1 Introduction to Data Mining

If we look at the change in processing speed and the speed of data collection, we can deduce that every 3 years the processing speed increases by a factor 4 and the collected data by a factor 8.

Definition 1.1 (Moore's Law)**.** Processing speed doubles every 18 months

Definition 1.2 (Kryder's Law)**.** Hard disk capacity doubles every 12 months

Definition 1.3 (Lyman & Varian)**.** The amount of collected data doubles every year

Meaning, every 3 years, the overflow of data (or flood) is doubling. Besides the total amount of data, the shape of the data has also changed over time. This new data is also called Big Data, generally characterized by the "Three Vs of Big Data" (IBM):

Definition 1.4 (Three Vs of Big Data (IBM))**.**

- **Volume** - Large size of data (terabytes, petabytes).
- **Velocity** - Speed of added data is fast, and needs to be processed fast.
- **Variety** - Structured and unstructured data, such as text, sensor data, audio, video, click streams, log files and more.

To be able to process such big data, we need to do fuzzy queries and we want to be able to run the processes in a streaming fashion. Traditionally these are easy to execute, but complex in this context. The trade-off therein is accuracy for speed (a rough answer in real-time is sufficient).

1.1 Examples

1.1.1 Recommender Systems

Given petabytes of sales data in the form *<user_id, item_id>* we want to do recommendations to the *current_user_id* that just clicked or bought *current_item_id* and we have only milliseconds to do this.

This is a fairly simplified scenario, as you generally have a lot more information on users to do these recommendations.

1.1.2 Data Stream Mining

In some situations, the incoming stream of data is too large or too fast to be stored, and needs to be analyzed on-the-fly with limited memory.

Examples of this are to detect fraud in electronic transactions, showing the correct advertisements, prognostic health monitoring of a fighter jet, detection of DoS attacks, or returning rankings from current interests or click streams from users.

1.2 Data Mining Paradigms

We can group data mining activities into two separate paradigms: supervised and unsupervised learning.

Definition 1.5 (Supervised Learning)**.** We have a set of defined inputs and defined outputs and try to build a model that converts input into these outputs

Definition 1.6 (Unsupervised Learning)**.** The outputs are characteristics and generally tries to learn similarity from the data.

1.3 Overfitting

If we would train our model on all the data that we have, we have the risk to overfit on the data. This means that it follows the data closely, but will not work with unseen data. Thus, we separate the training data from test and evaluation data, and use approaches to avoid overfitting on the training data.

For example, we can use k-Fold Cross-Validation where we split the training data (stratified) in to k folds and use $(k-1)$ folds for training and 1 for testing. Repeat this k times and average the results. A downside of this is that it is computationally expensive (but parallelizable).

2 Recommender Systems

There are different approaches to recommender system, the idea is basically that you have a mapping of users onto items and use this mapping to learn the value of an item to a user that is not in the mapping.

Generally, we have a matrix of the size users $\times$ items and it is generally filled with ratings, interest, history or some other value that represents the relation. Such matrices are very large and usually sparse.

2.1 Naive Approach

The baseline approach is to take the mean of the entire matrix, item or user. A bit more advanced we can do a linear regression modeled as:

Definition 2.1 (Naive Linear Recommender)**.**

$$R_{\text{user-item}}(u, i) = \alpha \times R_{\text{user}}(u, i) + \beta \times R_{\text{item}}(u, i)$$

The linear regression returns good results, is easy to calculate and update, and allows for simple interpretation of the model (good movie, harsh user, crowd follower).

2.2 Content-Based Approach

The intuition for Content-based Approaches is to construct a profile for every item and a profile of every user, and see which items are closest to the user profile. The profile generally contains labels such as budget, genre, origin, cast, et cetera. The baseline would be to take the mean of each dimension of the profile for a user.

2.2.1 Model-Based Approach

The content-based approach assumes that you can fill in these parameters for users yourself (or source it from the user themselves). However, we can also train a model per user that predicts rating from the item profiles. Pitfalls here are that it is expensive to build and maintain, the accuracy is low and it doesn't work with new users.

2.3 Collaborative Filtering

An alternative to item-specific profiles is to recommend items to users based on what similar users have liked. You can either do user-to-user collaborative filtering or item-to-item collaborative filtering.

User-User Recommendations Each user has a vector with ratings for every item that they have rated. Users with similar vectors are selected as neighbours. We then take the top L neighbours and aggregate their ratings to predict the rating.

