Web Intelligence - Crawler Miniproject

Erik Sidelmann Jensen
ejens11@student.aau.dk

Lasse Vang Gravesen
lgrave11@student.aau.dk

Dennis Jakobsen
djakob11@student.aau.dk

1 Crawler Miniproject

1.1 Crawler

The crawler retrieves the documents used for the search engine, it starts by using a list of seed urls. The seed site for ours was `http://en.wikipedia.org/wiki/Internet_of_Things`. Politeness was important, in that the crawler had to adhere to the 'robots.txt' file on websites and determine if the crawler was allowed to visit the url in question and at the same time the crawler should not visit a page too often, this was implemented by simply just waiting a second after each visit. A better approach to this would be to implement a map between a host and a timestamp, and then check the timestamp for the last visit of a host before visiting it again. Equality of content on pages had to be checked because there is no reason to have multiple pages with the same content in the search results, we did this using sketches with Shingling and Jaccard similarity. The general idea of shingling can be demonstrated using the following string: "This is a string", with a shingle size of 2, the resulting shingles would be "This is", "is a", "a string". These can then be used to check equality against a different document using the aforementioned Jaccard similarity.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The Jaccard similarity is then some value between 0 and 1, and if it is above the threshold, say 0.9, the two documents are said to be equal.

Sketching is then when you use multiple hashing functions on the shingles and get the minimum hashed shingle for each hashing function, and then run jaccard similarity on those.

The crawler also needs to extract urls from the document, it does this using a regular expression. To simplify the regular expression we only get the urls inside an href attribute. These urls then are normalized, fixed and added to the frontier if they have not already been visited or they are not in the frontier already.

The architecture is somewhat similar to the architecture shown in the slides, though there are some differences specifically with the 'robots.txt' check in that it does that check for each url it visits instead of performing the check before adding it to the frontier. See Figure 1.1.

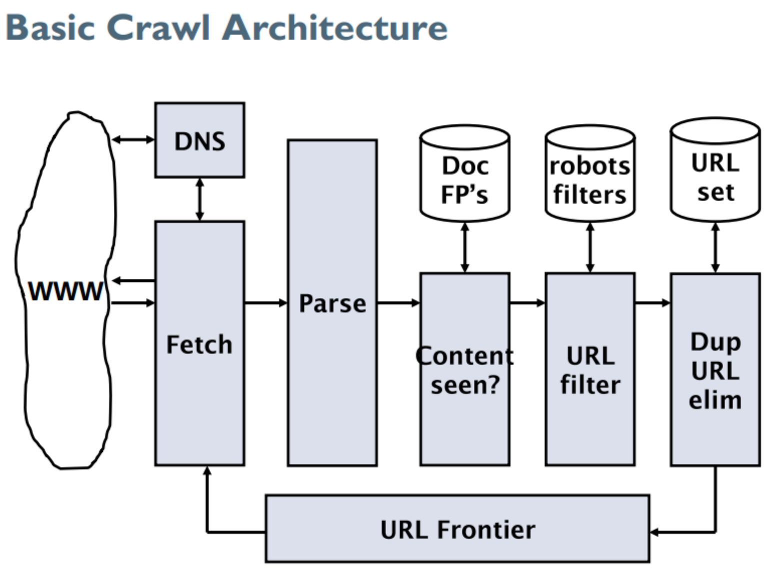


Figure 1.1: Basic Crawl Architecture from Slides

To sum up: It gets an url from the frontier, fetches the content if it can get past the 'robots.txt' checker, parses out the word content and checks if that has already been seen, and adds it to the url filter after normalizing and checking if it does not already exist in the frontier or already visited urls.

Cut Corners

The Crawler does not find every url in the visited urls, but rather just the ones that are inside href attribute.

The Crawler was limited to very few pages, 100-1000 because waiting a second after each page is limiting. This could be improved by using threading, it could also be improved by using a queue and if the crawler needs to wait just put the item back in the queue and get the next item instead.

The sketches for each document is kept in memory.

1.2 Indexer

The purpose of the indexer is to sort through the crawled documents, and create something that can be easily utilized by the search engine. It does this by providing an inverted index of the terms and which documents contain them and their frequency in said documents. We use an inverted index because then we do not need to investigate every document for matches but rather we can just look at the individual terms and see what documents match. The indexer works by retrieving the crawled pages in the database as an object that contains the html document, the url and the document id. The html document is converted into only words and these are tokenized by splitting them. These words are made into features, by stemming them using an English language stemmer. It also removes the English stopwords, such as ”the” or ”a”. It is important to note that the query gets the same treatment, in that it is tokenized, stemmed and that the stopwords are removed because if you do not do that the terms in the inverted index will not match. That feature(or term) is then added to the inverted index along with the id of the document.

