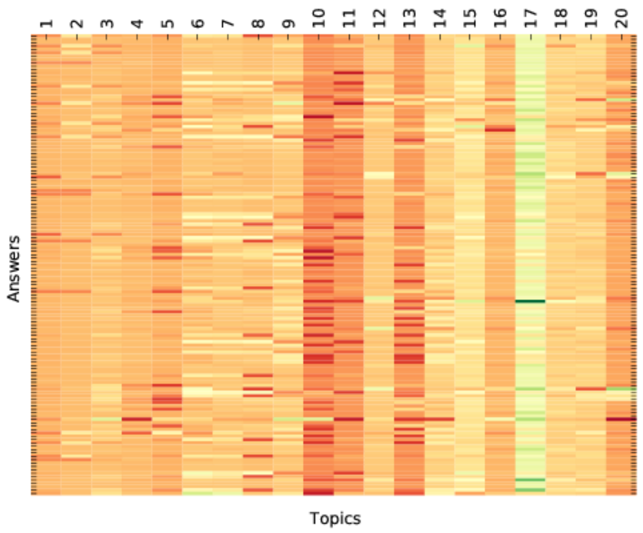

Figure 4.7: Heat-map of the fifth cluster using LSA. The cluster contains 137 answers.

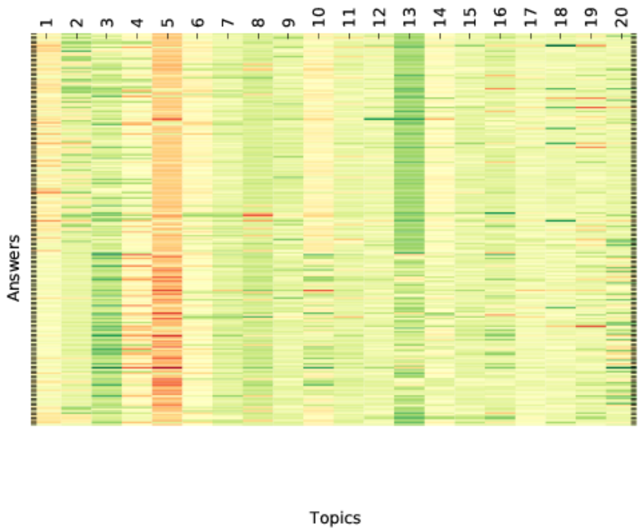

Figure 4.8: Heat-map of the sixth cluster using LSA. The cluster contains 208 answers.

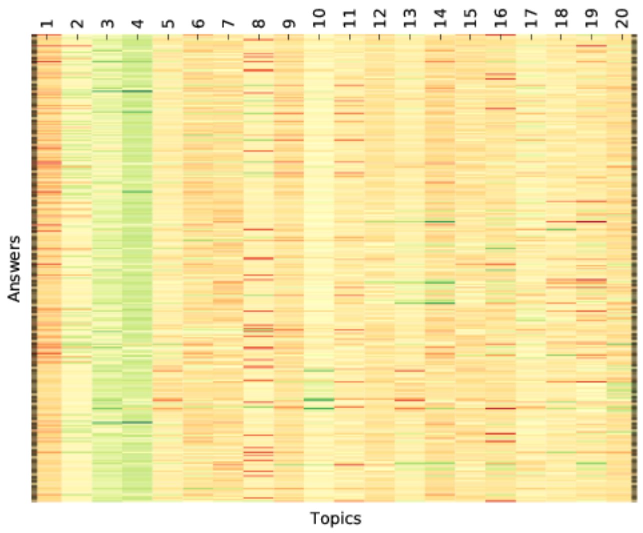

Figure 4.9: Heat-map of the seventh cluster using LSA. The cluster contains 298 answers.

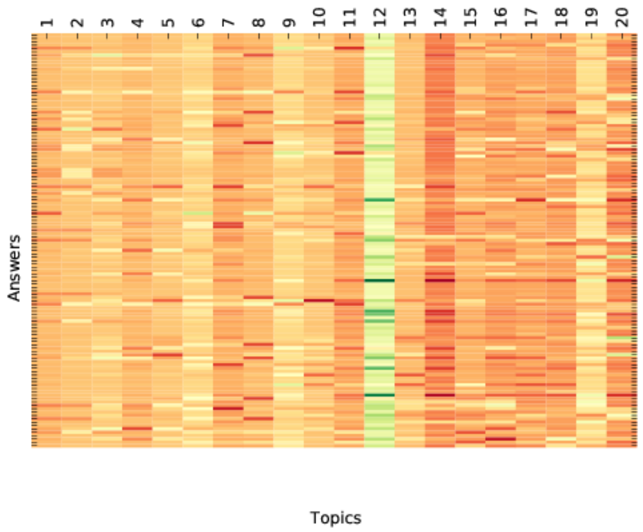

Figure 4.10: Heat-map of the eighth cluster using LSA. The cluster contains 123 answers.