Item-Item Recommendations Each item has a vector with ratings for every user that have rated it. Items with similar users are selected as neighbours. We then take the top L neighbours and aggregate their ratings to predict the rating.

The main differences here are that for either category insertion of new items or users creates a cold start problem respective of the type of collaborative filtering. Furthermore, user-to-user filtering has a higher personalization, suffers more from sparsity, is less scalable with larger user bases and may fluctuate with user behaviour. However, as it takes the user as the comparison base, the personalization is better.

2.4 Singular Value Decomposition

A large motivator for changes in recommendation algorithms was the Netflix Challenge. Before, as discussed in the introduction we can do recommendation by matching pairs of users and ratings to estimate ratings for new users (based on other ratings).

A possible approach to recommending movies by using the user ratings is the CINEMATCH system, which was state-of-the-art in 2006. While this approach worked well, Netflix decided to set up a challenge for scientists to improve the recommendation system. For this challenge you would predict ratings based on a training set and submit them. The improvement was measured using the RMSE.

Definition 2.2 (RMSE)**.**

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\text{predicted} - \text{true})^2}$$

2.5 Matrix Factorisation - UV Decomposition

One approach for recommendation is the use of matrix factorisation, where the users and ratings are combined into a utility matrix. This matrix is then split into a user matrix (with a fixed dimension) and a vector matrix (with a fixed dimension). Ratings can then be approximated by multiplying a specific user vector with the transpose of an item vector and summing the output. The dimension lengths are fixed before doing the decomposition but their semantic "meaning" and values are learned from the data.

If we encode a loss function to be a polynomial that measures the distance between the predicted values and the true values (like the MSE) we can do learn the correct values for U and V (where the MSE is minimal).

For example, we can do a simple line search, this initializes all variables randomly, chooses a random parameter (with the rest frozen) and then finds the optimal value for that parameter. It does that by calculating the derivative of that parameter and chooses the optimum.

The gradient descent algorithm is a better alternative, which takes the loss function and chooses an arbitrary point in the multi-dimensional space. Then, it finds a direction in which the loss function is decreasing the most rapidly using partial derivatives and makes a small step in that direction. It repeats this until a local minimum is reached.

Finally, an alternative is to use alternating least squares, which reduces the distance in an alternating fashion between the user and item matrices. First, freeze the user features and treat all item features as variables, solve the resulting least squares problem. Then, freeze the item features and treat the user features as variables and solve resulting least squares.

2.6 Boosting Accuracy - Blending Models

Better results were ultimately achieved by blending different algorithms together. For example, first you can train the initial values using a regression model instead of choosing random parameters.

3 Mining Data Streams

Instead of stored data that is used for off-line analysis we might sometimes need instant action on results or signals that we receive. We can use trained models (rules, decision trees, neural networks, et cetera) in real time. Specifically, we might be interested in queries over the last N records (sliding window) or a sample of everything that we have seen so far (i.e. the last N days).

3.1 Data Stream Sampling

Generally, high velocity data streams (such as global-scale events) are too large to mine, thus we need to sample events from these data streams. Naively, we can take a modulo, for example $(n mod 100) < c$ which will take c% samples from the stream. Alternatively, we could take random selections from a probability $c/100$.

However, there are problems with this as we run the risk of representing items from some groups more than others, and it might not be representative of the entire stream. We could also create groups, or a list of clients, and take a sample from them. Whenever an event comes by, we check if it is from a client that is in our hash map. This might become problematic if we have a large number of users, as the worst case hase to represent all users in the table. Moreover, the list of clients is a snapshot and might not reflect the current active users.

Alternatively, we could alter the hash function such that we don't store the value in a hashmap but rather map the unique clients to integers between 1 to 100 and use this to sample the first n integers. However, when the amount of data grows, the memory usage also grows if we keep the same sampling ratio.

3.1.1 Reservoir Sampling

Suppose the memory can store s records, initially we store all the records in the memory and the probability of an element entering is $\frac{s}{n} = 1$. When the $(i+1)$th element arrives, decide with probability $\frac{s}{i+1}$ to keep the record in

RAM. Otherwise, ignore it. If you choose to keep it, throw one of the previously stored records out, selected with equal probability, and use the free space for the new record.

Definition 3.1 (Reservoir Sampling)**.**

- Suppose we can store s records
- Store all incoming records until i is s
- For every subsequent record, decide with probability $\frac{s}{i+1}$ to keep the record.
- If the record is to be kept, replace a random record in the store.

