Webintelligence - Mini Project 1

Lars Andersen, Mathias Winde Pedersen & Søren Skibsted Als

4. december 2014

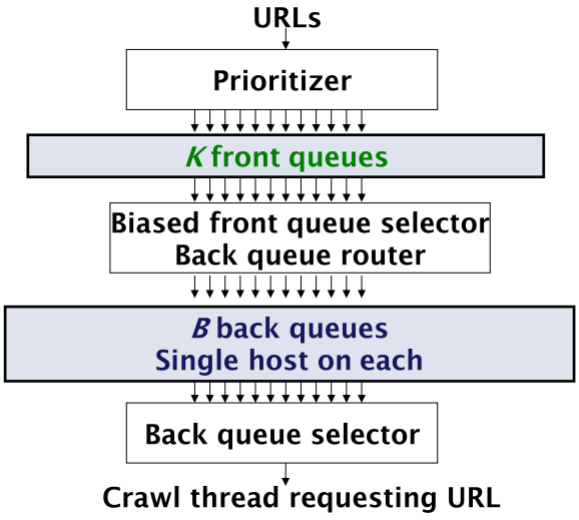
Crawler

The crawler serves as the obtainer of website links. When doing this, it has to make sure that it follows the robots.txt specification. The robots.txt specification file, is a file that is placed at the root of websites, who wants to indicate what pages they want to have crawled by different search engines. This ensures explicit politeness. In order to not consume too much space, near duplicate detection is also performed here, this is called Jaccard Similarity. The reason near duplicate detection is performed is that a large percentage of the internet content is duplicate information.

We decided to implement the crawler with Mercator scheme. Mercator works by having several front and backqueues. The front queues serves as ranking of what websites gets crawled first. As an example, news sites can be put into a frontqueue with high priority, as such sites changes regularly, whereas a static site can have a lower priority for crawling. However, for the front queues we use a single queue, resulting in a uniform ranking, as we have no detection of what is static and what is dynamic websites. However, we only visit each website once. What might be a vulnerability is hidden changes made to similar websites with different url, e.g. text that is transparent that the Jaccard Similarity will not exclude. The backqueues makes you able to crawl different domains while still ensuring implicit politeness for the crawler. Implicit politeness being not hitting the same domain more than once every two seconds.

Furthermore, the crawling is single threaded, as a proof of concept, but could be expanded to a multi threaded one. This is also related to DNS lookup times, as it was found that some DNS lookups delayed the crawler severely.

An illustration of the Mercator scheme can be seen in Figure 1



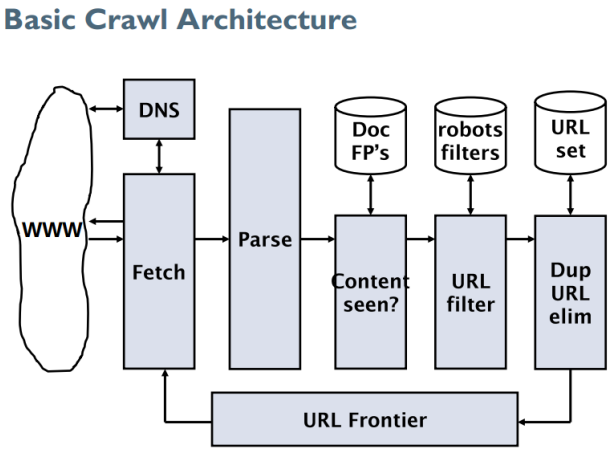
Figur 1: Basic structure of Mercator Scheme, picture taken from slides.

To check near duplicates we use shingles with sketches. Shingles ia a continuous set of words, which in our case is a set of 8 words per shingle, thus a shingle size of 8, which is the size recommended by Google. Sketches is then

to use hashing on these shingles, to reduce the storage necessary, as well as computation time, since the smallest of all the hashing values is chosen for each hash functions, and compares on this. However, Google recommends 84 hash functions for sketching, but we only use 19 hashes, as we did it as a proof of concept.