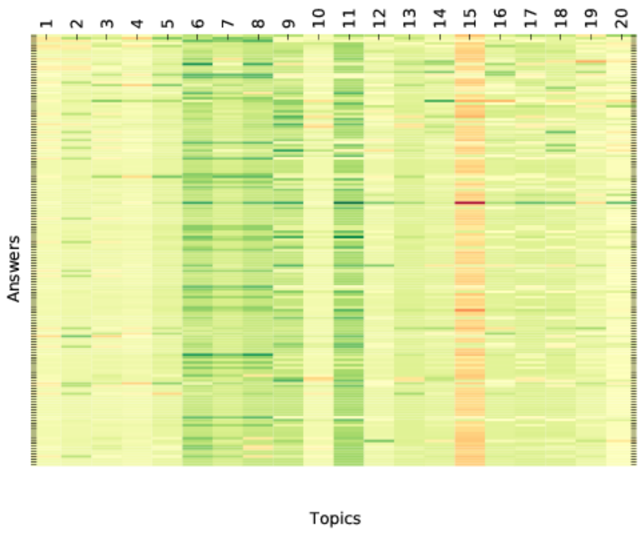

Figure 4.11: Heat-map of the ninth cluster using LSA. The cluster contains 165 answers.

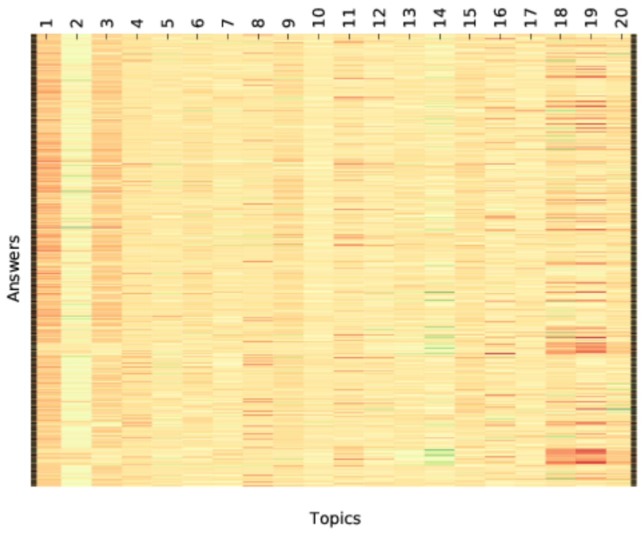

Figure 4.12: Heat-map of the tenth cluster using LSA. The cluster contains 480 answers.

4.2.2 NMF

I repeat the exercise with the NMF data. Since there are no negative values in NMF, the color-map is now from light-green to dark green, with darker meaning a higher value. This also mean the complexity of negative correlation is gone. Instead of looking at red columns vs. green columns, I just need to look for very dark green columns. A dark green columns means that there is a topic that this cluster does use a lot. The heat-maps can be seen in figures 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21 and 4.22.

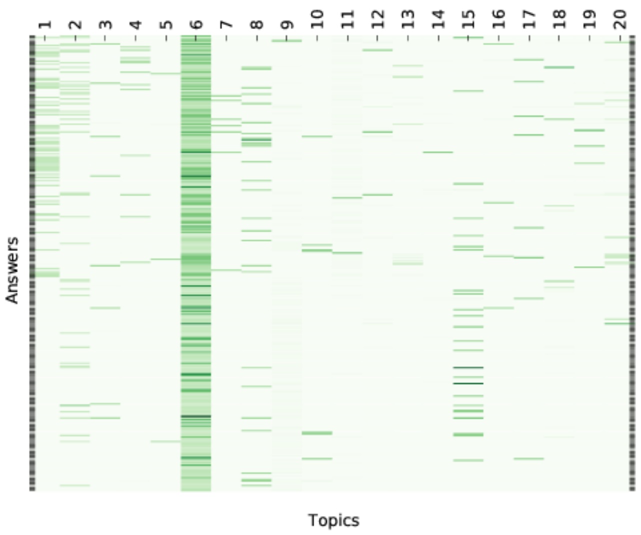

Figure 4.13: Heat-map of the first cluster using NMF. The cluster consists of 127 answers.

Only the fourth cluster (in figure 4.16) has no distinctive topic, with the third (in figure 4.15) only having a weak distinctive topic. Unfortunately, the third topic is the largest by far, containing $\sim$ 40% of the answers. The rest of the clusters all have only a few fairly distinctive topics, which is a very good result. With the constraint that NMF imposes, this makes the result very easy to interpret for anyone looking at a potential report containing these heat-maps.