3.2 Bloom Filters

Bloom Filters allow to classify entries as positive or negative, where we are certain that a real positives are never classified as negative, but have no problems with a small number of false positives. Generally, the idea is that we filter out the majority cases where we are sure that an (expensive) computation will be negative or not required.

First, decide on a memory space that you have and write a hash function that encodes the search input to a one-hot encoding in this memory space. Finally, set all bits to 1 for the positive values and whenever an input points to a positive bit, consider it positive.

To improve, we can use multiple hash functions simultaneously and in the case of a positive number, set all bits to one. In that case, all k positions in the memory need to be set to one to indicate a positive input.

Definition 3.2 (Bloom Filter - False Positives)**.**

$$\left(1-\frac{1}{N}\right)^k n = \left[\left(1-\frac{1}{N}\right)\right]^{\frac{kn}{N}} \approx e^{-\frac{kn}{N}}$$

$$\lim_{x\to\infty}\left(1-\frac{1}{x}\right)^x = e^{-1}$$

$$k_{opt} = \frac{N}{n}\times \ln 2$$

3.3 Probablistic Counting

Given a data stream of a size n we want to count how many distinct items we have seen. Generally, you would count n using a hash map, however if the size of n gets larger storing the hash map might not be feasible. If we accept that we have some loss of accuracy, we can use probablistic counting approaches.

3.3.1 MinTopK Estimate

Hash incoming objects into doubles in the interval $[0, 1]$ and count them, shrinking the interval if needed. Maintain only the K biggest values, say, $K = 1000$. If s is the minimum value of the tracked set, the number of distinct elements is approximately $K/(1 - s)$.

Definition 3.3 (MinTopK Estimate)**.**

- Pick a hash function h that maps m elements to a float in the interval $[0, 1]$.
- For each stream element a, calculate $h(a)$ and store the hash if it is in the top K hashes.
- The number of distinct elements is approximately $K/(1 - s)$.

3.3.2 Flajolet-Martin

The intuition is that you hash passing elements into short bitstrings, store only the length of the longest tail of zeroes, and the more distinct elements, the longer the longest tail of zeroes. To estimate the number of distinct elements, use $2^R/\Phi$:

Definition 3.4 (Flajolet-Martin)**.**

- Pick a hash function h that maps m elements to lgm bits.
- For each stream element a, let $r(a)$ be the number of trailing 0s in $h(a)$
- Record R = the maximum $r(a)$ seen
- Estimate the number of distinct elements as $\frac{1}{\Phi}2^R$ where $\Phi = 0.77351$

The probability that $h(a)$ ends in at least r zeroes is 2^{-r}. If there are m differente elements, the probability that none of them have r zeroes (so $R \geq r$) is $1 - (1 - 2^{-r})^m$. Pitfalls are that 2^R is a power of 2, meaning that there are larger jumps as we come across larger items. Bad luck can return in huge errors, but we can work around that by running parallel copies using different hash functions and average the results.

3.3.3 LogLog

An update to the Flajolet-Martin algorithm by Durand and Flajolet is the LogLog algorithm. This uses stochastic overaging and calibration, where the samples are partitioned into $n = 2^l$ groups using the first l bits of the hash function as a selector. If $n = 1024$, the relative error is around 3% to 4%

Definition 3.5 (LogLog)**.**

- Partition the samples into $n = 2^l$ groups, with the first l bits of the hash as selector
- Calculate $R_1, \ldots, R_n$
- Return $a_n * n * 2^{\text{mean}(R_1,\ldots,R_n)}$

This results in bit strings of length $logn$ and we maintain the length of the longest tail of zeroes: $loglogn$.

4 Similarity Search

In the task of determining the similarity between users U_1 and U_2 based on the sets of movies they have watched, a naive approach involves comparing all possible user pairs. This results in a computational complexity of $O(n^2)$, where n is the number of users. To address this challenge efficiently, we focus on identifying pairs of users with high similarity.

4.1 Jaccard Similarity

One commonly used metric for measuring the similarity between two sets is the Jaccard Similarity. This metric quantifies the overlap between two sets by comparing the size of their intersection to the size of their union.

Definition 4.1 (Jaccard Similarity)**.**

$$\text{Similarity}(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$$

The assumption here is that we don't have to bother about all the movies that we didn't watch together, it only takes into account the movies that we did see.

4.2 Candidate Selection for Similar Pairs

In this case we would still have to execute this similarity calculation on all possible user pairs, even those that have a very small overlap. To select a set of users that we will run similarity tests on, we can use min-hashing to generate signatures and use locality-sensitive hashing to generate candidate pairs to test for similarity.