Furthermore, tackling documents on the website which is not html or txt encoded is not handled, which gives crawled text encoded in a strange manner, e.g. pdf files. Image files are excluded, as some of those formats made the crawler otherwise crash, when trying to convert it to a string. A library could be used to translate such text into a readable format, to be used for the indexing, and also to make more sensible near duplicate detection. A more conservative approach would be to only crawl files that are HTML, however, this would lead to a more narrow crawling as parts of the files on the web are not HTML files.

For the seed we used `http://aau.dk/` and `http://stackoverflow.com/`, which is the starting frontier of the crawler. A figure of the basic structure of a crawler can be seen in Figure 2.



Figur 2: Basic structure of crawler, picture taken from slides.

Indexer

The indexer has the purpose of constructing the inverted index, in that regard it is important to keep statistics of term and doc frequency, such that this information can be used by the ranker to rank pages. An inverted index is just similar to the back of a book, when you lookup a term, the pages containing the term are referenced. This provides the advantage that only documents containing the term are listed, and thus greatly reducing the memory required for storage. An example is a term only occuring on 100 out of 1 million sites crawled, then you only have to list 100 documents in the index, instead of the whole million, containing a lot of zeroes.

For the indexer we cut corners as follows. We only stem on the English language. The reason we only stemmed on the english language was as a proof of concept, and since the library found was developed for english. Stemming

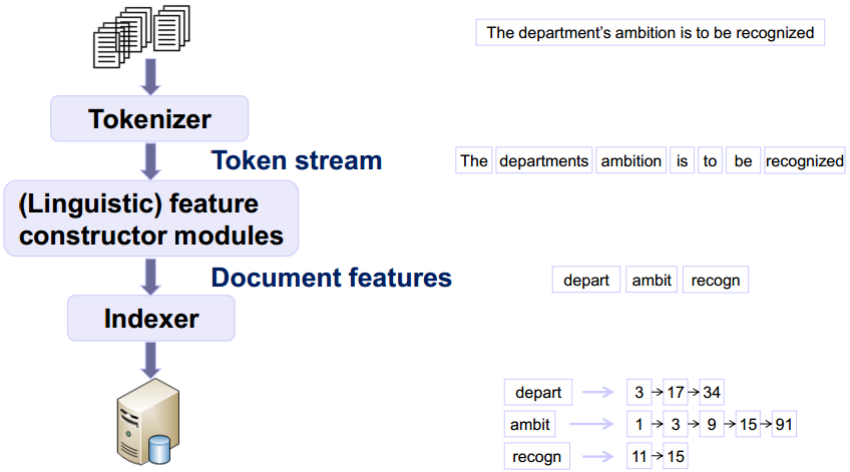
means cutting off unnecessary parts of the words to greatly reduce the index size. We do not develop our own stem method, but instead relies on an already developed one found on the web.

For stopwords it is used for English and Danish, Which are common words without much meaning, such as "the", "about", "then", "by"etc.

As we encode weird strings due to extracting pdf files without a library, we get some strange terms that fill the index unnecessarily. Services could be used to tackle such file formats, in order to extract the text from the pdf, word, and other such files.

Furthermore, the postingslist are all kept in main memory. This works fine for a small size of websites crawler, however, if you were to crawl the whole web, you would have to have servers farms or write to disk(making it much slower).

A figure showing the basic steps of the indexer can be seen in Figure 3. Where for the inverted index shown in the lower left, we also keep track of the amount of times the term occurs in the document, as that is used for the ranker. Our implementation also differs in that we do not store the results in a database/saved on disk, but instead stored in main memory, and the reason for this is that the crawler is developed as a proof of concept.



Figur 3: Basic structure of Indexer, picture taken from slides.

Ranker

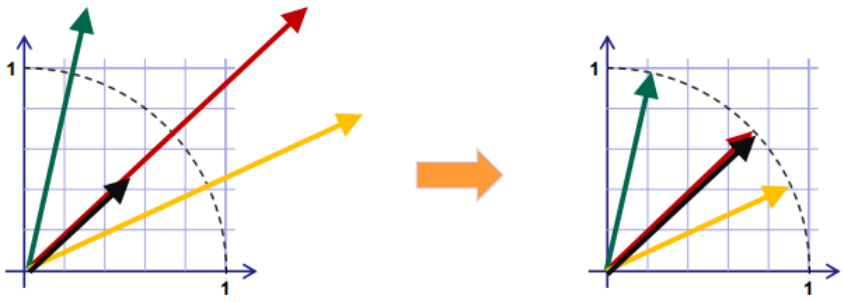
The ranker serves the purpose of ranking the pages, given a search query. The search query is tokenized, stemmed and filtered for stop words in the same way as the documents, such that the words can be matched.