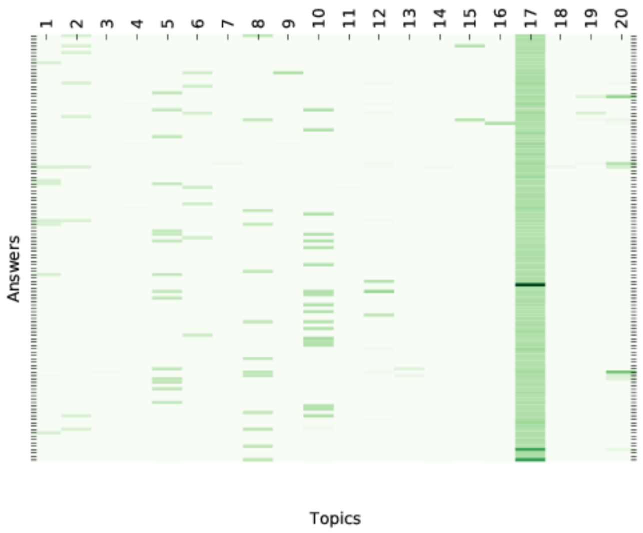

Figure 4.14: Heat-map of the second cluster using NMF. The cluster consists of 637 answers.

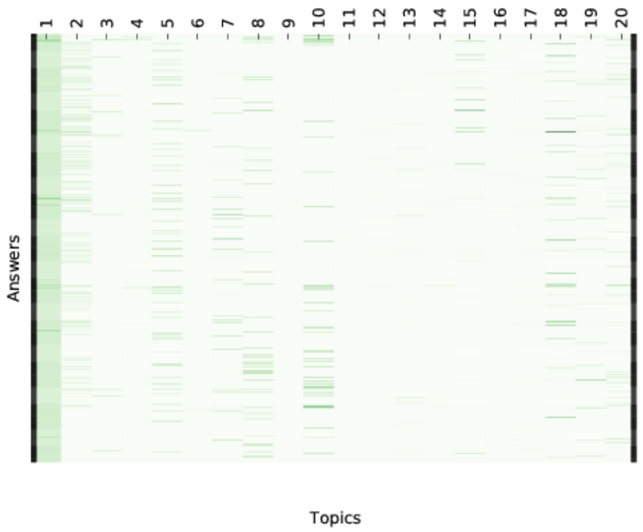

Figure 4.15: Heat-map of the third cluster using NMF. The cluster consists of 1450 answers.

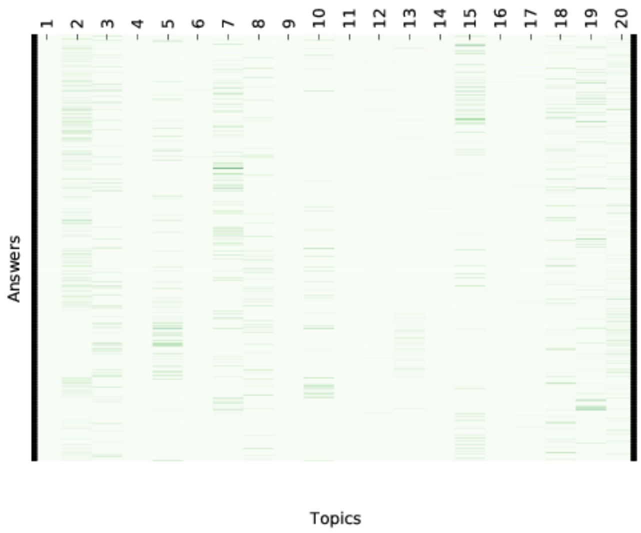

Figure 4.16: Heat-map of the fourth cluster using NMF. The cluster consists of 328 answers.

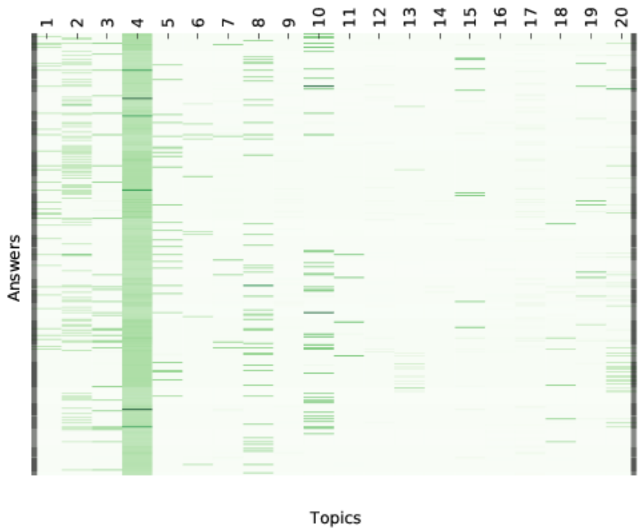

Figure 4.17: Heat-map of the fifth cluster using NMF. The cluster consists of 149 answers.