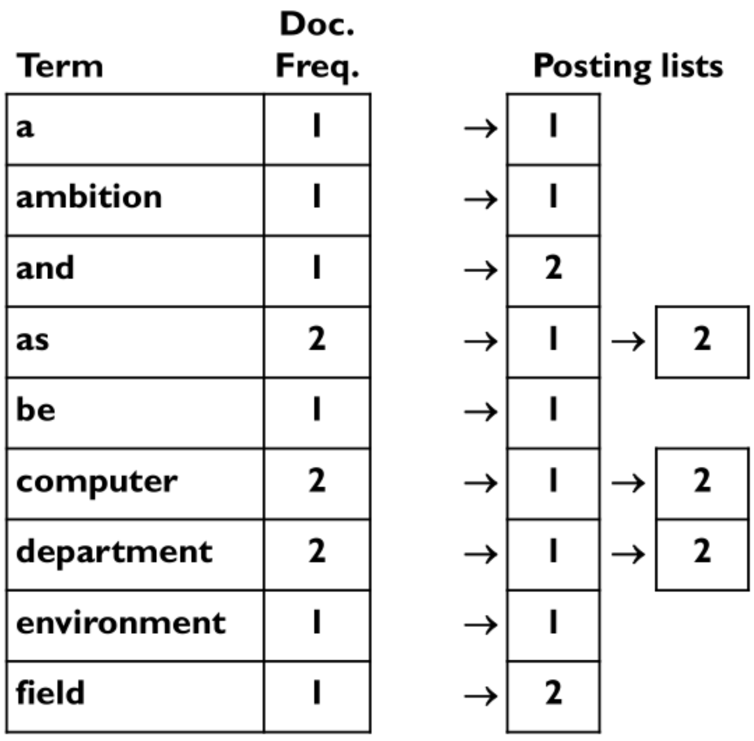
Adding to the inverted index works as such, it receives the term and the document id as mentioned, checks if the inverted index already contains the term.

If the inverted index does not already contain the term, it creates a list of postings, adds a posting with the document id, and adds that to inverted index using the term and a new posting list.

If the inverted index does contain it, it checks if the document id already has a posting because we might encounter the same word multiple times in the same document. It then sorts the postings.

Otherwise it increments the term frequency of that term for the current document.

The structure of the inverted index can be seen in Figure 1.2.



Term	Doc. Freq.	Posting lists
a	1	→ 1
ambition	1	→ 1
and	1	→ 2
as	2	→ 1 → 2
be	1	→ 1
computer	2	→ 1 → 2
department	2	→ 1 → 2
environment	1	→ 1
field	1	→ 2

Figure 1.2: The inverted index. Source: From the Lecture 3 slides page 36.

This inverted index is kept in memory.

The general idea of what the indexer does can be seen in Figure 1.3.

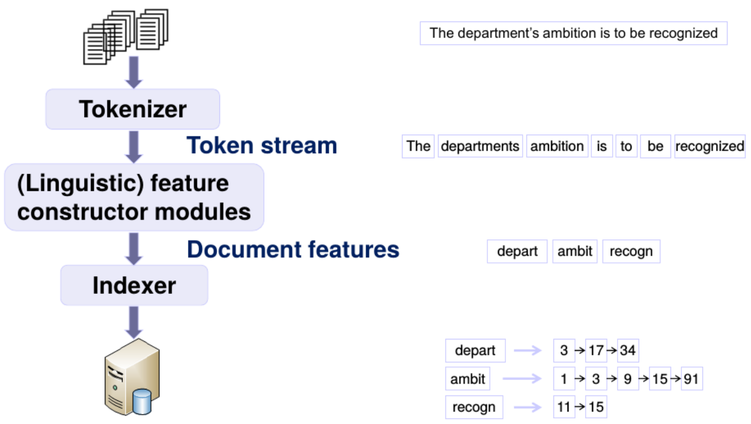

Tokenizer
Token stream
(Linguistic) feature constructor modules
Document features
Indexer
The department's ambition is to be recognized
The departments ambition is to be recognized
depart ambit recogn
depart → 3 → 17 → 34
ambit → 1 → 3 → 9 → 15 → 91
recogn → 11 → 15

Figure 1.3: Indexer. Source: From the Lecture 3 slides page 7.