This is meaningful, as it contrary to boolean matching can weight different documents, and as of such can rank the results on what matches the query best, and not just if the terms occur in the document or not.

For ranking we used the algorithm provided from the slideshow, which means we used the CosineScore, based on the tf-idf weighting. For contender pruning

we used a precomputed champion list, which is a list of the R documents for each term which has the highest weight.

A picture of CosineScores can be seen in Figure 4.



Figur 4: Example of CosineScore, picture taken from slides.

This made it possible to pre-compute this, and significantly reduce the computations needed when ranking, as the champion list limits each posting to a list of r docs, the docs with the most occurrences of the given term the posting is associated with.

A more specific description of the ranker comes here: The ranker takes a query as input and stems, uses stop-words and tokenizes, in the same manner as the indexer did for the documents, as the terms have to be treated in the same way, to locate them in the inverted index.

The term frequency weight (tf) is the amount a given document contains the given term, where a logarithm is used as relevance is not proportional to frequency.

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

The document frequency is (df) is the total sum of occurrences of a term across all documents. the inverse document frequency (idf) can then be calculated as follows

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

Where N is the total amount of documents, which is 1000 in our case.

The tf-idf weight, which is used for calculation of the score, can then be calculated as follows:

$$tf - idf = tf * idf$$

A small example, for calculating the score for a single document, given a query, can then be seen in Figure 5

Finally, to see how the crawler, indexer and ranker is structured in a broad sense, see Figure 6

tf-idf example: lnc.ltc

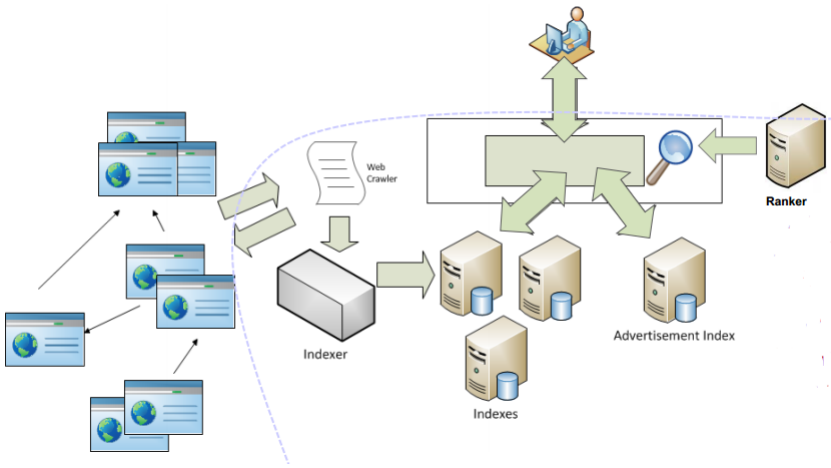
Document: *car insurance auto insurance*
Query: *best car insurance*

Term	Query						Document				Prod
	tf-raw	tf*-wt	df	idf	wt	norm wt	tf-raw	tf*-wt	wt	norm wt	
auto	0	0	5000	2.3	0	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0.34	0	0	0	0	0
car	1	1	10000	2.0	2.0	0.52	1	1	1	0.52	0.27
insurance	1	1	1000	3.0	3.0	0.78	2	1.3	1.3	0.68	0.53

Doc length = $\sqrt{1^2+0^2+1^2+1.3^2} \approx 1.92$

Score = 0+0+0.27+0.53 = 0.8

Figur 5: Example of ranking score calculation, picture taken from slides.



Figur 6: Basic structure of a search engine, picture taken from slides.