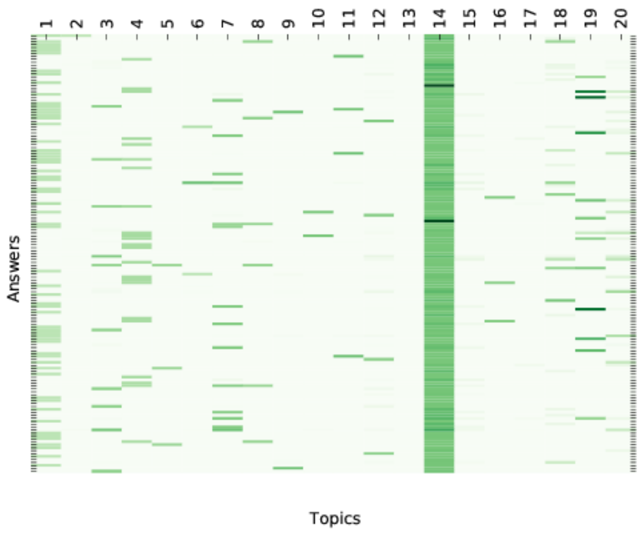

Figure 4.18: Heat-map of the sixth cluster using NMF. The cluster consists of 113 answers.

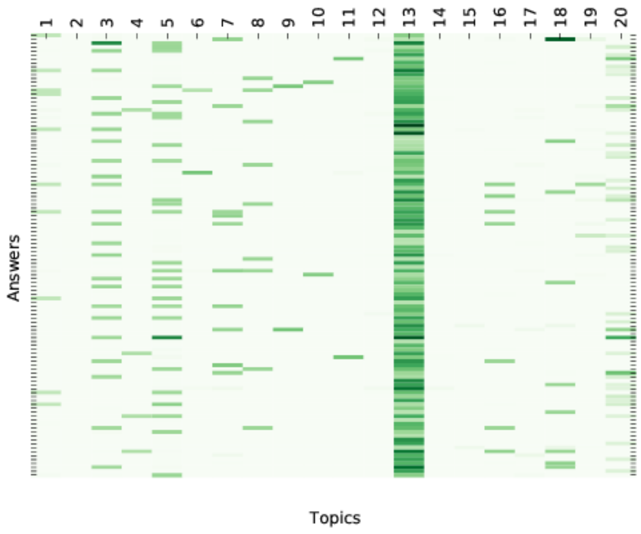

Figure 4.19: Heat-map of the seventh cluster using NMF. The cluster consists of 201 answers.

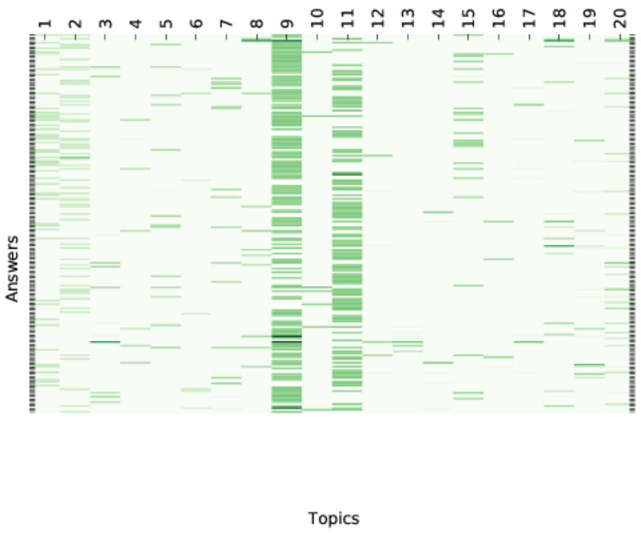

Figure 4.20: Heat-map of the eighth cluster using NMF. The cluster consists of 176 answers.

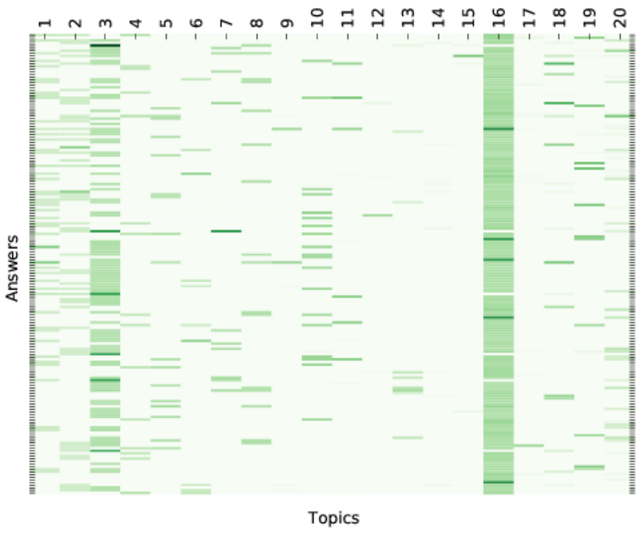

Figure 4.21: Heat-map of the ninth cluster using NMF. The cluster consists of 119 answers.