1.3 Ranker

The purpose of the ranker is to get the most relevant results for the user given some query. For the ranker we used the cosine score algorithm based on tf-idf weighting. The cosine score algorithm uses cosine similarity between the angle of tf-idf vectors.

The general idea behind that can be seen in Figure 1.4

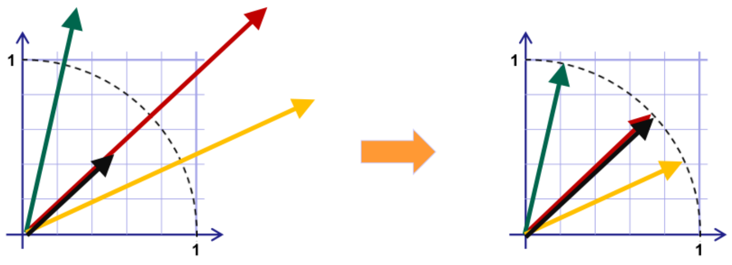


Figure 1.4: The idea behind cosine similarity. Source: From the Lecture 4 slides page 32.

What the ranker does specifically is that it takes a query, tokenizes it and constructs features using the stemmer and stopword removal because the query needs to be treated the same way as the documents were in the indexer.

The tf(Term Frequency) weight is the amount of times a document contains a term.

The tf* weight is

$$log_{10}(1 + tf_{t,d})$$

because relevance does not increase proportionally we need to log10 it.

The df(Document Frequency) weight is the amount of times a term is mentioned in all the documents.

The idf(Inverse Document Frequency) weight is

$$log_{10}(\frac{N}{df_t})$$

where N is the total amount of documents.

The tf-idf weight is then

$$tf * idf$$

Using tf-idf we can then calculate a score for each document using the cosine score algorithm mentioned previously, and this is then used to sort the results and provide a good output for the user, for example 10 documents.

tf-idf weighting has different variants, like ltc or lnc. The one we use is ltc.

We also implemented contender pruning using 'Champions Lists' where you precompute for each term t, r amount of docs of the highest weight(the tf weight) and this list is then looked at first instead of the default list of postings as a champion list.

Our implementation is vulnerable to keyword stuffing for the ranker, and this could be improved using for example the PageRank algorithm.

All this comes together to form a search engine, which can be seen in Figure 1.5

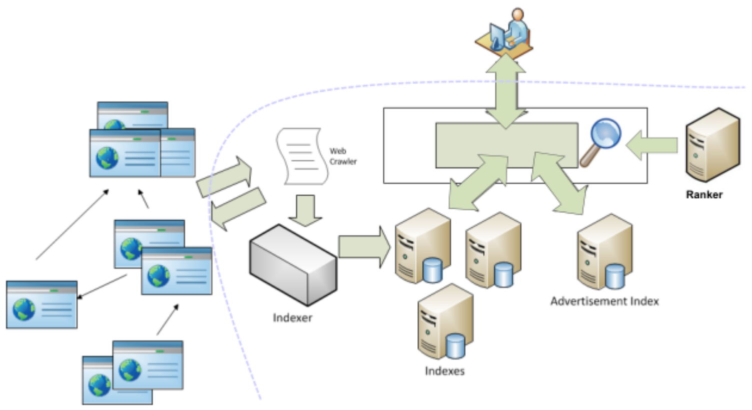


Figure 1.5: Search Engine Architecture. Source: From the Lecture 1 slides page 27.