Webintelligence - Mini Project 2

Lars Andersen, Mathias Winde Pedersen & Søren Skibsted Als

4. december 2014

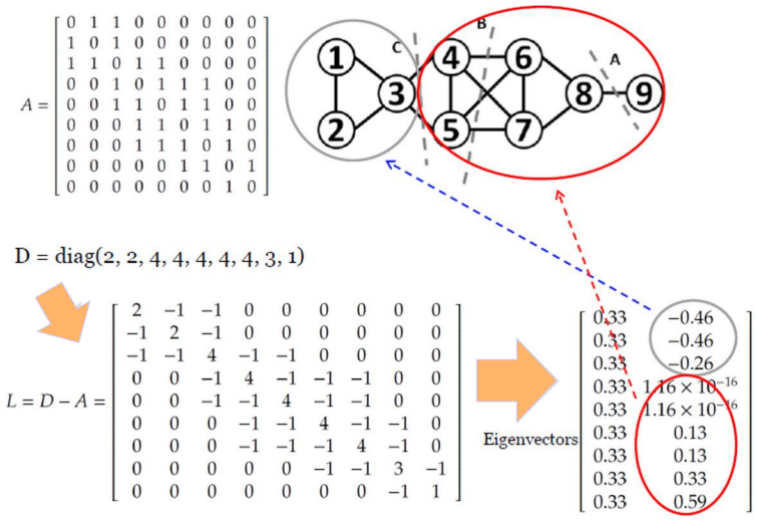
Part 1 - Community Detection

The task of detection communities is the task of finding clusters in the friendships network. In order to do this, we draw upun the spectral clustering power, and we use that for finding clusters.

But, before we discuss of the spectral clustering works, some theory needs to be explained.

Unnormalized Laplacian

We need to be able to compute the unnormalized Laplacian matrix, which is a matrix consisting of the diagonal matrix (D) and the relationship matrix (A), constructed as $L = D - A$. Where the diagonal matrix D has numbers on the diagonal equal to the sum of numbers for that given row in the A matrix. An example of this can be seen in Figure 1, where you for now can ignore the eigenvectors part.



Figur 1: Laplacian example

Eigen decomposition

Some other computations needed for the spectral clustering is the eigen vector decomposition (EVD). To calculate this, we used the mathnet.numerics library, as it was able to calculate the EVD multithreaded. What you calculate the EVD of is the unnormalized laplacian matrix.

The library used was also good in the sense that the vectors were sorted according to their respective eigenvalues.

Spectral Clustering

As mentioned, we used the unnormalized laplacian for our spectral clustering algorithm, resulting in a RatioCut. An alternative is to use the normalized

laplacian, resulting in a NormalizedCut. However, we had no incentive to use the NormalizedCut, as that cut considers the overall structure, but this is irrelevant for us.

As taken from the slides, the spectral clustering then works as follows:

1. Compute the unnormalized Laplacian L.
2. Compute the second eigenvector $v : 2$ of L (the one with the second smallest eigenvalue - not 0).
3. The j'th value in eigenvector corresponds to the j'th node.
4. Order the nodes according to their eigenvector-values
5. Cut at the largest gap

For the algorithm above, the largest gap can be now be seen in Figure 1, in the lower right corner.

The results of running this community detection is that we found 4 clusters, resulting in 4 communities. We also printed the relations to a png file and it also indicated 4 communities.

At this point in time, when we decide enough clustering have been performed, we use a threshold for the largest gap, and just decide on this. In the future, with more time, we would used the Modularity of the clusters for this. The idea of modularity is that in a real.world scenario, modularity is a measure of how much the edges in the graph falls withing the same cluster, opposed to a randomly across clusters, see the slides.

Part 2 - Sentiment Classifier

The idea of having a sentiment classifier for this miniproject, is that based on a review, we are able to tell whether the review is positive or negative. The baseline of the algorithm is the following: (taken from slides)

- Tokenization
- (Stemming usually not good for sentiment analysis)
  - Sometimes destroys the Positive/Negative distinction
- Feature Extraction
- Classification using different classifiers
  - Naive Bayes
  - MaxEnt

Each of these relevant parts will be described in turn.

Tokenization and feature extraction

For the tokenization, we used the proposed python library (Potts) and translated it into c#, such that it was easily available for use in our visual studio project. The library takes care of splitting the string into tokens, and in that regard uses several useful regex's, and also takes care of handling negation by adding a "not"suffix to each token. Another powerful token to recognize with this is capturing emoticons, which the library also takes care of.