First, we run a min-hashing algorithm to generate signatures, which are short integer vectors that represent the sets and reflects their similarity. Then, we run locality-sensitive hashing to get pairs of signatures that we need to test for similarity.

4.2.1 Shingles

If we don't look at user-movie selection, but rather document-text selection to do for example plagiarism checks, we can use shingles (or grams) as a measure of similarity. For example, if we have a document $abcab$ and we take $2-grams$ we get the set of ab, bc, ca. Alternatively, instead of characters, we can use a sequence of k consecutive words. For example, $childinhiseyes$ will result in $childin, inhis, hiseyes$. Finally, we represent the document by a binary column of the set of k-shingles. Documents with a lot of shingles in common have similar text, even if the text appears in a different order. However, you need to pick k large enough, or most document will have most shingles. To compress long shingles, we can hash them to uints as we can easily compare and store such values.

4.2.2 Minhashing

We can take the column of the user-movie, or document-shingles binary matrix and replace the binary columns with a short signature. The goal is that if the columns are similar, the signatures should be similar. We can't sample from the rows and let the signature be those bits only, as the input matrix is likely to be very sparse. Rather, we use the notion that the ordering of the matrix is not relevant to the similarity calculation.

Definition 4.2 (Minhashing)**.**

- Permute (swap) the rows randomly (or generate a random sequence of row indices)
- Define hash function $h(C)$ as the position of the first permuted row in C that has a 1.
- Run this several (i.e. 100) times independently and concatenate to create a signature
- The similarity of two signatures is then the fraction of signature elements that match

We see that the probability that $h(C_1) = h(C_2)$ is very close to $\text{Sim}(C_1, C_2)$. Consider that if we check two hashes and they are not the same, we note that down as a type S row. If they are equal we note that down as a type B row. The overall probability is then $\frac{B}{B+S}$ or, the equal over the total rows, which is the same as the Jaccard Similarity (intersection over union).

The main challenge of minhashing arises when dealing with a very large number of entries, as generating and storing a true random permutation of rows becomes computationally expensive and memory-intensive. Furthermore, accessing rows in the permuted order can lead to inefficient memory access patterns, such as thrashing or excessive swapping. To address this, instead of explicitly permuting rows, we use $n \to n$ hash functions to simulate the effect of permutations. By processing the matrix top-down row by row, we update the signature values for each column and each hash function in parallel, keeping track of the minimum hash values. This approach avoids the need for explicit permutations.

Definition 4.3 (Minhashing without Permutation)**.**

- Define a family of $n \to n$ hash functions $h_1, h_2, \ldots, h_k$ to simulate the effect of row permutations.
- Initialize the signature matrix Signature$[i][C]$ with ∞ for all hash functions i and columns C.
- Process the binary matrix row by row:
  - For each row r:
    - Compute $h_i(r)$ for all hash functions $h_1, h_2, \ldots, h_k$.
    - For each column C in that row where the matrix has a 1:
      - For each hash function h_i, update the signature value:

$$\text{Signature}[i][C] = \min(\text{Signature}[i][C], h_i(r))$$

- The resulting signature matrix stores the smallest hash values observed for each column and each hash function, effectively simulating the effect of random row permutations.

4.2.3 Locality-Sensitive Hashing

Now that we have converted the large and sparse binary matrix to a smaller matrix of similarity hashes, the memory constraint is removed. However, calculating the real similarities between the pairs is still an expensive task. The idea behind locality-sensitive hashing is to split the columns of the signature matrix into bands of the same size. If two columns are very similar, then it is likely that at least one band will be identical.

Instead of first calculating the bands, we can determine candidate pairs from columns that hash at least once to the same bucket. Split the rows up into bands and calculate a number of hashes for all columns in the bands. When two columns hit the same bucket, consider them similar and record the pair.

Definition 4.4 (Locality-Sensitive Hashing)**.**

- Split the signature matrix in b bands
- For over all the columns c in all bands calculate hashes that map to k buckets.
- Whenever a column c in a band maps to the same bucket as another column, consider them similar and record the pair.

Summarizing, the probability that the signatures agree on one row is s (the Jaccard similarity) and $P_{\text{band}} = s^k$ that they agree on one particular band (or $1 - s^k$ that they are not identical). The probability that they are not identical in any of the b bands is $(1 - P_{\text{band}})^b$. Finally, the probability of becoming a candidate pair is $t \approx \frac{1}{b}^{\frac{1}{r}} = \sqrt[r]{\frac{1}{b}}$