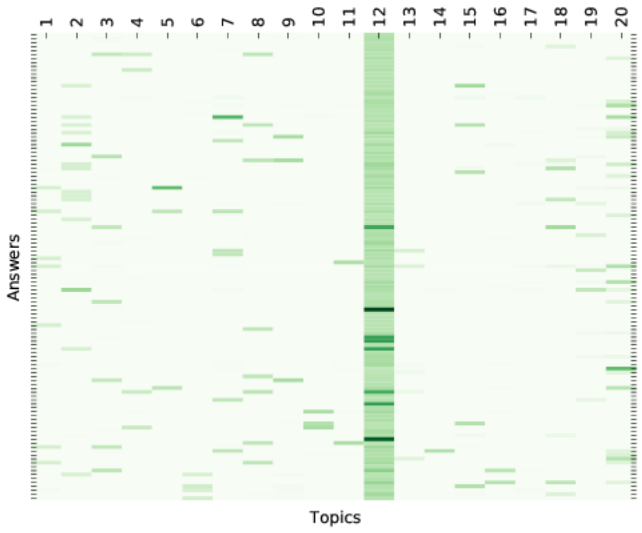

Figure 4.22: Heat-map of the tenth cluster using NMF. The cluster consists of 290 answers.

Again, I pick some sample answers from a couple of the clusters. First 10 answers from the third cluster (the largest cluster, with 1450 answers, ∼ 40%:

- Det er billigere end andre selvskaber
- billig mobilnet
- Billige løsninger
- bilig mobilabonnement
- Der er et abonnement der passer til alle til en billig pris
- Der er mange forskellige muligheder til billige penge
- 3 er billigt
- billig telefoni og data
- billigt
- priskrig og billig abonnement

All of these has the central term *'billig'*. By looking at the heat-map for the third cluster (4.15), it is evident that the main topic used is the first. Let's look at the first topic again:

Topic #1

bil:	5.03
telefoni:	0.445
mobiltelefoni:	0.200

Luckily, the first topic is indeed dominated by the stemmed form of *'billig'*. Next is 10 answers from the seventh cluster:

- 129 kr for 3 t tale mm
- 3 har et godt netværk
- rabat for eksisterende kunder
- mobilrabat til alle

- 3 giver rabat på nye mobiltelefoner
- mobiltelefon og abonomang
- rabat til alle
- hvis du ikke griber tilbudet NU er der en anden der snupper det og så er du lost
- rabat på taletid
- Nap tilbuddet når det er der det er snart forbi

Again there seems to be a central topic, this time discount. There are a couple of outliers, but that can't be avoided. One interesting fact is that answers with the term *'tilbud'* has been clustered with answers containing the term *'rabat'* - contextually two very similar terms, but very different in writing. Last, 10 answers from the tenth cluster:

- med 4G mobilbredbånd går det utroligt stærkt at være på internettet Det kommer til at blæse voldsomt uanset hvor du er
- At man kan bestille det nye 4G mobilbredbånd
- at det er en hurtig forbindelse
- Hurtige net og mobilen er vandafvisende
- 4 g
- at netforbindelsen er hurtig
- at 4g er hurtigere end 3g
- At 3 har super hurtigt mobilnet og vandsikre mobiler
- at mobil hos 3 er er sej og spændende
- Hunde bør vælge et bestemt selskab

While there isn't one central topic between these answers, they all seem to be fairly similar. Some are about 4G, some about a waterproof cellphone and some are simply about the company 3.

Again, it should be noted that this is anecdotal evidence for the performance of the clustering and should not be taken as more than an indicator. However, both for NMF and LSA the clustering seems to have performed reasonable when considering both the heat-maps and the samples of answers.

Chapter 5

Conclusion

I set out to create a system for Mindshare, that would enable them to incorporate open-ended questions in their questionnaires without having to worry about a huge overhead. While I have not implemented a plug-and-play executable for Mindshare, I have provided a framework that enables them to easily integrate latent analysis of answers to open-ended questions by using either LSA or NMF for the analysis and k-means++ clustering to group the answers. Both LSA and NMF produced reasonable results, but NMF outperformed LSA both in terms of accessibility (ease-of-use and understanding) and in terms of control of granularity.

With LSA, negative correlation is possible - and very likely - which can quickly confuse the interpreter. Suddenly a value from the document-feature matrix can not be taken at face value, but first has to be compared to the corresponding value in the feature-term matrix and vice versa. In addition to this, the only way to control the number of topics is to 'throw away' data - i.e. remove components. A benefit of this is that the number of topics can be changed after the computation expensive operation - SVD - has been performed, enabling control of the size of the topic base on the fly.
The only way of controlling granularity, or how much information is displayed, is to increase the minimum weight needed for a term to be shown.

This is in contrast to NMF where all values are positive. The result is suddenly much easier to interpret, since a large positive value in the document-feature matrix now, without a doubt, means that the corresponding topic is used extensively by the respondent. In addition to this, the number of topics is set when factorizing the original matrix. This means it is computationally expensive to change the number of topics, but also that we are not 'throwing away' data when including less topics. The granularity, then, can now be controlled both by adjusting the minimum weight and by adjusting the number of topics. To add to this, increasing or decreasing the number of topics truly expands or collapses topics instead of just ignoring 'smaller' topics (i.e. topics that explain less variance).