The tokenization and feature extraction is somewhat merged into one here, as the handling of negation and such is part of feature extraction, but is handled by the implemented tokenizer.

Classification

For the classification, which for us consist of prediction whether a review is positive or negative, can use several algorithms. We decided to use the Naive Bayes algorithm, as it is known to do decently well, although simple.

In order to predict the classification of reviews, training data is used. For the training data, you have a review and its associated score, which is used to calculate the probabilities for being positive and negative for each of the tokens.

The Naive Bayes classifier, builds upon bayes theorem that says for two stochastic variables C and X, bayes theorem says: $p(C|X) = \frac{p(X|C)*p(C)}{p(X)}$.

The score for class $c \in C$ given review x is:

$$score(x, c) = p(c|x) = \frac{p(x|c) * p(c)}{p(x)}$$

However, the division with $p(x)$ can be excluded, as that is the same for all c.

With the assumption that the features are independent given the class attribute. The score can then be calculated as follows:

$$score(x, c) = (\prod_{i=1}^{n} p(x_i|c))p(c)$$

And the way you then would decide on which class to pick, it is simply choosing the class with the highest score.

In order to reduce the amount of computations needed we use the following trick: As the reviews usually contain only a small subset of the vocabulary, the following computation is much more efficient.

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

if you have a review x consisting of the words $x_1, ..., x_k$, the score can be calculated as follows:

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

Additionally, we use another trick to ensure numerical stability. The idea is that instead of having multiplications, which can become very large, you switch

to log-space such. So this just uses the logarithmic rules to give you.

$$\log score(x, c) = \log p(c) + \sum_{i=1}^{n} \log p(x_i|c)$$

Learning the model

To learn the model, it is a matter of simple counting, with a modified Laplace smoothing. What you count is used to calculate the following, where $|C| = 2$ for our case, as we have two classes (positive and negative):

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

The dependent probability distributions is calculated as follows:

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

Where,

$N(x_i, c)$ is the number of times the word x_i appears across all reviews with sentiment c.

$|X|$ is the size of the vocabulary.

Corners Cut

We did not try and use the MaxEntropy algorithm, and could be interesting to use in the future, as the reading material states that its performance is usually better than the Naive Bayes algorithm.

Additionally, we did not use the reviews that had a score of 3.0 to learn our classifier. Those reviews may have been of use, to have a neutral category or something of the like, but for this task with only two classes, we did not see how we could use it, and as of such excluded it.

How good is the classifier

To determine the quality of the classifier, we used cross-validation, by splitting the data 10-fold. For each fold, we use that as a temporary test set, and the other 9 folds as training data, and then calculate the performance, after which we calculate the average performance of the 10 runs. This gave us an precision of about 80% for positives, but we had a precision of about 60% for negatives. A reason for this could be that the training set mainly consists of positive reviews, and as of such there is not as much training data for the negative reviews, making this classifier less accurate. We find this a good performance, considering the low complexity of the classifier.

Webintelligence - Mini Project 3

Lars Andersen, Mathias Winde Pedersen & Søren Skibsted Als

4. december 2014

Part 1 - Data Manipulation

The first part of the project is to load data. The data consists of two parts, training data and probe data. A subset of this is loaded, as if you were to load all of it, the loading and later training would take too much time.

One thing you have to be careful about, though, is that the probe data is part of the training data. This is bad as you would then get to train on the test set. In order to prevent this, we remove the probe part from the training set, and instead store its actual ratings in the probe set, to be used for later testing of the algorithm.

This transformation of the data is performed once, such that you do not need to perform this conversion each time you test your program.

Part 2 - Learning

Recommender Systems

Two sorts of recommender systems have been talked about during the lectures:

- Content-based filtering
- Collaborative filtering

The content-based filtering has to do with predictions based on the content, that is if you have reviews, geographics, gender, and other kinds of such information. Collaborative filtering is to predict based on other peoples ratings, where people more similar to you has a greater influence. For the netflix dataset, no real content is provided other than the title, however, a vast amount of user movie pair ratings are provided. This leads to the obvious choice of using collaborative filtering.

In a broader setting, where you have access to the content and other users ratings, a mix of the two is beneficial.