Web Intelligence - Social Media Miniproject

Erik Sidelmann Jensen
`ejens11@student.aau.dk`

Lasse Vang Gravesen
`lgrave11@student.aau.dk`

Dennis Jakobsen
`djakob11@student.aau.dk`

1 Social Media Miniproject

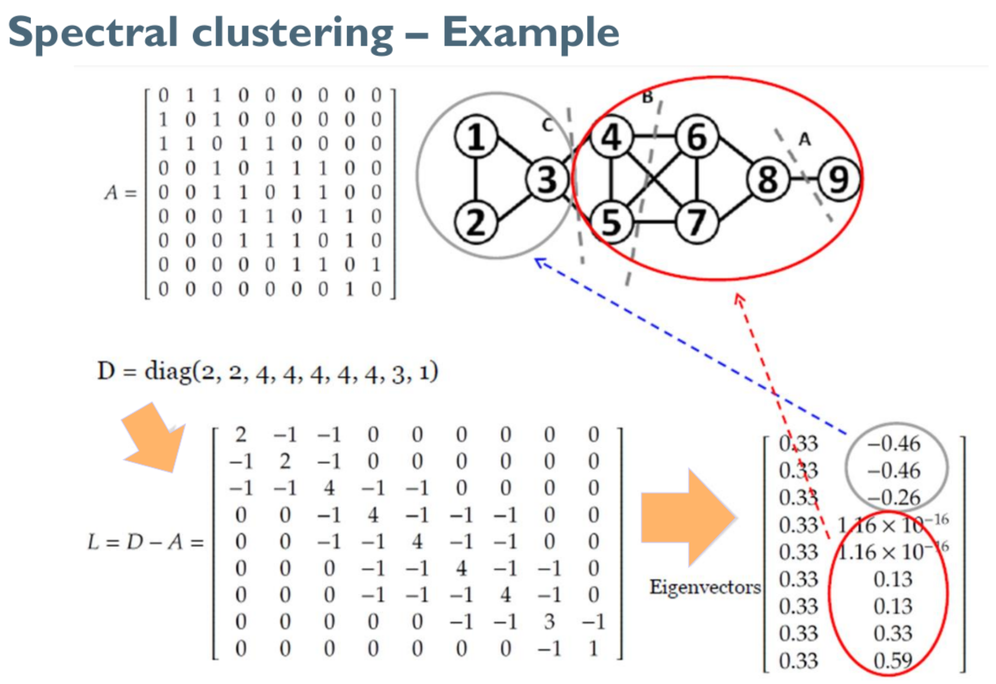
1.1 Network Communities Partitioning

There are several approaches to discover communities. For example Simple local(member) based clustering, Global(group) based clustering, and Hierarchical clustering. We did community partitioning using spectral clustering, which is community detection by graph cutting. The advantages of using spectral clustering is that it is easy to implement, gives good results, and is reasonably fast for sparse data sets of 1000+ elements. The disadvantages of using this approach is that it is computationally expensive for large data sets.

The general algorithm behind spectral clustering is this:

1. Given an adjacency matrix, A, representing a graph and an diagonal matrix D where the diagonal entry i is the sum of i'th row in A. Compute the unnormalized Laplacian L using $L = D - A$.
2. Compute the second eigenvector v_2 of L, the one with the second smallest eigenvalue - not 0.
3. The j'th value in the eigenvector corresponds to the j'th node.
4. Order the nodes according to their eigenvector values.
5. Cut at the largest gap.

An example of this can be seen in the figure below.



Spectral clustering – Example

$$A = \begin{bmatrix} 0&1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0&0 \\ 1&1&0&1&1&0&0&0&0 \\ 0&0&1&0&1&1&1&0&0 \\ 0&0&1&1&0&1&1&0&0 \\ 0&0&0&1&1&0&1&1&0 \\ 0&0&0&1&1&1&0&1&0 \\ 0&0&0&0&0&1&1&0&1 \\ 0&0&0&0&0&0&0&1&0 \end{bmatrix}$$

$$D = \mathrm{diag}(2, 2, 4, 4, 4, 4, 4, 3, 1)$$

$$L = D - A = \begin{bmatrix} 2&-1&-1&0&0&0&0&0&0 \\ -1&2&-1&0&0&0&0&0&0 \\ -1&-1&4&-1&-1&0&0&0&0 \\ 0&0&-1&4&-1&-1&-1&0&0 \\ 0&0&-1&-1&4&-1&-1&0&0 \\ 0&0&0&-1&-1&4&-1&-1&0 \\ 0&0&0&-1&-1&-1&4&-1&0 \\ 0&0&0&0&0&-1&-1&3&-1 \\ 0&0&0&0&0&0&0&-1&1 \end{bmatrix}$$