By looking at the heat-maps produced after clustering, it is obvious that NMF not only provides more control but also gives a better, and cleaner, result. The NMF clusters almost all have only a few distinct topics; a result that would be very easy for a human to interpret, especially considering the non-negative constraint of NMF. The LSA clustering is also decent, but the distinct topics are much weaker and the negative values makes the result more difficult to interpret. As mentioned, a fully red (negative) column does not mean anything in itself before comparing to the corresponding terms in the topic. After all, negative times negative will yield a positive value, so it may still be possible that the topic in fact contains a lot of terms with a high negative value, suddenly indicating a strong correlation.

As for the quality of the clustering itself, the result seems reasonable. The heat-maps gives an image of fairly similar clusters according to our analysis and by sampling some answers from different clusters I find them to be, anecdotally, similar.

If this framework is integrated in Mindshares future system and the cluster heat-maps as well as the topic lists are made a part of the monthly reports, a human will be able to quite easily determine the results of the answers to open-ended questions. For LSA, it would be to simply find distinct topics in each cluster and compare it to the topic list to see if this is a negative or positive correlation. NMF is even easier to interpret, having no negative values. Again distinct topics would be found in each cluster and then compared to the topic list - only this time there can only be a positive correlation.

Future work could include automating the process of choosing cluster size, e.g. by using an information criteria. A way to determine the number of topics to be found would be convenient as well, but I am not, as of yet, aware of a reliable way of doing this. In addition to this, I still believe the result could be improved by implementing a spelling corrector to make the term base more homogeneous.