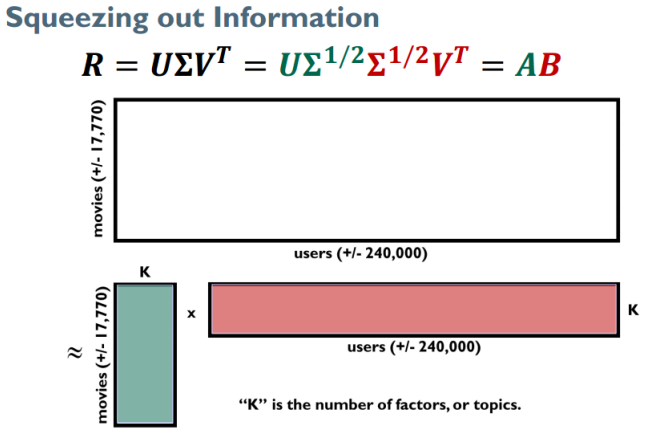
Matrix Factorization

In the area of collaborative filtering, several approaches can be taken. These include item-based, user-based and latent factor models(Matrix Factorization). The matrix factorization model is for this dataset preferred. This is due to the dataset being very large. As the dataset is very large, we trade a complex offline model for a fast online prediction. Additionally, if you were to use some of the other approaches, as the dataset is so large, it cannot fit in main memory, whereas the matrix factorization approach is more scalable, such as the netflix dataset with about 17k movies and 240k users. How the approach then works is to approachimate the single value decomposition matrix (SVD). Libraries exist to calculate this, however, as such a computation can be quite heavy, another approach is used called Funk-SVD.

Funk-SVD

The idea behind the algorithm is to tackle the problem that the user-movie matrix is very large but very sparse. That is mainly the result of each user

only watching a small fraction of the entire movie set. In order to utilise this to instead work on smaller matrices, a "squeezing"of the matrix is performed, such that we instead work on K factors/topics.



Figur 1: Matrix squeezing, taken from the slides.

An illustration of the theory for this can be seen in Figure 1. These two smaller matrices can then be interpreted in different ways. Each row in the movie-matrix $U * \Sigma^{1/2}$ then represents k genres, and the given movie's values for each genre, such that movies similar have similar values for the different genres. If you look at the user-matrix $\Sigma^{1/2} * V$ each column corresponds to the values for each movie genre of a user, such that users with similar tastes in movies will have similar values for their columns. These rows/columns could also be interpreted as movie/user communities, but we take the first interpretation approach.

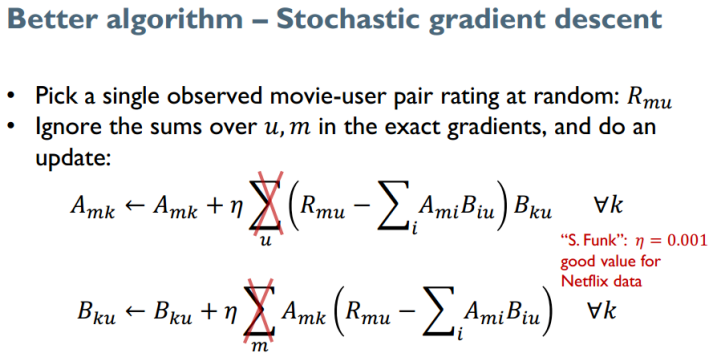
The matrix we then want to calculate is as follows:

$$\hat{R}_{mu} = \sum_{k=1}^{K} A_{mk} * B_{ku}$$

We then want to produce A and B to be able to reproduce the observed ratings as best as possible. To do this we want to minimize the squared error, this is involved with what K we choose, and is touched upon in the scoring section. The trick then comes into play. We train and minimize the error for the observed ratings only, and as of such ignores the non rated movie-user pairs. Then when you multiply A and B then non non-rated movie user pairs will have ratings calculated, these ratings then serves as our prediction! In other words, the non-rated movie-user pairs is then learnt from the observed ratings of similar users/movies.

In order to minimize the error, we use stochastic gradient descent to approximate A and B. The reason a stochastic gradient descent and not a usual gradient descent is used, is due to the fact that the regular gradient descent would not scale well with the large netflix dataset.