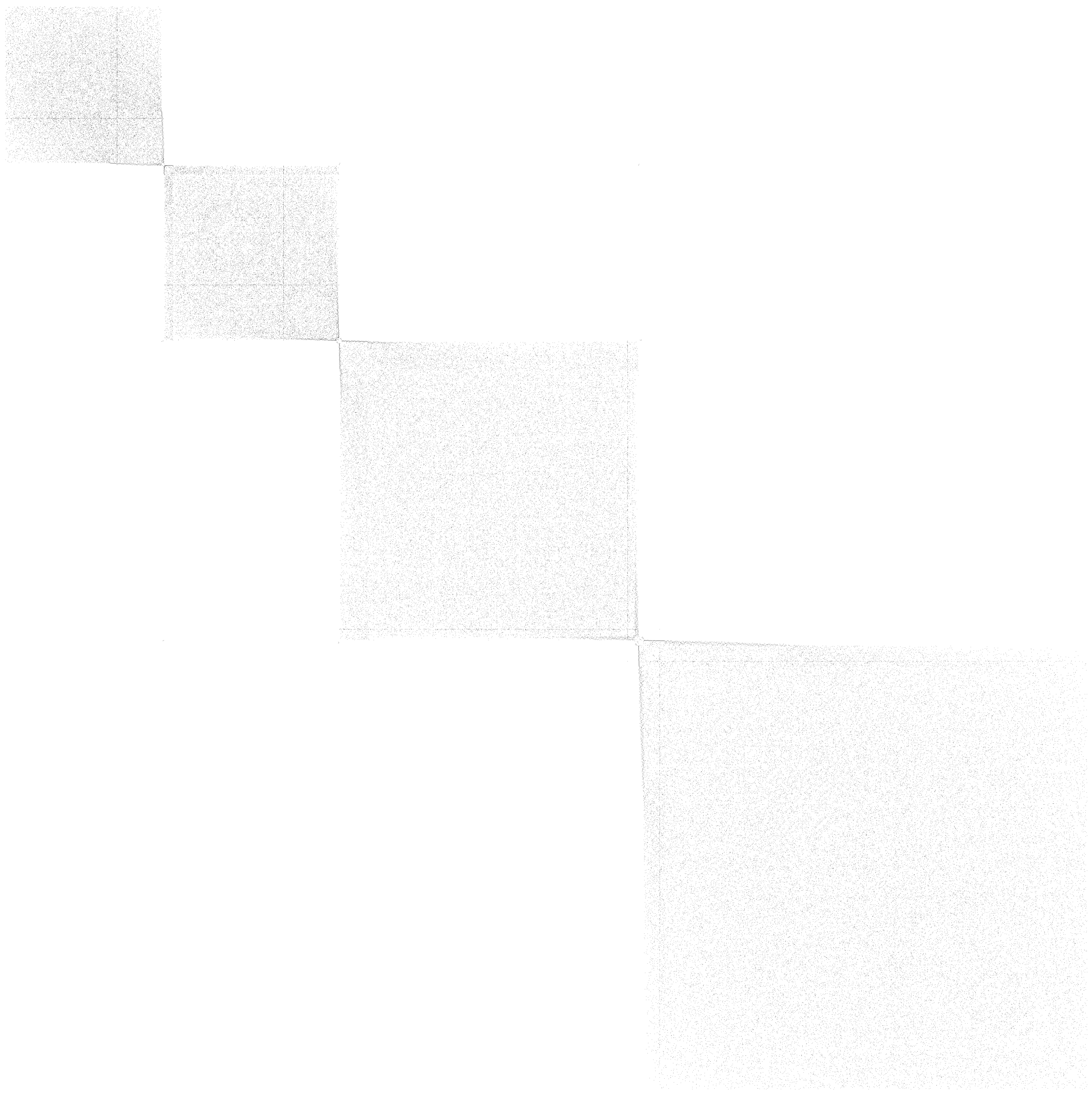
$$\begin{bmatrix} 0.33 & -0.46 \\ 0.33 & -0.46 \\ 0.33 & -0.26 \\ 0.33 & 1.16\times10^{-16} \\ 0.33 & 1.16\times10^{-16} \\ 0.33 & 0.13 \\ 0.33 & 0.13 \\ 0.33 & 0.33 \\ 0.33 & 0.59 \end{bmatrix}$$

To do this in practice the following was done:

A User class was created to hold the basic information, the Username, the list of friends, the review, the summary, and a sortable eigen value. Using this as a model the file containing the friendships and reviews could be loaded into a list of User objects. From this list of Users, the adjacency matrix A is created based on who they are friends with. From A, D is created The laplacian is calculated using L = D - A. The eigen decomposition is then found on L, and the second smallest eigenvector is found. This is the computation of the second vector. The corresponding eigenvector-value is then assigned to a User object in the user list, after which the user list can be sorted. Then the largest gap can be cut.