Appendix A

Code

```
from __future__ import division

import itertools
from collections import OrderedDict
import string
from math import log

import nimfa
from scipy.linalg import svd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import k_means
from nltk.tokenize import wordpunct_tokenize
from nltk.stem.snowball import SnowballStemmer
from nltk.corpus import stopwords

class BagOfWords:
    """Represents a bag-of-words matrix."""

    def __init__(self, answers):
        if isinstance(answers, basestring):
            # Assume data is a path, which means we need to load the da
            answers = load_list_from_file(answers)
        # Tokenize and filter every data entry.
        tokens = [self._tokenize(answer) for answer in answers]

        # Filter the concatenated tokens to create the list of terms.
        terms = sorted(list(set(itertools.chain.from_iterable(tokens))))

        # Generate a dict for looking up the index of an attribute.
        index_lookup = dict(zip(terms, range(len(terms))))

        # Now that we have the data in a tokenized form, we can begin crea
        matrix = np.zeros([len(tokens), len(terms)])
        for doc, row in enumerate(tokens):
            for token in row:
                term = index_lookup[token]
                matrix[doc, term] += 1

        # Set the document-term matrix.
        self.matrix = matrix
        # The list of terms, used for getting the term given an index.
        self.terms = terms
        # And the dictionary for looking up the index of a term.
        self._index_lookup = index_lookup

    def _tokenize(self, doc, sort=False, unique=False):
        stemmer = SnowballStemmer('danish')
        tokens = [token for token in wordpunct_tokenize(doc) if token.lowe

        tokens = [stemmer.stem(token.lower()) for token in tokens]
        return tokens

    def tfidf_transform(self):

        num_docs = len(self.matrix)
        tdo = [None] * len(self.terms)

        for doc, row in enumerate(self.matrix):

            num_words = reduce(lambda x, y: x + y, row)

            for term, freq in enumerate(row):
                if freq != 0:
                    # Calculate the number of answers a term occurs in..
                    if tdo[term] is None:
                        tdo[term] = sum([1 for d in self.matrix if d[te
                    tf = freq / num_words
                    idf = log(abs(num_docs / tdo[term]))
                    self.matrix[doc][term] = tf * idf

class LSA:

    def __init__(self, bag_of_words, rank=20, min_weight=0.1, K=30):
        self.bag = bag_of_words
        self.rank = rank
        self.min_weight = min_weight
        self.K = K

        # Calculate SVD
        self._u, self._s, self._vt = svd(bag_of_words.matrix,
                                         full_matrices=False)

        self.variance_by_component = (self._s * self._s) / (self._s * s

        # Find feature_term_matrix
        self.feature_term_matrix = self._find_topics(rank, min_weight)

        # Find doc weights
        self.document_feature_matrix = self._find_document_feature_matr

        # Compute clusters
        self.centroids, self.clusters, self.inertia = k_means(
            self.document_feature_matrix, K, init='k-means++')

    def _find_topics(self, rank, min_weight):
        # Find the first rank feature_term_matrix and exclude all term
        # below min_weight.

        # Initialize feature_term_matrix to be a list of empty dictiona
        feature_term_matrix = [{} for _ in range(rank)]

        # For each topic and list of terms in the first rank components
        for topic, terms in itertools.islice(enumerate(self._vt), 0, ra

            for term, weight in enumerate(terms):
                if abs(weight) > min_weight:
                    # Look up the name of the term and add it to the
                    # appropriate topic dictionary.
                    feature_term_matrix[topic][self.bag.terms[term]] =
        return feature_term_matrix

    def plot_clusters(self, path=None):
        document_feature_matrix = self.document_feature_matrix
        cluster_labels = self.clusters
        K = self.K
        cluster_file = open('../lsa_clusters.txt', 'w')
        for i in range(K):
            cluster_file.write('CLUSTER␣#{0}\n'.format(i+1))
            cluster_file.write('Centroids:␣{0}\n\n'.format(self.centroids[
            doc_lookup = []
            cluster = []
            for doc, feature_term_matrix in enumerate(document_feature_mat
                if cluster_labels[doc] == i:
                    doc_lookup.append(doc)
                    cluster.append(feature_term_matrix)
            new_path = None
            if path is not None:
                new_path = '{0}{1}.pdf'.format(path, i + 1)
                # path = ''.join([path[0], str(i),'.', path[1]])
            # print 'Calling plot....'
            for doc in doc_lookup:
                cluster_file.write('\t{0}'.format(doc))
            cluster_file.write('\n\n')
            self._plot_heatmap(cluster, i + 1, self.rank, doc_lookup, path
        cluster_file.close()
        plt.show()

    def _plot_heatmap(self, cluster, cluster_number, rank, doc_lookup, pat

        # print 'Plotting....'

        cluster = np.array(cluster)

        cluster_norm = (cluster - cluster.mean()) / (cluster.max() - clust

        fig, ax = plt.subplots()
        ax.pcolor(cluster_norm, cmap=plt.cm.RdYlGn)

        fig = plt.gcf()
        fig.set_size_inches(8, 11)

        ax.set_frame_on(False)
        ax.set_yticks(np.arange(cluster_norm.shape[0])+0.5, minor=False
        ax.set_xticks(np.arange(cluster_norm.shape[1])+0.5, minor=False

        ax.invert_yaxis()
        ax.xaxis.tick_top()

        xlabels = [topic + 1 for topic in range(rank)]
        # ylabels = [doc for doc in doc_lookup]

        ax.set_xticklabels(xlabels, minor=False)
        ax.set_yticklabels([], minor=False)
        ax.set_ylabel('Answers')
        ax.set_xlabel('Topics')

        plt.xticks(rotation=90)

        ax.grid(False)

        ax = plt.gca()

        for t in ax.xaxis.get_major_ticks():
            t.tick1On = False
            t.tick2On = False

        for t in ax.yaxis.get_major_ticks():
            t.tick1On = False
            t.tick2On = False

        if path is not None:
            print 'saving to path: {}'.format(path)
            plt.savefig(path)


    def _find_document_feature_matrix(self, rank):
        # Find the doc weights for the number of feature_term_matrix ch

        document_feature_matrix = []
        for doc, feature_term_matrix in enumerate(self._u):
            document_feature_matrix.append(list(feature_term_matrix[:ra

        return document_feature_matrix

    def print_topics(self):
        """Prints the feature_term_matrix found in a readable format."""

        for topic, terms in enumerate(self.feature_term_matrix):
            print 'TOPIC␣#{0}:'.format(topic + 1)
            print 'Variance␣explained:␣{0}'.format(self.variance_by_compon