How this stochastic gradient descent is used can be seen in Figure 2. One problem with this is that the stochastic gradient descent will not converge to



Figur 2: Stochastic Gradient Descent, taken from the slides.

the minimum, but instead "dingle"around it. This can then be resolved by using a weighted decay.

Apart from this some preprocessing steps of the ratings needs to be performed and is discussed hereafter.

Preprocessing Step

Before we use the stochastic gradient descent to approximate A and B we need to perform some preprocessing on R_{mu}. What we want to do is subtract the user and movie means and add the overall mean. The reason we want to substract the user mean and movie mean is to adjust the ratings to be around zero. However, as you subtract two times for each entry, you have adjust with the overall mean, which is why this is added. We did this preprocessing step and made sure that the overall sum of the preprocessed entries and found it was very close to zero (10^{-8}) which was sufficient, and ensured that the preprocessing did not "skew"the data too much. The reason it was not entirely zero is blamed on floating point numbers to be inaccurate. So the algorithm for this preprocessing step then is as follows:

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

U_m total number of observed ratings for movie m

M_u total number of observed ratings for user u

N total number of observed movie-user pairs

It is then important after the stochastic gradient descent to add these values that you removed. And thus finally you do the following calculation.

$$\hat{R}_{mu} = \hat{R}_{mu} + \frac{1}{U_m} * \sum_s R_{ms} + \frac{1}{M_u} * \sum_r R_{ru} - \frac{1}{N} * \sum_s \sum_r R_{sr}$$

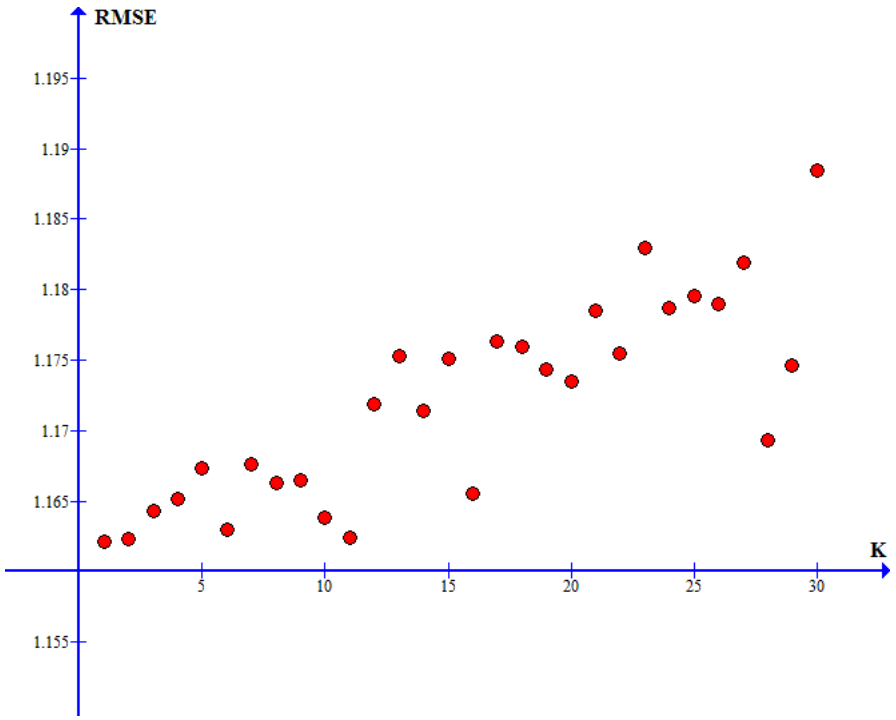
Part 3 - Scoring

For the scoring we evaluate RMSE for the probe data, which is the data we extracted from the training set, in order to avoid overfitting. We then calculate RMSE as follows:

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

Where $\hat{r}_i$ is the predicted value for a user-movie pair and r_i is the actual observed rating.

We do this evaluation for $K = 1 \ldots K = 50$ in order to determine the best value for K, which you recall is the amount of latent factors.



Figur 3: RMSE results for varying k.

As can be seen from Figure 3, we retrieved some strange results. We tried to increase the number of traversals of the stochastic gradient descent to 1 million traversals. However, the result is still strange, as you would expect that the result gets better with an increase in k. Further investigation of the code is needed, but no more time was available for this miniproject.