To find more communities, this is then run recursively until a threshold gap of 0.7 is reached after which it stops and returns the list of communities. Alternatively to running it recursively, multiple eigenvectors could have been used to perform the cut. Also alternatively to the threshold gap, modularity could have been used.

The results of this is 4 very segregated communities, on the first run-through the communities are obvious as can be seen in the figure below. We do not know if there are any flaws in our implementation but it seems to work.



The spectral clustering algorithm was used because it is simple to implement, and works well.

1.2 Sentiment Analysis

Sentiment analysis of text was done using a Naive Bayes Classifier. We could also have used a classifier called 'Conditional Maximum Entropy', but for our purposes Naive Bayes was sufficient.

This was the procedure: First step is tokenising the test data and constructing features, and for sentiment analysis it is important to remember things like emoticons and negative words because those have an impact on the rest of the content. As such we took a sentiment sensitive tokeniser made in Python and converted it to C#, and added sensitivity to negative words, in that every token followed by a negative word would get "_NEG" appended until the end of the sentence(punctuation or an emoticon).

The second step was parsing the learning data, which was done line by line, skipping the identifying text and taking the rest after having trimmed the result. Tokenising is also done in the parser, the tokeniser is called and these tokens are saved in a properties. Finally the known score of the review block is determined by the score. If the score is 3, the review is discarded. These reviews are then split into N partitions for easy learning and testing.

The third step is then constructing the Naive Bayes Classifier, which is essentially just counting. Specifically these properties of the learning data is determined through counting:

Estimation of the probability for all classes: $p(c)$. This is done through counting the amount of reviews available: N. The number of reviews with sentiment c: $N(c)$. We also have $|C|$, the amount of classes. Finally $p(c)$ can be estimated through

$$p(c) = \frac{N(c) + 1}{N + |C|}$$

We also estimate probability for all words in corpus X and all possible sentiment classes in C: $p(x_i|c)$. To do this we need to count the number of times x_i appears across all reviews with sentiment c: $N(x_i, c)$. We also need $|X|$, the size of the vocabulary. Using this we can then calculate $p(x_i|c)$:

$$(x_i|c) = \frac{N(x_i, c) + 1}{N(c) + |X|}$$

We also use Laplace smoothing, which ensures that no probability is exactly zero.

Then the score for a review can be determined through two tricks: First calculate score for the "empty" review because most words from the vocabulary are not present in a given review(Trick 1). This is done using summation instead of product to avoid numerical instability(Trick 2).

$$s^*(empty, c) = (\sum_{i=1}^{n} p(\text{not } x_i|c)) + p(c)$$

The score can then be calculated using:

$$s(x, c) = s^*(empty, c) + (\sum_{j=1}^{k} \frac{p(x_j|c)}{p(\text{not } x_j|c)})$$

And that is how the Naive Bayes Classifier works, you first use the learning data to construct the probabilities and then you use those probabilities to determine the score for individual reviews.

To test the classifier we used all possible combinations of test and learning data, where test data was 1 partition and learning data was the rest. So over N partitions, that means we iteratively ran over the partitions, selected the first for testdata, the rest for learning data and then in the second iteration we select the second for testdata and so on. This is also called cross validation.

Using this to improve the feature construction, we ended up with an accuracy rate of roughly 91% and an error rate of roughly 9% for each iteration. Though the classifier ended up being biased towards positive reviews in that it correctly classified them roughly 94% of the time, and for negative reviews it classified them correctly roughly 77% of the time.

Alternatively to the Naive Bayes Classifier, MaxEnt could have been used and would probably have been a good choice since it apparently gives better results than Naive Bayes. It is more complicated however, and Naive Bayes works fine for our purposes.

1.3 Evaluation of Reviews

We then used the community partitions and the sentiment classifier to evaluate the reviews.

We did this by using the entirety of the learning data to teach the Naive Bayes Classifier. We then ran through the list of users and classified the reviews of users that had reviews/summaries using the Naive Bayes Classifier. Then we determine the communities using the spectral clustering. Then we ran through the list of users again, this time only focusing on the users that did not have any reviews or summaries, and based on their friends average review scores we determine if they are likely to purchase. A user named 'kyle' and friends in other communities count 10 times more than a regular friend, finally we determine if they are likely to purchase by checking if the average is more than or equal to 3, and if it is the user is likely to purchase otherwise not.