            # Make sure the terms is printed in a sorted fashion.
            terms = OrderedDict(sorted(
                terms.items(), key=lambda t: t[1], reverse=True))
            for term in terms.keys():
                print '\t{0}:␣{1}'.format(term.encode('utf8'), terms[term]
            print ''

    def change(self, rank=None, min_weight=None, K=None):
        """Recalculate feature_term_matrix, doc weights and clusters to fi

        # Update parameters if a new value is given.
        self.rank = rank if rank is not None else self.rank
        self.min_weight = min_weight if min_weight is not None else self.m
        self.K = K if K is not None else self.K

        # Recalculate according to the new values.
        self.feature_term_matrix = self._find_topics(self.rank, self.min_
        self.document_feature_matrix = self._find_document_feature_matrix(
        self.centroids, self.clusters, self.inertia = k_means(
            self.document_feature_matrix, self.K, init='k-means++')



class NMF(object):
    """docstring for NMF"""

    def __init__(self, bag_of_words, rank=20, min_weight=1, K=10, seed='nn

        self.terms = bag_of_words.terms
        self.rank = rank
        self.min_weight = min_weight
        self.K = K
        V = bag_of_words.matrix
        fctr = nimfa.mf(V,
                        seed=seed,
                        rank=rank,
                        method="nmf",
                        max_iter=12,
                        initialize_only=True,
                    update=update,
                    objective=objective)

        fctr_res = nimfa.mf_run(fctr)
        self.W = fctr_res.basis().getA()
        self.H = fctr_res.coef().getA()
        self.feature_term_matrix = self._find_topics(self.H, rank, min_
        self.document_feature_matrix = self.W
        self.centroids, self.clusters, self.inertia = k_means(
            self.W, self.K, init='k-means++')

    def _find_topics(self, feature_term_matrix, rank, min_weight):
        # Find the first rank feature_term_matrix and exclude all term
        # below min_weight.

        # Initialize feature_term_matrix to be a list of empty dictiona
        feature_term_matrix = [{} for _ in range(rank)]

        # For each topic and list of terms in the first rank components
        for topic, terms in itertools.islice(enumerate(feature_term_ma

            for term, weight in enumerate(terms):
                if abs(weight) > min_weight:
                    # Look up the name of the term and add it to the
                    # appropriate topic dictionary.
                    feature_term_matrix[topic][self.terms[term]] = weig

        return feature_term_matrix

    def change(self, rank=None, min_weight=None, K=None):
        """Recalculate feature_term_matrix, doc weights and clusters to

        # Update parameters if a new value is given.
        self.rank = rank if rank is not None else self.rank
        self.min_weight = min_weight if min_weight is not None else sel
        # self.K = K if K is not None else self.K

        # Recalculate according to the new values.
        self.feature_term_matrix = self._find_topics(self.H, self.rank,
        # self.document_feature_matrix = self._find_document_feature_m
        # self.clusters, self.centroids, self.inertia = k_means(
        #     self.document_feature_matrix, self.K)
    def print_topics(self):
        """Prints the feature_term_matrix found in a readable format."""

        for topic, terms in enumerate(self.feature_term_matrix):
            print 'TOPIC␣#{0}:'.format(topic + 1)
            # Make sure the terms is printed in a sorted fashion.
            terms = OrderedDict(sorted(
                terms.items(), key=lambda t: t[1], reverse=True))
            for term in terms.keys():
                print '\t{0}:␣{1}'.format(term.encode('utf8'), terms[term]
            print ''

    def plot_clusters(self, path=None):
        document_feature_matrix = self.document_feature_matrix
        cluster_labels = self.clusters
        K = self.K
        cluster_file = open('../nmf_clusters.txt', 'w')
        for i in range(K):
            cluster_file.write('CLUSTER␣#{0}\n'.format(i+1))
            cluster_file.write('Centroids:␣{0}\n\n'.format(self.centroids[
            doc_lookup = []
            cluster = []
            for doc, feature_term_matrix in enumerate(document_feature_ma
                if cluster_labels[doc] == i:
                    doc_lookup.append(doc)
                    cluster.append(feature_term_matrix)
            new_path = None
            if path is not None:
                new_path = '{0}{1}.pdf'.format(path, i + 1)
                # path = ''.join([path[0], str(i), '.', path[1]])
            # print 'Calling plot....'
            for doc in doc_lookup:
                cluster_file.write('\t{0}'.format(doc))
            cluster_file.write('\n\n')
            self._plot_heatmap(cluster, i + 1, self.rank, doc_lookup, path
        cluster_file.close()
        plt.show()

    def _plot_heatmap(self, cluster, cluster_number, rank, doc_lookup, pat

        # print 'Plotting....'

        cluster = np.array(cluster)
```

```
cluster_norm = (cluster - cluster.mean()) / (cluster.max() - cl

fig, ax = plt.subplots()
ax.pcolor(cluster_norm, cmap=plt.cm.Greens)

fig = plt.gcf()
fig.set_size_inches(8, 11)

ax.set_frame_on(False)

ax.set_yticks(np.arange(cluster_norm.shape[0])+0.5, minor=False
ax.set_xticks(np.arange(cluster_norm.shape[1])+0.5, minor=False

ax.invert_yaxis()
ax.xaxis.tick_top()

xlabels = [topic + 1 for topic in range(rank)]
# ylabels = [doc for doc in doc_lookup]

ax.set_xticklabels(xlabels, minor=False)
ax.set_yticklabels([], minor=False)
ax.set_ylabel('Answers')
ax.set_xlabel('Topics')

plt.xticks(rotation=90)

ax.grid(False)

ax = plt.gca()

for t in ax.xaxis.get_major_ticks():
    t.tick1On = False
    t.tick2On = False

for t in ax.yaxis.get_major_ticks():
    t.tick1On = False
    t.tick2On = False

if path is not None:
    print 'saving␣to␣path:␣{}'.format(path)
    plt.savefig(path)
```

Bibliography

[ADHP09] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. 2009.

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. 2007.

[DD90] Scott Deerwester and Susan T. Dumais. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 1990.

[DHS05] Chris Ding, Xiaofeng He, and Horst D. Simon. On the equivalence of nonnegative matrix factorization and spectral clustering. *Proc. SIAM Data Mining Conf*, 2005.

[KKH] Pentti Kanerva, Jan Kristoferson, and Anders Holst. Random indexing of text samples for latent semantic analysis.

[LJLS11] Travis D. Leleu, Isabel G. Jacobson, Cynthia A. LeardMann, and Besa Smith. Application of latent semantic analysis for open-ended responses in a large, epidemiologic study. 2011.

[LS99] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. 1999.

[Mar79] Kanti Mardia. Multivariate analysis. 1979.

[MN09] Meena Mahajan and Prajakta Nimbhorkar. The planar k-means problem is np-hard. 2009.