Our result in summary is then as follows: We classified 1347 user reviews out of 1644 as 'Positive', and 297 user reviews as 'Negative'. Of the users that did not have a review or a summary, we classified 2451 out of 2575 as likely to purchase and 124 as unlikely to purchase.

Web Intelligence - Recommender Miniproject

Erik Sidelmann Jensen
`ejens11@student.aau.dk`

Lasse Vang Gravesen
`lgrave11@student.aau.dk`

Dennis Jakobsen
`djakob11@student.aau.dk`

1 Recommender Miniproject

1.1 Data Loading & Manipulation

The structure used to contain the data being loaded is based on Dictionaries, specifically because we are provided with a movie and a user id and it makes sense to structure it like that. Loading is done by first loading the probe data and relevant information is put into a UserRating class that contains everything that is known about that rating(which movie and user and the actual rating). Then the training data is loaded, which is done the same way as the probe data. We only load movies that are represented in the probe dataset, and also restrict it to very few files. Then we manipulate the data such that we can run test the accuracy later, specifically we take the ratings from the training data that are represented in the probe data and add it to the probe representation and then we remove it from the training data. This is done because we want to see how accurately the rating can be estimated, and if the rating is already in the training data when the learning takes place we will not know how accurately it is actually guessing unknown ratings.

The data being manipulated is not resaved because loading data is not a bottleneck.

1.2 Learning

There are two different recommender systems to use, collaborative filtering and content-based filtering. For content-based filtering the content of an item is what you recommend based on. This information is however not present in the Netflix competition data. Only a movie-id, user-id and a rating is present in this data. Collaborative filtering on the other hand is where you predict ratings for users based on other users' ratings, where users that are similar get similar ratings. Because the data contains a large amount of user movie rating pairs collaborative filtering makes a lot more sense than content-based filtering Therefore a collaborative filtering recommender system is built.

The approach we take to CF filtering is called 'matrix factorization'. We do this because it trades more complex offline model building for faster online prediction generation and we are interested in large scale, with missing ratings. Traditional matrix factorization through single value decomposition (SVD), which for large scale is very slow. So we need a variant that approximates SVD, called Funk-SVD, created by Simon Funk for a Netflix recommender competition.

Through Funk-SVD we want to create two matrixes, called A and B that contain only the most valuable information for predicting the ratings. See Figure 1.1 and Figure 1.2.

In an equation what we want from this is:

$$\hat{R}_{mu} \approx \Sigma_{k=1}^{K} A_{mk} B_{ku}]$$

We want to find A and B such that we can reproduce the observed ratings as best as we can, given K. We do this by minimizing the squared error through stochastic gradient descent.

The magic is then that from this process, unknown ratings have been filled in aswell.

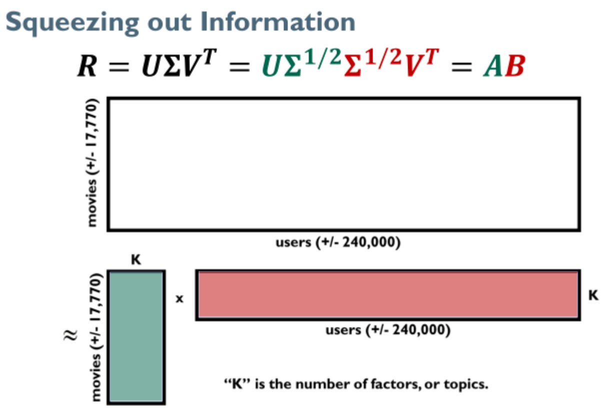
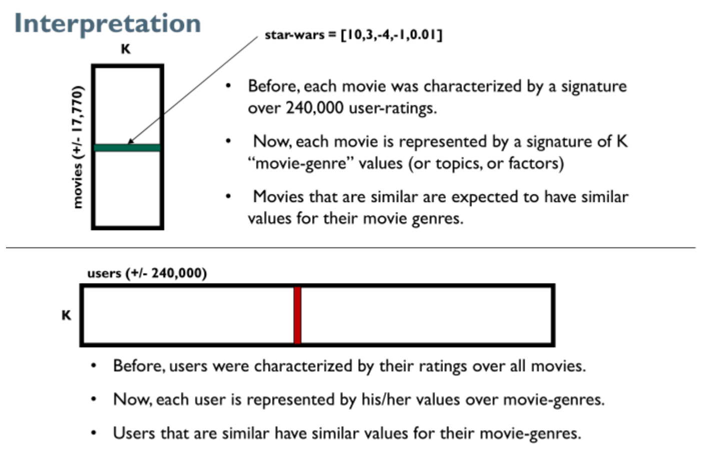

Squeezing out Information
$R = U\Sigma V^T = U\Sigma^{1/2}\Sigma^{1/2}V^T = AB$
movies (+/- 17,770)
users (+/- 240,000)
K
$\approx$
movies (+/- 17,770)
x
K
users (+/- 240,000)
"K" is the number of factors, or topics.

Figure 1.1: Squeezing out information to get A and B



- Before, each movie was characterized by a signature over 240,000 user-ratings.
- Now, each movie is represented by a signature of K "movie-genre" values (or topics, or factors)
- Movies that are similar are expected to have similar values for their movie genres.

- Before, users were characterized by their ratings over all movies.
- Now, each user is represented by his/her values over movie-genres.
- Users that are similar have similar values for their movie-genres.

Figure 1.2: Interpretation of A and B

To start, we loaded the data and the next thing we want to do is to remove the movie and user means from the training data, in order to do some pre-processing for the algorithm, see Figure 1.3. We remove these means because we want to normalise the ratings to take into account the differences between peoples enthusiasm when rating. For example, one person could rate a movie with two star, where another person could rate the same movie four stars, but the meaning of the stars would be the same, because the first user has a tendency to rate low whereas the other user has a tendency to rate high.

$$R_{mu} \leftarrow R_{mu} - \frac{1}{U_m}\sum_s R_{ms} - \frac{1}{M_u}\sum_r R_{ru} + \frac{1}{N}\sum_s\sum_r R_{sr}$$

Figure 1.3: Removing the obvious structure from the ratings.

U_m: total number of of observed ratings for movie m.

M_u: total number of observed ratings for user u.

N: total number of observed movie-user pairs.

We construct the movie feature A and the user feature B by first filling it with random values between -0.5 and 0.5. Then we use the stochastic gradient descent algorithm to calculate A and B, see Figure 1.4. This algorithm iterates until the error reaches a local minimum. We do not check the error for every iteration, but instead check it with some interval. It is not necessary to monitor this error for a minimum, since a predetermined amount of iterations can be used, for example 1 000 000, and just adjust this value until a minimum error has been reached.

$$A_{mk} \leftarrow A_{mk} + \eta \cancel{\sum_{u}} \left(R_{mu} - \sum_{i} A_{mi} B_{iu} \right) B_{ku} \qquad \forall k$$

“S. Funk”: $\eta = 0.001$
good value for
Netflix data

$$B_{ku} \leftarrow B_{ku} + \eta \cancel{\sum_{m}} A_{mk} \left(R_{mu} - \sum_{i} A_{mi} B_{iu} \right) \qquad \forall k$$

Figure 1.4: Stochastic gradient descent.

k: Latent factors (between 10 and 50).

R_{mu}: Rating for movie m and user u.

η: Learning rate.

At last, we add in the obvious structure from the pre-processing step again, and truncate the calculated rating to fit the Netflix rating scale. Meaning that every rating above five results in five, and all ratings below one results in one.

Alternative algorithms could have been used such as user-based nearest neighbors CF (kNN) and item-based nearest neighbors CF (kNN). Although for these techniques there are issues with scalability when having millions of users and thousands of movies. Furthermore there are a possible problem of coverage where the kNN algorithm does not find enough neighbors and for example with user with preferences for niche products.

Using this model-based algorithm matrix factorisation scales to larger data set. When recommending a movie to a user, only the learned model is used, which makes it much faster, although the model is re-trained or updated periodically.

The model-based approach seems as the best choice for this kind of recommender system, although we did not manage to get useful results out of it.

1.3 Scoring

We evaluated the RMSE(Root-Mean-Square Error) for the probe data and the result ranged between 1.15 and 1.50, where it should roughly have been 0.90.

$$RMSE = \sqrt{\frac{\Sigma_{i=1}^{n}(\hat{r}_i - r_i)^2}{n}}$$

For one subset of the overall dataset we did manage to get good results, specifically using midsized movie files and 25 of them we could get an RMSE score of 0.90, though changing the latent factors seemed not to make much of a difference in the final result.

In all, the results seemed to vary wildly based on what dataset was picked out, and very few of them gave good results. To perfect this, more time would have been required to do different things in the code, but there was none left.